OAuth 2.0

Piya Lumyong

Agenda

- **OAuth2 Authorization Framework**
- **Spring Security OAuth2**
- **Single Sign-On**
- **JSON Web Tokens JWT**
- **API Gateway**
- **Google OAuth2 provider**

Scenario

AwesomeApp wants Peters profile data from facebook for Peters profile in AwesomeApp

Introduction to OAuth2

OAuth2 is a protocol enabling
a Client application to act on behalf of a User

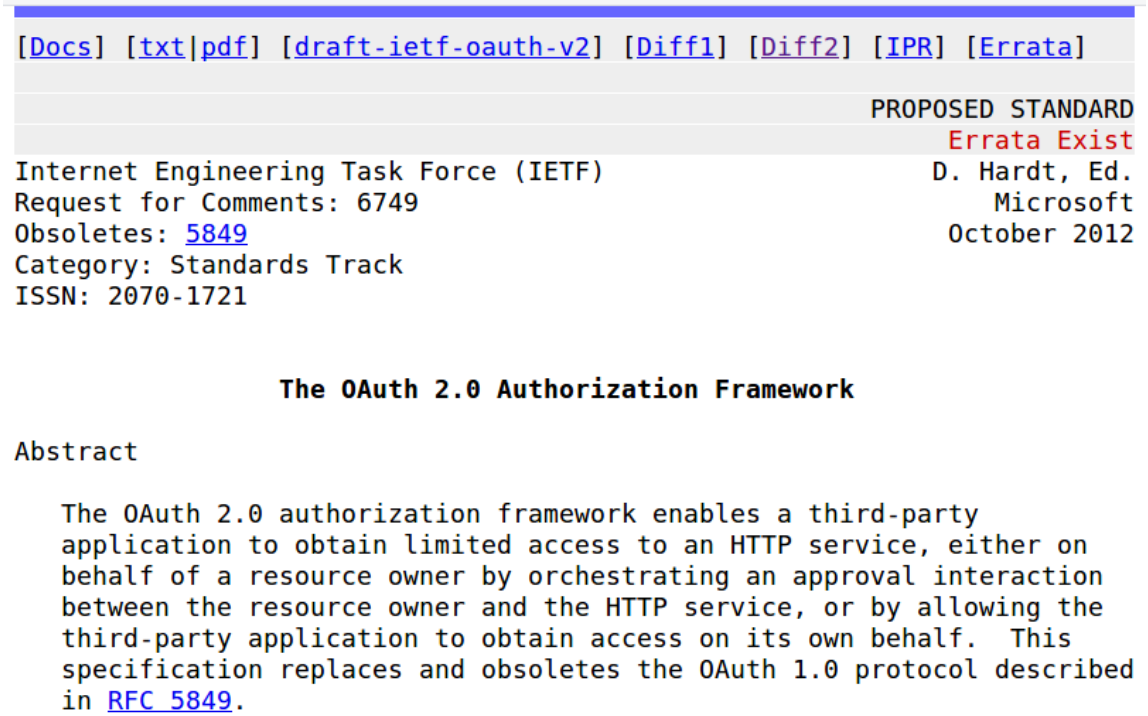
OAuth2 is a protocol enabling a Client application, often a web application, to act on behalf of a User, but with the User's permission. The actions a Client is allowed to perform are carried out on a Resource Server (another web application or web service), and the User approves the actions by telling an Authorization Server that he trusts the Client to do what it is asking. Clients can also act as themselves (not on behalf of a User) if they are permitted to do so by the Authorization Server.

Common examples of Authorization Servers

- Facebook and Google, both of which also provide Resource Servers (the Graph API in the case of Facebook and the Google APIs in the case of Google).

OAuth 2.0 Open Standard

- **https://tools.ietf.org/html/rfc6749**



[Docs] [txt|pdf] [draft-ietf-oauth-v2] [Diff1] [Diff2] [IPR] [Errata]

PROPOSED STANDARD
Errata Exist

Internet Engineering Task Force (IETF) — D. Hardt, Ed.
Request for Comments: 6749 — Microsoft
Obsoletes: 5849 — October 2012
Category: Standards Track
ISSN: 2070-1721

The OAuth 2.0 Authorization Framework

Abstract

The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP 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. This specification replaces and obsoletes the OAuth 1.0 protocol described in RFC 5849.

What is OAuth2?

Delegate Authorization

- A protocol for conveying authorization decisions (via token)
- Standard means of obtaining a token (aka the 4 OAuth2 grant types)
- Users and Clients are separate entities
  - *“I am authorizing this app to perform these actions on my behalf”*

What is OAuth2 Not?

