EECS3999 - Modern Cyber Foundation
IAM Workshop Lab: Advanced Cloud PKCE-based IAM with OIDC/OAuth 2.1

2025-03-14

Contents

1 Lab Overview: Identity and Access Management (IAM)

This lab provides an in-depth, hands-on workshop on **Identity and Access Management (IAM)** and its critical role in securing modern applications. Students will gain a foundational understanding of IAM principles, including **authentication, authorization, and access control**.

The lab will focus on implementing **advanced PKCE-based OpenID Connect (OIDC) / OAuth 2.1** for secure authentication and authorization. Additionally, it will cover **Role-Based Access Control (RBAC)** to enforce user permissions and manage access rights for **administrators and staff members**.

We will explore the risks associated with **authorization flow interception** and demonstrate how **Proof Key for Code Exchange (PKCE)** mitigates such threats. The hands-on implementation will leverage **IBM Cloud App ID** as the identity provider, with the flexibility to adapt the solution to other cloud providers.

Furthermore, the lab will introduce key security enhancements such as **Multi-Factor Authentication (MFA)** and **One-Time Passwords (OTP)**, ensuring an additional layer of protection against unauthorized access and credential-based attacks.

2 Lab Roadmaps

In future labs, we will architect and implement a complete **identity infrastructure** from the ground up using **enterprise-grade open-source solutions**, such as the **Red Hat Authorization Server**, built on the **Keycloak IAM platform**. This will provide a deeper understanding of self-hosted identity management, fine-grained authorization policies, and seamless integration with modern authentication protocols.

Furthermore, subsequent labs will delve into the deployment and optimization of **advanced API gateways** and endpoint protection, fortified with **Next-Generation Web Application Firewalls (NG-WAFs)** such as **AWS WAF and AWS Shield**. These technologies will be leveraged to enforce **OWASP Top 10 protections, dynamic threat mitigation, and intelligent rate throttling**, ensuring robust security against evolving cyber threats.

Towards the latter part of the course, we will dedicate an entire lab to **advanced anomaly detection**, demonstrating the power of **AI-driven Intrusion Detection and Prevention Systems (ID/IPS)**. This session will showcase how **machine learning models** can detect **behavioral anomalies, adaptive attack patterns, and automated botnet activities**, providing **real-time threat intelligence** and **proactive security enforcement**.

3 Learning Objectives and Outcomes

By the end of this lab, students will be able to:

1. Understand the fundamentals of IAM and its importance in securing modern applications.
2. Implement OAuth 2.1 / OIDC authentication flows using PKCE to prevent authorization flow interception attacks.
3. Configure Role-Based Access Control (RBAC) to enforce granular permissions for different user roles.
4. Secure APIs with token-based authentication and session management.
5. Integrate cloud solutions as identity providers while ensuring extensibility for other OIDC/OAuth solutions.
6. Enhance authentication security with Multi-Factor Authentication (MFA) and One-Time Passwords (OTP).
7. Gain exposure to other IAM solutions, including self-hosted authentication infrastructure using IBM Red Hat Keycloak.

After completing this lab, students will have acquired the following skills and knowledge:

- A solid grasp of IAM principles and modern authentication frameworks.
- The ability to implement a secure and scalable authentication system in real-world web applications.
- Hands-on experience configuring and deploying OAuth 2.1 / OIDC authentication with PKCE.
- Proficiency in enforcing RBAC policies to control user access effectively.
- The capability to integrate MFA and OTP-based security enhancements for stronger authentication measures.

4 Background - Paradigm of Modern Security

As seen in Figure 1, Modern applications face a vast and ever-evolving spectrum of security threats, shaped by rapid technological advancements and increasingly sophisticated attack vectors. Security risks emerge from a combination of architectural weaknesses, software vulnerabilities, and the persistence of threat actors employing novel exploitation techniques. Organizations must navigate these challenges by implementing robust security strategies to safeguard sensitive data, ensure system availability, and maintain trust.

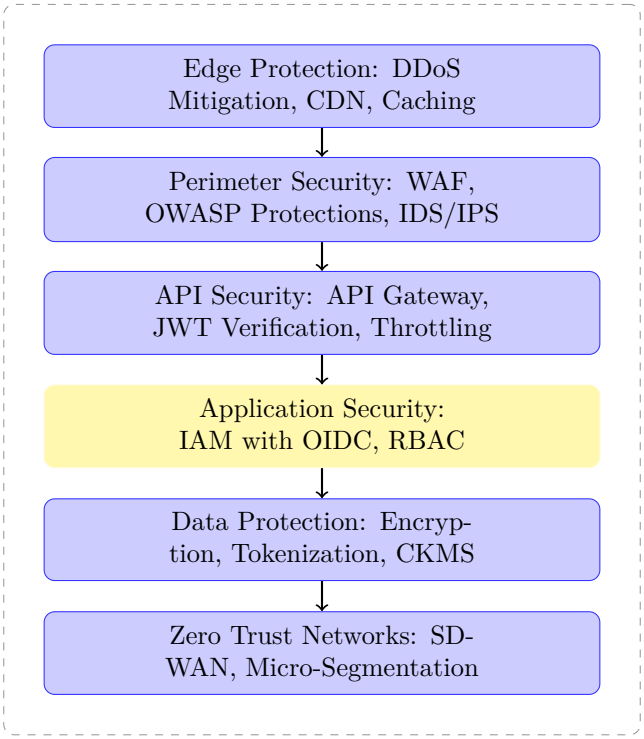

Figure 1: Modern Cybersecurity Layered Model

One of the most pressing concerns is **data breaches**, where unauthorized entities gain access to sensitive information. Such breaches often result in financial losses, identity theft, and significant reputational damage. Common causes include:

- SQL injection attacks
- Misconfigured cloud storage
- Inadequate encryption of data at rest

- Weak encryption key management practices
- Insufficient access controls

Ensuring proper tokenization, implementing strong encryption mechanisms, and enforcing strict access policies are critical to mitigating these risks.

API security flaws pose another major challenge, as APIs serve as the backbone of modern applications. Attackers frequently exploit:

- Inadequate authentication mechanisms
- Excessive data exposure
- Unprotected endpoints

Without proper API security measures, such vulnerabilities can lead to unauthorized access, data leakage, and service disruptions.

Distributed Denial of Service (DDoS) attacks remain a formidable threat, as they aim to overwhelm systems with excessive traffic, causing downtime and service disruption. Attackers leverage botnets to execute:

- Large-scale volumetric attacks
- Protocol-based attacks
- Application-layer floods

Mitigation requires scalable security solutions such as rate limiting and distributed filtering.

The **OWASP Top 10** categorizes the most critical security risks in web applications, highlighting vulnerabilities such as:

- Broken authentication
- Security misconfigurations
- Insufficient logging and monitoring
- Exposure to injection attacks

Organizations must align their security practices with OWASP guidelines to enhance their resilience against these common attack vectors.

A paradigm shift in cybersecurity has led to the rise of **Zero Trust security models**, which reject the traditional perimeter-based approach in favor of:

- Least privilege access principles
- Mandatory multi-factor authentication (MFA)
- Micro-segmentation
- Continuous monitoring

Zero Trust minimizes the risk of lateral movement by attackers.

In this evolving security landscape, **Identity and Access Management (IAM)** and **API protection** are indispensable components. IAM solutions provide:

- Centralized management of user identities
- Enforcement of strong authentication policies
- Governance over API access through OAuth2, OpenID Connect (OIDC), and other security frameworks

By implementing a robust IAM framework, organizations can mitigate unauthorized access and protect sensitive assets. Please noe that this will be our focus in this lab and we primarly work on enhaced OIDC-cloud-based IAM.

4.1 Layered Security Model

In this lab, while our primary focus is on Identity and Access Management (IAM), it is essential to understand how it fits within the broader security landscape. IAM is a critical component of modern security architectures, which typically follow a **multi-layered security model**. This approach integrates multiple defense mechanisms to mitigate threats at various levels, reducing the attack surface and ensuring that no single point of failure compromises the entire system. By briefly reviewing key concepts in multi-layered security architectures, we can better appreciate the role of IAM within a defense-in-depth strategy that provides robust protection against a wide range of cyber threats.

4.1.1 Edge Protection Layer

At the **edge protection layer**, DDoS mitigation services such as AWS Shield and Cloudflare provide automated protection against volumetric attacks. Additionally, Content Delivery Networks (CDNs) like CloudFront and Akamai help distribute traffic, reduce latency, and add an extra security layer against web-based threats.

4.1.2 Perimeter Security

Perimeter security measures involve next-generation Web Application Firewalls (WAFs) that protect against OWASP vulnerabilities while integrating with:

- Intrusion Detection Systems (IDS)
- Intrusion Prevention Systems (IPS)

These solutions monitor network traffic, detect anomalies, and actively block malicious activity in real time.

4.1.3 API Security

API security is enforced through API gateways that implement:

- Rate limiting
- Authentication mechanisms such as JWT validation
- API keys to prevent abuse

Security protocols like OAuth2 and OIDC enhance authentication mechanisms, ensuring secure access control.

4.1.4 Application Security

Application security focuses on:

- Enforcing Role-Based Access Control (RBAC) through IAM
- Implementing secure coding practices
- Adopting vulnerability management frameworks to address potential exploits before attackers can leverage them

4.1.5 Data Protection

Data protection mechanisms include:

- Encryption for data in transit (such as **TLS 1.3**) and at rest (such as **AES-256**)
- Tokenization techniques that replace sensitive data with non-sensitive tokens
- Centralized Key Management Systems (CKMS) to securely handle and rotate cryptographic keys

4.1.6 Zero Trust and Micro Segmented Networks

Zero Trust networks leverage:

- Software-Defined Wide Area Networks (SD-WAN) for dynamic traffic routing based on security policies
- Micro-segmentation techniques to restrict lateral movement within a network, ensuring that a compromised node cannot grant attackers unrestricted access to critical assets

4.1.7 DDoS Protection and CDNs

DDoS attacks are designed to disrupt online services by overwhelming networks, servers, or applications with excessive requests. These attacks result in significant downtime, financial losses, and reputational damage. DDoS threats are categorized into three primary types:

1. **Volumetric Attacks**
    - **UDP Flood**: Attackers send large volumes of UDP packets to overwhelm the target.
    - **ICMP Flood**: Exploits the ICMP protocol to exhaust bandwidth.
2. **Protocol Attacks**
    - **SYN Flood**: Exploits the TCP handshake process to deplete server resources.
    - **Slowloris**: Keeps HTTP connections open for prolonged durations, exhausting server capacity.
3. **Application-Layer Attacks**
    - **HTTP Floods**: Bombard web applications with excessive HTTP requests, consuming server resources.

To counteract DDoS threats, organizations deploy mitigation techniques such as:

- **Rate limiting**: Restricts the number of requests from a single IP address.
- **Traffic scrubbing**: Analyzes and filters malicious traffic in real-time.
- **Anycast network distribution**: Distributes traffic across multiple locations, reducing the impact of large-scale attacks.

Content Delivery Networks (CDNs) further enhance security by caching content at edge locations, reducing latency, and mitigating traffic spikes. Load balancing mechanisms distribute traffic evenly across servers, preventing single points of failure. Additionally, content optimization techniques reduce server load by caching static assets, including images, JavaScript, and CSS files.

4.1.8 Next-Gen Web Application Firewalls (NGWAF) and API Security

A **Web Application Firewall (WAF)** is essential for protecting web applications from various attack vectors. NGWAFs leverage machine learning-based anomaly detection to identify deviations from normal traffic patterns, allowing for proactive defense against cyber threats. NGWAFs prevent **OWASP Top 10 attacks**, including SQL Injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), and Broken Authentication.

By integrating with Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS), NGWAFs provide deeper monitoring capabilities using deep packet inspection. Behavioral analysis and AI-driven threat intelligence help detect automated bot activities and abnormal traffic patterns.

