

Segurança Informática e nas Organizações

Departamento de Eletrónica, Telecomunicações e Informática



universidade
de aveiro

1º Projeto – Vulnerabilidades

Ano letivo 2023/2024

Autores:

Rúben Marinho (40115)

Pedro Carneiro (73775)

Inês Águia (73882)

Filipe Posio (80709)

7 de Novembro de 2023

Índice

Índice.....	2
1 Contexto e Metodologia	3
2 Vulnerabilidades.....	4
2.1 CWE – 256: PlainText Storage of a Password	4
2.2 CWE - 521: Weak Password Requirements	7
2.3 CWE - 200: Exposure of Sensitive Information to an Unauthorized Actor}	10
2.4 CWE - 20: Improper Input Validation	12
2.5 CWE- 601: URL Redirection to Untrusted Site('Open Redirect')	13
2.6 CWE - 79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	14
2.7 CWE - 89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	17
3 Conclusão	20

1 Contexto e Metodologia

Este projeto surgiu no âmbito da disciplina de Segurança Informática e nas Organizações, de forma a desenvolver e assim consolidar os conceitos aprendidos nos primeiros meses do 1º Semestre na componente prática.

Serve o presente relatório para contextualizar o seu desenvolvimento, as ferramentas utilizadas, bem como as vulnerabilidades encontradas.

Através da análise individual de cada uma delas, o porquê de surgirem e as correções feitas para as evitar.

2 Vulnerabilidades

2.1 CWE – 256: PlainText Storage of a Password

Esta vulnerabilidade é conhecida como Plaintext Storage of a Password, ocorre quando as senhas dos utilizadores estão em formato de texto simples e podem ser facilmente lidas e consultadas por qualquer pessoa com acesso ao banco de dados ou arquivo onde se encontrem armazenadas.

Optar por armazenar senhas desta forma, torna-se um risco tanto para o utilizador como para o próprio sistema. Basta alguém com piores intenções (funcionário da empresa ou hacker), usar qualquer tipo de credenciais, incluindo de alguém com mais permissões no sistema.

No nosso caso em particular para esta vulnerabilidade foi criada a classe PasswordHasher, com os métodos ComputeHash and GenerateSalt.

De cada vez que um novo utilizador se regista, antes da password ser armazenada a classe é chamada.

```
public static class PasswordHasher
{
    5 referencias
    public static string ComputeHash(string password, string salt, string pepper, int iteration)
    {
        if (iteration <= 0) return password;

        using var sha256 = SHA256.Create();
        var passwordSaltPepper = $"{password}{salt}{pepper}";
        var byteValue = Encoding.UTF8.GetBytes(passwordSaltPepper);
        var byteHash = sha256.ComputeHash(byteValue);
        var hash = Convert.ToBase64String(byteHash);
        return ComputeHash(hash, salt, pepper, iteration - 1);
    }

    3 referencias
    public static string GenerateSalt()
    {
        using var rng = RandomNumberGenerator.Create();
        var byteSalt = new byte[16];
        rng.GetBytes(byteSalt);
        var salt = Convert.ToBase64String(byteSalt);
        return salt;
    }
}
```