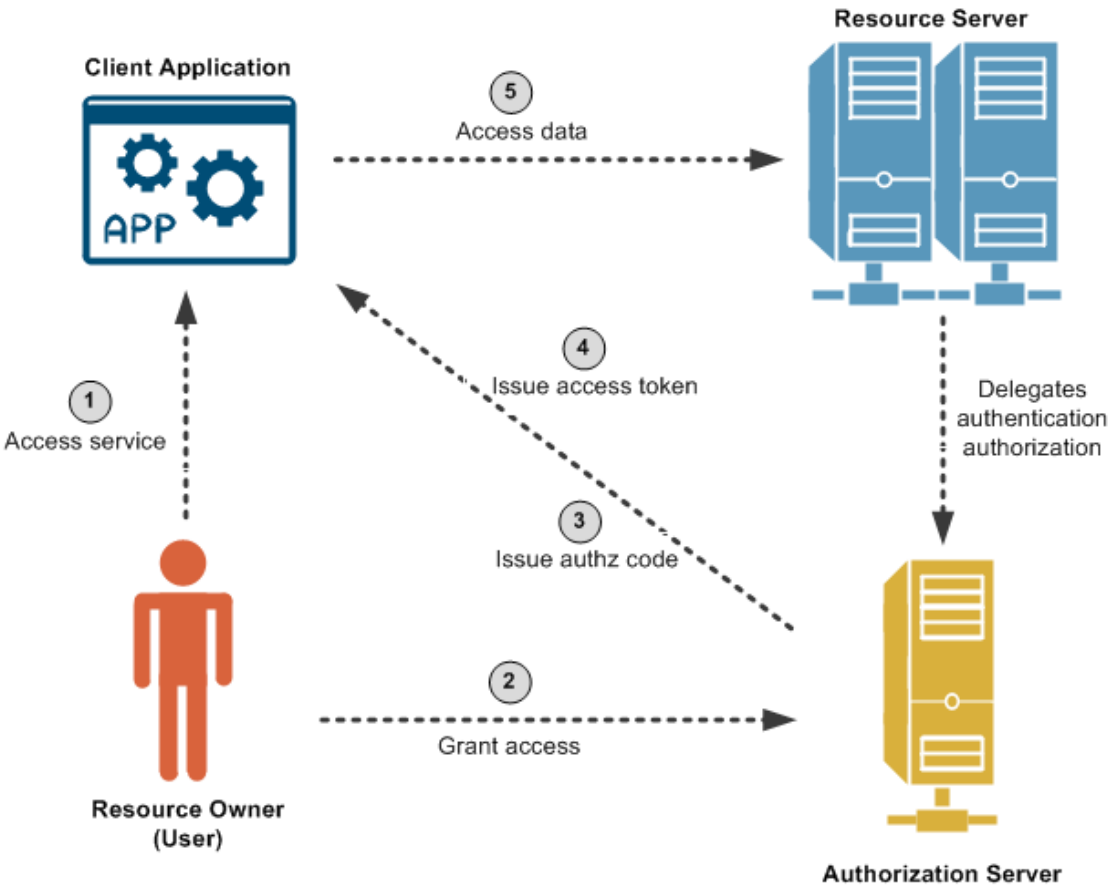
OAuth2 is not Authentication

- The user must be authenticated to obtain a token
- How user is authenticated is outside of the spec
- How the token is validated is outside the spec
- What the token contains is outside the spec
- Read more: http://oauth.net/articles/authentication/

OAuth 2.0 Roles

- **Resource Owner**
  - *User*
- **Resource Server**
  - *API*
- **Client Application**
  - *3rd party application*
- **Authorization Server**
  - *Auth API (may be in scope of Resource Server)*

OAuth 2.0 Roles

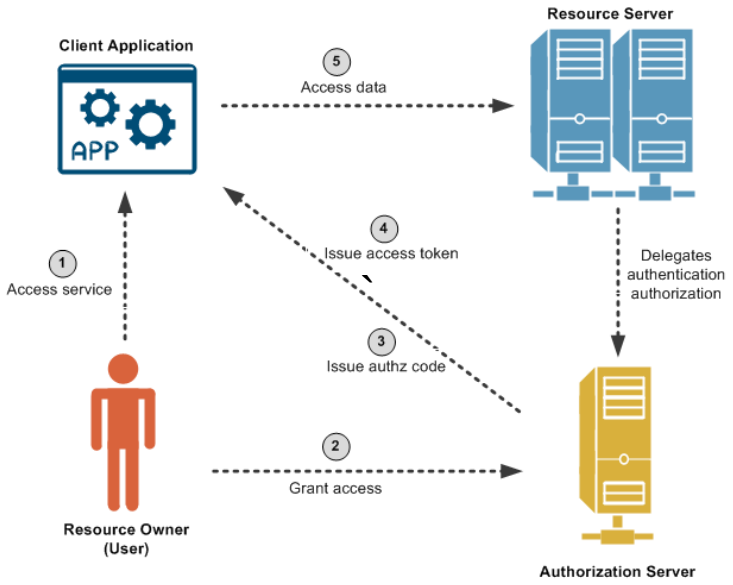


Oracle: Introduction to API Gateway OAuth 2.0 client

OAuth 2.0 Roles

Role of Client Application

- Register with Authorization Server (get a client_id and maybe a client client_secret)
- Do not collect user credentials
- Obtain a token from Authorization Server
- Use it to access Resource Server

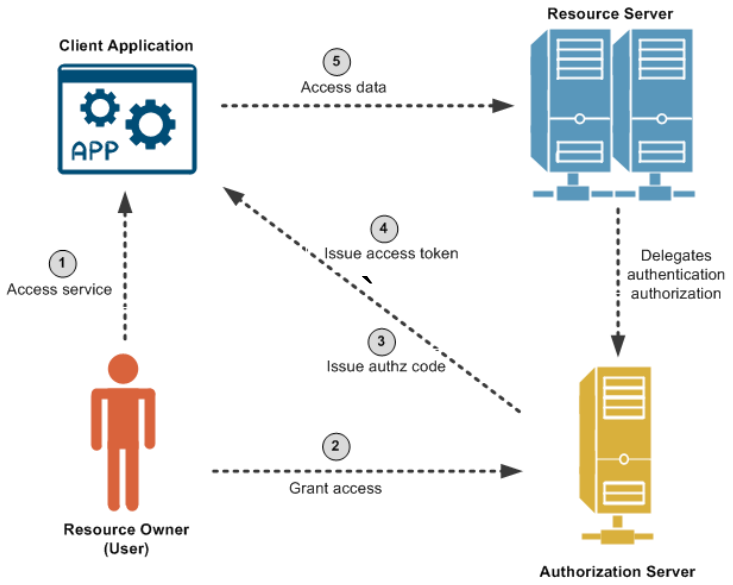


OAuth 2.0 Roles

Role of Resource Server

- Extract token from request and decode it
- Make access control decision
  - Scope
  - Audience
  - User account information (id, roles etc.)
  - Client information (id, roles etc.)
- Send 403 (FORBIDDEN)