As you may say, WAFs are essential for protecting web applications from various attack vectors. NGWAFs further can leverage machine learning-based anomaly detection to identify deviations from normal traffic patterns, allowing for proactive defense against zero-day cyber threats. By integrating with Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS), NGWAFs provide deeper monitoring

capabilities using deep packet inspection. Behavioral analysis and AI-driven threat intelligence help detect automated bot activities and abnormal traffic patterns.

API security is a critical aspect of modern cybersecurity, as APIs serve as gateways to sensitive data and system functionality, making them prime targets for cyberattacks. Due to their widespread integration with various services, APIs are vulnerable to threats such as credential stuffing, injection attacks, and API scraping, where attackers exploit vulnerabilities to extract large volumes of data. To mitigate these risks, organizations implement API gateways that enforce robust security measures, including OAuth2 authentication, API key enforcement, rate limiting, request throttling, and always-on JWT validation with HMAC to prevent replay attacks. These security controls ensure that only authorized users and applications can access APIs while preventing abuse, unauthorized access, and session hijacking attempts, ultimately safeguarding sensitive data and maintaining the integrity of API-driven systems.

As you can imagine, by implementing a multi-layered security approach, enterprise organizations can effectively mitigate threats, ensure regulatory compliance, and safeguard critical assets against the ever-evolving landscape of cyber threats. In this lab, we will begin with the foundation of security—**Identity and Access Management (IAM)**—and progressively build a fully functional, secure application together.

5 Introduction to Identity and Access Management (IAM)

In this lab, we will explore the foundational principles of Identity and Access Management (IAM), a critical first step in application development to ensure that access is restricted to authorized users only. Effective IAM implementation is fundamental to securing modern applications, as it governs authentication, authorization, and access control. Our goal is to demonstrate how IAM serves as the backbone of application security, enabling developers to establish robust identity verification mechanisms and enforce granular access policies. Understanding IAM is essential for students who aspire to build secure, scalable applications that adhere to best practices in cybersecurity.

We will demonstrate an advanced implementation of **PKCE-based OpenID Connect (OIDC) / OAuth 2.1** for secure authentication and authorization within applications. Additionally, we will implement **Role-Based Access Control (RBAC)** to differentiate user permissions between administrators and staff members.

Furthermore, we will discuss how attackers attempt to intercept authorization flows and how implementing **Proof Key for Code Exchange (PKCE)** mitigates these threats. This lab will include a fully functional code demonstration using **IBM Cloud** as the identity provider (which can be replaced with other cloud providers as well). The application can also be extended to support **Multi-Factor Authentication (MFA)** and **One-Time Passwords (OTP)** for enhanced security. Let's start by reveiwing the basics.

IAM is a framework of policies and technologies that ensures the right individuals and devices have appropriate access to a system. It governs:

- **Authentication**: Verifying that a user is who they claim to be (e.g., logging in with a username and password, biometrics, or OAuth-based authentication).
- **Authorization**: Granting users specific permissions based on their identity and role (e.g., an admin can delete users, but staff members cannot).
- **Access Control**: Restricting access to sensitive data and resources based on predefined security policies.

5.1 OAuth 2.1 and OpenID Connect (OIDC)

OAuth 2.1 is an authorization framework that enables secure access delegation, allowing users to grant applications limited access to their data without exposing credentials. OpenID Connect (OIDC) is an authentication layer built on OAuth, enabling user identity verification and single sign-on (SSO).

OAuth and OIDC use various authorization flows, and in this lab, we focus on the secure **PKCE-based Authorization Code Flow** for enhanced security.

5.2 PKCE (Proof Key for Code Exchange)

PKCE is a security enhancement for OAuth 2.1 and OIDC that prevents authorization code interception attacks. When a client requests an authorization code, PKCE ensures that only the legitimate client can exchange the code for an access token, mitigating attacks such as:

- **Authorization Code Interception**: Attackers steal the authorization code and use it to gain unauthorized access.
- **Man-in-the-Middle (MITM) Attacks**: Attackers intercept network traffic to steal access tokens.

PKCE works by adding a **code verifier** and a **code challenge** in the authentication process. The client generates a cryptographically random code verifier, derives a code challenge from it using SHA-256, and sends it to the authorization server. During token exchange, the client must provide the original code verifier, ensuring that only the legitimate client can complete the process.

5.3 Role-Based Access Control (RBAC)

RBAC is a security model that assigns permissions to users based on their roles. In this lab, we will define two roles:

- **Admin Role**: Users with administrative privileges who can perform high-privilege actions (e.g., managing users, modifying settings).
- **Staff Role**: Users with limited access who can perform specific, restricted actions.

RBAC ensures that different users have different levels of access to secure resources, reducing the risk of unauthorized operations.

5.4 Introduction to Identity and Access Management (IAM)

In this lab, we will explore the foundational principles of **Identity and Access Management (IAM)** and how it plays a crucial role in securing modern applications. Understanding IAM is essential for students aspiring to build secure applications, as it governs user authentication, authorization, and access control.

We will demonstrate an advanced implementation of **PKCE-based OpenID Connect (OIDC) / OAuth 2.1** for secure authentication and authorization within applications. Additionally, we will implement **Role-Based Access Control (RBAC)** to differentiate user permissions between administrators and staff members.

Furthermore, we will discuss how attackers attempt to intercept authorization flows and how implementing **Proof Key for Code Exchange (PKCE)** mitigates these threats. This lab will include a fully functional code demonstration using **IBM Cloud** as the identity provider (which can be replaced with other cloud providers as well). The application can also be extended to support **Multi-Factor Authentication (MFA)** and **One-Time Passwords (OTP)** for enhanced security.

5.5 Cloud-based Identity Providers

IBM Cloud App ID is a robust cloud-based **Identity Provider (IdP)** that simplifies authentication and access management for modern applications. By leveraging IBM Cloud App ID, developers can seamlessly integrate industry-standard authentication protocols such as:

- OAuth 2.1
- OpenID Connect (OIDC)
- Multi-Factor Authentication (MFA)
- One-Time Passwords (OTP)

This eliminates the need for organizations to host and manage their own authorization servers, reducing infrastructure complexity while ensuring secure and scalable authentication mechanisms.

IBM Cloud App ID provides a managed authentication service, offering features such as:

- User authentication
- Identity federation
- Role-Based Access Control (RBAC)
- Token management

With support for both cloud-native and traditional applications, it enables enterprises to secure APIs, web applications, and mobile applications without implementing custom authentication logic. Below diagram demostrate a typical OAuth flow.

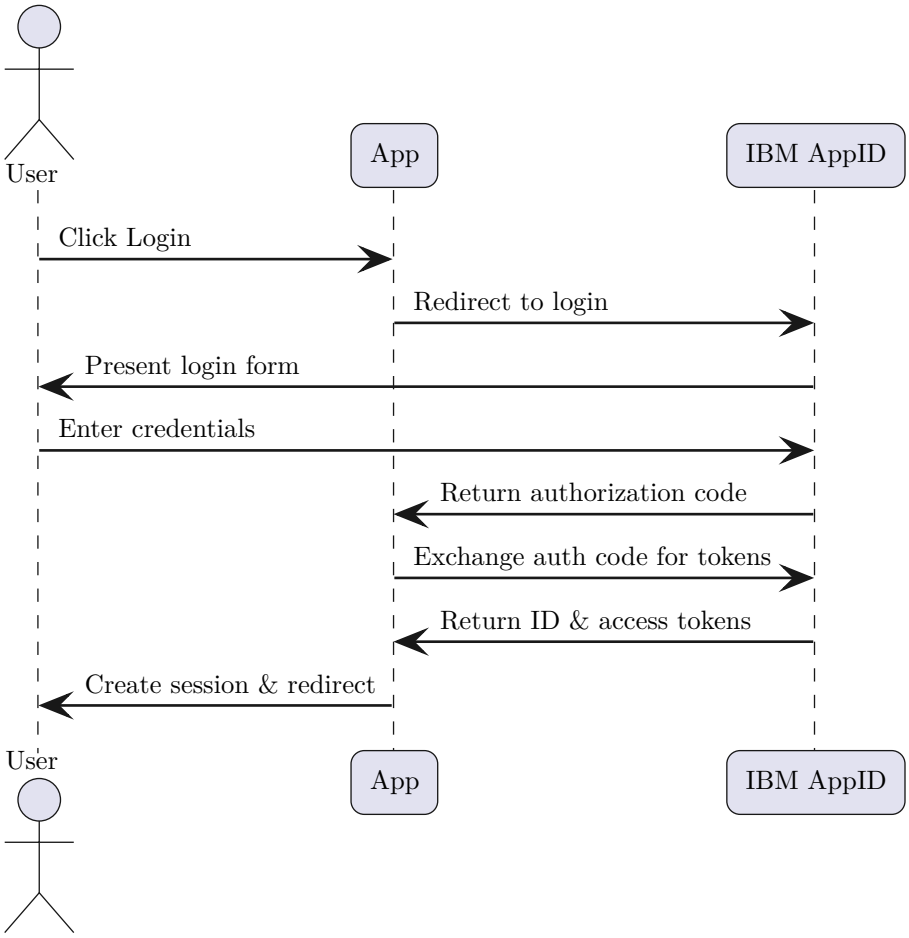

Figure 2: OpenID Connect Authentication Flow

As can be seen in Figure 2, we use OIDC for authentication between a User, an Application (App), and IBM AppID as the identity provider. The process begins when the User clicks "Login", prompting the App to redirect the user to IBM AppID for authentication. IBM AppID presents a login form, and the User enters credentials, after which IBM AppID returns an authorization code to the App. The App then exchanges this code for ID and access tokens from IBM AppID. Once the tokens are received, the App creates a session for the User and redirects them to the authenticated state, completing the authentication flow securely.

In upcoming labs, we will delve deeper into building our own Identity and Access Management (IAM) infrastructure from scratch. We will achieve this by deploying an open-source **Red Hat Authorization**

Server, which is built on the powerful **Keycloak** open-source IAM platform. We will demonstrate Keycloak as an enterprise-grade IAM solution that supports advanced authentication mechanisms, including:

- Single Sign-On (SSO)
- Adaptive authentication
- Fine-grained authorization policies
- Integration with various identity providers

However, in this lab, we will use IBM Cloud App ID to reduce the setup time so that you can quickly understand the overall OIDC flows quicker.

- Authenticate users using OAuth 2.1 and OIDC.
- Implement PKCE to enhance security.
- Enforce RBAC to control access based on user roles.
- Extend authentication with MFA and OTP for additional security layers.

We will discuss each topics in details in subsequent sections.

6 Basics Needed to Understand OIDC/OAuth-based IAM

JWT (JSON Web Token) and JWKS (JSON Web Key Set) are powerful tools for secure and scalable authentication in modern web applications. JWTs are used for representing claims securely between parties. They consist of three parts: a header, a payload, and a signature. The header typically consists of the type of token (JWT) and the signing algorithm being used. The payload contains the claims, which are statements about an entity (typically, the user) and additional data. The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.

JWKS, on the other hand, provides a standardized way to publish public keys for verifying JWT signatures. A JWKS endpoint is a URL where a set of keys is published. Each key can be used to validate the signature of a JWT. This mechanism allows servers to rotate keys and clients to automatically discover new keys, ensuring secure and seamless operations without downtime or the need for manual intervention.

OAuth (Open Authorization) and OIDC (OpenID Connect) further enhance the security and scalability of web authentication. OAuth 2.1 is an authorization framework that enables third-party applications to obtain limited access to a service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. It allows users to authorize third-party applications to access their information without sharing their credentials.

OIDC (OpenID Connect) is an identity layer on top of OAuth. It allows clients to verify the identity of the end-user based on the authentication performed by an authorization server and to obtain basic profile information about the end-user. OIDC introduces the concept of an ID Token, which is a JWT that contains user authentication information, such as the user's identifier, the time the token was issued, and any other claims necessary for the client to understand who the user is. It simplifies the process of federated authentication by providing a standardized identity protocol, making it easier to implement and ensuring interoperability. Common use cases include single sign-on (SSO) and delegated access to resources on behalf of the user.

