**COMP.SEC.300**

## General description

The app is a simple demonstration for creating a user login and registration and using JWT-tokens for authentication. The UI consists of the main screen where the user can choose to either login or register a new user and when either of these are successful the user is directed to a new page in the application.

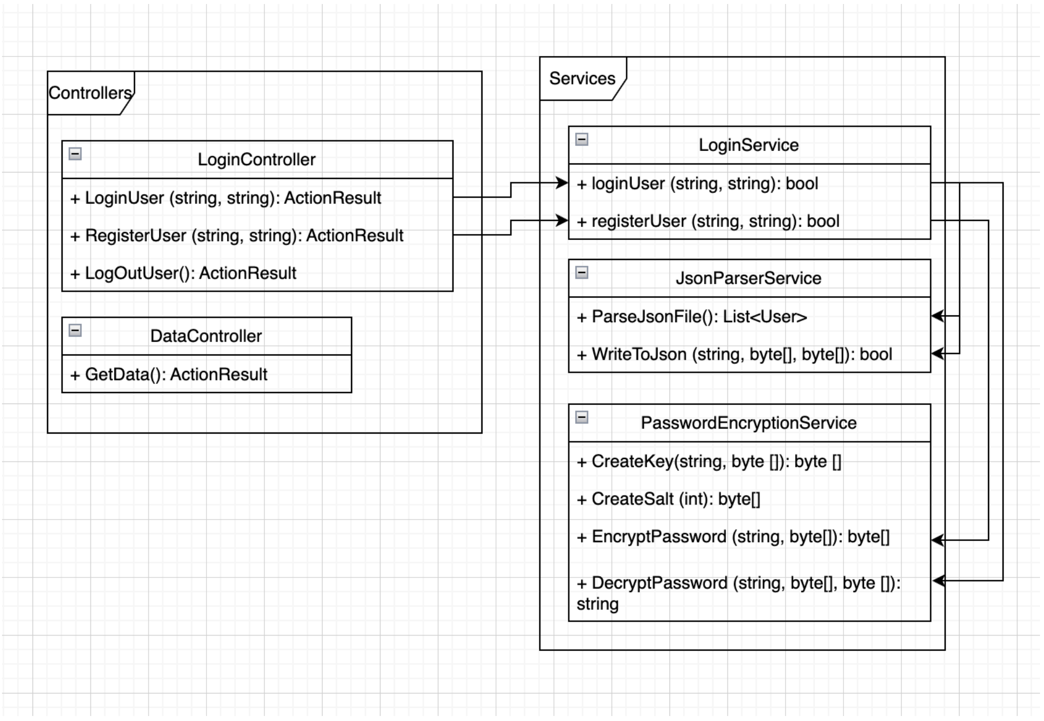
The app focuses on creating secure login functionalities by using encryption and decryption methods with salt and keys and also creating JWT-tokens for users and using authorized access to specific resources.

## Structure of the program

The program is created with a .NET backend and React frontend.

The backend follows a simple controller and service approach, with having two controllers, one for login and one for displaying data and three services, one for login, one for encrypting and decrypting passwords and one for parsing the JSON-file where the user, password and salts are stored. The program uses a JSON-file as storing mechanism as the aim for this application was to only practice encrypting and decrypting passwords and generating JWT-tokens so therefore the storage itself was not in focus.

Below is a simple diagram of the architecture for the backend implementation:


Controllers
LoginController
+ LoginUser (string, string): ActionResult
+ RegisterUser (string, string): ActionResult
+ LogOutUser(): ActionResult
DataController
+ GetData(): ActionResult
Services
LoginService
+ loginUser (string, string): bool
+ registerUser (string, string): bool
JsonParserService
+ ParseJsonFile(): List<User>
+ WriteToJson (string, byte[], byte[]): bool
PasswordEncryptionService
+ CreateKey(string, byte []): byte []
+ CreateSalt (int): byte[]
+ EncryptPassword (string, byte[]): byte[]
+ DecryptPassword (string, byte[], byte []): string

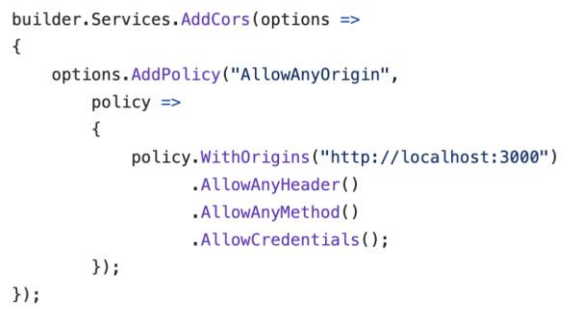
The frontend is also simplified to only one component being the main App, which handles the user login, logout and registration functionalities as well as retrieving the data after successful authentication.