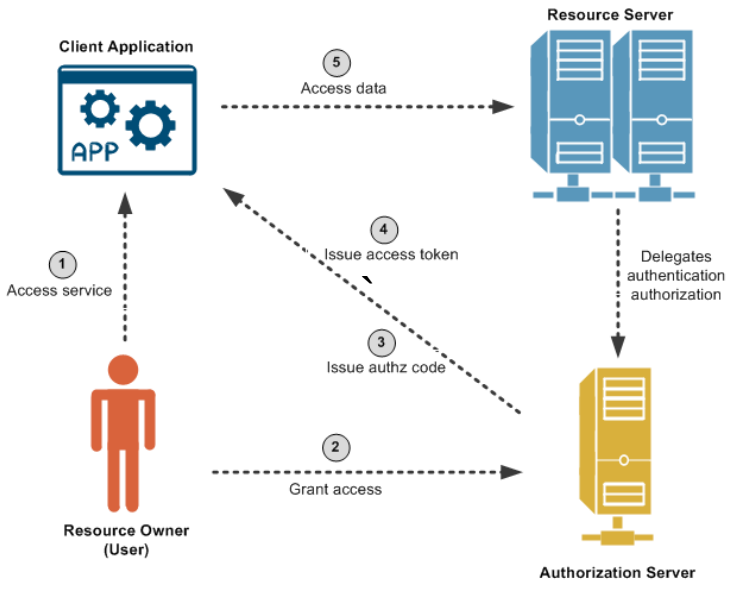
  if token not sufficient



OAuth 2.0 Roles

Role of the Authorization Server

- Compute token content and grant tokens
- Interface for users to confirm that they authorize the Client to act on their behalf (optional)
- Authenticate users (optional)
- Authenticate clients

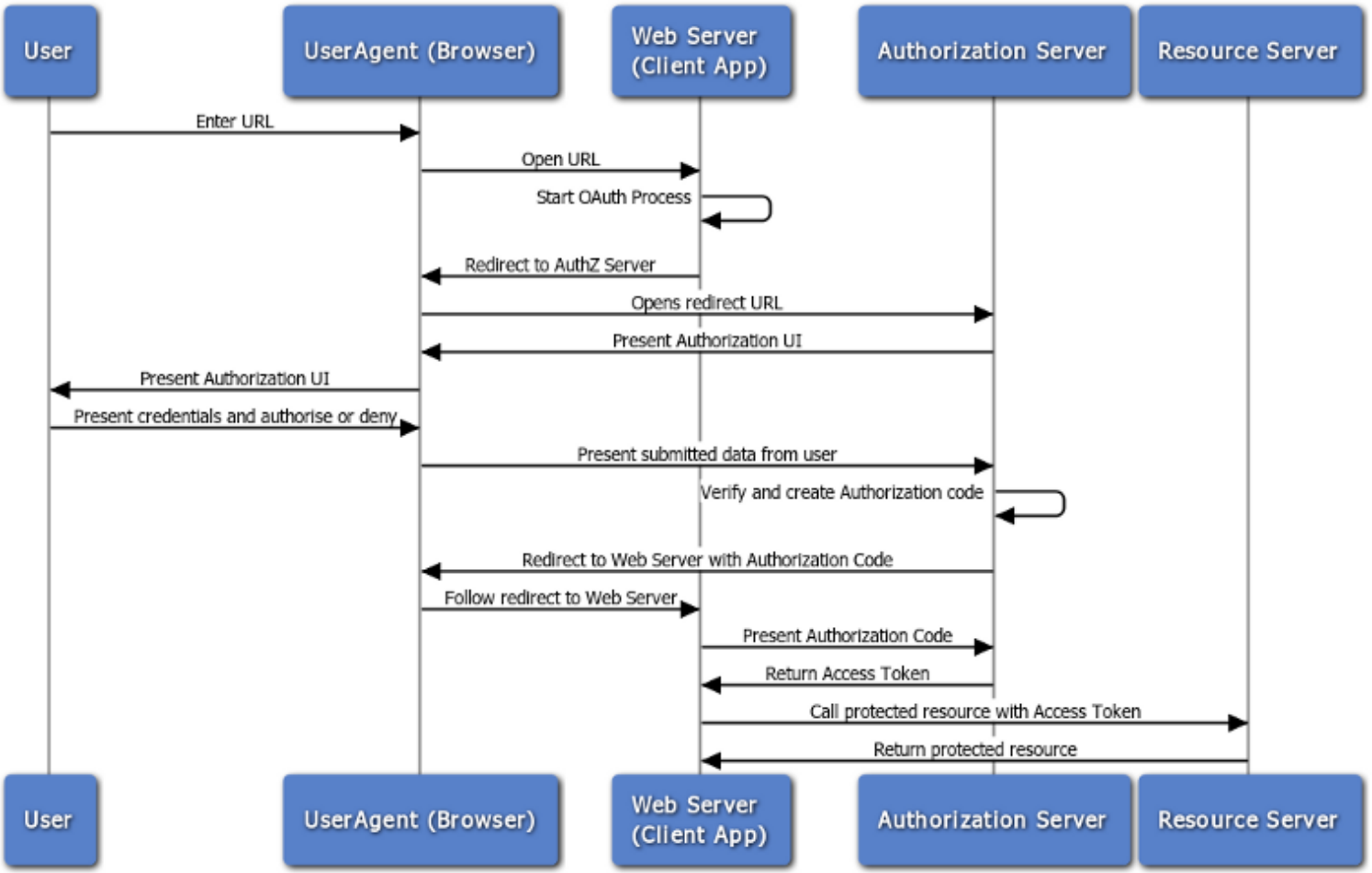


OAuth 2.0 Grant Flows

- **Authorization code grant flow**
  - Web-server apps
- **Implicit grant flow**
  - Browser-based apps
  - Mobile apps
- **Resource owner password grant flow**
  - Username/password access
- **Client credentials grant flow**
  - Application access