Figura 1 - Classe PasswordHasher

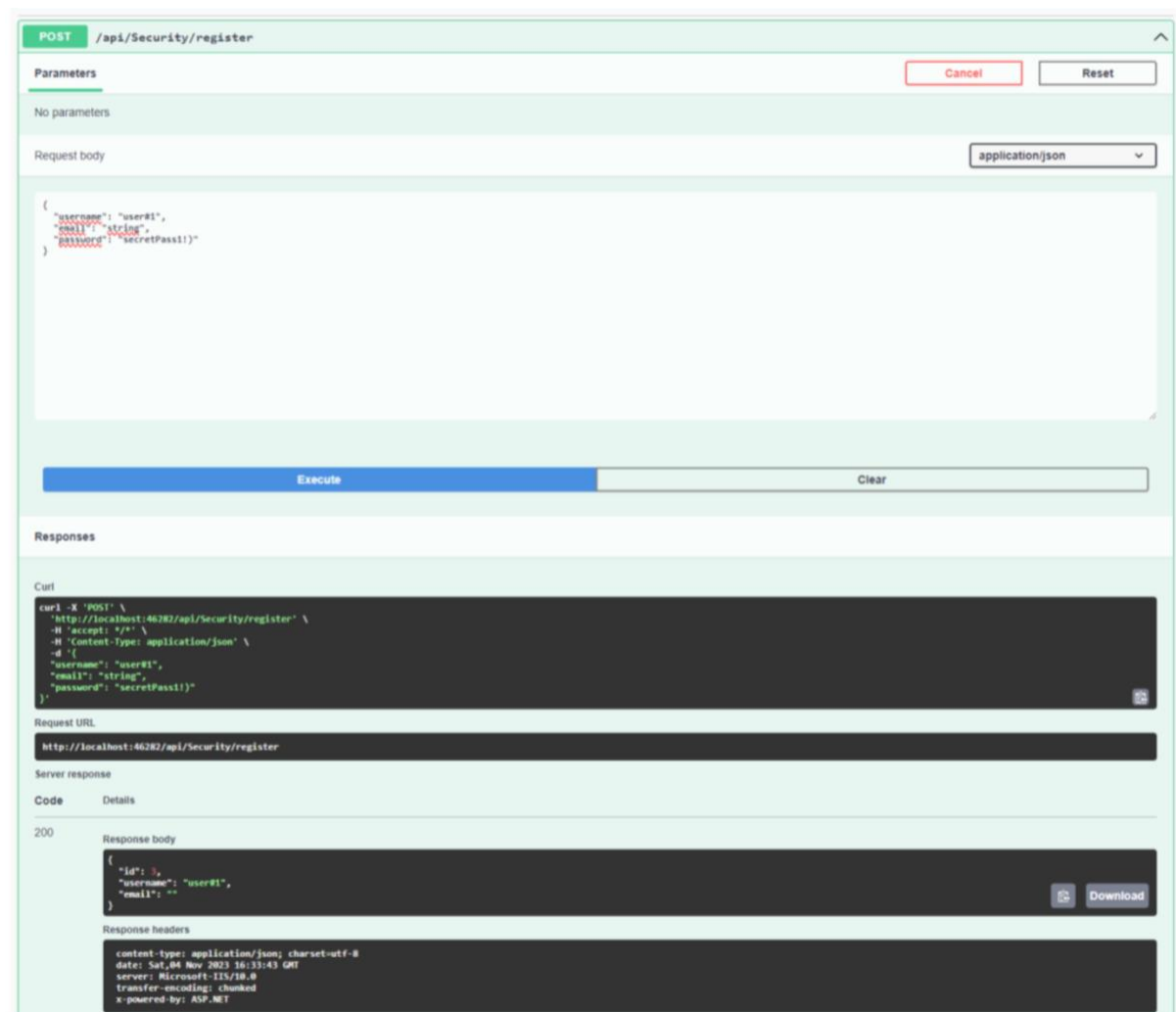


Figura 2 - Http Request versão insegura

users						
username = 'user#1'						
Grid	1	user_id	username	password_hash	password_salt	UserRole
1	3	user#1	secretPass1!	vY4a2OGDxNgkuAZnzXGg==	123	1

Figura 3 - Dados na Base de Dados, sem PasswordHasher

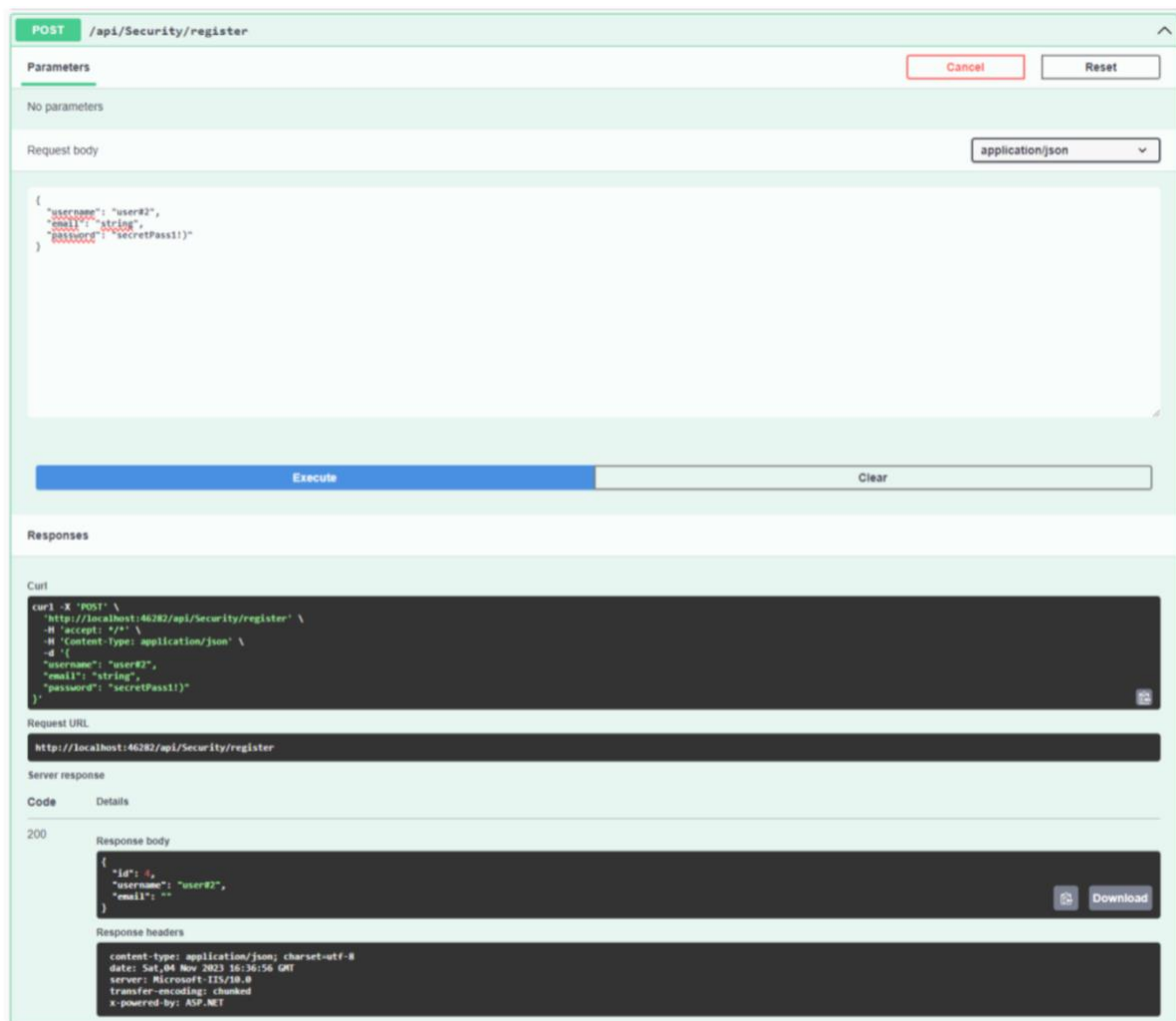


Figura 4 - Http Request versão segura

users						
username = 'user#2'						
	user_id	username	password_hash	password_salt	UserRole	
1	4	user#2	aJJWfoO44kDc/LqskBen9BCWxzrpgD/zZw515sYJgU=	kfsQ9RID1BiHeNPWStVnvA=	1	

Figura 5 - Dados na base de Dados, versão segura

Neste caso os logs mantêm-se iguais, a única mudança é a password que vai ser "hashed" nos parâmetros.