## Secure programming solutions

OWASP TOP 10:

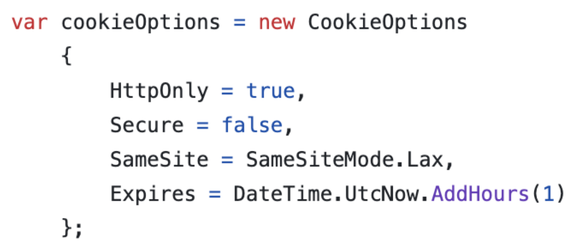
1. **Broken access control**

The broken access control means that users are able to act outside of their permissions and view data that they are not entitled to. The LogIn-app has prevented this by implementing a CORS-policy as the LogIn-app accepts requests only from the localhost:3000 which is the URL where the frontend is running. This prevents access from any other sites trying to make requests to the backend.



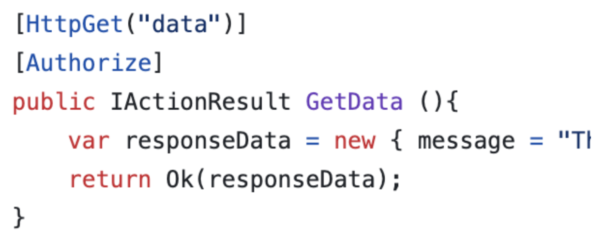
```
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowAnyOrigin",
        policy =>
        {
            policy.WithOrigins("http://localhost:3000")
                  .AllowAnyHeader()
                  .AllowAnyMethod()
                  .AllowCredentials();
        });
});
```

The LogIn-app also implements a JWT-token that is HTTP-only. This means that the JWT-token can be set and modified only from the backend and the tokens can’t be tampered by the user.



```
var cookieOptions = new CookieOptions
    {
        HttpOnly = true,
        Secure = false,
        SameSite = SameSiteMode.Lax,
        Expires = DateTime.UtcNow.AddHours(1)
    };
```

The data controller has the authorized annotation implemented which means that the controller accepts requests only from authorized users. This prevents access from users who are not allowed to get access to the data.



```
[HttpGet("data")]
[Authorize]
public IActionResult GetData (){
    var responseData = new { message = "Th
    return Ok(responseData);
}
```

## 2. Cryptographic failures

This refers to data being encrypted accordingly in transit and at rest. The LogIn-app enforces proper encryption by encrypting all passwords for the user before storing them in a JSON-file.

```
public async Task<byte[]> EncryptPassword (string password, byte[] salt){
    using Aes aes = Aes.Create();

    aes.Key = CreateKey(password, salt);
    aes.IV = CreateSalt(16);

    using MemoryStream output = new();
    using (CryptoStream cryptoStream = new(output, aes.CreateEncryptor(), CryptoStreamMode.Write)){

    await cryptoStream.WriteAsync(salt);
    await cryptoStream.WriteAsync(aes.IV);

    byte[] passwordBytes = Encoding.Unicode.GetBytes(password);
    await cryptoStream.WriteAsync(passwordBytes, 0, passwordBytes.Length);
    await cryptoStream.FlushFinalBlockAsync();}

    return output.ToArray();
}
```

The passwords are stored with strong and adaptive hashing functions as the LogIn-app uses PBKDF2 as a hashing function for the passwords. The PBKDF2-library was chosen as it is recommended by OWASP.

```
return Rfc2898DeriveBytes.Pbkdf2(Encoding.Unicode.GetBytes(password),
                                  salt,
                                  iterations,
                                  hashMethod,
                                  desiredKeyLength);
```

Keys are generated cryptographically randomly and the salt is created by getting random bytes from the RNGCryptoServiceProvider-class. The RNGCryptoServiceProvider-class was chosen as it is compliant with security standards, and it provides a higher quality of randomness than the System.Random-class.