OAuth 2.0 Grant Flows

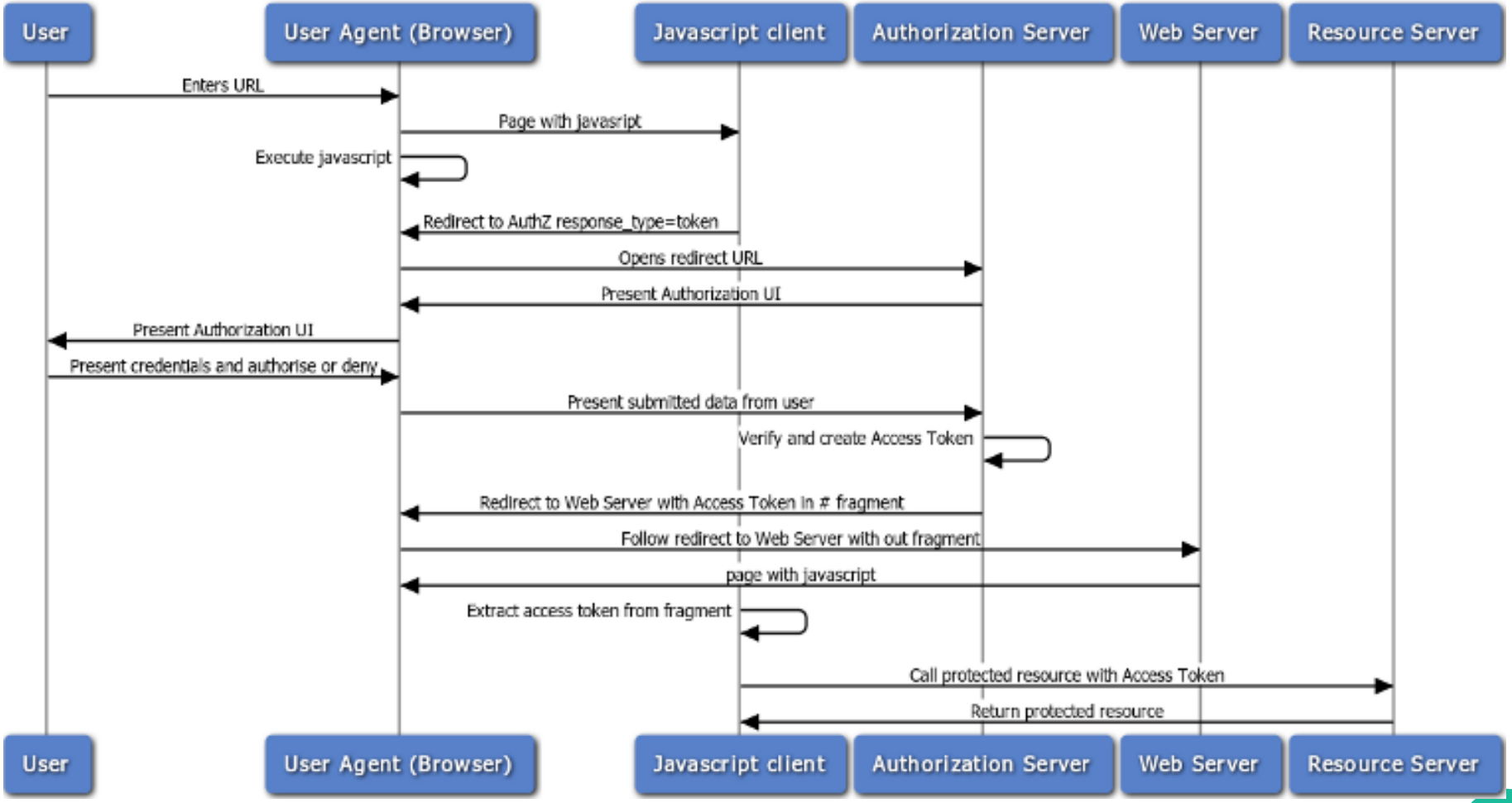
- **Authorization code grant flow**



Oracle: API Gateway OAuth 2.0 Authentication Flows

OAuth 2.0 Grant Flows

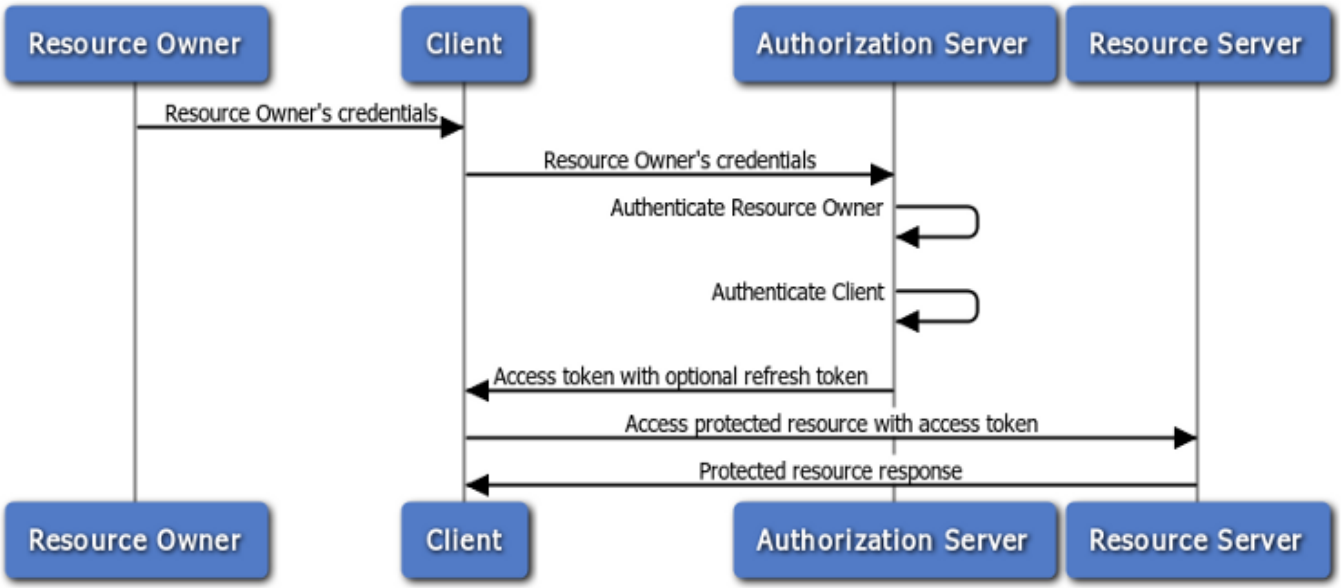
- **Implicit grant flow**



Oracle: API Gateway OAuth 2.0 Authentication Flows

OAuth 2.0 Grant Flows

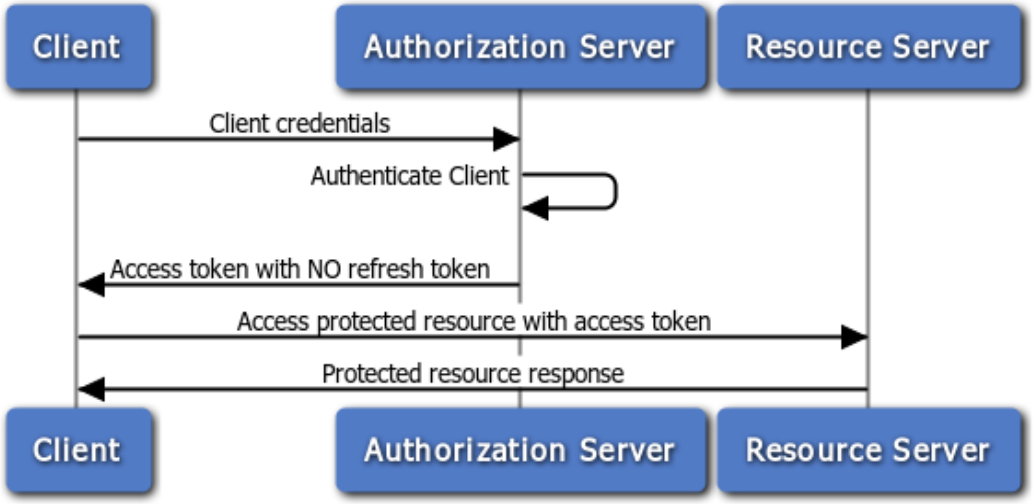
- **Resource owner password grant flow**



Oracle: API Gateway OAuth 2.0 Authentication Flows

OAuth 2.0 Grant Flows

- **Client credentials grant flow**



Oracle: API Gateway OAuth 2.0 Authentication Flows

Native Application

- **Authorization code flow with PKCE**

[Docs] [txt|pdf|xml|html] [Tracker] [WG] [Email] [Diff1] [Diff2] [Nits]

Versions: (draft-wdenniss-oauth-native-apps)
00 01 02 03 04 05 06 07 08 09 10 11
12

OAuth Working Group — W. Denniss