2.2 CWE- 521: Weak Password Requirements

Esta vulnerabilidade é conhecida como “Requisitos fracos de Password”.

Consiste em políticas de passwords “fracas”, isto é, se o utilizador tiver uma password fácil de adivinhar, que não seja robusta ou longa o suficiente pode-se tornar uma vulnerabilidade e mais suscetível a acessos não autorizados.

Para combater isso, foi implementado uma validação de password para quando o utilizador se tenta registar.

Como podemos ver na imagem abaixo foram definidos os seguintes parâmetros para a definição da password:

- Não estar vazia;
- Ter um mínimo de 8 caracteres;
- Ter um máximo de 16 caracteres;
- Ter no mínimo uma minúscula;
- Ter no mínimo uma maiúscula;
- Ter no mínimo um caracter especial (!?*.)

```
public class PasswordValidator : AbstractValidator<RegisterModel>
{
    1 referência
    public PasswordValidator()
    {
        RuleFor(p => p.Password).NotEmpty().WithMessage("Your password cannot be empty")
            .MinimumLength(8).WithMessage("Your password length must be at least 8.")
            .MaximumLength(16).WithMessage("Your password length must not exceed 16.")
            .Matches(@"[A-Z]+").WithMessage("Your password must contain at least one uppercase letter.")
            .Matches(@"[a-z]+").WithMessage("Your password must contain at least one lowercase letter.")
            .Matches(@"[0-9]+").WithMessage("Your password must contain at least one number.")
            .Matches(@"[!\?*\.\.]+").WithMessage("Your password must contain at least one (!? *.).");
    }
}
```

Figura 6 - Validação da Password

Para este pedido não seguro, só verifica se a password não está vazia e permite ao utilizador registar-se.

The screenshot displays a REST client interface for a POST request to `/api/Security/register`. The request body is a JSON object: `{ "username": "user3", "email": "string", "password": "pass" }`. The response is a 200 status code with a JSON body: `{ "id": 0, "username": "user3", "email": "" }`. The response headers include `content-type: application/json; charset=utf-8`, `date: Sat, 04 Nov 2023 16:39:56 GMT`, `server: Microsoft-IIS/10.0`, `transfer-encoding: chunked`, and `x-powered-by: ASP.NET`.

Parameters

No parameters

Request body

application/json

```
{
  "username": "user3",
  "email": "string",
  "password": "pass"
}
```

Execute Clear

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:46282/api/Security/register' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
    "username": "user3",
    "email": "string",
    "password": "pass"
  }'
```

Request URL

`http://localhost:46282/api/Security/register`

Server response

Code Details

200

Response body

```
{
  "id": 0,
  "username": "user3",
  "email": ""
}
```

Response headers

```
content-type: application/json; charset=utf-8
date: Sat, 04 Nov 2023 16:39:56 GMT
server: Microsoft-IIS/10.0
transfer-encoding: chunked
x-powered-by: ASP.NET
```

Responses

Figura 7 - Http Request inseguro da password

Na versão segura podemos confirmar que após tentativa de registo com uma password simples, retorna um 'BadRequest' e os parâmetros para criação da password.

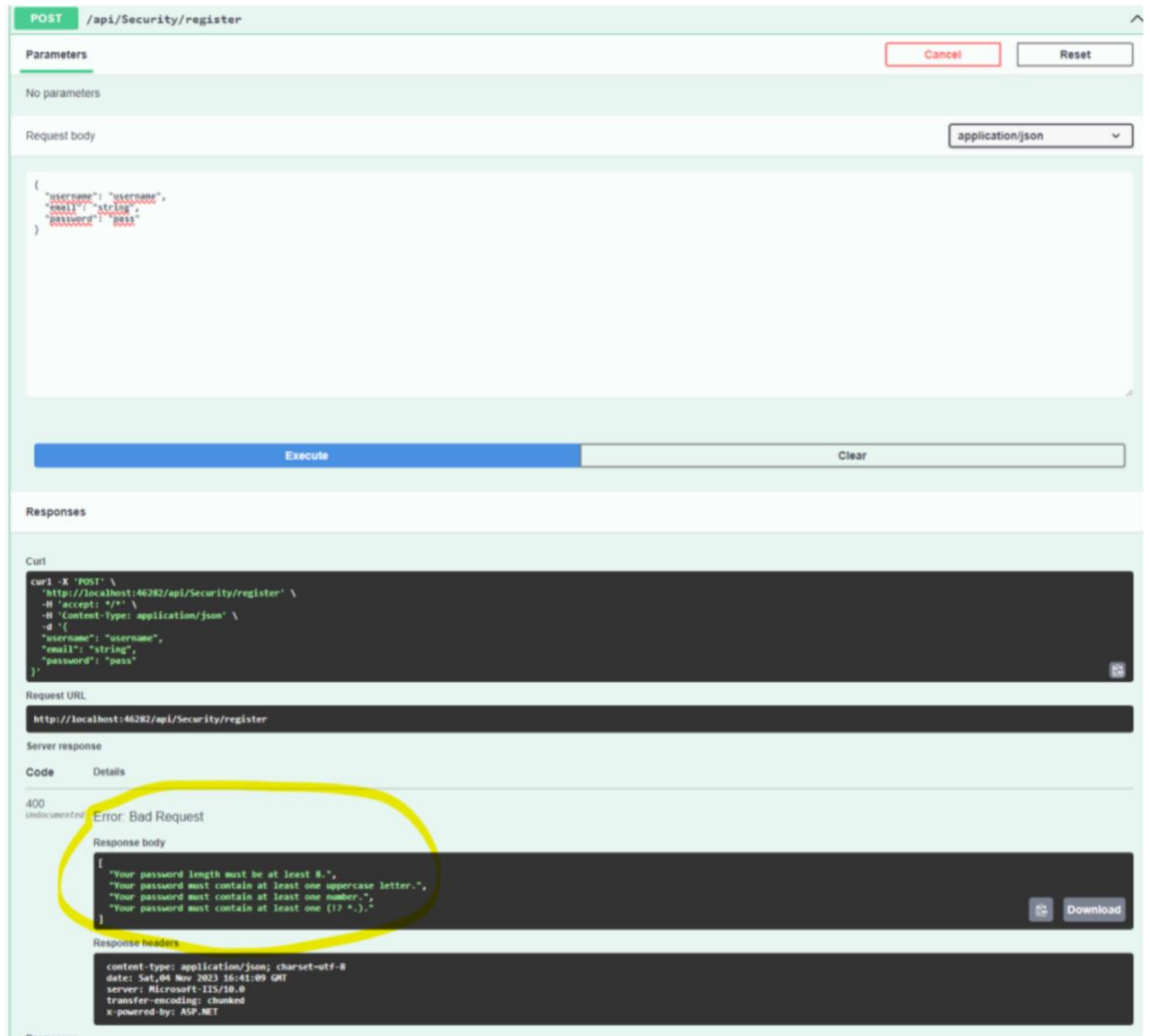


Figura 8 - Http Request versão segura da password

2.3 CWE- 200: Exposure of Sensitive Information to an Unauthorized Actor}

No que toca a esta vulnerabilidade, sabemos que consiste na partilha de informações sensíveis de forma inadequada, quer por mensagens de erro ou outras.

Como é possível observar na Figura 9, na versão insegura são mostrados parâmetros que comprometem a exposição de informação confidencial.

Podendo o atacante saber se os utilizadores estão ou não registados.