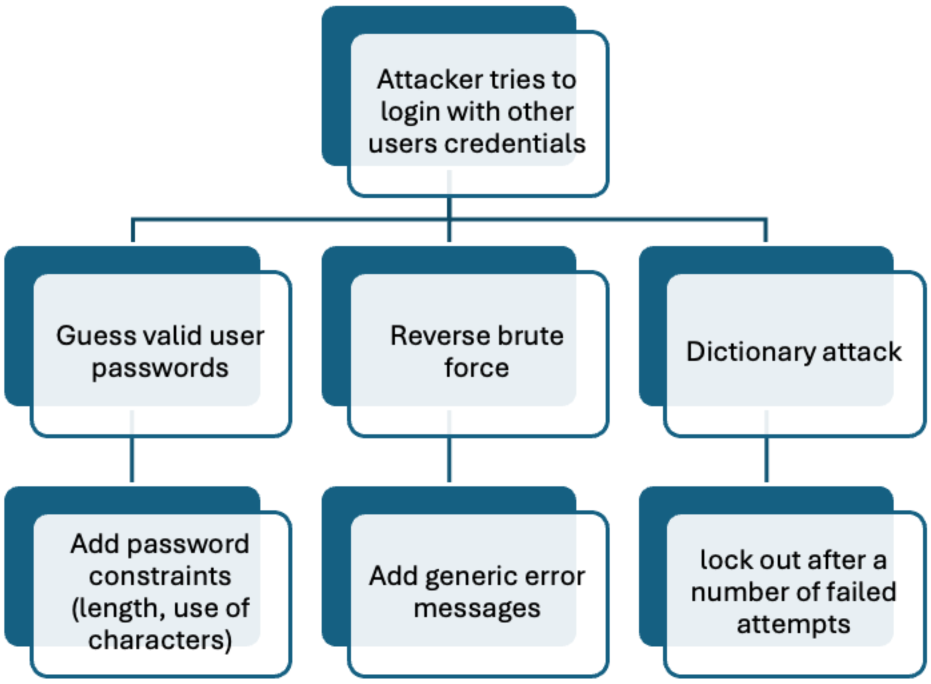
```
public byte[] CreateSalt (int length){
    using var rng = new RNGCryptoServiceProvider();
    byte [] randomBytes = new byte[length];

    rng.GetBytes(randomBytes);
    return randomBytes;
}
```

## 3. Injection

Injection is a vulnerability where the user can input malicious code or data to the database due to the user input not being validated and sanitized correctly. The LogIn-app doesn't prevent or address this issue since it does not have a database installed.

4. **Insecure design**

Insecure design means a lack of taking security risks into account when designing the application. A secure design should evaluate any threats the developed application might expose and how these can be prevented. In the case of LogIn-app a threat model tree could look something like this:


Attacker tries to login with other users credentials
Guess valid user passwords
Reverse brute force
Dictionary attack
Add password constraints (length, use of characters)
Add generic error messages
lock out after a number of failed attempts

The LogIn-app addresses two of the scenarios (password constraints and generic error messages) but having the lockout after a number of failed attacks is something that could be still implemented in the LogIn-app.

5. **Security Misconfiguration**

Security misconfiguration refers to an application having improper configuration settings such as leaving default settings or using default users with their default passwords. The LogIn-app prevents improper configuration by having correct security headers in place. Also, the error-handling of the application doesn't reveal any error messages to the users and the application only includes dependencies that are used in the application.

```
      content="Web site created using create-react-app"
    />
    <meta http-equiv="Content-Security-Policy" content="default-src 'self'; scrip
    <title>React App</title>
  </head>
```

## 6. Vulnerable and outdated components

This security issue is exposed by using unknown versions of packages or not updating and tracking packages accordingly. The LogIn-app has a DevSecOps-pipeline which creates a SBOM-file of all dependencies used in the application and it also performs an OWASP Dependency check which alerts for any possible vulnerabilities in the libraries that are used in the application.

## 7. Identification and authentication failures

The identification and authentication failures may occur if the users are allowed to use weak passwords or otherwise vulnerable authentication mechanism. The LogIn-app prevents identification and authentication failures by forcing the user to use a strong enough password before storing the user credentials in the JSON-file.

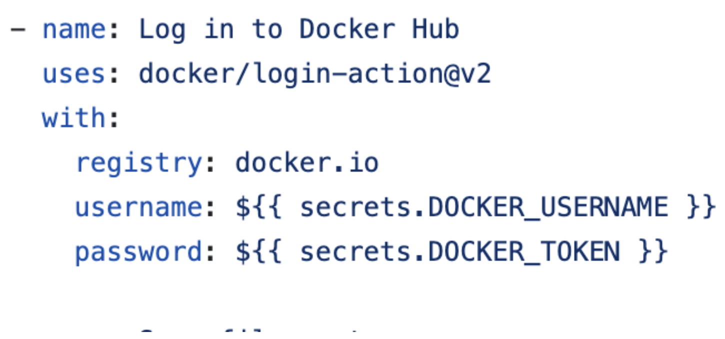
```
Regex regex = new Regex(@"^(?=.*?[A-Z])(?=.*?[a-z])(?=.*?[0-9])(?=.*?[#?!@$%^&*-]).{8,}$");
Match match = regex.Match(password);

 if(!match.Success){
     return BadRequest(new {message = "Invalid password format. Password should include at least
```