In below subsections we discuss the details.

6.1 JSON Web Token (JWT) and Related Cryptography Concepts

JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure, enabling the claims to be digitally signed or integrity-protected with a Message Authentication Code (MAC) and/or encrypted. A JWT is composed of three parts and each part is encoded in Base64Url and separated by dots:

SomeHeader.SomePayload.SomeSignature

To gain a deeper understanding of the signature in JWTs, it is essential to first comprehend the cryptographic principles involved in signing a JWT. Let's explore the required cryptographic concepts.

6.1.1 HS256 Overview

Before we start, let's review HMAC, or **Hash-based Message Authentication Code**, which is a method used to verify the integrity and authenticity of a message using a cryptographic hash function along with a secret key. The purpose of HMAC is to provide a means of checking whether a message has been tampered with, as well as ensuring the authenticity of the message's origin.

1. **Hash Function**: HMAC can be used with any cryptographic hash function (e.g., MD5, SHA-1, SHA-256), though SHA-256 is often preferred for its security.
2. **Secret Key**: HMAC is symmetric, meaning the same secret key is used for both generating and verifying the code. Only parties that have the key can generate or verify an HMAC, which provides mutual trust.
3. **Construction**: HMAC combines the message with the secret key and applies the hash function to produce a code. This output code is unique to the message and key, and if either is changed, the HMAC result changes.
4. **Verification**: The recipient, with access to the shared secret key, can recompute the HMAC on the received message. If the computed HMAC matches the one sent with the message, the message is validated as genuine.

HS256, which is HMAC with SHA-256 is a specific implementation of HMAC that uses SHA-256 as the hashing function. It can be used for:

1. **Signing a Message**:
   - The sender creates a hash of the message, combined with the secret key, using SHA-256.
   - This produces a unique code (or signature) based on the message content and the secret key.
2. **Validating a Message**:
   - The receiver uses the same SHA-256 hashing function and the shared secret key on the received message.
   - If the generated HMAC matches the provided signature, it verifies the message's authenticity and integrity.

HS256 is commonly used in **JSON Web Tokens (JWTs)** for secure communication between a client and server. In a JWT, the payload is signed with HS256, allowing the server to verify that the token was created by a trusted source and has not been altered. It provides a symmetric key mechanism where both parties share a secret key, which is then used to generate and verify signatures, ensuring message integrity and authenticity.

6.1.2 RS256 Overview

RS256 is a cryptographic signing algorithm that combines RSA (Rivest-Shamir-Adleman) encryption with SHA-256 hashing to create a secure digital signature. It's an asymmetric signing algorithm commonly used in JSON Web Tokens and other protocols where secure verification of message authenticity is needed. RS256 is particularly favored for scenarios where the verifying party should not have access to the signing key, as it uses separate keys for signing and verification.

Unlike HMAC, RS256 is an asymmetric algorithm, meaning it uses two different keys:

- **Private Key**: Used by the sender to generate a digital signature.

- **Public Key**: Used by the receiver to verify the digital signature.

The private key must remain secure, as it is used to sign the message, while the public key is distributed to others for verification. This separation allows third parties to verify the authenticity of a message without having access to the private key, enhancing security.

As we know, RSA is a widely used public-key cryptosystem, chosen in RS256 because of its robustness in securing data. It provides the foundation for the asymmetric nature of the algorithm.

RS256 uses the SHA-256 hashing function to compute a fixed-length digest of the message content. The digest is then encrypted with the RSA private key to create a signature. SHA-256 is chosen here for its balance of security and efficiency. It can be used for:

Signing a Message:

1. The sender hashes the message with SHA-256, producing a fixed-length hash.
2. The sender then uses the RSA private key to encrypt this hash. The encrypted hash becomes the digital signature.
3. The message and the signature are then sent together to the receiver.

Validating a Message:

1. The receiver uses the sender's public key to decrypt the signature, retrieving the original SHA-256 hash created by the sender.
2. The receiver then hashes the received message independently using SHA-256.
3. If the newly computed hash matches the decrypted hash, the message is authentic and has not been altered in transit.

RS256 can offer strong security for applications requiring verifiable, tamper-resistant messages, with an extra layer of trust facilitated by the use of separate signing and verification keys for scenarios like secure authentication:

- A server signs a JWT containing user information with its private key.
- Clients can then use the server's public key to verify the token's authenticity, ensuring that it was indeed generated by the server and hasn't been tampered with.

6.1.3 JWT Header

Back to our JWT explnation, the header typically consists of two parts: the type of token and the signing algorithm (e.g., HMAC SHA256).


```
{
  "alg": "HS256",
  "typ": "JWT"
}
```


6.1.4 JWT Payload

The payload contains the claims. Claims are statements about an entity (typically, the user) and additional data.


```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```


6.1.5 JWT Signature

To create the signature part, you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that:

HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)

6.2 HS256 vs. RS256

As we discussed, RS256 and HS256 are both cryptographic signing algorithms often used in JSON Web Tokens and some other secure communication protocols. The choice between RS256 and HS256 depends on specific requirements for **security**, **performance**, and **key management complexity**.

- **HS256** is ideal for performance-critical applications with controlled access to a shared key.
- **RS256** is preferred in distributed systems that require strong security and public verification.

In terms of their security model we can say:

- **HS256 (HMAC with SHA-256)**
  - **Symmetric**: HS256 is symmetric, meaning the same secret key is used for signing and verifying. This requires both the sender and receiver to share the secret key, which increases risk if the key is exposed.
  - **Pros**: Simpler to implement since only one key is required.
  - **Cons**: Key management is challenging in distributed systems, as all parties need to protect the shared key.
- **RS256 (RSA with SHA-256)**
  - **Asymmetric**: RS256 uses a private key for signing and a public key for verification. Only the sender holds the private key, while the public key can be distributed for verification.
  - **Pros**: More secure for distributed systems; only the private key needs protection.
  - **Cons**: Requires managing a key pair, which adds complexity.

In terms of their efficiency we can say:

- **HS256**
  - **Performance**: Faster, as symmetric operations are quicker than asymmetric.
  - **Efficiency**: Ideal for high-speed processing in low-latency environments.
  - **Resource Usage**: Requires less computational power, making it suitable for embedded systems or devices with limited resources.
- **RS256**
  - **Performance**: Slower due to computationally intensive RSA operations.
  - **Efficiency**: Less efficient for systems needing fast verification of many signatures.
  - **Resource Usage**: Higher resource usage, which may affect real-time systems or hardware-constrained environments.

In terms of their key length and signature size we can say:

- **HS256**
  - **Key Length**: Typically uses a 256-bit secret key.
  - **Signature Size**: Fixed at 256 bits (32 bytes) with SHA-256, regardless of message length.

- **RS256**
  - **Key Length**: Commonly 2048 or 4096 bits, depending on security needs.
  - **Signature Size**: Depends on RSA key length (e.g., 256 bytes for a 2048-bit key).

In terms of their usecase model we can say:

- **HS256**
  - **Best Suited For**: High-performance applications like IoT, embedded systems, or real-time processing where key sharing is manageable.
  - **Examples**: Often used in internal API authentication where the secret key can be securely shared.
- **RS256**
  - **Best Suited For**: Distributed systems or applications requiring public verification without exposing the private key.
  - **Examples**: Used in OAuth 2.1 and JWTs in open systems where verification by multiple parties is necessary.

Feature	HS256 (Symmetric)	RS256 (Asymmetric)
Key Management	Simple, single shared secret key	Requires key pair (private public keys)
Security	Secure, but depends on secret key secrecy	More secure for distributed verification
Performance	Faster, low computational overhead	Slower, higher computational overhead
Resource Usage	Lightweight	Heavier resource consumption
Key Length	Shorter (e.g., 256 bits)	Longer (e.g., 2048+ bits)
Signature Size	Fixed and small (32 bytes)	Larger (depends on RSA key size)
Use Cases	Best for internal or high-performance apps	Best for distributed or public verification

6.3 JSON Web Key Set (JWKS)

JSON Web Key Set (JWKS) is a JSON structure that represents a set of JSON Web Keys (JWK). This data structure can be used to represent one or more keys, which can be used for various purposes, including signature verification. A JWKS typically contains a keys property, which is an array of JWK objects. Here is an example:

```
{
  "keys": [
  {
    "kty": "RSA",
    "kid": "1234ABCD",
    "use": "sig",
    "n": "0vx7agoebGcQSuuPiLJXZptN4nrh9m5dI5ySTwh8dG...",
    "e": "AQAB"
  }
  ]
}
```

Please note:

- kty: Key Type (e.g., "RSA")
- kid: Key ID, used to match a specific key (e.g., "1234ABCD")

- use: Public Key Use, sig (signature) or enc (encryption) (e.g., ”sig”)
- n: Modulus (base64url-encoded) for RSA
- e: Exponent (base64url-encoded) for RSA

6.4 How JWT and JWKS Work Together

Using JWKS, services can dynamically retrieve and rotate public keys without downtime. This enhances security and flexibility, which is particularly useful in distributed systems and microservices architectures. Tokens such as JWT can be verified as well

6.4.1 Token Issuance and Verification

When a client authenticates and requests a token, the server creates a JWT signed with its private key. Further, when the client makes a request with the JWT, the server can verify the token's signature using the corresponding public key found in the JWKS. Here is a typical workflow:

1. Resource server fetches the JWKS from the authorization server.
2. Finds the public key with the matching kid.
3. Verifies the JWT's signature using this key. If valid, processes the request.

6.4.2 Client Authentication

Client authenticates with the server, while, server issues a JWT to the client, signed with the server's private key. Here is a example of client making a request to a resource server with the JWT in the Authorization header.

```
GET /protected/resource HTTP/1.1
Host: resource.server.com
Authorization: Bearer eyJhbGciOiJSUzI1NiIsImt...
```

6.4.3 Key Rotation and Public Key Distribution (PKD)

The JWKS allows for easy rotation and invalidation of keys without impacting clients, as they can dynamically fetch the latest set of keys. This helps with Public Key Distribution (PKD). For instance, the authorization server provides a well-known JWKS endpoint where it publishes its public keys. These keys are used by clients and resource servers to verify the signatures of JWTs. The JWKS allows for smooth key rotation. The authorization server can further update the keys in the JWKS endpoint, and clients can periodically fetch the latest keys without service disruption.

6.5 JWKS with Elliptic Curve Cryptography (ECC)

Elliptic Curve Cryptography (ECC) is a form of public key cryptography based on the algebraic structure of elliptic curves over finite fields. ECC keys are smaller, which results in faster computation, lower power consumption, and smaller memory requirements. Similar to RSA structure, a JWKS for ECC contains an array of JWK objects, each representing an elliptic curve key. The keys will have parameters specific to ECC, such as crv (curve), x, and y coordinates. Here is an example:


```
{
  ”keys”: [
  {
    ”kty”: ”EC”,
    ”kid”: ”1”,
    ”use”: ”sig”,
    ”crv”: ”P-256”,
    ”x”: ”f83OJ3D2xFMB2TcQvV5G9QtxHWEwYI6NL_dwPmg6rmI”,
```



