**PXL Secure Coding**

### RISK #1: SQL Injection

**Exploit**

' or '1'='1

' or '1'='1

**Message**: wireless charging is overrated -> message

**SQL sanitization:**

app.get('/authenticate/:username/:password', async (request, response) => {

const username = request.params.username;  
const password = request.params.password;

const query = 'SELECT \* FROM users WHERE user\_name=$1 and password=$2';  
console.log(query);

pool.query(query, [username, password], (error, results) => {

if (error) {

throw error

}

response.status(200).json(results.rows)});

});

### RISK #2: Insecure Storage

CREATE DATABASE pxldb;

\c pxldb;

CREATE USER secadv WITH PASSWORD 'ilovesecurity';

GRANT ALL PRIVILEGES ON DATABASE pxldb TO secadv;

CREATE TABLE users (

id SERIAL PRIMARY KEY,

user\_name TEXT NOT NULL UNIQUE,

password\_hash TEXT NOT NULL

);

GRANT ALL PRIVILEGES ON TABLE users TO secadv;

BEGIN;

INSERT INTO users (user\_name, password\_hash)

VALUES ('pxl-admin'**, crypt('insecureandlovinit'**, gen\_salt('bf', 8))),

('george', **crypt('iwishihadbetteradmins'**, gen\_salt('bf', 8)));

COMMIT;

Note that the password column has been replaced with a password\_hash column, which will store the bcrypt-hashed password. The crypt() function is used to perform the hashing, with a randomly generated salt of length 8 using the gen\_salt() function. The number 8 is the cost factor, which determines the computational cost of the hashing process. A higher cost factor makes the hashing process more computationally expensive, making it harder for attackers to crack the passwords.

### RISK #3: CORS

const pg = require('pg');

const express = require('express');

const bodyParser = require('body-parser');

const app = express();

const cors = require('cors')

const port=3000;

const pool = new pg.Pool({

user: 'secadv',

host: 'db',

database: 'pxldb',

password: 'ilovesecurity',

port: 5432,

connectionTimeoutMillis: 5000

})

console.log("Connecting...:")

//CORS

const allowedOrigins = ['http://localhost:8080'];

const corsOptions = {

origin: function (origin, callback) {

if (allowedOrigins.indexOf(origin) !== -1 || !origin) {

callback(null, true);

} else {

callback(new Error('Not allowed by CORS'));

}

}

};

//CORS

app.use(cors(corsOptions));

app.use(bodyParser.json());

app.use(

bodyParser.urlencoded({

extended: true,

})

)

app.get('/authenticate/:username/:password', async (request, response) => {

const username = request.params.username;

const password = request.params.password;

const query = 'SELECT \* FROM users WHERE user\_name=$1 and password=$2';

console.log(query);

pool.query(query, [username, password], (error, results) => {

if (error) {

throw error

}

response.status(200).json(results.rows)});

});

// app.get('/authenticate/:username/:password', async (request, response) => {

// const username = request.params.username;

// const password = request.params.password;

// const query = `SELECT \* FROM users WHERE user\_name='${username}' and password='${password}'`;

// console.log(query);

// pool.query(query, (error, results) => {

// if (error) {

// throw error

// }

// response.status(200).json(results.rows)});

// });

app.listen(port, () => {

console.log(`App running on port ${port}.`)

})

### RISK #4: Credentials in Version Control

Methods: Commit with git-ignore, use ansible-vault, environment variables

### …

**PXL Oauth**

CORS?

CORS stands for Cross-Origin Resource Sharing. It is a mechanism that allows web browsers to securely request resources (such as data, fonts, or images) from a different domain than the one that served the original web page. It is an important security feature implemented in web browsers to prevent cross-origin attacks.

By default, web browsers restrict requests to resources from the same origin or domain for security reasons. However, there are legitimate use cases where a web application may need to access resources from a different domain. CORS provides a way for servers to declare which cross-origin requests are allowed and which are not.

When a web browser makes a cross-origin request, it first sends a preflight request (an HTTP OPTIONS request) to the server to check if the server allows the actual request. The server responds with CORS headers that specify the permissions granted to the requesting domain. If the server approves the request, the browser allows the subsequent actual request to proceed.