Internet-Draft — Google
Updates: 6749 (if approved) — J. Bradley
Intended status: Best Current Practice — Ping Identity
Expires: December 11, 2017 — June 9, 2017

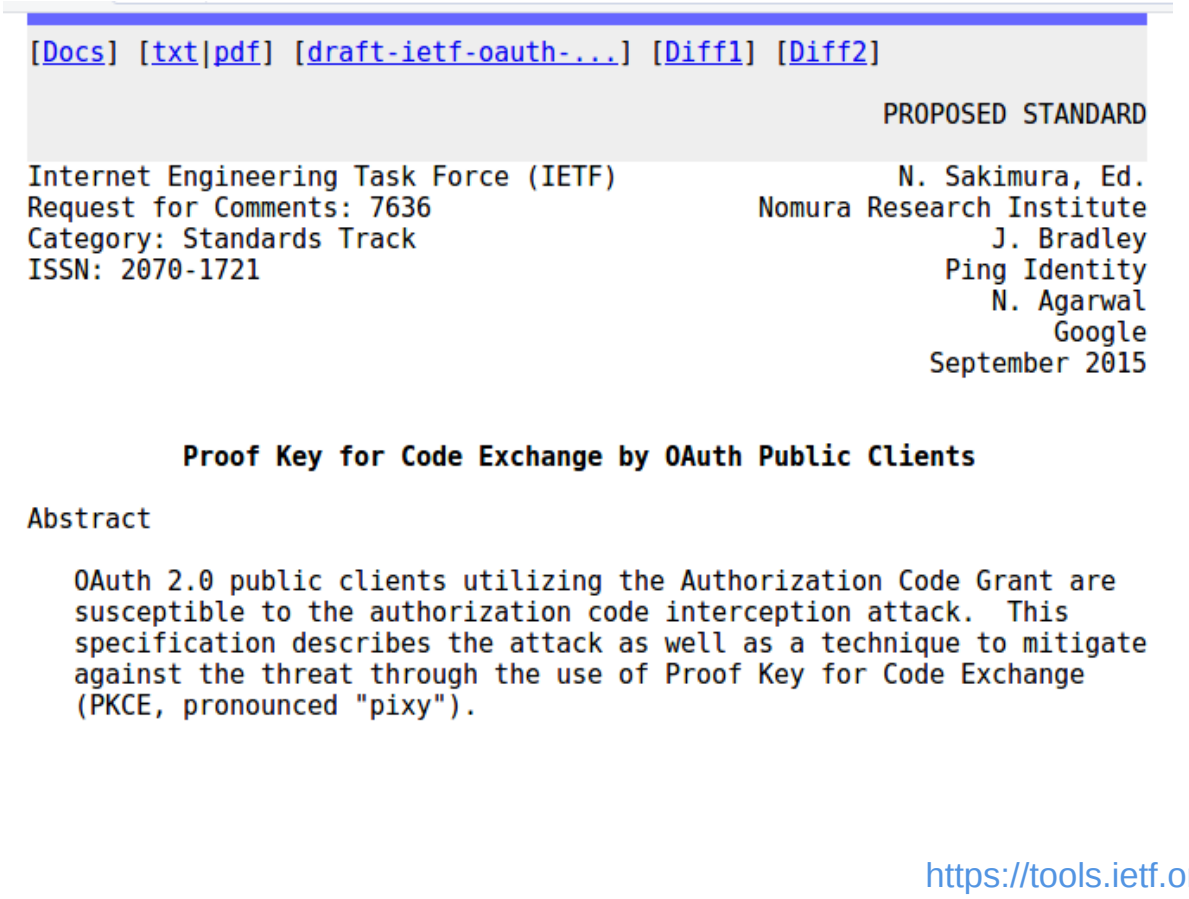
OAuth 2.0 for Native Apps
draft-ietf-oauth-native-apps-12

Abstract

OAuth 2.0 authorization requests from native apps should only be made through external user-agents, primarily the user's browser. This specification details the security and usability reasons why this is the case, and how native apps and authorization servers can implement this best practice.

https://tools.ietf.org/html/draft-ietf-oauth-native-apps-12

Proof Key for Code Exchange



[Docs] [txt|pdf] [draft-ietf-oauth-...] [Diff1] [Diff2]

PROPOSED STANDARD

Internet Engineering Task Force (IETF)
Request for Comments: 7636
Category: Standards Track
ISSN: 2070-1721