```
    "y": "x_FEzRu9kiPSt9I1gTYV8ZOmNbebcfiZJ-2V0y8BEp8",
    "alg": "ES256"
  }
 ]
}
```


Please note:

- kty: Key Type (e.g., "EC" for elliptic curve)
- kid: Key ID, used to match a specific key (e.g., "1")
- use: Public Key Use, "sig" (signature) or "enc" (encryption) (e.g., "sig")
- crv: Curve, indicates the name of the curve (e.g., "P-256")
- x: X coordinate of the elliptic curve point (base64url-encoded)
- y: Y coordinate of the elliptic curve point (base64url-encoded)
- alg: Algorithm, specifies the algorithm used with the key (e.g., "ES256" for ECDSA using P-256 and SHA-256)

6.6 Role of JWKS and JWT in OIDC

We will discuss OIDC in more details in the subsequent sections. For now, as can be seen in Figure 3, we can say OpenID Connect (OIDC) is an identity layer on top of the OAuth 2.1 protocol, enabling clients to verify the identity of an end-user based on the authentication performed by an authorization server, as well as to obtain basic profile information about the end-user.

The primary use of JWT in OIDC is for the ID Token, which is a security token that contains identity claims about the end-user. The ID Token is digitally signed using the server's private key. OIDC also uses JWTs for access tokens, which are used to authorize access to protected resources. Payloads typically contain claims about the user and other metadata, such as:

- iss (Issuer): URL of the issuing authorization server.
- sub (Subject): Identifier for the user.
- aud (Audience): Intended audience for the token.
- exp (Expiration time): When the token expires.
- iat (Issued at): When the token was issued.
- nonce: A random value to mitigate replay attacks.
- Signature: Ensures the token's integrity and authenticity.

Please note, for user authentication, the user logs in to an OIDC-compliant identity provider. The identity provider authenticates the user and issues a JWT ID Token to the client. The client receives the ID Token and needs to verify its authenticity. The client retrieves the JWKS from the identity provider's well-known configuration endpoint (e.g., https://auth.server.com/.well-known/openid-configuration), which includes the URL of the JWKS endpoint. The client finds the public key corresponding to the kid in the JWT header from the JWKS. The client uses the public key to verify the JWT's signature, ensuring the token's integrity and authenticity.

To access protected resources, the client includes the JWT access token in requests to the resource server. The resource server retrieves the JWKS from the identity provider. The resource server verifies the access token's signature using the appropriate public key from the JWKS. If the token is valid, the resource server processes the request.

The IETF draft of OAuth 2.1 for Browser-Based Apps outlines best practices for implementing OAuth 2.1 in browser-based applications. It uses OAuth 2.1 authorization code flow with Proof Key for Code

Exchange (PKCE) to enhance security. It handles authorization and refresh tokens, protecting against common vulnerabilities like Cross-Site Request Forgery (CSRF), and how to better secure storage of tokens. It also covers architecture patterns for JavaScript applications with or without a backend. The goal is to provide guidelines to securely manage OAuth flows within browser environments.

In the OAuth 2.1 authorization framework, the typical flow involves the user granting the application permission via the authorization server, which then issues an access token. The application uses this token to access resources from the resource server on behalf of the user. Here are some common terminology that we use:

- **User (Resource Owner - RO)**: The individual who owns the data and authorizes the application to access their resources.
- **Application (Client)**: The application requesting access to the user's resources. It can be a web app, mobile app, or other types of software.
- **Authorization Server - AS**: The server responsible for authenticating the user, obtaining their consent, and issuing access tokens to the client application. It manages the OAuth 2.1 flows.
- **Resource Server - RS**: The server hosting the protected resources. It verifies the access token and serves the requested resources to the client.

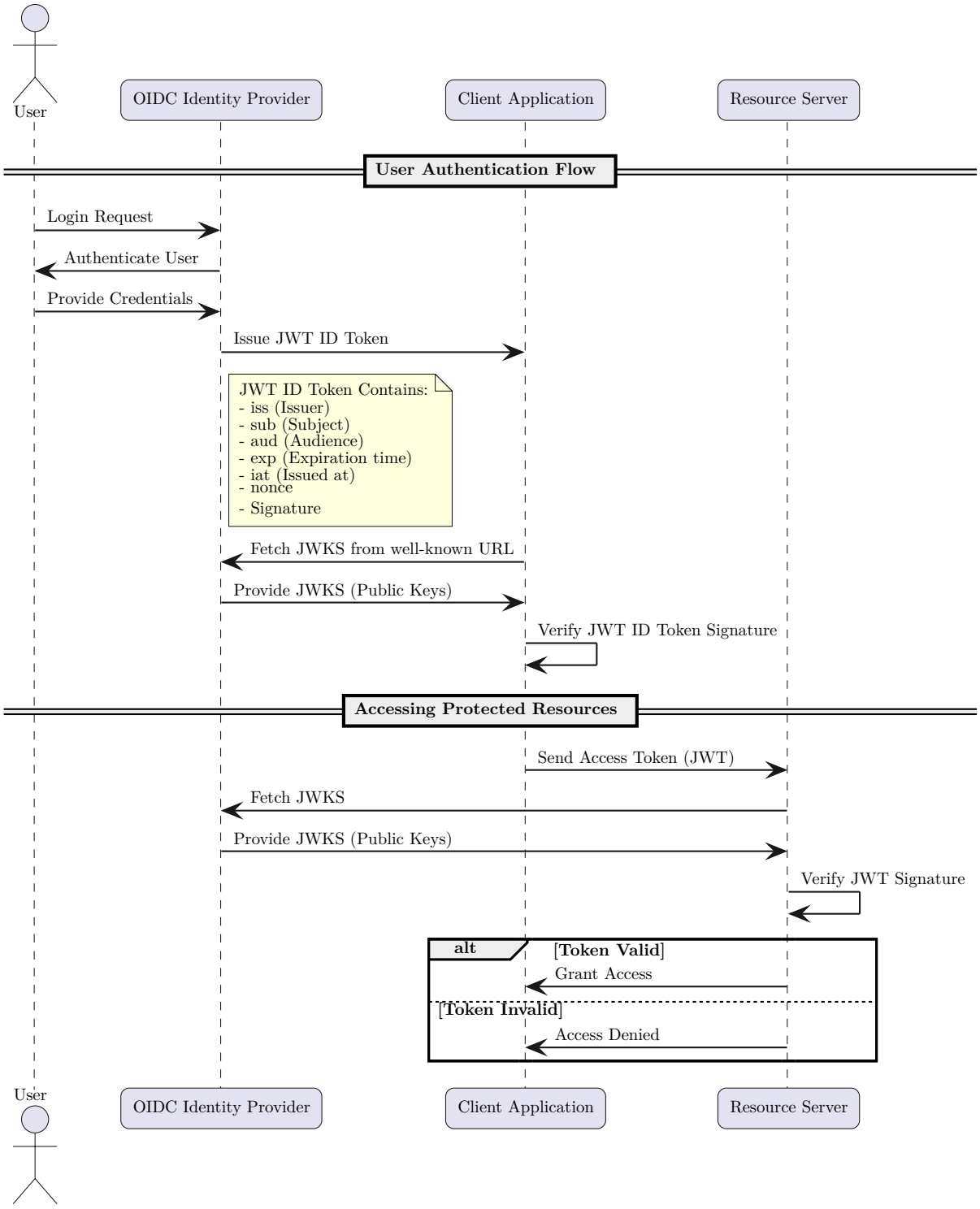

Figure 3: Role of JWKS and JWT in OIDC

6.7 OpenID Connect (OIDC)

OpenID Connect (OIDC) standard aims to add an identity layer on top of the OAuth 2.1 protocol. While OAuth 2.1 is primarily designed for authorization (granting access to resources), OIDC provides authenti-

cation, allowing clients to verify the identity of the end-user based on the authentication performed by an authorization server. OIDC also allows for single sign-on (SSO) capabilities and more secure handling of user identities via additional tokens such as:

- **ID Token**: A JWT that is used to authenticate the user and provide information about their identity. ID Token contains claims about the user information (e.g., sub, name, email).
- **UserInfo Endpoint**: An endpoint to fetch additional user attributes.

OIDC allows clients to verify the identity of the end-user based on the authentication performed by an authorization server, as well as to obtain basic profile information about the end-user. The **OIDC Scope** is a mechanism for defining what access privileges are being requested by the client application. When a client requests authorization from an end-user, it specifies one or more scopes. These scopes (typically simple strings) define the level of access the client application is requesting. Common OIDC scopes include:

- openid: Required scope for OIDC, which signifies that the application wants to use OIDC to verify the user's identity.
- profile: Requests access to the user's default profile claims, such as name, family name, given name, etc.
- email: Requests access to the user's email address.
- address: Requests access to the user's address information.
- phone: Requests access to the user's phone number.
- offline_access: Requests a refresh token to obtain access tokens without user interaction.

The **OIDC Context** refers to the environment and conditions under which authentication and authorization occur. This can include several factors, such as:

- **Authentication Context Class Reference (ACR)**: This specifies the requirements for the authentication method used by the identity provider. For example, it may require multi-factor authentication (MFA) or a specific level of assurance.
- **Claims Request**: The claims that are requested by the client can be specified in a more granular way, beyond just using scopes. This allows the client to ask for specific pieces of information about the user.
- **Authorization Request Parameters**: Additional parameters in the authorization request can provide context, such as prompt (to force user interaction), max_age (maximum allowable time since the user was last authenticated), and ui_locales (preferred user interface languages).

The **OIDC Roles** defines several roles involved in the authentication and authorization process:

- **End-User**: The human user who wants to access a resource and authenticate their identity.
- **Relying Party (RP)**: Also known as the client, this is the application that wants to authenticate the end-user and access resources on their behalf. It relies on the identity provider to verify the user's identity.
- **Identity Provider (IdP)**: Also known as the OpenID Provider (OP), this is the service that authenticates the end-user and issues identity tokens, access tokens, and sometimes refresh tokens to the relying party.
- **Resource Server**: This is the server hosting the protected resources that the relying party wants to access. It validates the access token issued by the identity provider.

To recap we can say that: **Scopes** define what access is being requested by the client application. **Context** encompasses the conditions and parameters surrounding the authentication and authorization process. **Roles** include the end-user, relying party, identity provider, and resource server, each playing a specific part in the OIDC framework.

6.7.1 OIDC Standard Endpoints

In OIDC, there are several standard endpoints exposed by the authorization server:

- **Authorization Endpoint**: Used by the client to obtain authorization from the resource owner via user agent redirection.
  - **URL Format**: https://example.com/oauth2/authorize
  - **Includes**: client_id, response_type, redirect_uri, scope, state, code_challenge, code_challenge_method (for PKCE).
- **Token Endpoint**: Used by the client to exchange an authorization code for an access token, refresh token, and optionally an ID token.
  - **URL Format**: https://example.com/oauth2/token
  - **Includes**: client_id, client_secret, grant_type, code, redirect_uri, code_verifier (for PKCE).
- **UserInfo Endpoint (OIDC)**: Used by the client to retrieve user profile information.
  - **URL Format**: https://example.com/oauth2/userinfo
  - **Includes**: access_token in the Authorization header.
- **Token Revocation Endpoint**: Used by the client to notify the authorization server that a previously obtained token is no longer needed.
  - **URL Format**: https://example.com/oauth2/revoke
  - **Includes**: token, token_type_hint.
- **Token Introspection Endpoint**: Used by the client or resource server to check the active state of an access token or refresh token.
  - **URL Format**: https://example.com/oauth2/introspect
  - **Includes**: token, token_type_hint.
- **Discovery Endpoint (OIDC)**: Provides metadata about the authorization server to simplify the configuration of clients.
  - **URL Format**: https://example.com/.well-known/openid-configuration
  - **Includes**: JSON document with URLs for all the other endpoints, supported scopes, response types, grant types, etc.

6.8 Possible Security Risk in OAuth Authorization Code Flow

In OAuth standard authorization code flow, a client exchanges an authorization code for an access token. However, without Proof Key for Code Exchange, an attacker intercepting the authorization code could potentially exchange it for a token, leading to security vulnerabilities. PKCE mitigates this risk by introducing a dynamic, client-generated secret (code verifier) and a hashed version (code challenge), ensuring that only the legitimate client that initiated the request can complete the exchange.

PKCE prevents authorization code interception attacks by requiring the client to prove possession of the original code verifier when exchanging the code for a token. During the authorization request, the client sends a code challenge (a hashed and encoded version of the code verifier) to the authorization server. When redeeming the authorization code, the client must provide the original code verifier, which the server validates against the previously sent challenge. If they do not match, the exchange fails. This mechanism ensures that even if an attacker steals the authorization code, they cannot complete the token exchange without the corresponding code verifier, thereby strengthening the security of OIDC flows.

Figure **??** shows some of the typical interactions that happens during OAuth 2.1 Authorization Code Flow with PKCE to better protect authorization codes in public clients, such as browser-based or mobile applications:

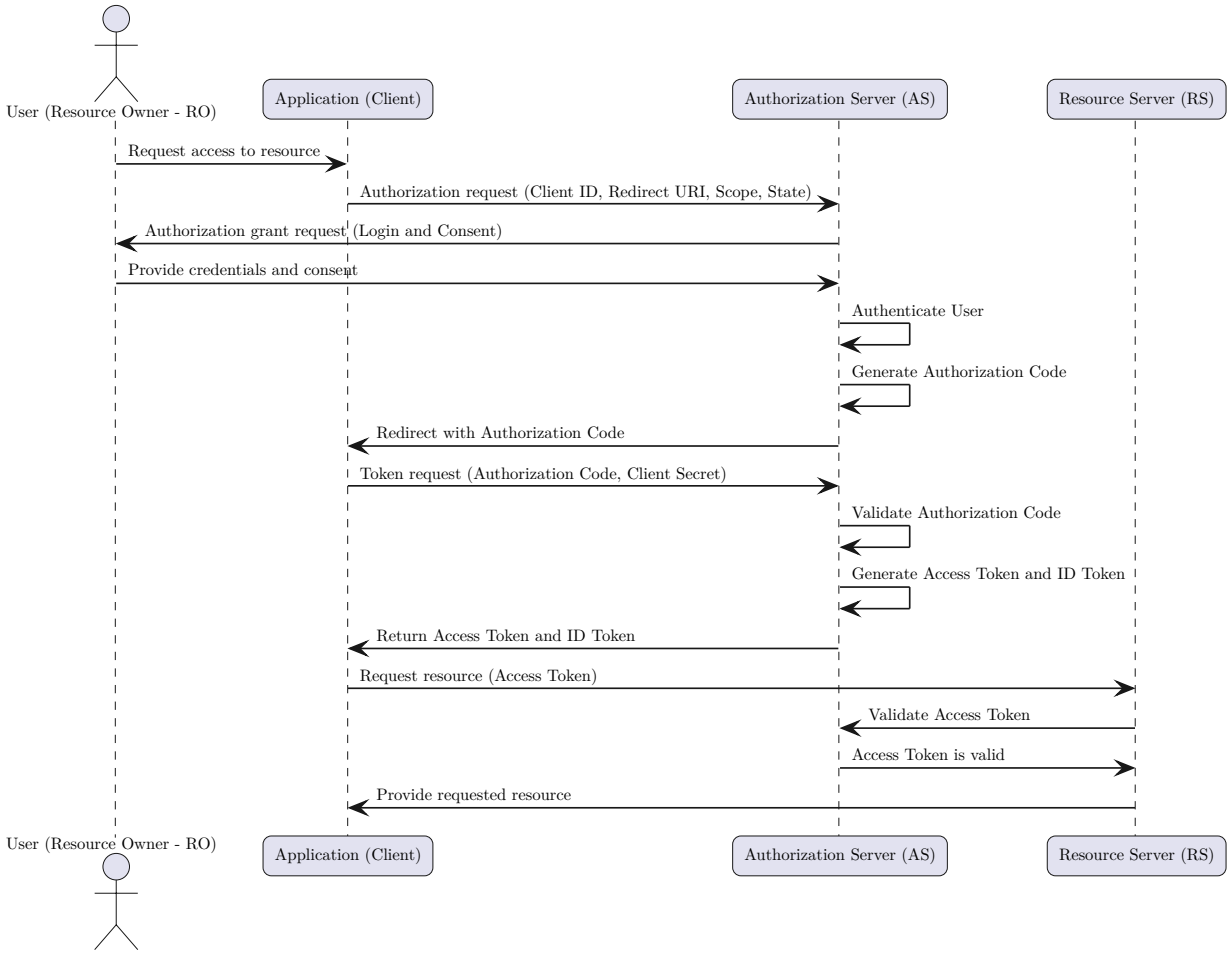

Figure 4: Typical Interactions in OAuth Authorization Code Flow

Please note:

1. **Client Initiation**: The client generates a unique code verifier and a corresponding code challenge. The code challenge is derived from the code verifier using a hashing algorithm (e.g., SHA-256).

2. **Authorization Request**: The client initiates an authorization request to the authorization server, including the code challenge and the method used to generate it.

3. **Authorization Response**: The authorization server authenticates the user and redirects them back to the client address with an authorization code.

4. **Token Request**: The client sends the authorization code along with the code verifier to the token endpoint of the authorization server.

5. **Token Response**: The authorization server validates the code verifier against the code challenge and, if valid, issues an access token to the client. This means possible intercepted authorization codes cannot be used without the code verifier, thus enhancing security.

6. **Accessing Resources**: After the client receives the access token, it can use this token to make authenticated requests to the resource server on behalf of the user of the application. The access token is typically included in the HTTP headers of the requests. The resource server then validates the access token with the authorization server to ensure its legitimacy. If the token is valid, the resource server processes the request and returns the requested resources to the client. If the access token expires,

the client can use a refresh token, if issued, to obtain a new access token without requiring the user to reauthenticate. Access token format is often set to JWT and its lifetime is typically short-lived to enhance security. The refresh token is used to obtain a new access token when the current access token expires and it is typically a simple opaque string, which is longer-lived and do require secure storage.

6.9 Why Authorization Code + PKCE?

Please keep in mind that with Mobile or Web-based Single Page Applications, most of the security comes from the API's server, so you should also secure your API server. From a security perspective, single-page applications alone cannot store secrets securely. The Authorization Code + PKCE flow introduces a variation of the basic OAuth 2.1 Authorization Code flow that replaces client secrets with a one-time code verifier and challenge to address this concern. This mechanism enhances security by preventing several common attack vectors that exploit weaknesses in token exchange processes.

- **Preventing Code Interception Attacks**: Without PKCE, an attacker intercepting an authorization code (e.g., via a malicious browser extension, compromised network, or improperly secured redirect URI) could exchange it for an access token. PKCE mitigates this by requiring the original code verifier during the token request, ensuring that only the legitimate client that initiated the request can complete the exchange.
- **Mitigating Authorization Code Injection**: Attackers who gain access to a victim's authorization code (e.g., through open redirect vulnerabilities or improperly validated redirect URIs) could attempt to inject it into their own token request. PKCE prevents this by enforcing proof of possession, ensuring that only the rightful client with the correct code verifier can successfully redeem the authorization code.
- **Protection Against Man-in-the-Middle (MitM) Attacks**: In scenarios where an attacker can intercept traffic between the client and authorization server, they might attempt to steal and reuse authorization codes. PKCE ensures that even if an attacker obtains the code, they cannot exchange it for a token without the corresponding code verifier.
- **Implicit Flow Issues**: The Implicit flow is no longer recommended due to security flaws, such as token leakage through URL fragments and susceptibility to redirect URI attacks, where attackers could gain unauthorized access to tokens. PKCE eliminates the need for exposing tokens in the URL and strengthens authorization security.
- **Cross-Site Request Forgery (CSRF) Protection**: Since PKCE binds the authorization request to the original client using the code verifier, it significantly reduces the risk of CSRF attacks, where an attacker tricks the user into authorizing an attacker's client instead of the legitimate one.

6.10 Typical OIDC/OAuth 2.1 with PKCE Sequence Diagram

A typical OIDC/OAuth PKCE flow is demonstrated in Figure 5. Please note that:

- **User Action**: A user attempts to log in to the single-page application.
- **SDK Generates Code Verifier**: The App ID SDK creates a code verifier for the authorization request, a plain text version of the code challenge. The client sends the code challenge and the challenge method used to encode the challenge with the authorization request.
- **Authorization Request**: The authentication flow is started by App ID in a new window.
- **User Authentication**: The user chooses an identity provider to authenticate with and completes the sign-in process.
- **Grant Code Received**: The App ID SDK on the application receives the grant code.
- **Token Request**: The SDK makes an XHR request to the App ID token endpoint along with the grant code and the code verifier to obtain access and identity tokens.

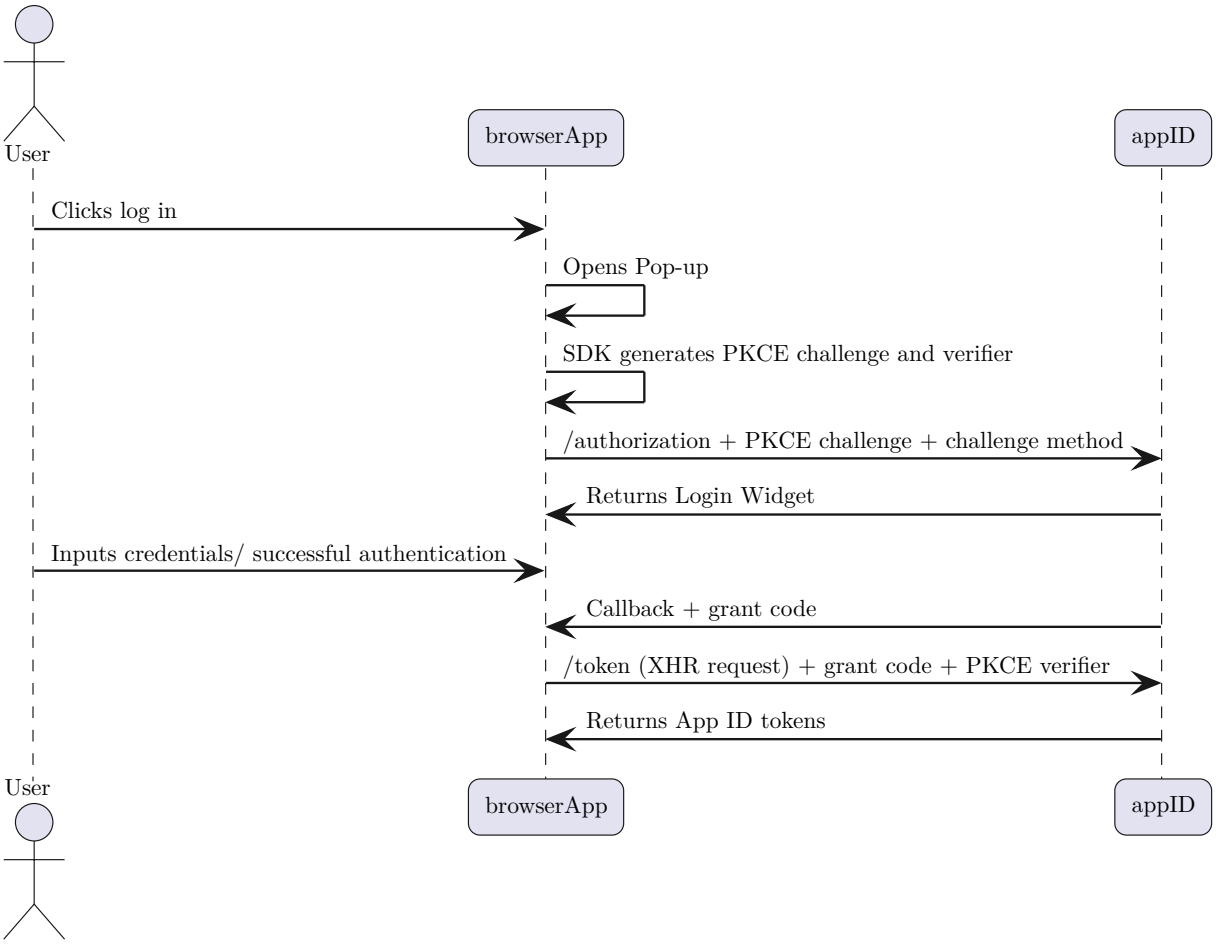

Figure 5: Sample SPA with PKCE

After the use clicks on the login button, a one-time code verifier and challenge is created for the OIDC Authorization Code with PKCE flow:

6.10.1 Step 1: Client Creates a Code Verifier

The client generates a random string as the code verifier.

code_verifier = ”dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk”

6.10.2 Step 2: Client Creates a Code Challenge

The client creates a code challenge from the code verifier using SHA-256.

code_challenge = BASE64URL-ENCODE(SHA256(ASCII(code_verifier)))
code_challenge = ”E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM”

6.10.3 Step 3: Client Initiates Authorization Request

The client sends the user to the authorization server with the following parameters:

GET /authorize?response_type=code
&client_id=client123
&redirect_uri=https://client.example.com/callback

&scope=openid
&code_challenge=E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM
&code_challenge_method=S256

6.10.4 Step 4: User Authenticates

The user authenticates and the authorization server redirects back to the client with an authorization code.

https://client.example.com/callback?code=AUTH_CODE

6.10.5 Step 5: Client Requests Access Token

The client exchanges the authorization code for an access token by sending the code verifier along with the code to the token endpoint.

POST /token
Host: authorization-server.com
Content-Type: application/x-www-form-urlencoded
grant_type=authorization_code
&code=AUTH_CODE
&redirect_uri=https://client.example.com/callback
&client_id=client123
&code_verifier=dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk

6.10.6 Step 6: Authorization Server Verifies and Responds

The authorization server verifies the code verifier against the code challenge and, if valid, issues an access token.