The LogIn-app also handles sessions by JWT-tokens that are managed and created on the server-side. The token is invalidated after logout and has a time out.

```
[HttpGet("logout-user")]
public async Task<IActionResult> LogoutUser(){
    var cookieOptions = new CookieOptions
            {
                HttpOnly = true,
                Secure = false,
                SameSite = SameSiteMode.Lax,
                Expires = DateTime.UtcNow.AddDays(-1)
            };

    Response.Cookies.Append("jwt", "", cookieOptions);
    return Ok();

}
```

## 8. Software and data integrity failures

Software and data integrity failures occur when an application relies on any resources from untrusted resources or the CI/CD-pipeline exposes security risks. The LogIn-app has a stage of OWASP dependency check to report any components with known vulnerabilities. The CI/CD-pipeline also has all secrets stored as Git secrets rather than having them hardcoded in the pipeline.



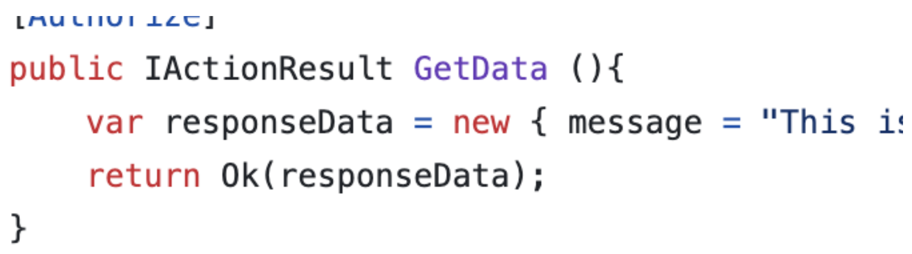
```
- name: Log in to Docker Hub
  uses: docker/login-action@v2
  with:
    registry: docker.io
    username: ${{ secrets.DOCKER_USERNAME }}
    password: ${{ secrets.DOCKER_TOKEN }}
```

9. **Security logging and monitoring failures**

Applications should log and monitor applications activity in order to respond to any active breaches. As the LogIn-app is not intended to be published the application doesn't include any logging or monitoring activity at the moment.

10. **Server-side request forgery**

These flaws allows an attacker to make the application send any requests to unexpected destinations. The LogIn-app prevents the user from getting any information of the internal services by sending validated responses instead of raw responses.



```
[Authorize]
public IActionResult GetData (){
    var responseData = new { message = "This i
    return Ok(responseData);
}
```

## Initial running of the DevSecOps-pipeline

I created a DevSecOps-pipeline for the LogIn-app which consisted of five components:

1. SonarQube – SonarQube was used for running a static application security testing to detect any possible vulnerabilities in the code
2. OWASP Dependency check – this was used to detect any possible vulnerabilities or issues with the libraries used in the application
3. SBOM – the SBOM-file was created to have an easy access for all the libraries etc. dependencies used in the application

4. OWASP Zap – OWASP Zap for used for dynamic application security testing to detect any vulnerabilities in the code that happen at runtime
5. Trivy – Trivy scanning was performed to find any vulnerabilities in the container images

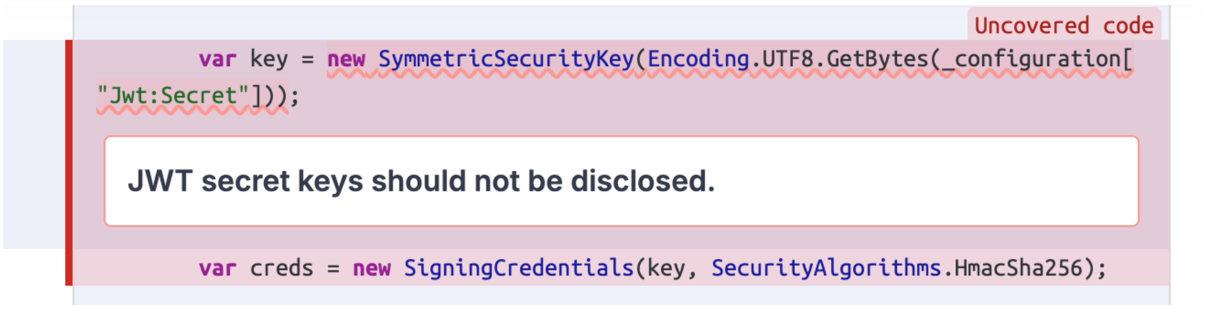
Once I was finished with the coding work, I ran the whole pipeline to find any vulnerabilities or issues with the code. The issues that the pipeline detected are described below.

1. SonarQube scanning

SonarQube scanning reported one security blocker, three medium security issues and one low priority security issue to review for the backend. For the frontend the scanning reported two medium security issues which were basically the same as for the backend relating to the docker file.