```
public async Task<UserModel> Login(LoginModel resource, CancellationToken cancellationToken = default)
{
    var user = await _context.Users
        .FirstOrDefaultAsync(x => x.Username == resource.Username, cancellationToken);

    if (user == null)
        throw new Exception("Login Fail - Unknown Username.");

    var passwordHash = PasswordHasher.ComputeHash(resource.Password, user.PasswordSalt, _pepper, _iteration);
    if (user.PasswordHash != passwordHash)
        throw new Exception("Login Fail - Password Incorrect.");

    return new UserModel(user.UserId, user.Username, string.Empty);
}
```

Figura 9 - Revela informação privilegiada

Através da correção é possível colmatar esta vulnerabilidade pois o resultado é mais vago, não permitindo ao atacante saber se o utilizador já se encontra ou não registado ou se a password é ou não a correta.

```
public async Task<UserModel> Login(LoginModel resource, CancellationToken cancellationToken = default)
{
    var user = await _context.Users
        .FirstOrDefaultAsync(x => x.Username == resource.Username, cancellationToken);

    if (user == null)
        throw new Exception("Username or password did not match.");

    var passwordHash = PasswordHasher.ComputeHash(resource.Password, user.PasswordSalt, _pepper, _iteration);
    if (user.PasswordHash != passwordHash)
        throw new Exception("Username or password did not match.");