```
{
  "access_token": "ACCESS_TOKEN",
  "token_type": "Bearer",
  "expires_in": 3600,
  "id_token": "ID_TOKEN"
}
```


7 Prelab Activities

Before beginning the lab on **Advanced Cloud PKCE-based IAM with OIDC/OAuth 2.1**, students should familiarize themselves with key concepts related to Identity and Access Management (IAM), OAuth 2.1, OpenID Connect (OIDC), and security paradigms to better understand the hands-on implementation. Students should read and review the following sections from this lab manual:

7.1 Reading - Layered Security Model and Zero Trust

- Read about the layered security approach and its importance in modern security architectures.
- Study concepts such as Zero Trust, API Security, and Perimeter Protection.

7.2 Reading - Identity and Access Management (IAM)

- Understand the importance of IAM in securing modern applications.
- Learn about authentication, authorization, and access control principles.

7.3 Reading - OAuth 2.1 and OpenID Connect (OIDC)

- Study the OAuth 2.1 framework and how it enables secure authentication.
- Understand the OIDC layer built on OAuth 2.1 for identity verification.
- Read about different authentication flows, focusing on Authorization Code Flow with PKCE.

7.4 Reading - PKCE (Proof Key for Code Exchange)

- Understand how PKCE mitigates security threats in OAuth 2.1.
- Learn about the process of code verifier and code challenge creation.

7.5 Reading - Role-Based Access Control (RBAC)

- Study how RBAC enforces security by assigning user roles.
- Differentiate between Admin and Staff roles in access control policies.

7.6 Reading - JSON Web Tokens (JWT) and JSON Web Key Sets (JWKS)

- Understand JWT structure: header, payload, and signature.
- Learn about cryptographic signing mechanisms (HS256 vs RS256).
- Study how JWKS enables secure key distribution and validation.

7.7 Prelab Questions

To reinforce understanding, students should answer the following questions before starting the lab:

1. What are the key differences between authentication and authorization?
2. How does PKCE prevent authorization code interception attacks?
3. Describe the flow of OAuth 2.1 Authorization Code Flow with PKCE.
4. What role does JWT play in authentication and authorization?
5. How does Role-Based Access Control (RBAC) enhance security in IAM?
6. Explain the concept of Zero Trust security and its significance.

7.8 Prepare for Hands-on Lab and Setups

Before starting the lab, students should:

- Ensure they have access to an IBM Cloud account or an alternative OIDC-compliant identity provider.
- Install required development tools (Node.js, npm, Git).
- Set up their local development environment following the provided installation instructions.

8 IAM Complete Example via IBM Cloud AppID Identity Provider

Cloud companies provide software as a service identity solutions with easy to use client SDKs. These SaaS models, simplify the integration process by abstracting the complexities of OAuth and OIDC. Below we discuss a sample high-level Next.JS/React web application and how it uses IBM Cloud AppID SDK to achieve enhanced PKCE-based OIDC/OAuth compliant IAM with RBAC and MFA/OTP:

The YorkU Secure App is a custom Next.js application that we made for this lab. It implements enterprise-level authentication and authorization using IBM Cloud App ID and OpenID Connect (OIDC). We will dicuss the technical details about the project structure, installation process, and code implementation. You can also download the source code from:

8.1 YorkU Secure App Project Structure

The project follows a standard Next.js file structure with additional security-focused components:

- **src/pages/** - Next.js pages and API routes
  - _app.tsx - Application entry point
  - index.tsx - Home page
  - staff.tsx - Staff dashboard
  - admin.tsx - Admin dashboard
  - profile.tsx - User profile page
  - unauthorized.tsx - Access denied page
- **src/pages/api/** - Backend API routes
  - auth/[...nextauth].ts - Authentication handling
  - protected/ - Protected API endpoints
- **/components/** - Reusable React components
  - Layout.tsx - Main layout component
  - Navigation.tsx - Navigation bar
  - AuthCheck.tsx - Authentication wrapper
- **/lib/** - Utility functions and configurations
  - passport.ts - Passport.js configuration
  - auth.ts - Authentication utilities

8.2 Installation and Setup Steps and Prerequisites

1. Node.js (v18.0.0 or higher)
2. npm (v9.0.0 or higher)
3. Git