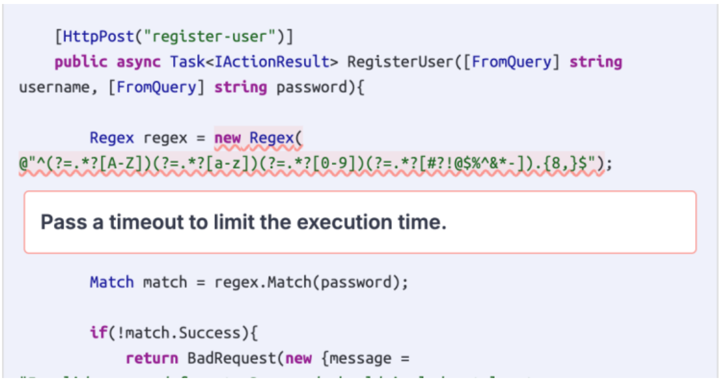
1.1. JWT secret key disclosed - BLOCKER

The JWT-key is stored in a configuration file and accessed from the configuration file in the application. This makes leaking the secret key more likely.


```
                                                                  Uncovered code
        var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration[
"Jwt:Secret"]));
```
JWT secret keys should not be disclosed.
```
        var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);
```

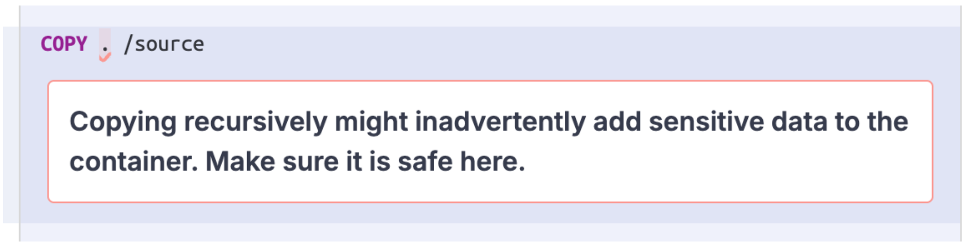
1.2. Passing timeout to limit execution time - MEDIUM

The registration form for the user checks whether the given password has a valid format. This regex has no time limit for execution so the scanner alerts this as a possibility for a hacker to perform a DDoS-attack.


```
    [HttpPost("register-user")]
    public async Task<IActionResult> RegisterUser([FromQuery] string
username, [FromQuery] string password){

        Regex regex = new Regex(
@"^(?=.*?[A-Z])(?=.*?[a-z])(?=.*?[0-9])(?=.*?[#?!@$%^&*-]).{8,}$");
```
Pass a timeout to limit the execution time.
```
        Match match = regex.Match(password);

        if(!match.Success){
            return BadRequest(new {message =
```

1.3. Recursive copying – MEDIUM

The docker file copies the whole context directory to the docker daemon before the build. Since the entire top-level directories are copied, unexpected files might also get copied to the filesystem. This issue was reported for both backend and frontend.

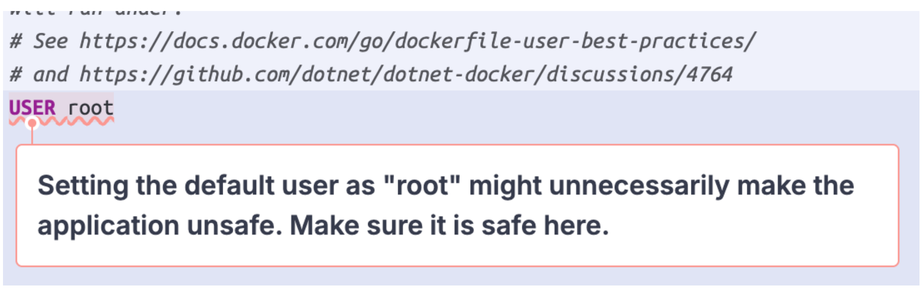


```
COPY . /source
```

Copying recursively might inadvertently add sensitive data to the container. Make sure it is safe here.

1.4. Setting default user as root – MEDIUM

The docker file sets the default user as root which means that any user whose code is running on the container can perform administrative actions. This weakens the containers' runtime security.