    return new UserModel(user.UserId, user.Username, string.Empty);
}
```

Figura 10 - Informação do utilizador/password protegida

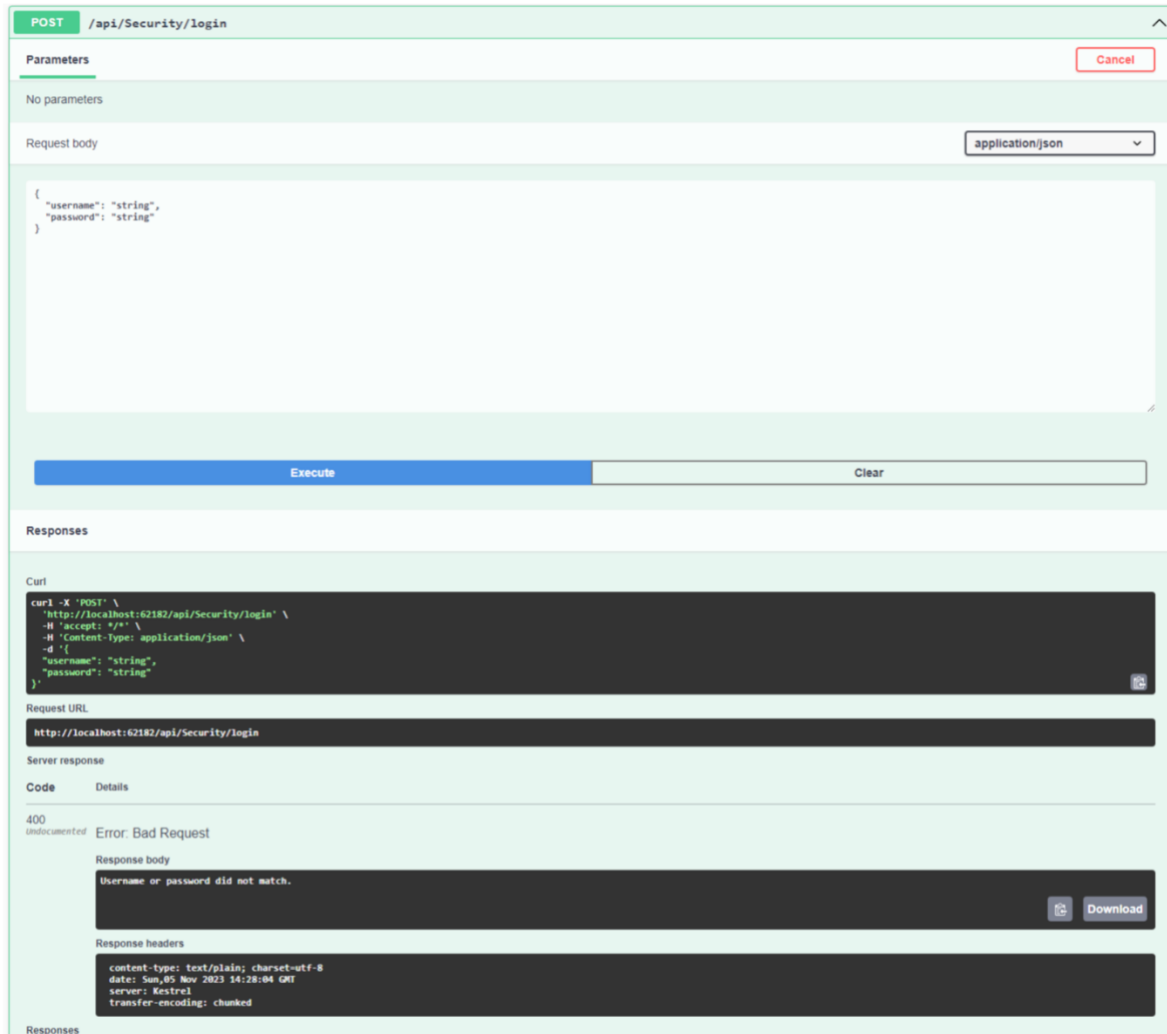


Figura 11 - Http request seguro

2.4 CWE- 20: Improper Input Validation

Para esta CWE, sabemos que consiste numa falha onde o sistema não verifica ou valida corretamente as entradas dadas pelos utilizadores. Assim sendo, um utilizador pode fornecer dados com os quais os sistema não sabe lidar levando por vezes a comportamentos inesperados e até inseguros.

```
<input type="text" name="username" placeholder="Username">
```

Figura 12 - Código não Seguro

De forma a colmatar este problema, foi alterado no lado do cliente.

```
<input type="text" name="username" placeholder="Username" required minlength="4" maxlength="25"
pattern="^(?!.*://).{4,25}$" title="Username must be between 4 and 25 characters and must not contain a URL.">
```

Figura 13 - Código corrigido

2.5 CWE- 601: URL Redirection to Untrusted Site('Open Redirect')

Esta vulnerabilidade ocorre quando é possível redirecionar os utilizadores da página para qualquer outro site. Problema este que pode ser explorado de forma a reencaminhar os utilizadores para sites maliciosos, onde podem ser vítimas de diversos tipos de ataques. De maneira a impedir este tipo de acontecimento, foi colocado no form "on submit" um script de validação. Evitando assim redireccionamentos indesejados.

```
<form id="addProductForm" onsubmit="return validateDescription()">
  <label for="productName">Name:</label>
  <input type="text" id="productName" name="name" required>

  <label for="productDescription">Description:</label>
  <textarea id="productDescription" name="description" required></textarea>

  <label for="productPrice">Price:</label>
  <input type="number" step="0.01" id="productPrice" name="price" required>

  <label for="productCategory">Category ID:</label>
  <input type="number" id="productCategory" name="categoryId" required>

  <label for="productStock">Stock:</label>
  <input type="number" id="productStock" name="stock" required>

  <label for="productImage">Image URL:</label>
  <textarea id="imagedescription" name="img" ></textarea>

  <button type="submit">Add Product</button>
</form>
```

Figura 14 - onsubmit form

```
function validateDescription() {
  var description = document.getElementById('productDescription').value;
  var urlPattern = /https?:\/\/\|www\./i;

  if (urlPattern.test(description)) {
    alert('Please remove URLs from the description.');
```

Figura 15 - Função de Validação

2.6 CWE- 79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

É uma vulnerabilidade que ocorre quando a informação fornecida pelo utilizador não é tratada corretamente, isto é, se alguém de forma maliciosa colocar código prejudicial num campo de entrada, esse mesmo código pode ser executado nos navegadores de outros utilizadores que estejam a ver a página.

Implicando uma série de consequências como por exemplo, roubo de informação de uma sessão, manipulação de conteúdo e até redireccionamento para conteúdo malicioso.

Para mitigar este problema foi implementada a classe AntiXssMiddleware, que verifica se no corpo do HTTP Request existe algum código HTML. Caso haja, retorna um 'BadRequest'.