```
# We use nvm
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.2/install.sh | bash

# in lieu of restarting the shell
\. "$HOME/.nvm/nvm.sh"

# Download and install Node.js:
```



```
nvm install 23

# Verify the Node.js version:
node -v # Should print "v23.10.0".
nvm current # Should print "v23.10.0".

# Download and install pnpm:
corepack enable pnpm

# Verify pnpm version:
pnpm -v
```

8.3 Project Setup

```
# Clone the repository
git clone https://github.com/navidmo/nextjs-ibm-auth.git
cd nextjs-ibm-auth

# Install dependencies
npm install

# Create environment file
cp .env.example .env.local

# Start development server
npm run dev
```

8.4 Environment Configuration

Create a .env.local file with the following variables:

```
# IBM App ID Configuration
APP_ID_ISSUER=https://us-south.appid.cloud.ibm.com/oauth/v4/[TENANT_ID]
APP_ID_CLIENT_ID=[YOUR_CLIENT_ID]
APP_ID_CLIENT_SECRET=[YOUR_CLIENT_SECRET]

# NextAuth.js Configuration
NEXTAUTH_URL=http://localhost:3000
NEXTAUTH_SECRET=[GENERATED_SECRET]
```

8.5 Project Files

8.5.1 Authentication Configuration

```
/**
 * Passport.js is a popular authentication middleware for Node.js that provides a flexible way to integrate various
 * authentication mechanisms, including OpenID Connect (OIDC). In the context of OpenID Connect, Passport.js acts as
 * an Identity and Access Management (IAM) solution by leveraging authentication providers that follow the OIDC standard.
 * The given code configures Passport.js to use an **OpenID Connect strategy**, which enables user authentication via
 * an OIDC-compliant Identity Provider (IdP). The `OpenIDConnectStrategy` is initialized with key endpoints (issuer,
 * authorization URL, token URL, and user info URL), as well as OAuth 2.0 credentials (`clientID` and `clientSecret`).
```

```
 * Once a user is authenticated, Passport retrieves the user's profile from the IdP and invokes the `done` callback
 * to pass the user object for further processing. Additionally, `passport.serializeUser` and `passport.deserializeUser`
 * enable session management by converting the user object into a storable format and reconstructing it when needed.
 *
 * The concept of **strategy** in OIDC IAM refers to the abstraction that Passport.js uses to handle different
 * authentication methods. Each strategy encapsulates a particular authentication flow, allowing developers to plug in
 * different authentication providers with minimal changes to their application logic. In the context of OIDC, the
 * strategy defines how the authentication request is made, how tokens are exchanged, and how user information is
 * retrieved. The **OpenIDConnectStrategy** in this code follows the OIDC authorization code flow, where a user is
 * redirected to the IdP, grants authorization, and receives an ID token along with access tokens. This strategy-based
 * approach in Passport.js allows applications to be modular and extensible, supporting multiple authentication
 * mechanisms such as OAuth, SAML, or LDAP, depending on the requirements of the IAM system.
 */

import passport from 'passport';
import { Strategy as OpenIDConnectStrategy, VerifyCallback, Profile } from 'passport-openidconnect';

// Configure OpenID Connect strategy
passport.use(
  new OpenIDConnectStrategy(
    {
      issuer: process.env.APP_ID_ISSUER!,
      clientID: process.env.APP_ID_CLIENT_ID!,
      clientSecret: process.env.APP_ID_CLIENT_SECRET!,
      authorizationURL: `${process.env.APP_ID_ISSUER}/authorization`,
      tokenURL: `${process.env.APP_ID_ISSUER}/token`,
      userInfoURL: `${process.env.APP_ID_ISSUER}/userinfo`,
      callbackURL: '/api/auth/callback',
    },
    (
      issuer: string,
      profile: Profile,
      done: VerifyCallback
    ) => {
      return done(null, profile as any);
    }
  )
);

// Serialize and deserialize user
passport.serializeUser((user: any, done) => {
  done(null, user);
});

passport.deserializeUser((obj: any, done) => {
  done(null, obj);
});

export default passport;
```

8.5.2 Home Page

```
/**
 * This file implements the main landing page of the application, serving as the entry point for users.
```

```
 * It provides authentication status display, role-based navigation options, and core functionality access.
 * The implementation utilizes Next.js features and next-auth for authentication, ensuring secure access
 * control and session management throughout the application.
 *
 * The component includes conditional rendering based on user authentication status and roles, displaying
 * appropriate navigation options and content for administrators, staff members, and regular users. It
 * implements a clean and responsive user interface using Bootstrap styling, with proper loading states
 * and session handling to ensure a smooth user experience.
 */

import { signIn, signOut, useSession } from "next-auth/react";
import Layout from "../components/Layout";

export default function Home() {
  const { data: session, status } = useSession();

  const isAdmin = session?.user?.roles?.some(role => role.name === "admin");
  const isStaff = session?.user?.roles?.some(role => role.name === "staff");

  if (status === "loading") {
    return (
      <Layout>
        <div className="container mt-5">
          <div className="d-flex justify-content-center">
            <div className="spinner-border" role="status">
              <span className="visually-hidden">Loading...</span>
            </div>
          </div>
        </div>
      </Layout>
    );
  }

  return (
    <Layout title="YorkU Secure App Demo">
      <div className="container mt-4">
        <div className="row justify-content-center">
          <div className="col-md-8 col-lg-6">
            <div className="card">
              <div className="card-header bg-primary text-white">
                <h1 className="h4 mb-0">YorkU Secure App Demo</h1>
                <p className="small mb-0">Utilizing IBM Cloud AppID, IAM, RBAC, MFA, OTP, and
                ↪  OIDC/OAuth 2.1</p>
              </div>

              <div className="card-body">
                {session ? (
                  <>
                    <div className="alert alert-success mb-4">
                      <p className="mb-0">Signed in as {session.user.email}</p>
                    </div>

                    <div className="d-grid gap-3">
                      <a href="/profile" className="btn btn-outline-primary">
                        My Profile
                      </a>

                      <a href="/activity-log" className="btn btn-outline-info">
                        Activity Log
                      </a>
```

```
                <a href="/protected" className="btn btn-outline-primary">
                  View Protected Page
                </a>

                {isAdmin && (
                  <a href="/admin" className="btn btn-outline-danger">
                    Admin Dashboard
                  </a>
                )}

                {isStaff && (
                  <a href="/staff" className="btn btn-outline-success">
                    Staff Dashboard
                  </a>
                )}

                <a href="/userinfo" className="btn btn-outline-info">
                  View User Info
                </a>

                <a href="/oidc-details" className="btn btn-outline-secondary">
                  View OIDC Details
                </a>