N. Sakimura, Ed.
Nomura Research Institute
J. Bradley
Ping Identity
N. Agarwal
Google
September 2015

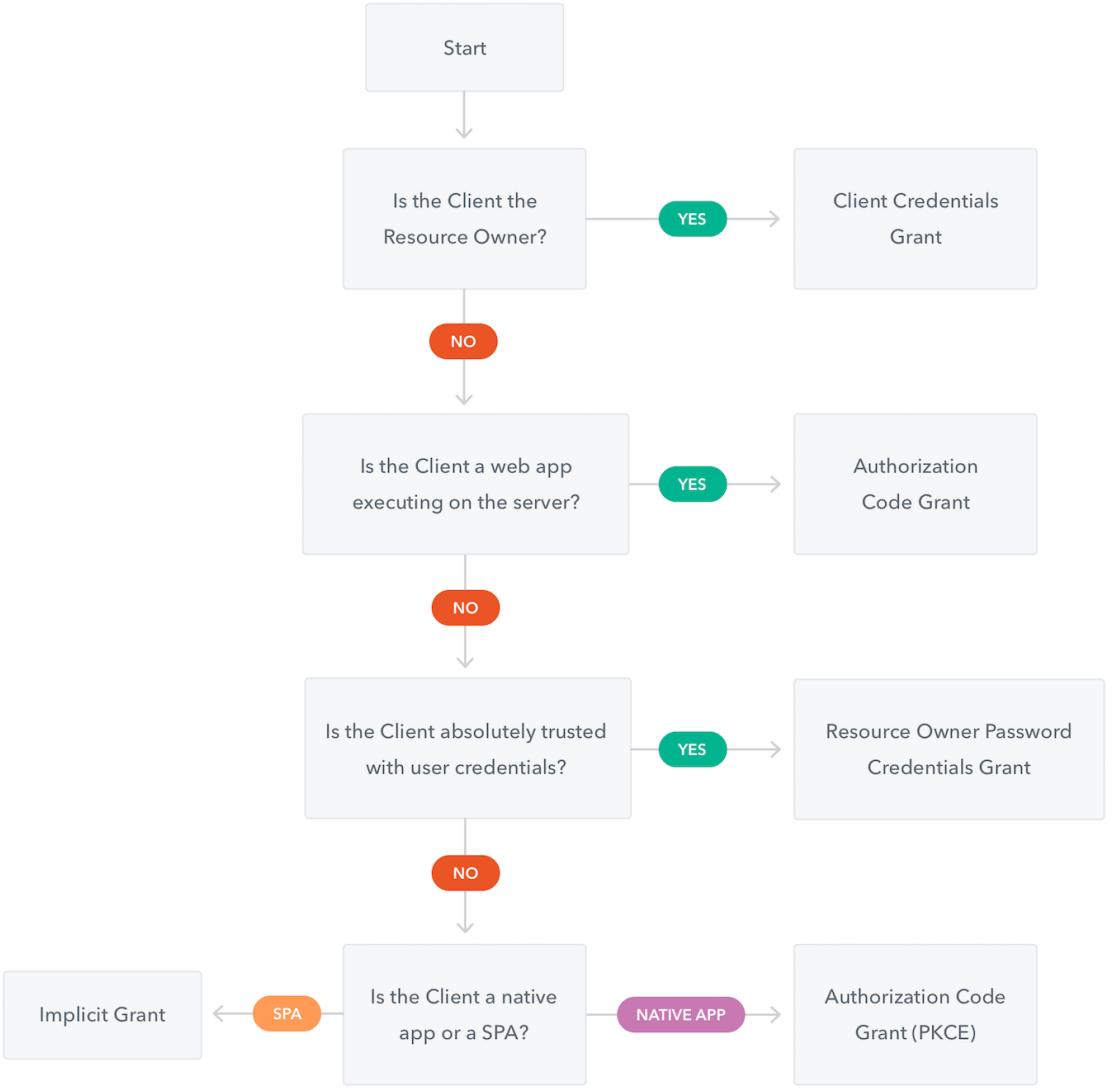
Proof Key for Code Exchange by OAuth Public Clients

Abstract

OAuth 2.0 public clients utilizing the Authorization Code Grant are susceptible to the authorization code interception attack. This specification describes the attack as well as a technique to mitigate against the threat through the use of Proof Key for Code Exchange (PKCE, pronounced "pixy").

https://tools.ietf.org/html/rfc7636

Which OAuth 2.0 flow should I use?



Which OAuth 2.0 flow should I use?

Lab: Basic OAuth

Grant Type: Authorization Code

- **Request authorization code /oauth/authorize**
  - Parameter
    - **response_type required “code”**
    - **client_id required**
    - **redirect_uri** optional
    - **scope** optional
    - **state** optional

```
curl -v -u tae:secret \
      http://localhost:8080/uaa/oauth/authorize \
      -d "client_id=my-client&redirect_uri=http://example.com&response_type=code"
```

Grant Type: Authorization Code

- **Request access token /oauth/token**
  - Parameter
    - **grant_type required “authorization_code”**
    - **code required**
    - **redirect_uri required**
    - **client_id** optional
    - **client_secret** optional

```
curl -v -u my-client:my-client-pass \
        http://localhost:8080/uaa/oauth/token \
        -d "grant_type=authorization_code&redirect_uri=http://example.com&code=ilYvKC"
```

Grant Type: Authorization Code

- **Token**


```
{
    "access_token": "a838f3f6-c248-41ce-9d7a-b40ce99b3ca4",
    "expires_in": 42710,
    "refresh_token": "b6177be8-4d74-4716-9525-eccfceff43cb",
    "scope": "openid",
    "token_type": "bearer"
}
```


Grant Type: Authorization Code

- **Request access resource**
  - Header
    - **authorization: bearer access_token**

```
curl -v -H "authorization: bearer 28bcab1b-6ab5-458c-b940-025f8c5b504c" \
        http://localhost:8888/me
```

Grant Type: Implicit

- **Request access token /oauth/authorize**
  - Parameter
    - **response_type required “token”**
    - **client_id required**
    - **redirect_uri** optional
    - **scope** optional
    - **state** optional