```
namespace sio_proj1_webapi.Middleware
{
    2 referências
    public class AntiXssMiddleware
    {
        private readonly RequestDelegate _next;

        0 referências
        public AntiXssMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        0 referências
        public async Task InvokeAsync(HttpContext context)
        {
            var originalBody = context.Request.Body;
            try
            {
                var content = await ReadRequestBodyAsync(context);

                var sanitizer = new HtmlSanitizer();
                var sanitized = sanitizer.Sanitize(content);
                if (content != sanitized.Replace("&", "&"))
                {
                    await RespondWithError(context);
                }
                await _next(context);
            }
            finally
            {
                context.Request.Body = originalBody;
            }
        }

        1 referência
        private static async Task<string> ReadRequestBodyAsync(HttpContext context)
        {
            var buffer = new MemoryStream();
            await context.Request.Body.CopyToAsync(buffer);
            context.Request.Body = buffer;
            buffer.Position = 0;

            var encoding = Encoding.UTF8;

            var requestContent = await new StreamReader(buffer, encoding).ReadToEndAsync();
            context.Request.Body.Position = 0;
            return requestContent;
        }
    }
}
```

Figura 16 – Classe AntiXssMiddleware

Como é possível ver pelo código, é feita uma avaliação, se o corpo antes de ser avaliado não for igual ao corpo depois de avaliado, retorna um erro.

Na versão insegura abaixo é possível ver algum código html no corpo e o request é permitido retornando OK.

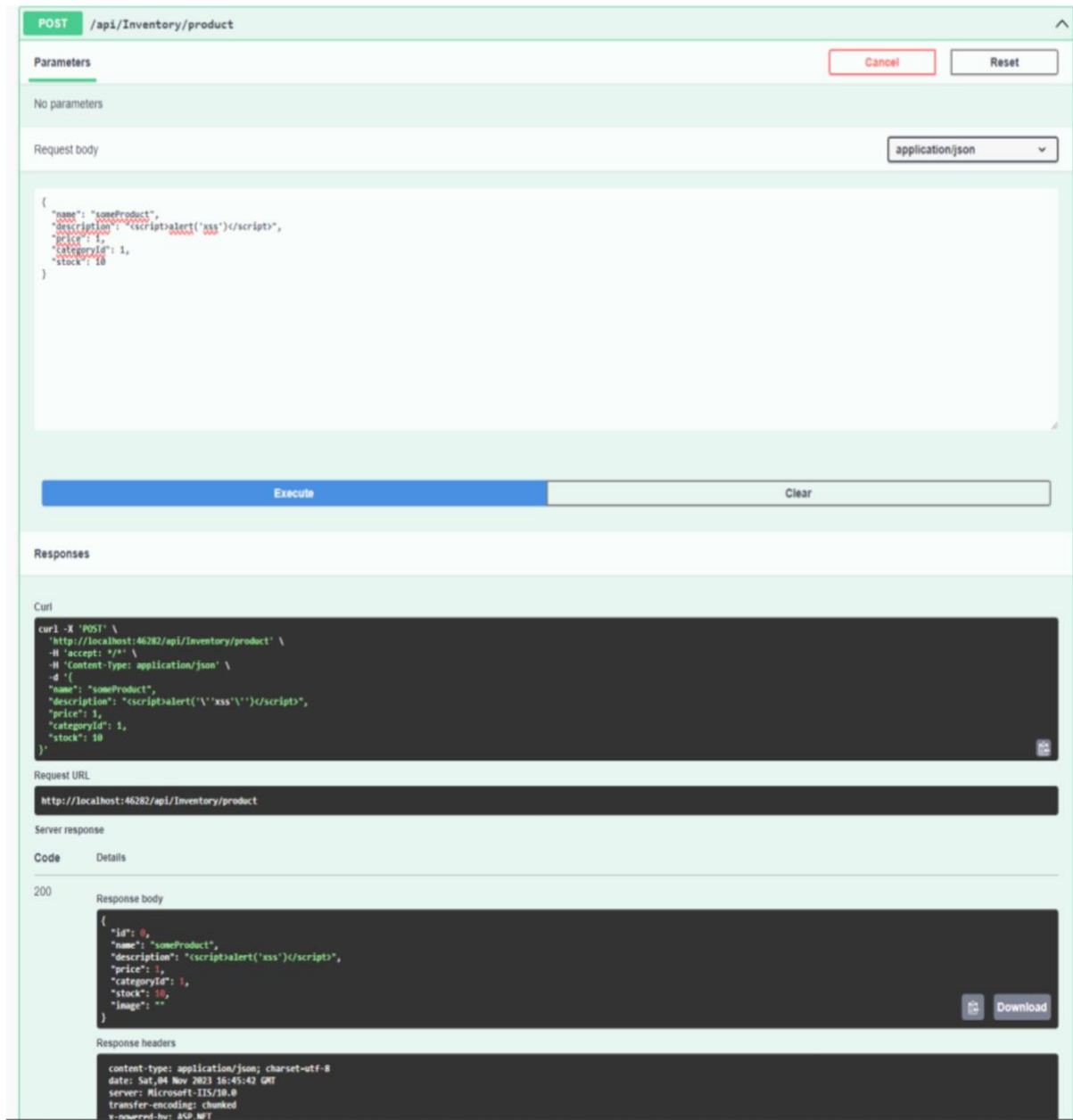
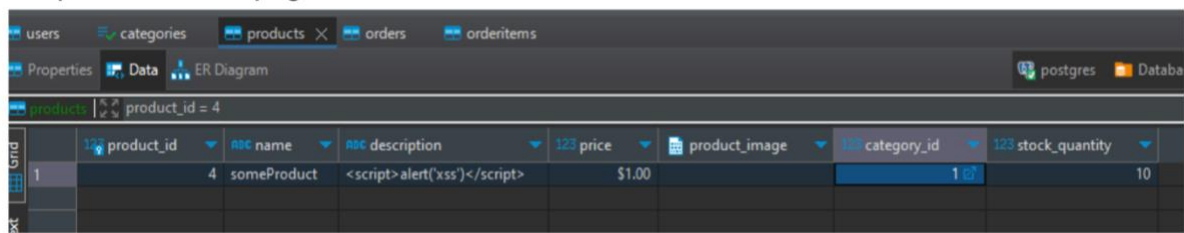


Figura 17 - Http Request inseguro

O código html foi incorretamente guardado na base de dados, permitindo que quando um utilizador tente carregar este produto, a página vai correr este código.



product_id	name	description	price	product_image	category_id	stock_quantity
4	someProduct	<script>alert('xss')</script>	\$1.00		1	10

Figura 18 - Resultado do pedido malicioso na base de dados

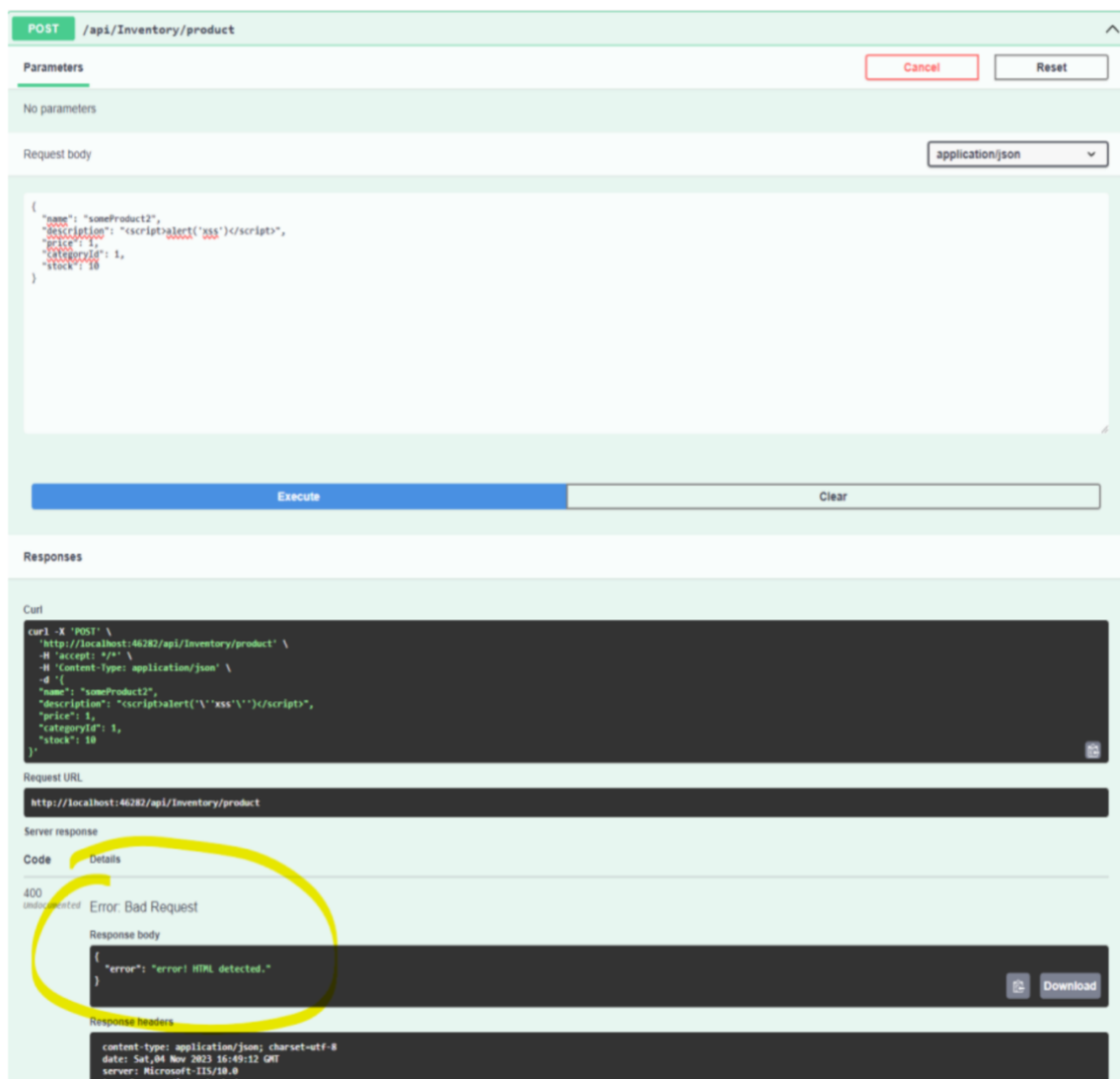


Figura 19 - Http Request após correção

Na versão corrigida, quando deteta código html, retorna um 'BadRequest'.

2.7 CWE- 89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Este tipo de vulnerabilidade surge quando os caracteres especiais não são devidamente validados ou neutralizados quando introduzidos pelos utilizadores em parâmetros de entrada que interagem com dados SQL.

Isto pode levar a graves consequências pois um atacante pode executar comandos SQL e ter acesso de leitura e manipulação de dados não autorizada.

Da maneira inicialmente implementada era possível realizar uma SQL Injection, pois as consultas não estão parametrizadas como deviam.