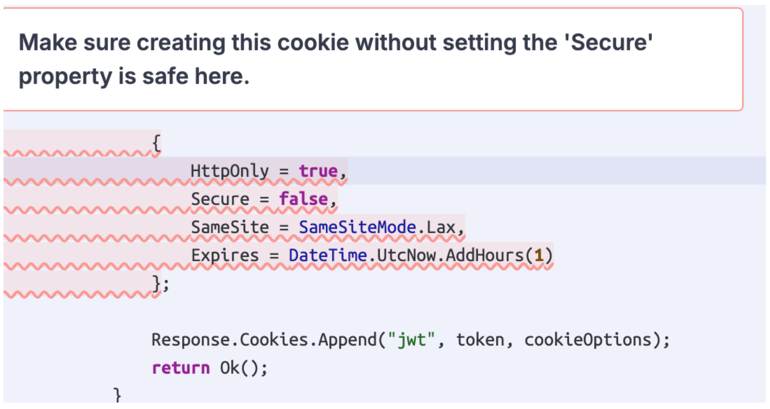


```
# See https://docs.docker.com/go/dockerfile-user-best-practices/
# and https://github.com/dotnet/dotnet-docker/discussions/4764
USER root
```

Setting the default user as "root" might unnecessarily make the application unsafe. Make sure it is safe here.

1.5. Creating cookie with secure attribute as false – LOW

The cookie created in the application is sent with the attribute secure set to false. This exposes vulnerabilities as the cookie can be observed by an unauthorized person during a man-in-the-middle attack.

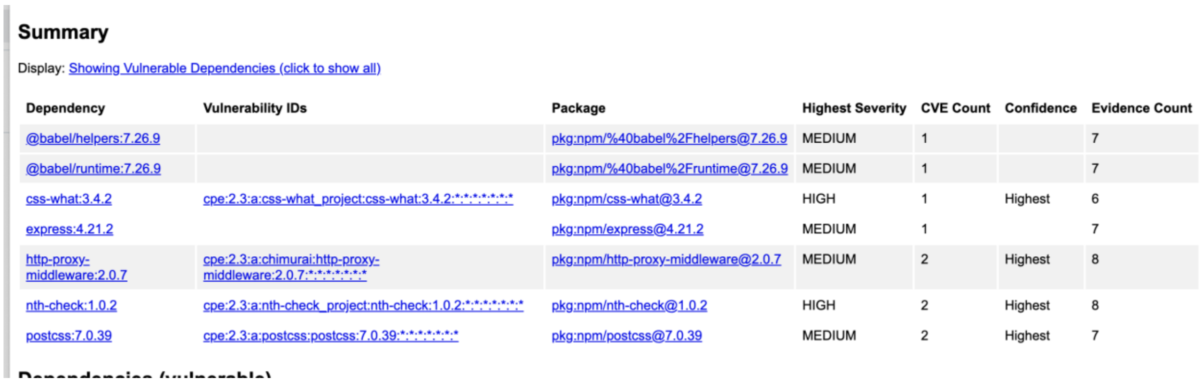


Make sure creating this cookie without setting the 'Secure' property is safe here.

```
                {
                    HttpOnly = true,
                    Secure = false,
                    SameSite = SameSiteMode.Lax,
                    Expires = DateTime.UtcNow.AddHours(1)
                };

                Response.Cookies.Append("jwt", token, cookieOptions);
                return Ok();
            }
```

2. OWASP Dependency check

The owas dependency check reported the following libraries as vulnerabilities for the frontend:

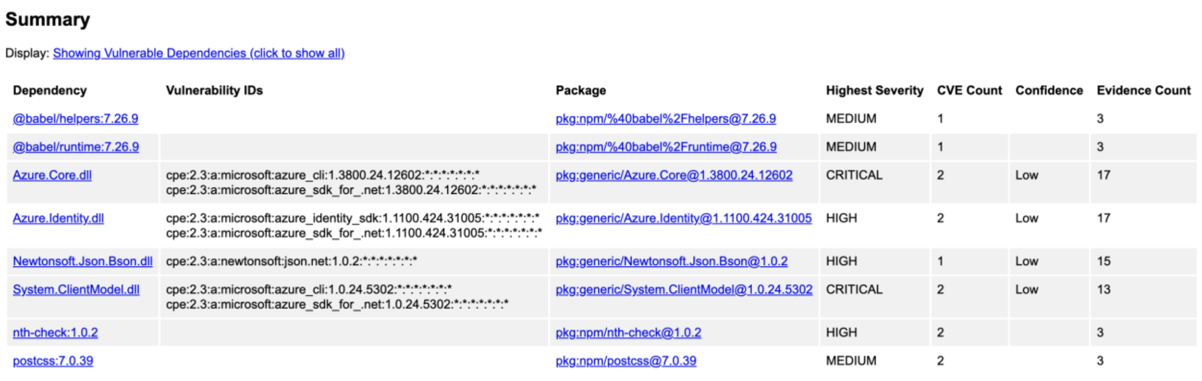


**Summary**

Display: Showing Vulnerable Dependencies (click to show all)