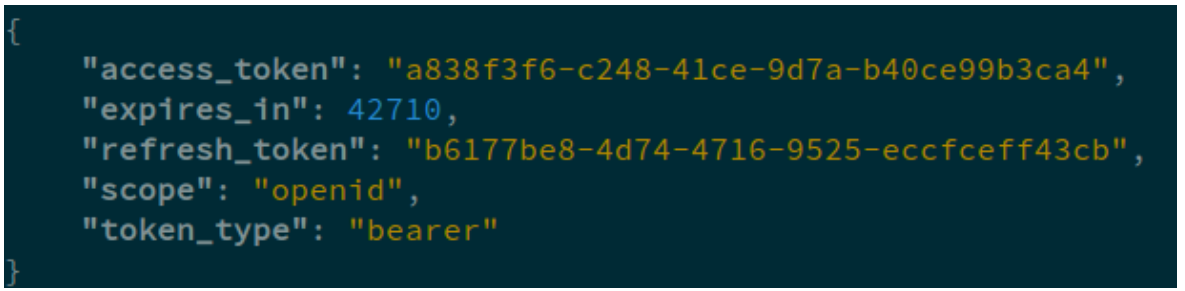
```
curl -v -u tae:secret http://localhost:8080/uaa/oauth/authorize \
       -d "response_type=token&client_id=my-client&redirect_uri=http://example.com"
```

```
http://example.com#access_token=30ee8c5f-e8d0-4479-b52e-9b19793f4c73&token_type=bearer&expires_in=43195&scope=openid
```

Grant Type: Password Credentials

- **Request access token /oauth/token**
  - Parameter
    - **grant_type required “password”**
    - **username required**
    - **password required**
    - **client_id optional**
    - **client_secret optional**
    - **scope optional**

```
curl -v -u my-client:my-client-pass http://localhost:8080/uaa/oauth/token \
     -d "grant_type=password&username=tae&password=secret"
```



Grant Type: Client Credentials

- **Request access token /oauth/token**
  - Parameter
    - **grant_type required “client_credentials”**
    - **client_id optional**
    - **client_secret optional**

```
curl -v -u my-client:my-client-pass http://localhost:8080/uaa/oauth/token \
     -d "grant_type=client_credentials"
```


```
{
    "access_token": "28517370-e965-40f9-a8b0-336bc019e7a6",
    "expires_in": 42201,
    "scope": "openid",
    "token_type": "bearer"
}
```


Spring Security OAuth2

Feature

- **SSO with OAuth2 and OpenID Connect servers**
  - With a single annotation (and some config)
- **Secure Resource Servers with tokens**
  - With a single annotation (and some config)
- **Relay tokens between SSO enabled webapps and resource servers**
  - With an autoconfigured OAuth2RestTemplate

Lab: Basic OAuth

Lab: Basic OAuth (form)

Lab: Basic OAuth (sso)

What tokens to use?

- **AtomicLong**
- **Random numbers**
- **Hash**
- **UUID**
- **Any?**

JSON Web Tokens(JWT)

[Docs] [txt|pdf] [draft-ietf-oauth-...] [Diff1] [Diff2] [IPR]

Updated by: 7797 PROPOSED STANDARD

Internet Engineering Task Force (IETF)
Request for Comments: 7519
Category: Standards Track
ISSN: 2070-1721

M. Jones
Microsoft
J. Bradley
Ping Identity
N. Sakimura
NRI
May 2015

JSON Web Token (JWT)

Abstract

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 or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted.

https://tools.ietf.org/html/rfc7519

JSON Web Tokens(JWT)

What is JWT

- JSON Web Token (RFC7519), standardized May 2015
- Base64 encoded form is easy to transmit in headers
- Standardized generation and verification of signatures
- Can encapsulate any claim (scopes, identity)
- Can expire
- Enable scalable, stateless authentication and authorization
  - Client can verify tokens themselves
  - With the treadeoff of losing token revocation

JWT Benefits

- **Standard approach**
- **Self-contained - no need for token/session storage**
- **Passed with each request to the server**
- **Plays nice with OAuth 2.0**

JWT Token Structure

- **Header**
- **Payload**
- **Signature**

```
Base64(Header) . Base64(Payload) . Base64(Signature)
```

JWT Example

JWT.IO

ALGORITHM HS256

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

Decoded EDIT THE PAYLOAD AND SECRET (ONLY HS256 SUPPORTED)