```
public Task<List<ProductModel>> GetProductAsync(string productId)
{
    var result = _context.Products.FromSqlRaw("select * from public.products where product_id = " + productId);

    if (result == null)
    {
        throw new Exception($"No product found for key {productId}");
    }

    return Task.FromResult(result.Select(x => new ProductModel(x.ProductId, x.Name, x.Description, x.Price, x.CategoryId.Ge
```

Figura 20 - Método Inseguro

Sendo um pedido inseguro, retorna todos os produtos como podemos ver na figura abaixo.

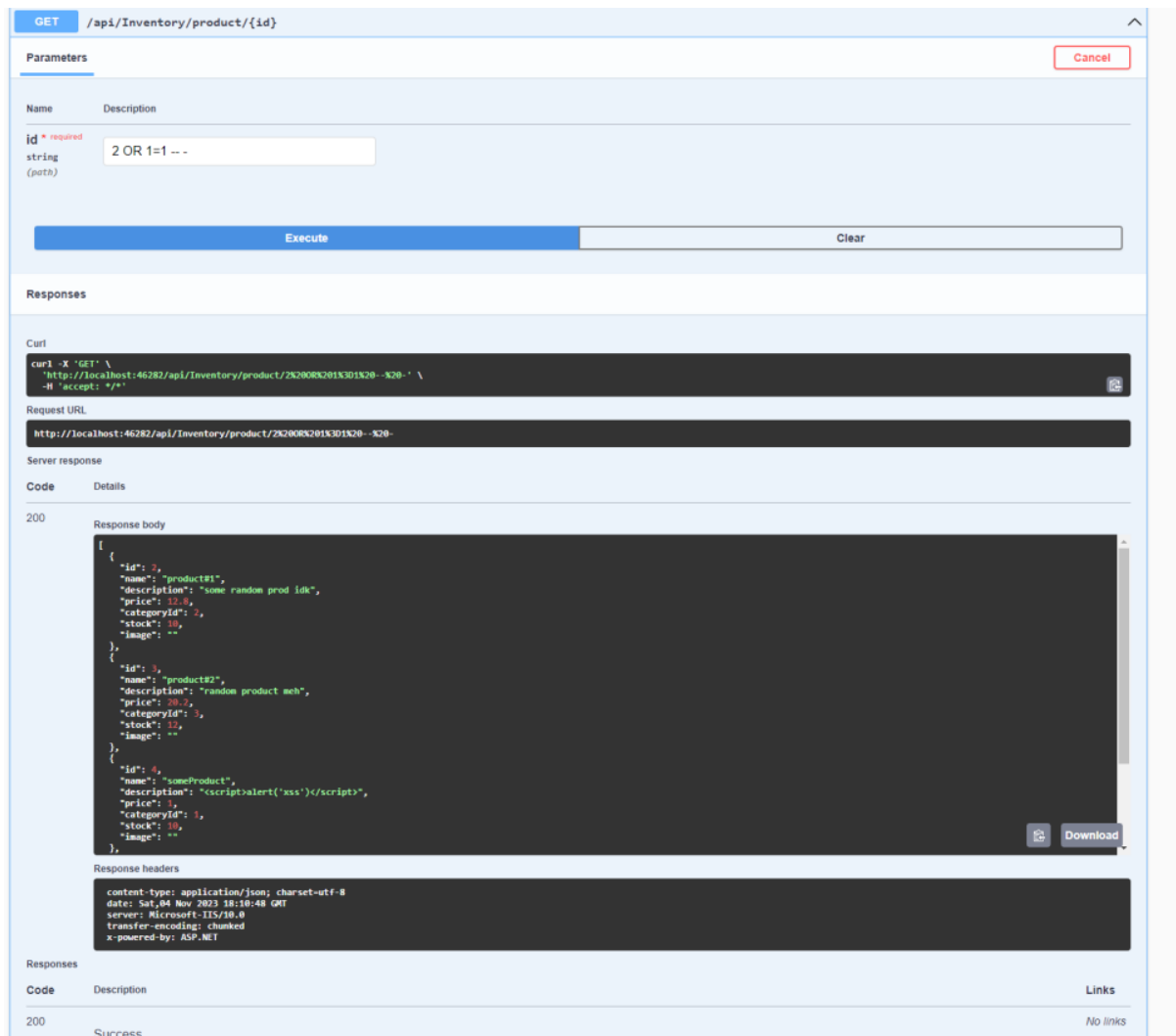


Figura 21 - Request Inseguro

Para colmatar esta situação, utilizamos um ORM que utiliza parâmetros para as consultas.

```
public Task<List<ProductModel>> GetProductAsync(string productId)
{
    var result = _context.Products.Where(x => x.ProductId.ToString() == productId);

    if (result == null)
    {
        throw new Exception($"No product found for key {productId}");
    }

    return Task.FromResult(result.Select(x => new ProductModel(x.ProductId, x.Name, x.Description, x.Price, x.CategoryId.Get
});
```

Figura 22 – Código atualizado

GET

/api/Inventory/product/{id}

⌵

Parameters

Cancel

Name	Description
id <small>* required</small>	
string	2 OR 1=1 -- -
(path)	

Execute

Clear

Responses

Curl

```
curl -X 'GET' \
'http://localhost:46282/api/Inventory/product/2K200Rk201X301X20--X20-' \
-H 'accept: */*'
```

Request URL

```
http://localhost:46282/api/Inventory/product/2K200Rk201X301X20--X20-
```

Server response

Code	Details
200	<div><div>Response body</div><div><pre>[]</pre></div><div><div>Download</div></div></div> <div><div>Response headers</div><div><pre>content-type: application/json; charset=utf-8 date: Sat, 04 Nov 2023 18:13:03 GMT server: Microsoft-IIS/10.0 transfer-encoding: chunked x-powered-by: ASP.NET</pre></div></div>

Responses

Code	Description	Links
200	Success	No links

Figura 23 - Request Seguro

3 Conclusão

O projeto tinha como propósito detetar e corrigir vulnerabilidades no desenvolvimento de uma loja online.

Uma das maiores dificuldades encontradas foi conseguir coordenar o trabalho dado que somos todos trabalhadores-estudantes com horários distintos e nos é praticamente impossível frequentar as aulas, o que tornou o processo de desenvolvimento mais difícil e lento do que gostaríamos.

Na resolução do trabalho poderiam ter sido usadas diferentes abordagens, e foram precisas diversas tentativas até atingir um resultado perto de satisfatório.

No entanto, é possível afirmar que o trabalho obedece aos objetivos propostos, e que foram superadas a maioria das adversidades encontradas.