Vulnerability Report

Archivo: UserController.cs

Code Analyzed:

```
?using AutoMapper;
using FermaOrders.API.Controllers.Response;
using FermaOrders.Application.DTOs.Auth;
using FermaOrders.Application.Interface.Auth;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using System.Net;
using System.Web.Http;
using FromBodyAttribute = Microsoft.AspNetCore.Mvc.FromBodyAttribute;
using HttpPostAttribute = Microsoft.AspNetCore.Mvc.HttpPostAttribute;
using RouteAttribute = Microsoft.AspNetCore.Mvc.RouteAttribute;

namespace FermaOrders.API.Controllers.Auth
{
    [Route("api/[controller]")]
    [ApiController]
    public class UserController : ControllerBase
    {
        private readonly IWebHostEnvironment _env;
        private readonly IUsuarioService _usuarioService;
        private readonly IMapper _mapper;
        protected RespuestaAPI _respuestaAPI;

        public UserController(IUsuarioService usuarioService, IMapper mapper,
IWebHostEnvironment env)
        {
            _usuarioService = usuarioService;
            _mapper = mapper;
            _respuestaAPI = new();
            _env = env;
        }

        [AllowAnonymous]
        [HttpPost("Registro")]
        [ProducesResponseType(StatusCodes.Status201Created)]
        [ProducesResponseType(StatusCodes.Status400BadRequest)]
        [ProducesResponseType(StatusCodes.Status500InternalServerError)]
        public async Task<IActionResult> Registro([FromForm] UsuarioRegistroDto
usuarioRegistroDto)
        {

            if (!ModelState.IsValid)
            {
                _respuestaAPI.StatusCode = HttpStatusCode.BadRequest;
                _respuestaAPI.IsSuccess = false;
                _respuestaAPI.ErrorMessages.Add("Datos de registro inválidos");
                return BadRequest(_respuestaAPI);
            }


            if (!_usuarioService.IsUniqueUser(usuarioRegistroDto.NombreUsuario))
            {
                _respuestaAPI.StatusCode = HttpStatusCode.BadRequest;
                _respuestaAPI.IsSuccess = false;
                _respuestaAPI.ErrorMessages.Add("El nombre de usuario ya existe");
                return BadRequest(_respuestaAPI);
            }

            var usuarioCreado = await _usuarioService.RegistroAsync(usuarioRegistroDto);
            if (usuarioCreado == null)
            {
                _respuestaAPI.StatusCode = HttpStatusCode.InternalServerError;
                _respuestaAPI.IsSuccess = false;
                _respuestaAPI.ErrorMessages.Add("Error en el registro de usuario");
                return StatusCode(StatusCodes.Status500InternalServerError, _respuestaAPI);
            }


            _respuestaAPI.StatusCode = HttpStatusCode.Created;
```

```
            _respuestaAPI.IsSuccess = true;
            _respuestaAPI.Result = usuarioCreado;
            return CreatedAtAction(nameof(Registro), new { id = usuarioCreado.Id },
_respuestaAPI);
        }


        [AllowAnonymous]
        [HttpPost("Login")]
        [ProducesResponseType(StatusCodes.Status201Created)]
        [ProducesResponseType(StatusCodes.Status400BadRequest)]
        [ProducesResponseType(StatusCodes.Status500InternalServerError)]
        public async Task<IActionResult> Login([FromForm] UsuarioLoginDto usuarioLoginDto)
        {
            var respuestaLogin = await _usuarioService.LoginAsync(usuarioLoginDto);

            if (respuestaLogin == null || string.IsNullOrEmpty(respuestaLogin.Token))
            {
                _respuestaAPI.StatusCode = HttpStatusCode.BadRequest;
                _respuestaAPI.IsSuccess = false;
                _respuestaAPI.ErrorMessages.Add("El nombre de Usuario o Password son
Incorrectos");
                return BadRequest(_respuestaAPI);
            }

            _respuestaAPI.StatusCode = HttpStatusCode.OK;
            _respuestaAPI.IsSuccess = true;
            ///Usamos esto para que nos devuelva el login
            _respuestaAPI.Result = respuestaLogin;
            return Ok(_respuestaAPI);
        }

        [Authorize]
        [HttpPost("logout")]
        public async Task<IActionResult> Logout()
        {
            var token = Request.Headers["Authorization"].ToString().Replace("Bearer ", "");

            await _usuarioService.LogoutAsync(token);

            return Ok(new { message = "Sesión cerrada correctamente." });
        }
    }
}
```

