Security & Authentication in ASP.NET Core Web API

Security is critical in modern web applications, especially for APIs that expose data over the internet. ASP.NET Core provides robust tools for securing APIs, including authentication (verifying "who" a user is) and authorization (determining "what" they can do). This material focuses on JWT authentication, ASP.NET Identity, role-based access control, and securing APIs, using a simple setup.

---

1. Overview of Security Concepts

Key Terms

- **Authentication**: Proving a user's identity (e.g., login with username/password).
- **Authorization**: Defining what an authenticated user can do (e.g., roles like "Admin").
- **JWT (JSON Web Token)**: A token-based authentication method where a signed token is sent with each request.
- **ASP.NET Identity**: A framework for managing users, passwords, and roles.
- **Role-Based Access Control (RBAC)**: Restricting access based on user roles.

Why Security Matters

- Protects sensitive data (e.g., product prices in an API).
- Prevents unauthorized access or modifications.

---

2. Setting Up the Project

Create a New Web API Project

```
dotnet new webapi -o SecureApi
cd SecureApi
```

Add Required Packages

Install NuGet packages for JWT and ASP.NET Identity:

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
dotnet add package Microsoft.AspNetCore.Identity
```

---

3. JWT Authentication

What is JWT?

- A JWT is a compact, self-contained token with three parts: **Header**, **Payload**, and **Signature** (e.g., `xxxxx.yyyyy.zzzzz`).
- Used to authenticate users by sending the token in the `Authorization` header of HTTP requests.

Step 1: Configure JWT in Program.cs

Add JWT authentication services and middleware.

```
// Program.cs
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using System.Text;

var builder = WebApplication.CreateBuilder(args);

// Add services
builder.Services.AddControllers();

// Configure JWT authentication
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = "MySecureApi", // Issuer name
            ValidAudience = "MyApiUsers", // Audience name
            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes("MySuperSecretKey12345!")) // Secret key
(min 16 chars)
        };
    });

var app = builder.Build();

// Configure middleware
app.UseHttpsRedirection();
app.UseAuthentication(); // Add this before UseAuthorization
app.UseAuthorization();
app.MapControllers();

app.Run();
```

Step 2: Create a Token Generation Endpoint

Add a simple user model and a controller to issue JWTs.

```
// Models/LoginModel.cs
namespace SecureApi.Models
{
    public class LoginModel
    {
        public string Username { get; set; }
        public string Password { get; set; }
    }
}
```

```
// Controllers/AuthController.cs
using Microsoft.AspNetCore.Mvc;
using Microsoft.IdentityModel.Tokens;
using SecureApi.Models;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;

namespace SecureApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class AuthController : ControllerBase
    {
        [HttpPost("login")]
        public IActionResult Login([FromBody] LoginModel model)
        {
            // Simple hardcoded check (for demo only)
            if (model.Username != "user" || model.Password != "password")
                return Unauthorized("Invalid credentials");

            var claims = new[]
            {
                new Claim(ClaimTypes.Name, model.Username),
                new Claim(ClaimTypes.Role, "User") // Add role for RBAC later
            };

            var key = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes("MySuperSecretKey12345!"));
            var creds = new SigningCredentials(key,
SecurityAlgorithms.HmacSha256);

            var token = new JwtSecurityToken(
                issuer: "MySecureApi",
                audience: "MyApiUsers",
                claims: claims,
                expires: DateTime.Now.AddMinutes(30),
                signingCredentials: creds);

            return Ok(new
            {
```

```
                token = new JwtSecurityTokenHandler().WriteToken(token)
            });
        }
    }
}
```

Testing JWT

1. Run the API: `dotnet run`.
2. POST to `https://localhost:5001/api/auth/login` with:

```
{
    "username": "user",
    "password": "password"
}
```

3. Response:

```
{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
}
```

4. Use the token in the `Authorization` header for other requests: `Bearer <token>`.

---

4. ASP.NET Identity (Simplified)

What is ASP.NET Identity?

- A framework for user management (e.g., registration, login, roles).
- Typically uses a database, but we'll simulate it with an in-memory list for simplicity.

Step 1: Define a User Model

```
// Models/AppUser.cs
namespace SecureApi.Models
{
    public class AppUser
    {
        public int Id { get; set; }
        public string Username { get; set; }
        public string Password { get; set; } // In reality, this would be hashed
        public string Role { get; set; }
    }
}
```

Step 2: Simulate Identity with a Service

```
// Services/UserService.cs
namespace SecureApi.Services
{
    public class UserService
    {
        private readonly List<AppUser> _users = new List<AppUser>
        {
            new AppUser { Id = 1, Username = "admin", Password = "admin123", Role
= "Admin" },
            new AppUser { Id = 2, Username = "user", Password = "user123", Role =
"User" }
        };

        public AppUser Authenticate(string username, string password)
        {
            return _users.FirstOrDefault(u => u.Username == username && u.Password
== password);
        }
    }
}
```

Step 3: Register the Service

Update `Program.cs`:

```
builder.Services.AddSingleton<UserService>();
```

Step 4: Update AuthController with Identity

```
// Controllers/AuthController.cs (updated)
using Microsoft.AspNetCore.Mvc;
using Microsoft.IdentityModel.Tokens;
using SecureApi.Models;
using SecureApi.Services;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;

namespace SecureApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class AuthController : ControllerBase
    {
        private readonly UserService _userService;
```

```
        public AuthController(UserService userService)
        {
            _userService = userService;
        }

        [HttpPost("login")]
        public IActionResult Login([FromBody] LoginModel model)
        {
            var user = _userService.Authenticate(model.Username, model.Password);
            if (user == null) return Unauthorized("Invalid credentials");

            var claims = new[]
            {
                new Claim(ClaimTypes.Name, user.Username),
                new Claim(ClaimTypes.Role, user.Role) // Include role from
"Identity"
            };

            var key = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes("MySuperSecretKey12345!"));
            var creds = new SigningCredentials(key,
SecurityAlgorithms.HmacSha256);

            var token = new JwtSecurityToken(
                issuer: "MySecureApi",
                audience: "MyApiUsers",
                claims: claims,
                expires: DateTime.Now.AddMinutes(30),
                signingCredentials: creds);

            return Ok(new
            {
                token = new JwtSecurityTokenHandler().WriteToken(token)
            });
        }
    }
}
```

---

5. Role-Based Access Control (RBAC)

What is RBAC?

- Assigns permissions based on roles (e.g., "Admin" can delete, "User" can only read).

Step 1: Secure a Product API

Add a `ProductsController` with role restrictions.

```
// Controllers/ProductsController.cs
using Microsoft.AspNetCore.Authorization;
```

```
using Microsoft.AspNetCore.Mvc;

namespace SecureApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    [Authorize] // Requires authentication for all actions
    public class ProductsController : ControllerBase
    {
        private static List<Product> _products = new List<Product>
        {
            new Product { Id = 1, Name = "Laptop", Price = 999.99m },
            new Product { Id = 2, Name = "Mouse", Price = 19.99m }
        };

        [HttpGet]
        public IActionResult GetAll()
        {
            return Ok(_products);
        }

        [HttpPost]
        [Authorize(Roles = "Admin")] // Only Admins can create
        public IActionResult Create([FromBody] Product product)
        {
            product.Id = _products.Max(p => p.Id) + 1;
            _products.Add(product);
            return CreatedAtAction(nameof(GetAll), product);
        }
    }

    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
    }
}
```

Step 2: Test RBAC

- **GET** `/api/products`: Works with any authenticated user's token (e.g., "user" or "admin").
- **POST** `/api/products`: Only works with an "Admin" token (e.g., login with "admin"/"admin123").
    - Non-admin token returns HTTP 403 (Forbidden).

---

6. Handling Secure APIs

Best Practices (Simplified)

1. **Use HTTPS**: Enforce with `app.UseHttpsRedirection();`.
2. **Validate Tokens**: Configured in JWT setup (`TokenValidationParameters`).

3. **Secure Endpoints**: Use `[Authorize]` and `[Authorize(Roles = "RoleName")]`.
4. **Sensitive Data**: Avoid logging tokens or passwords.
5. **Error Handling**: Return generic error messages (e.g., "Unauthorized") instead of detailed info.

Example: Secure API Flow

1. User logs in via `POST /api/auth/login`.
2. Receives a JWT with their role (e.g., "User" or "Admin").
3. Sends the token in the `Authorization` header: `Bearer <token>`.
4. API validates the token and enforces role-based access.

---

7. Running and Testing

Run the API

```
dotnet run
```

Test Endpoints

1. **Login as User**:
   - POST `https://localhost:5001/api/auth/login`
   - Body: `{"username": "user", "password": "user123"}`
   - Get token.
2. **Get Products**: `GET https://localhost:5001/api/products` (with token) → Success.
3. **Create Product**: `POST https://localhost:5001/api/products` (with "User" token) → 403 Forbidden.
4. **Login as Admin**: Use `{"username": "admin", "password": "admin123"}` → Token with "Admin" role.
5. **Create Product**: `POST https://localhost:5001/api/products` (with "Admin" token) → Success.

Summary Table

Topic	Description	Key Feature
JWT Authentication	Token-based auth	`AddJwtBearer`, token generation
ASP.NET Identity	User management framework	Simulated with `UserService`
Role-Based Access Control	Restrict access by role	`[Authorize(Roles = "Admin")]`
Secure APIs	Protect endpoints	HTTPS, token validation

Exercises

1. Add a "Manager" role with access to GET and POST, but not DELETE.
2. Create a logout endpoint that “invalidates” a token (hint: track tokens in memory).

3. Add a custom claim (e.g., "Department") to the JWT and restrict an endpoint based on it.