| Dependency | Vulnerability IDs | Package | Highest Severity | CVE Count | Confidence | Evidence Count |
|---|---|---|---|---|---|---|
| @babel/helpers:7.26.9 | | pkg:npm/%40babel%2Fhelpers@7.26.9 | MEDIUM | 1 | | 7 |
| @babel/runtime:7.26.9 | | pkg:npm/%40babel%2Fruntime@7.26.9 | MEDIUM | 1 | | 7 |
| css-what:3.4.2 | cpe:2.3:a:css-what_project:css-what:3.4.2:*:*:*:*:*:*:* | pkg:npm/css-what@3.4.2 | HIGH | 1 | Highest | 6 |
| express:4.21.2 | | pkg:npm/express@4.21.2 | MEDIUM | 1 | | 7 |
| http-proxy-middleware:2.0.7 | cpe:2.3:a:chimurai:http-proxy-middleware:2.0.7:*:*:*:*:*:*:* | pkg:npm/http-proxy-middleware@2.0.7 | MEDIUM | 2 | Highest | 8 |
| nth-check:1.0.2 | cpe:2.3:a:nth-check_project:nth-check:1.0.2:*:*:*:*:*:*:* | pkg:npm/nth-check@1.0.2 | HIGH | 2 | Highest | 8 |
| postcss:7.0.39 | cpe:2.3:a:postcss:postcss:7.0.39:*:*:*:*:*:*:* | pkg:npm/postcss@7.0.39 | MEDIUM | 2 | Highest | 7 |

And the following for the backend:



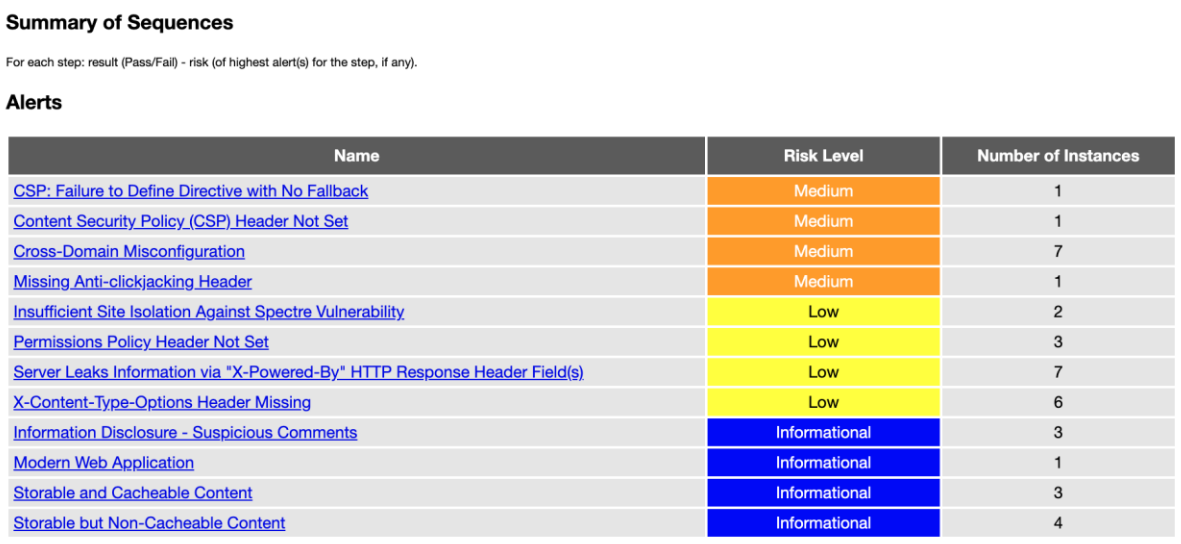
**Summary**

Display: Showing Vulnerable Dependencies (click to show all)

| Dependency | Vulnerability IDs | Package | Highest Severity | CVE Count | Confidence | Evidence Count |
|---|---|---|---|---|---|---|
| @babel/helpers:7.26.9 | | pkg:npm/%40babel%2Fhelpers@7.26.9 | MEDIUM | 1 | | 3 |
| @babel/runtime:7.26.9 | | pkg:npm/%40babel%2Fruntime@7.26.9 | MEDIUM | 1 | | 3 |
| Azure.Core.dll | cpe:2.3:a:microsoft:azure_cli:1.3800.24.12602:*:*:*:*:*:*:* cpe:2.3:a:microsoft:azure_sdk_for_.net:1.3800.24.12602:*:*:*:*:*:*:* | pkg:generic/Azure.Core@1.3800.24.12602 | CRITICAL | 2 | Low | 17 |
| Azure.Identity.dll | cpe:2.3:a:microsoft:azure_identity_sdk:1.1100.424.31005:*:*:*:*:*:*:* cpe:2.3:a:microsoft:azure_sdk_for_.net:1.1100.424.31005:*:*:*:*:*:*:* | pkg:generic/Azure.Identity@1.1100.424.31005 | HIGH | 2 | Low | 17 |
| Newtonsoft.Json.Bson.dll | cpe:2.3:a:newtonsoft:json.net:1.0.2:*:*:*:*:*:*:* | pkg:generic/Newtonsoft.Json.Bson@1.0.2 | HIGH | 1 | Low | 15 |
| System.ClientModel.dll | cpe:2.3:a:microsoft:azure_cli:1.0.24.5302:*:*:*:*:*:*:* cpe:2.3:a:microsoft:azure_sdk_for_.net:1.0.24.5302:*:*:*:*:*:*:* | pkg:generic/System.ClientModel@1.0.24.5302 | CRITICAL | 2 | Low | 13 |
| nth-check:1.0.2 | | pkg:npm/nth-check@1.0.2 | HIGH | 2 | | 3 |
| postcss:7.0.39 | | pkg:npm/postcss@7.0.39 | MEDIUM | 2 | | 3 |

