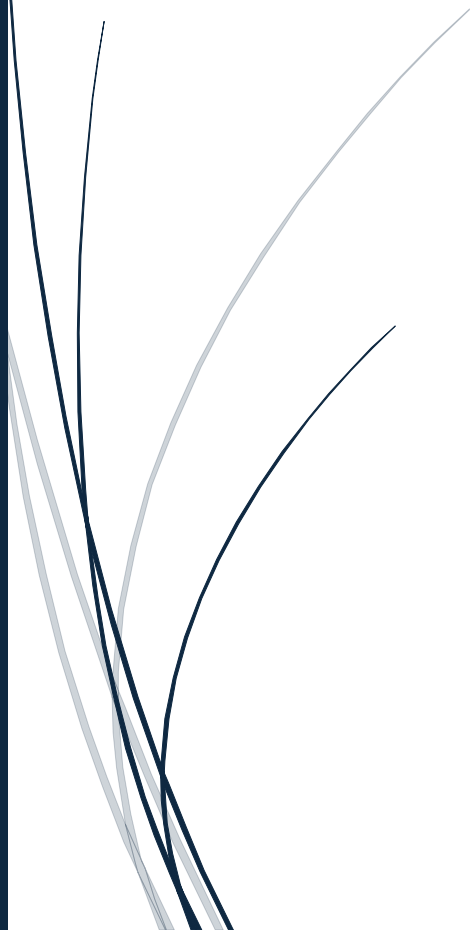


6/15/2024

Security Implementation For Sensitive Data and OWASP

Individual: FestivalConnect



Name: Lucas Jacobs

Class: S-A-RB06

PCN: 490692

Student number: 4607368

Technical teachers: Felipe Ebert, Bartosz Paszkowski

Semester coach: Gerard Elbers

Table of Contents

Introduction.....	1
Sensitive Data.....	2
OWASP.....	3
A01-Broken Access Control	3
A02-Cryptographic Failures	4
A03-Injection.....	4
A04-Insecure Design	4
A05-Security Misconfiguration	4
A06-Vulnerable and Outdated Components	4
A07- Identification and Authentication Failures.....	5
A08-Software and Data Integrity Failures	5
A10-Server-Side Request Forgery (SSRF)	5
References	5

Introduction

This document quickly explains the implementations I have taken for security. I specifically specified the requirements in my (Jacobs, Security Design).

Sensitive Data

To prevent unwanted users from entering your passwords and secrets, I used variable names specified in my code, such that if people gain access to the code, they don't gain access to the secrets.

For code, I used a key vault of Azure to store my secrets in there. This has several benefits to store them in a key vault of azure:

- Improved Security: I will have a central space to store my keys, using industry-standard algorithms to store my secrets securely, and manage who can access the key vault by using Azure Active Directory to authenticate and authorize access.
- Simple way of managing my secrets: functions such as versioning, easy integration to other services, and the option to have automated secret rotation.
- The price is also not a lot for retrieving the keys.

Secrets operations

€0.028/10,000 transactions

(Key Vault, n.d.)

See the following picture of how I connected to my key vault in my application

```
var config = builder.Configuration;

var credential = new ClientSecretCredential(config["AzureKeyVault:TenantId"], config["AzureKeyVault:ClientId"], config["AzureKeyVault:ClientSecret"]);
var client = new SecretClient(new Uri($"https://{config["AzureKeyVault:VaultName"]}.vault.azure.net/"), credential);
builder.Configuration.AddAzureKeyVault(client, new AzureKeyVaultConfigurationOptions());
```

After that you can easily specify the name of the secret.

```
var connectionString = config["IDENTITY-DB-AZURE"];
builder.Services.AddDbContext<DatabaseContext>(options =>
{
    options.UseMySQL(connectionString, ServerVersion.AutoDetect(connectionString));
});
```

The secrets can all be found back in the key vault itself.

Name	Type	Status	Expiration date
COMMUNITY-DB-IMAGE		✓ Enabled	
COMMUNITY-DB-LOCAL		✓ Enabled	
IDENTITY-DB-AZURE		✓ Enabled	
IDENTITY-DB-IMAGE		✓ Enabled	
IDENTITY-DB-LOCAL		✓ Enabled	
JWT-KEY		✓ Enabled	
NOTIFICATION-DB-IMAGE		✓ Enabled	
NOTIFICATION-DB-LOCAL		✓ Enabled	
POST-DB		✓ Enabled	
RABBITMQ-IMAGE		✓ Enabled	
RABBITMQ-LOCAL		✓ Enabled	
USER-CACHE		✓ Enabled	
USER-DB-AZURE		✓ Enabled	
USER-DB-IMAGE		✓ Enabled	
USER-DB-LOCAL		✓ Enabled	

Furthermore, in my YAML files I also used variables, see the following picture:

```
- az login --service-principal -u "$AZURE_APP_ID" -p "$AZURE_PASSWORD" --tenant "$AZURE_TENANT_ID"
```

OWASP

A01-Broken Access Control

For this, I am specifying only the roles that can execute this operation for each request. I am making sure to apply the least privileged access control, meaning only the minimum permissions can access the task. Furthermore, I am using the built-in ASP.NET Core authorization and authentication features.