                <button
                  onClick={() => signOut()}
                  className="btn btn-danger"
                >
                  Logout
                </button>
              </div>
            </>
          ) : (
            <div className="text-center">
              <p className="mb-4">You are not signed in.</p>
              <button
                onClick={() => signIn("ibm-appid")}
                className="btn btn-primary btn-lg"
              >
                Login with IBM App ID
              </button>
            </div>
          )}
        </div>
      </div>
    </div>
  </div>
</Layout>
);
}
```

8.5.3 Staff Dashboard

```
/**
 * This file implements the staff dashboard page of the application, providing a secure interface
 * exclusively for staff members. It utilizes Next.js features and next-auth for authentication,
 * implementing role-based access control to ensure only users with staff privileges can access
```

```
 * the dashboard and its features.
 *
 * The implementation includes client-side authentication checks, proper handling of loading states,
 * and unauthorized access attempts. It displays staff-specific information and tools, implementing
 * a clean and responsive interface using Bootstrap styling while maintaining security through
 * continuous session and role verification.
 */

import { useSession } from "next-auth/react";
import { useRouter } from "next/router";
import { useEffect } from "react";

export default function StaffPage() {
  const { data: session, status } = useSession();
  const router = useRouter();

  useEffect(() => {
    if (status === "loading") return;

    const isStaff = session?.user?.roles?.some(role => role.name === "staff");
    if (!session || !isStaff) {
      router.push("/unauthorized");
    }
  }, [session, status, router]);

  if (status === "loading") {
    return (
      <div className="container mt-5">
        <div className="d-flex justify-content-center">
          <div className="spinner-border" role="status">
            <span className="visually-hidden">Loading...</span>
          </div>
        </div>
      </div>
    );
  }

  if (!session?.user?.roles?.some(role => role.name === "staff")) {
    return null; // Router will redirect
  }

  return (
    <div className="container mt-4">
      <div className="row">
        <div className="col-12">
          <div className="card">
            <div className="card-header bg-primary text-white">
              <h1 className="h4 mb-0">Staff Dashboard</h1>
            </div>
            <div className="card-body">
              <h2 className="h5 mb-4">Welcome, {session.user.name}</h2>

              <div className="row">
                <div className="col-md-6 mb-4">
                  <div className="card h-100">
                    <div className="card-body">
                      <h3 className="h6 mb-3">Staff Information</h3>
                      <p className="mb-2"><strong>Email:</strong> {session.user.email}</p>
                      <p className="mb-2"><strong>Role:</strong> Staff</p>
                    </div>
```

```
              </div>
            </div>

            <div className="col-md-6 mb-4">
              <div className="card h-100">
                <div className="card-body">
                  <h3 className="h6 mb-3">Quick Actions</h3>
                  <div className="d-grid gap-2">
                    <a href="/" className="btn btn-outline-primary">
                      Return to Home
                    </a>
                  </div>
                </div>
              </div>
            </div>
          </div>
        </div>
      </div>
    </div>
  );
}
```

8.5.4 Profile Page

```
/**
 * This file implements the profile page of the application, providing a secure interface for users
 * to view and manage their account information. It utilizes Next.js features and next-auth for
 * authentication, implementing client-side access control to ensure only authorized users can view
 * their profile details.
 *
**/
import { useSession } from "next-auth/react";
import { useRouter } from "next/router";
import { useEffect } from "react";
import Layout from "../components/Layout";

export default function ProfilePage() {
  const { data: session, status } = useSession();
  const router = useRouter();

  useEffect(() => {
    if (status === "loading") return;
    if (!session) {
      router.push("/unauthorized");
    }
  }, [session, status, router]);

  if (status === "loading") {
    return (
      <Layout>
        <div className="container mt-5">
          <div className="d-flex justify-content-center">
            <div className="spinner-border" role="status">
              <span className="visually-hidden">Loading...</span>
            </div>
          </div>
        </div>
```

```
      </Layout>
    );
  }

  if (!session) {
    return null;
  }

  return (
    <Layout title="Profile - YorkU Secure App">
      <div className="container mt-4">
        <div className="row justify-content-center">
          <div className="col-md-8">
            <div className="card">
              <div className="card-header bg-primary text-white d-flex justify-content-between align-items-center">
                <h1 className="h4 mb-0">Profile Management</h1>
              </div>

              <div className="card-body">
                <div className="row">
                  <div className="col-md-6">
                    <div className="card mb-4">
                      <div className="card-body">
                        <h2 className="h5 mb-3">Personal Information</h2>
                        <dl className="row">
                          <dt className="col-sm-4">Name</dt>
                          <dd className="col-sm-8">{session.user.name}</dd>

                          <dt className="col-sm-4">Email</dt>
                          <dd className="col-sm-8">
                            {session.user.email}
                            {session.user.email_verified && (
                              <span className="badge bg-success ms-2">Verified</span>
                            )}
                          </dd>

                          <dt className="col-sm-4">User ID</dt>
                          <dd className="col-sm-8">{session.user.id}</dd>
                        </dl>
                      </div>
                    </div>
                  </div>

                  <div className="col-md-6">
                    <div className="card mb-4">
                      <div className="card-body">
                        <h2 className="h5 mb-3">Roles & Permissions</h2>
                        <div className="mb-3">
                          {session.user.roles?.length ? (
                            <div className="d-flex flex-wrap gap-2">
                              {session.user.roles.map((role) => (
                                <span key={role.id} className="badge bg-info">
                                  {role.name}
                                </span>
                              ))}
                            </div>
                          ) : (
                            <p className="text-muted">No special roles assigned</p>
                          )}
                        </div>
```

```
                </div>
              </div>
            </div>
          </div>

          <div className="card mb-4">
            <div className="card-body">
              <h2 className="h5 mb-3">Identity Providers</h2>
              <div className="table-responsive">
                <table className="table table-sm">
                  <thead>
                    <tr>
                      <th>Provider</th>
                      <th>ID</th>
                      <th>Status</th>
                    </tr>
                  </thead>
                  <tbody>
                    {session.user.identities?.map((identity) => (
                      <tr key={identity.id}>
                        <td>{identity.provider}</td>
                        <td>{identity.id}</td>
                        <td>
                          <span className="badge bg-success">Active</span>
                        </td>
                      </tr>
                    ))}
                  </tbody>
                </table>
              </div>
            </div>
          </div>

          <div className="d-flex justify-content-between">
            <a href="/" className="btn btn-outline-primary">
              Return to Home
            </a>
          </div>
        </div>
      </div>
    </div>
  </Layout>
  );
}
```

8.5.5 Unauthorized Page

```
/**
 * This file implements the unauthorized access page of the application, serving as a central
 * landing point for users who attempt to access restricted content without proper authentication
 * or authorization. It provides clear feedback to users about their access status and guides
 * them on how to proceed, whether they need to sign in or lack the necessary permissions.
 *
 * The implementation utilizes next-auth for session management and implements a responsive
 * interface using Bootstrap styling. It dynamically adjusts the displayed message based on
 * the user's authentication status, providing appropriate guidance for both authenticated
 * users lacking permissions and unauthenticated users attempting to access protected routes.
```



```
 */

import { useSession } from "next-auth/react";
import Head from "next/head";

export default function UnauthorizedPage() {
  const { data: session } = useSession();

  return (
    <>
      <Head>
        <title>Unauthorized Access - YorkU Secure App</title>
      </Head>

      <div className="container mt-5">
        <div className="row justify-content-center">
          <div className="col-md-6">
            <div className="card border-danger">
              <div className="card-header bg-danger text-white">
                <h1 className="h4 mb-0"> Unauthorized Access</h1>
              </div>
              <div className="card-body">
                <div className="text-center mb-4">
                  <p className="lead">
                    {session ?
                      "You don't have permission to access this page." :
                      "Please sign in to access this page."
                    }
                  </p>
                </div>

                <div className="d-grid gap-2">
                  <a href="/" className="btn btn-primary">
                    Return to Home
                  </a>
                </div>
              </div>
            </div>
          </div>
        </div>
      </div>
    </>
  );
}
```

8.5.6 NextAuth Configuration

```
/**
 * This file implements the authentication configuration for Next.js using NextAuth.js, specifically
 * integrating with IBM App ID for secure user authentication. It handles the authentication flow,
 * session management, and role-based access control for the application.
 *
 * The implementation includes custom interfaces for handling IBM App ID specific profile structures
 * and token information. It defines the IBMAppIDProfile interface which contains user information
 * including sub, name, email, verification status, and roles, as well as identity provider specific
 * information through the identities array.
 *
 * Token management is handled through the TokenInfo interface, which maintains both access and ID
 * tokens received from the authentication provider. These tokens are crucial for maintaining secure
```

```
 * sessions and making authorized API calls.
 *
 * The file implements a fetchUserRoles function that extracts role information from the access token.
 * This function decodes the JWT token to access the embedded role information, providing a secure way
 * to implement role-based access control throughout the application.
 *
 * Error handling is implemented throughout the authentication flow, with appropriate logging and
 * fallback mechanisms to ensure graceful degradation in case of authentication failures or invalid
 * token formats.
 *
 * The authentication configuration supports session management, allowing the application to maintain
 * user state across requests while ensuring security through proper token validation and renewal
 * processes.
 *
 * Integration with IBM App ID is configured to support various authentication flows including
 * username/password, social login, and enterprise SSO options, providing flexibility in how users
 * can authenticate with the application.
 *
 * The implementation includes proper typing through TypeScript interfaces, ensuring type safety
 * throughout the authentication flow and making the code more maintainable and less prone to
 * runtime errors.
 *
 * Security best practices are followed throughout, including proper token handling, secure session
 * management, and appropriate error handling to prevent security vulnerabilities.
 *
 * The file serves as the central authentication configuration for the application, working in
 * conjunction with other components like the Layout component and various admin pages to provide
 * a secure, role-based application architecture.
 */

import NextAuth from "next-auth";
import { decode } from "jsonwebtoken";

interface IBMAppIDProfile {
  sub?: string;
  name?: string;
  email?: string;
  email_verified?: boolean;
  given_name?: string;
  family_name?: string;
  roles?: Array<{ id: string; name: string }>;
  identities?: Array<{
    provider: string;
    id: string;
    idpUserInfo?: {
      displayName: string;
      active: boolean;
      emails: Array<{ value: string; primary: boolean }>;
      name: {
        givenName: string;
        familyName: string;
        formatted: string;
      };
      status: string;
      idpType: string;
    };
  }>;
}

interface TokenInfo {
```

```
  access_token?: string;
  id_token?: string;
}

async function fetchUserRoles(userId: string | undefined, accessToken: string | undefined): Promise<Array<{ id:
↪   string; name: string }>> {
  if (!userId || !accessToken) {
    console.error("Missing userId or accessToken");
    return [];
  }

  try {
    // Extract roles from the access token scope
    const [, payload] = accessToken.split('.');
    if (!payload) {
      console.error("Invalid access token format");
      return [];
    }

    try {
      const decodedToken = JSON.parse(Buffer.from(payload, 'base64').toString());
      console.log('Decoded access token:', JSON.stringify(decodedToken, null, 2));

      const scopes = decodedToken.scope;
      if (!scopes) {
        console.log('No scopes found in token');
        return [];
      }

      // Convert scopes string to array and map to roles
      const scopeArray = typeof scopes === 'string' ? scopes.split(' ') : [];
      const roles = [];

      // Check for admin scope
      if (scopeArray.includes('admin')) {
        roles.push({ id: 'admin', name: 'admin' });
      }

      // Check for staff scope
      if (scopeArray.includes('staff')) {
        roles.push({ id: 'staff', name: 'staff' });
      }

      // Check for other role-related scopes
      if (scopeArray.includes('appid_manage_roles')) {
        roles.push({ id: 'role_manager', name: 'role_manager' });
      }

      console.log('Extracted roles from scopes:', roles);
      return roles;
    } catch (e) {
      console.error('Error decoding token payload:', e);
      return [];
    }
  } catch (error) {
    console.error("Error processing roles:", error);
    return [];
  }
}

```