## 4. OWASP Zap scanning

The scanning for the application reported 4 medium risk level, 4 low risk level and 4 informational alerts. Many of these alerts were related to missing headers such as content security policy, anti-clickjacking and permission policy headers being not set.



**Summary of Sequences**

For each step: result (Pass/Fail) - risk (of highest alert(s) for the step, if any).

**Alerts**

| Name | Risk Level | Number of Instances |
|---|---|---|
| CSP: Failure to Define Directive with No Fallback | Medium | 1 |
| Content Security Policy (CSP) Header Not Set | Medium | 1 |
| Cross-Domain Misconfiguration | Medium | 7 |
| Missing Anti-clickjacking Header | Medium | 1 |
| Insufficient Site Isolation Against Spectre Vulnerability | Low | 2 |
| Permissions Policy Header Not Set | Low | 3 |
| Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s) | Low | 7 |
| X-Content-Type-Options Header Missing | Low | 6 |
| Information Disclosure - Suspicious Comments | Informational | 3 |
| Modern Web Application | Informational | 1 |
| Storable and Cacheable Content | Informational | 3 |
| Storable but Non-Cacheable Content | Informational | 4 |

## 5. Trivy scanning

The container scan didn't alert any vulnerabilities for the backend image, but the frontend image reported 8 issues. The issues were the same as the vulnerabilities reported in the previous dependency check.

## Fixes and remaining vulnerabilities

Below is a table representing the alerts picked from the pipeline and the fixes that I was able to make.

| **Pipeline step** | **Issue** | **Fixed** | **Comments** |
|---|---|---|---|
| SonarQube | JWT Secret key disclosed | No | This was not fixed in the current project as I am not planning on publishing the app. In case of making the app public, I would store the JWT key into some key handler (key vault etc.) |
| SonarQube | Pass timeout to limit execution time | Yes | |
| SonarQube | Recursive copying in docker file | No | This was not fixed as the folder itself does not contain anything but the app itself so copying the entire folder is safe |
| SonarQube | Default user as root | No | This was not fixed as the app was running always in a new docker container which was set up just for the app so using the root is safe here |
| SonarQube | Cookie with secure attribute as false | No | I was testing the app only in my local environment, so I didn't use any https-connections. If the app were public, I would switch the attribute to true. |
| OWASP Zap | Missing CSP | Yes | The CSP was added as a meta-tag to the index.html file |
| OWAS Zap | Cross-Domain Misconfiguration | No | The CORS-policy is added to the backend-server and the alert is regarding the frontend |
| OWASP Zap | Missing Anti-Clickjacking header | Yes | The header was set in the webpack.config.js -file |
| OWASP Zap | Low- and informational alerts | No | Concentrated on the medium-level issues so these I didn't look into |

For the dependency check I updated any outdated packages in both npm and dotnet and run the dependency check again. Since the OWASP dependency check might contain false positives, I searched the packages from OSS index (https://ossindex.sonatype.org/) to see if the packages were actually marked as vulnerable. The project remained then with the following vulnerabilities:

| **Package** | **Comment** |
|---|---|

| | |
|---|---|
| nth-check: 1.0.2 | Would require updating package: css-select |
| post-css: 7.0.39 | Would require updating package: resolve-url-loader |

Updating the dependencies could be possible but with doing some researching I wasn't able to fix these dependencies myself.

## Further improvements

To improve the applications security the LogIn-app could implement the following functionalities:

1. For making the login functionality even more secure the LogIn-app could add a multi-factor authentication. This would make hacking another user's account more difficult as the login would not rely only on usernames and passwords.
2. The LogIn-app could also implement a lockout of a specific amount of time for each user if they fail to login within a set amount of retries. This would make fetching user identities by guessing a username and password correctly slower for the attacker.