This looks like the following when a user wants to delete his account, all the roles are allowed to do this.

```
[HttpDelete]
[Authorize(Roles = "FESTIVALGOER,FESTIVALORGANIZER,ADMIN")]
0 references
public async Task<ActionResult<bool>> DeleteUser()
```

Also in my gateway, I check if the request has a valid JWT.

```
{
  "DownstreamPathTemplate": "/identity",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    {
      "Host": "identityapi",
      "Port": 8080
    }
  ],
  "AuthenticationOptions": {
    "AuthenticationProviderKey": "Bearer"
  },
  "UpstreamPathTemplate": "/identity",
  "UpstreamHttpMethod": [ "Delete" ],
  "RateLimitOptions": {
    "EnableRateLimiting": true,
    "Period": "1s",
    "PeriodTimespan": 1,
    "Limit": 1
  },
  "LoadBalancerOptions": {
    "Type": "LeastConnection"
  }
}
```

With these checks, we improve security by checking that the request is from a valid source and the performance to reduce unauthorized users by validating the token. Furthermore, I used rate limiting to minimize the harm from automated attacks, which can be found back in the evidence of 'Preventing DDOS attack'.

Also in the front end, when a user logs in, I am using cookies to save their JWT.

```
import {CookiesProvider, useCookies} from 'react-cookie'
```

```
const [cookies, setCookie] = useCookies(['token'])
```

```
setCookie('token', response.data.token, { path: '/', secure: true, maxAge: 60 * 60 * 24 * 1 }); //set cookie for 1 day
```

Using cookies gives several benefits.

- Performance: Cookies are efficient, and do not use a lot of resources, which increases faster load times on pages. Cookies also easily scale by storing them on the client-side (browser), which means you do not consume server resources to store or manage them.
- Security: By having an expiration date, non-executable, using an encrypted transmission, and controlled by a browser that has built-in security features.

It contributes to this since cookies can make use of various attributes, encrypting transmissions (HTTPS), limiting the life span of the cookie to prevent session hijacking.

A02-Cryptographic Failures

As specified earlier, I am using a key vault, preventing hard-coded passwords in my code. Furthermore, for passwords that are stored in the database, I am using hashing and salting, which will prevent attackers from gaining access to passwords even when they gained access to the database. See the following picture how it is stored in the database

id	email	password	role_id
1	user@example.com	\$2a\$11\$KD7tQZdqH9zc3aICsQ5IOMnoTGVihM...	0

This encryption happens using a library. This looks like the following when adding a hashed password to the database

```

Password = BCrypt.Net.BCrypt.HashPassword(request.Password),
RoleId = request.RoleId
registered = _identityRepository.

```

string BCrypt.Net.BCrypt.HashPassword(string inputKey) (+ 3 overloads)
Hash a password using the OpenBSD BCrypt scheme and a salt generated by BCrypt.Net.BCrypt.GenerateSalt().

A03-Injection

For input validation, we use attributes in my models to validate that the incoming request data is valid and sanitized.

```

[Required, EmailAddress]
[StringLength(255)]
4 references
public string Email { get; set; } = string.Empty;

```

This property will be checked that is required in the request, is a valid email, and is not bigger than 255 characters long.

A04-Insecure Design

I have tried to prevent this by doing several tests such as unit testing, integration tests, E2E tests, performance tests, and user acceptance tests. This can be found back in the code itself, in (Jacobs, Acceptance Testplan) and in (Jacobs, Monitoring Azure Deployment). Furthermore, I did security checks in my application which can be viewed in (Jacobs, Implementation CI/CD).

A05-Security Misconfiguration

I implemented a full CI/CD pipeline that contained a secure way of performing several tests and operations to build and deploy the application, in (Jacobs, Implementation CI/CD).

A06-Vulnerable and Outdated Components

With this use of CI/CD, we have audit scans to check if libraries (third-party software) are up-to-date, or have other vulnerabilities.

A07- Identification and Authentication Failures

For passwords, I implemented a policy to be at least 8 characters long and need to contain one special character. This makes sure that a

```
[Required]
[StringLength(255, MinimumLength = 8)]
[RegularExpression(@"(?=[!@#$%^&*()_?":{}|<>]).{8,255}$", ErrorMessage = "Password must be between 8 and 255 characters long and contain at least one special character.")]
public string Password { get; set; } = string.Empty;
```

This will reduce the risk of brute force attacks, which will improve:

- Reliability: to have a consistent password policy which will make sure that there is consistent security across the application.
- Performance: By preventing weak passwords, the likelihood of a successful brute force decrease, which will reduce the load on the identity service and improve the system performance.
- Security: I not only make that application make it more secure, but it will also contribute to meeting the GDPR regulation which mentions having at least a password of eight characters.

A08-Software and Data Integrity Failures

As specified earlier I am using software and audit checks to check on plugins and libraries. This is also in my CI/CD pipeline.

A09-Security Logging and Monitoring Failures

With the use of monitoring, I can notice certain suspicious events. This is shown in (Jacobs, Monitoring Azure Deployment).

A10-Server-Side Request Forgery (SSRF)

As mentioned in other points, I implemented data sanitation and I will use verified sources when using third-party software.

References

Key Vault. (n.d.). Retrieved from azure.microsoft: <https://azure.microsoft.com/en-us/products/key-vault>

Jacobs, L. (2024). Security Design (Unpublished manuscript), FontysICT.

Jacobs, L. (2024). Monitoring Azure Deployment (Unpublished manuscript), FontysICT.

Jacobs, L. (2024). Acceptance Testplan (Unpublished manuscript), FontysICT.