Analysis: ```html

Security Vulnerabilities

1. Lack of Input Validation and Sanitization

Type: Input Validation Vulnerability
Approximate Line: The `Registro` action method accepts a `UsuarioRegistroDto` from the request body. The `Login` action method accepts a `UsuarioLoginDto` from the request body.
Description: The code relies on `ModelState.IsValid` for validation, but this might not be sufficient to prevent malicious input. Specifically, there is no explicit sanitization of the input data within the DTOs or the controller, which can leave the application vulnerable to Cross-Site Scripting (XSS), SQL Injection (if the service interacts directly with a database using raw queries) and other injection attacks.
Mitigation:

1. Implement robust input validation within the `UsuarioRegistroDto` and `UsuarioLoginDto` classes. Use data annotations (e.g., `[Required]`, `[StringLength]`, `[EmailAddress]`, `[RegularExpression]`) to define acceptable input formats and lengths.
2. Sanitize the input data before processing. Encode the data to prevent XSS attacks and use parameterized queries or an ORM to avoid SQL Injection.
3. Consider using a validation library or framework for more advanced validation rules.

2. Insufficient Password Handling

Type: Authentication Vulnerability
Approximate Line: The `_usuarioService.RegistroAsync` and `_usuarioService.LoginAsync` methods (implementation not provided).
Description: Without examining the implementation of `_usuarioService`, it's impossible to verify secure password handling practices. Storing passwords in plain text or using weak hashing algorithms is a critical vulnerability.
Mitigation:

1. Never store passwords in plain text.
2. Use a strong password hashing algorithm like Argon2, bcrypt, or scrypt with a unique salt for each password.
3. Implement password complexity requirements (e.g., minimum length, special characters, uppercase/lowercase letters).
4. Consider using an existing identity management library like ASP.NET Core Identity to handle user authentication and authorization securely.

3. Logout Vulnerability

Type: Authentication Vulnerability
Approximate Line: The `Logout` action method retrieves the token from the `Authorization` header.
Description: The logout implementation relies on the client sending the token. A malicious client can send arbitrary tokens to trigger the `LogoutAsync` function. It also assumes Bearer authentication. If other methods are allowed, it could cause unintended issues. If `LogoutAsync` doesn't properly invalidate or revoke the token server-side, the token may remain valid until its natural expiration, creating a vulnerability.
Mitigation:

1. Ensure `LogoutAsync` properly invalidates or revokes the token server-side. This could involve storing invalidated tokens in a blacklist or updating user session data.
2. Consider implementing proper session management on the server.
3. Validate the format of the Authorization header and gracefully handle other authentication methods that may be allowed.

Code Quality Metrics

1. Coupling

Description: The `UserController` is tightly coupled to the `IUsuarioService`, `IMapper`, and `IWebHostEnvironment`. Changes to these dependencies could potentially require changes to the controller. Specifically the dependency on `IWebHostEnvironment` is not needed in the demonstrated methods.
Improvement: Apply the Dependency Inversion Principle. Aim to reduce direct dependencies on concrete implementations. Consider using abstractions or interfaces for `IWebHostEnvironment` where appropriate and only if it's critical, otherwise, remove it as a dependency.

2. Readability

Description: The code is generally readable, but could be improved by adding more comments to explain complex logic or decisions. The repeated code blocks for setting `_respuestaAPI` properties could be refactored.
Improvement:

- Add more comments to clarify non-obvious logic.
- Refactor the repeated `_respuestaAPI` setting code into a helper method to reduce duplication and improve readability.
- Use consistent naming conventions.

3. Duplication

Description: There is duplication of code related to setting the `_respuestaAPI` properties in multiple action methods (e.g., setting `StatusCode`, `IsSuccess`, and `ErrorMessages`).
Improvement: Create a helper method within the controller or a separate utility class to handle setting these properties. This would reduce duplication and improve maintainability.

Proposed Solution

To address the identified vulnerabilities and improve code quality, I propose the following:

1. **Implement comprehensive input validation and sanitization:** Use data annotations in DTOs, sanitize user inputs, and consider a validation library.
2. **Enforce secure password handling:** Use a strong password hashing algorithm (e.g., Argon2) with unique salts. Consider using ASP.NET Core Identity.
3. **Improve logout implementation:** Ensure token invalidation on the server-side during logout.
4. **Reduce coupling:** Review the dependencies. If `IWebHostEnvironment` is not crucial, remove it.
5. **Enhance readability:** Add comments, refactor repeated code blocks, and use consistent naming.
6. **Remove code duplication:** Create helper methods for setting `_respuestaAPI` properties.

```
```