CORS headers include:

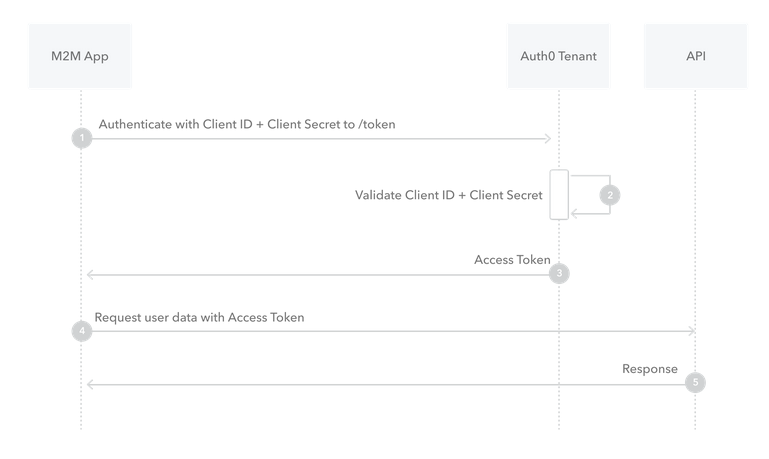
1. Access-Control-Allow-Origin: Specifies which origins are allowed to access the resource. It can be a specific origin, "\*", to allow any origin, or null, which means the resource is not available to any origin.
2. Access-Control-Allow-Methods: Specifies the HTTP methods (GET, POST, PUT, DELETE, etc.) that are allowed for the requested resource.
3. Access-Control-Allow-Headers: Specifies the HTTP headers that are allowed in the request.
4. Access-Control-Allow-Credentials: Indicates whether the request can include credentials, such as cookies or authorization headers.

The use of localhost:8080 is specific to your local development environment, where you may have configured the web app to run on localhost:8080 for testing purposes. However, within the Docker container, the web app is accessed through the container network, and the hostname web is used to refer to it.

Therefore, the request would be sent from http://web:80 to http://api:80 within the container network. The actual communication between the containers is facilitated through the internal Docker network and does not involve localhost or external network addresses.

Client credentials

The Client Credentials Flow (defined in [OAuth 2.0 RFC 6749, section 4.4](https://tools.ietf.org/html/rfc6749" \l "section-4.4)) involves an application exchanging its application credentials, such as client ID and client secret, for an access token.

This flow is best suited for Machine-to-Machine (M2M) applications, such as CLIs, daemons, or backend services, because the system must authenticate and authorize the application instead of a user.

## Registratie van Client en Resource

De identity server weet in feite niets af van de api. Het enige wat de identity server heeft, is een lijst van clients met voor elke client een lijst van scopes. Denk bijvoorbeeld aan je studentenpas, daar staat op wat je wel of niet mag doen. De identity server keert de tokens uit en op dat token staat dan de scope: "wat kan er met dit token gedaan worden?". Het is de verantwoordelijkheid van de api om die scope te controleren.

Autorize op de API

services.AddAuthentication("Bearer")

.AddJwtBearer("Bearer", options =>

{

options.Authority = "http://localhost:5002";

options.RequireHttpsMetadata = false;

options.TokenValidationParameters = new TokenValidationParameters

{

ValidateAudience = false

};

});

services.AddAuthorization(options => {

options.AddPolicy("ApiScope", policy =>

{

policy.RequireAuthenticatedUser();

policy.RequireClaim("scope", "krc-genk");

});

});

De eerste regel zorgt ervoor dat we een nieuw "authentication scheme" definiëren. Net zoals bij de CORS policy is dit louter de definitie van de middleware, nog niet de toepassing er van. We lezen hier dat authorizatie gebeurt aan de hand van JWT bearer tokens (access tokens) en dat we enkel tokens aanvaarden die uitgekeerd zijn door http://localhost:5002. In een realitischer scenario zou dat bijvoorbeeld Google kunnen zijn. We zetten ook HTTPS validatie uit, dat is niet zo verstanding, maar de uitbreiding naar HTTPS is een opdracht voor de PE. De validate audience is voor ons niet zo belangrijk en ook buiten de scope van dit vak.