HEADER: ALGORITHM & TOKEN TYPE

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

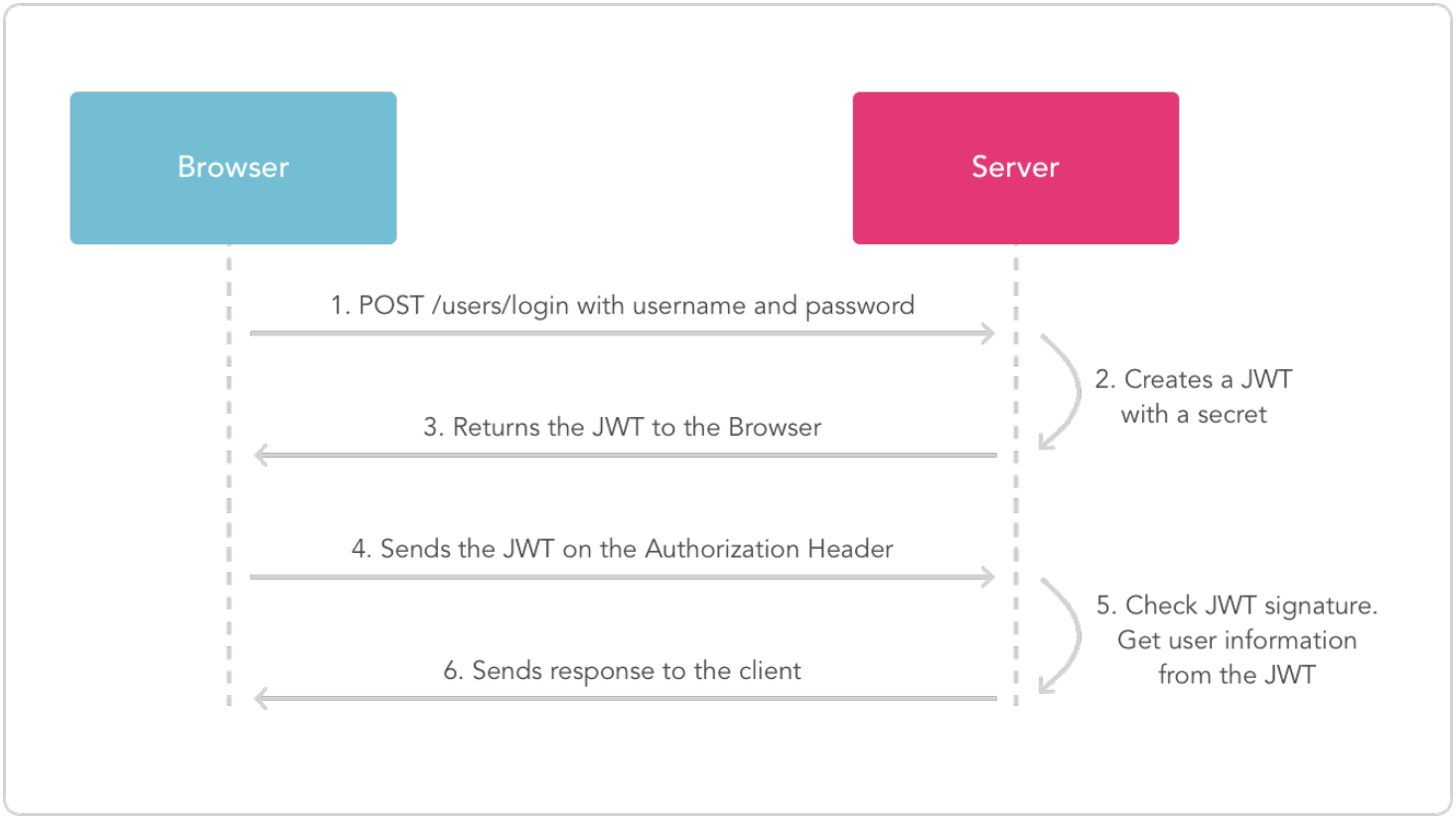
PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
) secret base64 encoded
```

JWT Simple Flow



Lab: Basic OAuth (jwt)

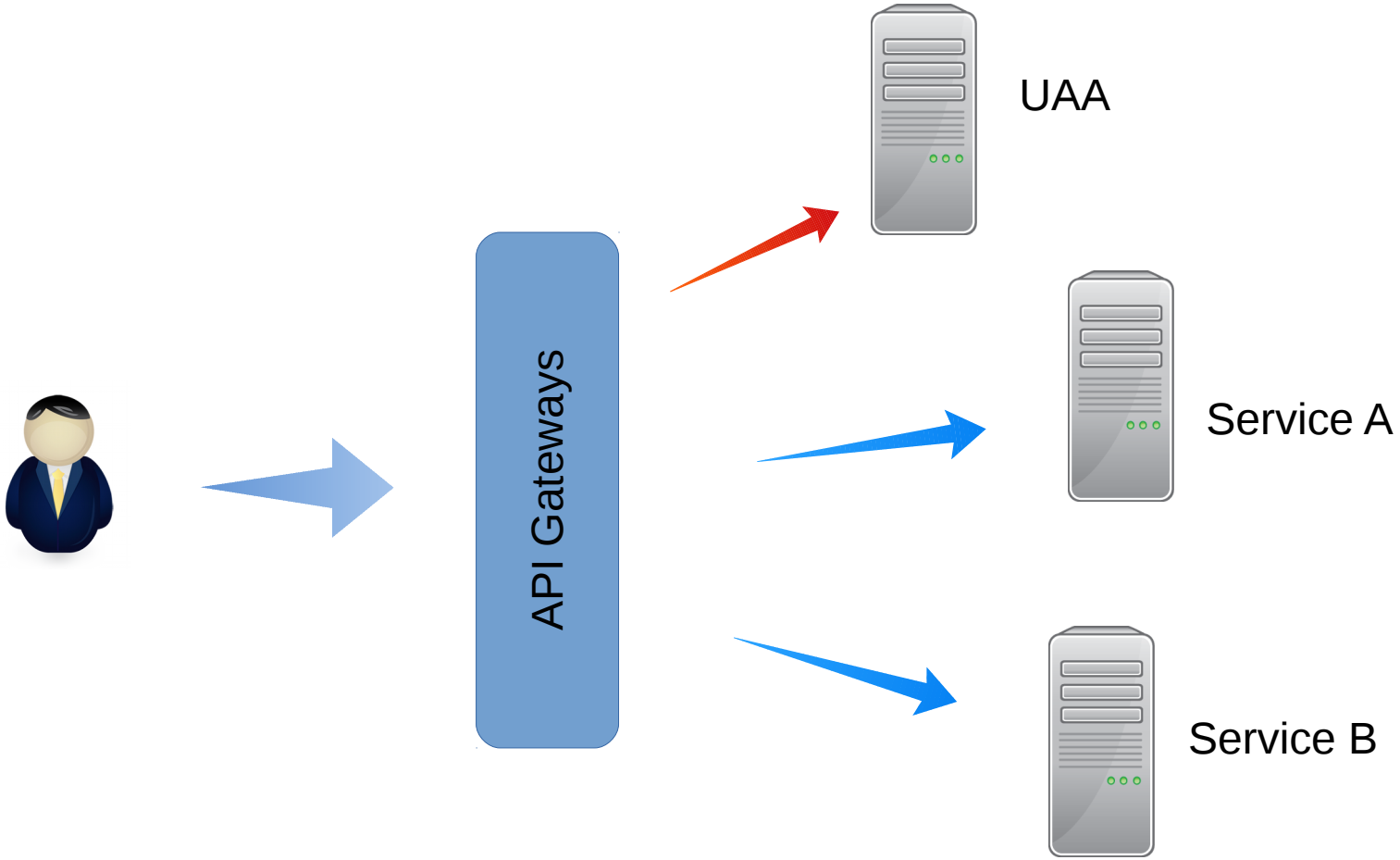
Trust JWT Token ?

Lab: Basic OAuth (jwt-keypair)

Lab: Connect Google OAuth

Spring Cloud Zuul

NETFLIX OSS



Lab: Basic OAuth (zuul)

Q & A

Additional References

- **Github Baeldung/spring-security-oauth**
- **Spring Security OAuth Reference**
- **Securing Microservices with Spring Cloud Security by Pivotal**
- **Modern Security with OAuth 2.0 and JWT and Spring by Dmitry Buzdin**