```
async function safelyFetchRoles(profile: IBMAppIDProfile, accessToken: string | undefined) {
  return await fetchUserRoles(profile.sub, accessToken);
}

async function profile(profile: IBMAppIDProfile, tokens: TokenInfo): Promise<any> {
  // Get roles from the access token
  const roles = await fetchUserRoles(profile.sub, tokens.access_token);
  console.log('Profile roles:', roles);

  return {
    id: profile.sub,
    name: profile.name || `${profile.given_name} ${profile.family_name}`,
    email: profile.email,
    email_verified: profile.email_verified,
    roles: roles,
    identities: profile.identities || [],
  };
}

export default NextAuth({
  debug: true,
  providers: [
    {
      id: "ibm-appid",
      name: "IBM App ID",
      type: "oauth",
      version: "2.0",
      clientId: process.env.APP_ID_CLIENT_ID,
      clientSecret: process.env.APP_ID_CLIENT_SECRET,
      issuer: process.env.APP_ID_ISSUER,
      wellKnown: `${process.env.APP_ID_ISSUER}/.well-known/openid-configuration`,
      authorization: {
        url: `${process.env.APP_ID_ISSUER}/authorization`,
        params: {
          scope: "openid email profile appid_authenticated appid_manage_roles admin staff",
          response_type: "code",
          code_challenge_method: "S256",
        },
      },
      token: `${process.env.APP_ID_ISSUER}/token`,
      userinfo: `${process.env.APP_ID_ISSUER}/userinfo`,
      checks: ["pkce"],
      profile,
    },
  ],
  callbacks: {
    async jwt({ token, account, profile, user }) {
      if (account && profile) {
        // Get roles directly from the access token
        const roles = await fetchUserRoles(profile.sub, account.access_token);
        console.log('JWT callback roles:', roles);

        token.user = {
          ...user,
          roles,
          id: profile.sub,
          email: profile.email ?? user.email ?? undefined,
          name: profile.name ?? user.name ?? undefined,
        };
        token.accessToken = account.access_token;
```

```
      }
      return token;
    },
    async session({ session, token }) {
      if (token.user) {
        session.user = token.user;
      }
      session.accessToken = token.accessToken;
      return session;
    },
  },
});
```

8.5.7 Layout Component

```
/**
 * This Layout component serves as the main structural wrapper for the application, providing consistent styling,
 * navigation, and authentication state management across all pages. It utilizes Next.js features like Head for
 * document metadata management and useSession for authentication state handling, while also implementing
 ↪  role-based
 * access control through user role checks (admin and staff).
 *
 * The component incorporates Bootstrap for styling and responsive design, featuring a header with our Lassonde
 * School of Engineering YorkU logo and a navigation bar. It accepts children components and an optional title
 * prop, making it flexible for different page requirements while maintaining a consistent user interface
 * throughout the application.
 */

import { useSession } from "next-auth/react";
import Head from "next/head";
import { useRouter } from "next/router";

interface LayoutProps {
  children: React.ReactNode;
  title?: string;
}

export default function Layout({ children, title = "YorkU Secure App" }: LayoutProps) {
  const { data: session } = useSession();
  const router = useRouter();

  const isAdmin = session?.user?.roles?.some(role => role.name === "admin");
  const isStaff = session?.user?.roles?.some(role => role.name === "staff");

  return (
    <>
      <Head>
        <title>{title}</title>
        <link
          href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css"
          rel="stylesheet"
        />
      </Head>

      <div className="container-fluid bg-white py-3">
        <div className="container text-center">
          <img
            src="/lassonde.png"
            alt="Lassonde School of Engineering"
```

```
            style={{
              width: "80%",
              height: "auto",
              maxWidth: "800px"
            }}
          />
        </div>
      </div>

      <nav className="navbar navbar-expand-lg navbar-dark bg-primary">
        <div className="container">
          <a className="navbar-brand" href="/">YorkU Secure App</a>

          {session && (
            <>
              <button
                className="navbar-toggler"
                type="button"
                data-bs-toggle="collapse"
                data-bs-target="#navbarNav"
              >
                <span className="navbar-toggler-icon"></span>
              </button>

              <div className="collapse navbar-collapse" id="navbarNav">
                <ul className="navbar-nav me-auto">
                  <li className="nav-item">
                    <a
                      className={`nav-link ${router.pathname === "/dashboard" ? "active" : ""}`}
                      href="/dashboard"
                    >
                      Dashboard
                    </a>
                  </li>

                  {isAdmin && (
                    <li className="nav-item dropdown">
                      <a
                        className="nav-link dropdown-toggle"
                        href="#"
                        role="button"
                        data-bs-toggle="dropdown"
                      >
                        Admin
                      </a>
                      <ul className="dropdown-menu">
                        <li>
                          <a className="dropdown-item" href="/admin">Dashboard</a>
                        </li>
                        <li>
                          <a className="dropdown-item" href="/admin/users">Users</a>
                        </li>
                        <li>
                          <a className="dropdown-item" href="/admin/roles">Roles</a>
                        </li>
                        <li>
                          <a className="dropdown-item" href="/admin/audit">Audit Log</a>
                        </li>
                      </ul>
                    </li>
```

```
104             )}
105
106             {isStaff && (
107               <li className="nav-item">
108                 <a
109                   className={`nav-link ${router.pathname === "/staff" ? "active" : ""}`}
110                   href="/staff"
111                 >
112                   Staff
113                 </a>
114               </li>
115             )}
116
117             <li className="nav-item">
118               <a
119                 className={`nav-link ${router.pathname === "/reports" ? "active" : ""}`}
120                 href="/reports"
121               >
122                 Reports
123               </a>
124             </li>
125
126             <li className="nav-item">
127               <a
128                 className={`nav-link ${router.pathname === "/activity-log" ? "active" : ""}`}
129                 href="/activity-log"
130               >
131                 Activity
132               </a>
133             </li>
134
135             <li className="nav-item">
136               <a
137                 className={`nav-link ${router.pathname === "/help" ? "active" : ""}`}
138                 href="/help"
139               >
140                 Help
141               </a>
142             </li>
143           </ul>
144
145           <ul className="navbar-nav">
146             <li className="nav-item dropdown">
147               <a
148                 className="nav-link dropdown-toggle"
149                 href="#"
150                 role="button"
151                 data-bs-toggle="dropdown"
152               >
153                 {session.user.name || session.user.email}
154               </a>
155               <ul className="dropdown-menu dropdown-menu-end">
156                 <li>
157                   <a className="dropdown-item" href="/profile">Profile</a>
158                 </li>
159                 <li>
160                   <a className="dropdown-item" href="/settings">Settings</a>
161                 </li>
162                 <li>
163                   <hr className="dropdown-divider" />
```

```
                  </li>
                  <li>
                    <a className="dropdown-item" href="/api/auth/signout">Logout</a>
                  </li>
                </ul>
              </li>
            </ul>
          </div>
        </>
      )}
    </div>
  </nav>

  <main className="py-4">
    {children}
  </main>

  <footer className="bg-light py-3 mt-auto">
    <div className="container text-center">
      <p className="text-muted mb-0">
        &copy; {new Date().getFullYear()} Navid Mohaghegh. All rights reserved.
      </p>
    </div>
  </footer>

  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"></script>
 </>
 );
}
```

8.6 Running the Application

```
npm run dev
# Application will be available at http://localhost:3000
```

8.7 Production Build

```
# Create production build
npm run build

# Start production server
npm start
```

8.8 Testing

```
# Run tests
npm test

# Run tests with coverage
npm run test:coverage
```

8.9 Additional Resources

- Next.js Documentation
- IBM App ID Documentation
- NextAuth.js Documentation

- TypeScript Documentation
- IBM Cloud App ID Client SDK for JavaScript
- IBM Cloud App ID Server SDK for Node.js
- IBM Cloud App ID Single Page Application Documentation
- OAuth 2.1 for Browser-Based Apps IETF Draft
- IETF RFC6749 - OAuth 2.0
- IETF RFC7636 - PKCE
- IETF DRAFT-OAuth-v2.1
- OpenID Connect Specification
- OIDC Guides

9 Lab Activities

As we discussed, the goal of this lab is to secure a web application (both frontend and backend) using OpenID Connect with OAuth 2.1, implementing Proof Key for Code Exchange and Role-Based Access Control. Students will leverage IBM Cloud AppID as the identity provider to authenticate users and enforce access policies.

9.1 Prerequisites

- Make sure that you completed pre-lab activities and have a general understanding of the OIDC with PKCE flow.
- You have a crisp understanding of the basics of IAM.
- Familiarity with the general OAuth 2.1 and OIDC authentication flows.
- General understanding of JWT and PKCE.
- Installed tools as mentioned in the pre-lab activities:
  - Node.js
  - npm or yarn
  - Git
  - Free/Basic IBM Cloud account (or equivalent OIDC-compliant identity providers such as AWS Cognito, or Microsoft Azure Entra ID and AD)

9.2 Lab Instructions

9.2.1 Step 1: Setting Up the Development Environment

1. Install Node.js and npm.
2. Clone the repository from:
   git clone https://github.com/navidmo/nextjs-ibm-auth.git
3. Navigate to the project directory:
   cd nextjs-ibm-auth

4. Install dependencies:
   npm install
5. Create an environment configuration file:
   cp .env.example .env.local

9.2.2 Step 2: Configuring IBM Cloud App ID

1. Sign in to IBM Cloud and navigate to the App ID service.
2. Create a new App ID instance and obtain:
   - Client ID
   - Client Secret
   - OAuth Server URL (Issuer)
3. Update the .env.local file with the credentials (we also provided a sample file in the git repository):

```
APP_ID_ISSUER=https://us-south.appid.cloud.ibm.com/oauth/v4/{TENANT_ID}
APP_ID_CLIENT_ID={YOUR_CLIENT_ID}
NEXTAUTH_URL=http://localhost:3000
NEXTAUTH_SECRET={GENERATED_SECRET}
```

9.2.3 Step 3: Implementing Authentication with OIDC and PKCE

1. Modify the frontend to use IBM App ID authentication.
2. Implement PKCE-based Authorization Code Flow.
3. Ensure login redirects users to the identity provider and retrieves JWT tokens.
4. Implement secure token storage to manage session authentication.

9.2.4 Step 4: Securing API Endpoints

1. Implement JWT authentication middleware for API routes.
2. Secure backend API endpoints using access tokens.
3. Validate JWTs against the IBM Cloud App ID public keys using JSON Web Key Sets (JWKS).
4. Implement error handling for expired or invalid tokens.

9.2.5 Step 5: Implementing Role-Based Access Control (RBAC)

1. Define user roles: **Admin**, **Staff**, **User**.
2. Assign role-based permissions in the backend.
3. Protect admin routes to allow access only to users with the admin role.
4. Modify frontend components to display UI elements based on user roles.

9.2.6 Step 6: Enhancing Security (Optional)

- Implement Multi-Factor Authentication (MFA) with IBM App ID.
- Enable One-Time Passwords (OTP) for additional security.
- Implement rate limiting to protect against brute-force attacks.
- Set up logging and monitoring for authentication attempts.

9.3 Lab Submission

Students should submit:

- A report detailing their implementation, including source codes, README files and screenshots.
- Source code of their modified web application.
- A demonstration video showcasing authentication and RBAC enforcement and explaining their codes. Feel free to use OBS Studio or other tools to record your screen.

9.4 Grading Criteria

Each part below has equal weight:

Criteria	**Description**
OIDC/OAuth 2.1 Authentication	Implemented authentication with PKCE and IBM Cloud App ID.
JWT-based API Security	Backend API routes are secured with JWT validation.
RBAC Implementation	User roles and access control mechanisms are correctly enforced.
Frontend Security	UI restricts access to users based on roles.
Security Enhancements (Bonus)	Additional security features such as MFA, OTP, or rate limiting.

Table 1: Grading Criteria for Lab

We hope that after finishing this lab, students will gain hands-on experience in securing applications with OpenID Connect, OAuth 2.1, PKCE, and RBAC. This knowledge is critical for building secure, scalable, and robust authentication systems in modern applications and cloud-based systems.