**LET OP!** Om de AddJwtBearer methode te laten werken hebben je het Microsoft.AspNetCore.Authentication.JwtBearer package nodige (versie 3.0.0). Normaal gezien is dat pakket echter al geïnstalleerd. Als de compiler klaagt over deze methode moet je dat pakket zelf nog even toevoegen.

We voegen nog een Authorization scheme toe (niet te verwarren met bovenstaande authentication!). We zeggen hier eigenlijk dat enkele geauthenticeerde gebruikers toegang krijgen en die gebruikers moeten ook de machtiging krc-genk hebben.

## Is dit veilig?

Neen.

Uiteraard printen we nu de access token af in de developer console, maar dat is zeker niet het grote probleem. Ga naar http://localhost:8080 en open de developer console. In chrome, ga naar het networks tabblad. Refresh de pagina. Daar staat een request tussen: token. Klik op deze request en kijk eens naar het tabblad Payload. Daar staat ons client\_secret te grabbel.

De Client Credentials flow is niet aangeraden voor single page applications. Bovendien doen we hier ook niet aan authenticatie van de gebruiker. De reden dat we dit zo opgebouwd hebben in dit lab is omdat het ons in staat stelt om eens geconfronteerd te worden met de developer nachtmerrie die CORS is en het ons in staat stelt om eens een JWT Bearer token flow op te stellen in een complexer ecosysteem. We hopen vooral dat je ziet dat het instellen van security in een productieomgeving een veel complexere taak is dan je services werkende te krijgen op je eigen machine.

Zou het gebruik van HTTPS ons helpen om de client secret in de web app te verbergen?

Welke andere OAuth flow zou hier toepasselijk zijn? Is dit wel een correcte case om OAuth toe te passen?

Would using HTTPS help us hide the client secret in the web app?

Yes, using HTTPS (HTTP over SSL/TLS) would help in securing the communication between the web app and the server. When using HTTPS, the data transmitted between the client and server is encrypted, providing confidentiality. This encryption helps protect sensitive information, including the client secret, from being easily intercepted or exposed.

Which other OAuth flow would be appropriate here? Is this a suitable case for applying OAuth?

In the scenario you described, the Client Credentials flow is not recommended for single-page applications (SPAs) because it lacks user authentication. The Client Credentials flow is typically used for server-to-server communication.

For a suitable case in this scenario, where user authentication is involved, the Authorization Code flow or the Implicit flow would be more appropriate for SPAs.

1. Authorization Code Flow: This flow involves the redirection of the user to the authorization server, authentication, and consent. The authorization server then provides an authorization code to the client, which is exchanged for an access token. This flow is more secure as it involves both user authentication and authorization.
2. Implicit Flow: This flow is designed for clients that cannot securely store the client secret, such as SPAs running in a web browser. It involves the user being redirected to the authorization server, authentication, and consent. The authorization server then directly provides the access token to the client as a fragment identifier in the redirection URI.

Both of these flows provide a better mechanism for user authentication and authorization in SPAs compared to the Client Credentials flow.

Here's a high-level overview of the OAuth process:

1. Client registration: The client (third-party application) registers itself with the authorization server and obtains a client identifier and a client secret. This registration step establishes trust between the client and the authorization server.
2. Authorization request: The client initiates the OAuth flow by redirecting the user (resource owner) to the authorization server. The request includes the client identifier, the desired scope of access, and a redirect URL to return the user after authentication.
3. User authentication and consent: The user is redirected to the authorization server's authentication page, where they enter their credentials to authenticate themselves. After successful authentication, the authorization server presents the user with a consent screen that explains the requested permissions and asks for their approval to grant access to the client.
4. Authorization grant: If the user consents, the authorization server issues an authorization grant (usually in the form of an access token) to the client. The authorization grant represents the user's consent to access their protected resources.
5. Access token request: The client sends a request to the authorization server, providing the authorization grant and its client credentials (client identifier and client secret).
6. Access token issuance: The authorization server validates the request and, if valid, issues an access token to the client. The access token is a credential that allows the client to access the protected resources on behalf of the user.
7. Resource access: The client includes the access token in subsequent requests to the resource server. The resource server verifies the access token's validity and permissions. If valid, the resource server provides the requested resources to the client.

OAuth is designed to be secure and allows users to control the level of access granted to third-party applications. It eliminates the need for sharing sensitive credentials and provides a standardized way for users to authorize application access to their resources.