SECURE CODE REVIEW TACTICS

Performing a secure coding review on a Windows-based application developed in C# (a common language for Windows applications). The application is a simplified ASP.NET Core Web API that handles user authentication and file uploads. We'll review the code for potential vulnerabilities using manual code analysis and static code analyzers like SonarQube or Visual Studio Code Analysis.

Code: C# ASP.NET Core Application

Sample Code: User Authentication and File Upload API

C#

```
using Microsoft.AspNetCore.Mvc;
using System.IO;

namespace SecureApp.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class UserController : ControllerBase
    {
        private static readonly Dictionary<string, string> Users = new()
        {
            { "admin", "password123" } // Hardcoded credentials (for
demonstration)
        };

        [HttpPost("login")]
        public IActionResult Login([FromBody] LoginRequest request)
        {
            if (Users.ContainsKey(request.Username) &&
Users[request.Username] == request.Password)
            {
                return Ok(new { Message = "Login successful" });
            }
            return Unauthorized(new { Message = "Invalid credentials" });
        }

        [HttpPost("upload")]
        public IActionResult UploadFile(IFormFile file)
        {
            var uploadsFolder =
Path.Combine(Directory.GetCurrentDirectory(), "uploads");
            if (!Directory.Exists(uploadsFolder))
            {
                Directory.CreateDirectory(uploadsFolder);
            }
```

```
            var filePath = Path.Combine(uploadsFolder, file.FileName);
            using (var stream = new FileStream(filePath, FileMode.Create))
            {
                file.CopyTo(stream);
            }

            return Ok(new { Message = "File uploaded successfully" });
        }
    }

    public class LoginRequest
    {
        public string Username { get; set; }
        public string Password { get; set; }
    }
}
```

Step 1: Manual Code Review

1. ***Identified Vulnerabilities***
2. ***Hardcoded Credentials:***

Issue: Admin username and password are hardcoded in memory.

Risk: Exposes sensitive information to potential attackers if the source code is accessed.

Recommendation: Use environment variables or secure vault services (e.g., Azure Key Vault) to store credentials securely.

3. ***Weak Password Storage:***

Issue: Passwords are stored in plaintext.

Risk: If the application is compromised, attackers can easily obtain all user credentials.

Recommendation: Hash passwords using secure algorithms like PBKDF2, bcrypt, or Argon2.

4. ***Insecure File Upload:***

Issue: Uploaded file names are not sanitized or validated.

Risk: Allows directory traversal attacks or overwriting sensitive files on the server.

Recommendation: Use a filename sanitizer and restrict upload paths.

5. ***Exposed Exception Details:***

Issue: The application may expose detailed exception messages if errors occur.

Risk: Can reveal sensitive implementation details to attackers.

Recommendation: Use generic error messages and log detailed errors to a secure file or logging service.

6. ***No Input Validation:***

Issue: The API does not validate Username, Password, or file input.

Risk: Opens the application to injection attacks or malformed input handling issues.

Recommendation: Validate and sanitize all user inputs (e.g., use regular expressions, length checks, or whitelisting).

7. ***Unrestricted File Uploads:***

Issue: There are no restrictions on file types or size.

Risk: Can allow malicious files (e.g., executable scripts) to be uploaded.

Recommendation: Restrict uploads to safe file types and enforce file size limits.

Step 2: Static Code Analysis

Using SonarQube or Visual Studio Code Analysis:

1. Install SonarQube:

Download and set up SonarQube from SonarQube.

Integrate the project with SonarQube for automated scans.

2. Run Analysis:

Run SonarQube scan on the project folder or use Visual Studio's built-in Code Analysis tools.

Example Result (SonarQube Output):

[Critical] Hardcoded credentials in UserController.cs (line 9).

[High] No input validation for LoginRequest.Username in UserController.cs (line 16).

[Medium] Unsafe file path in UserController.cs (line 26).

Step 3: Recommendations for Secure Coding Practices

1. Secure Credentials Management:

Store sensitive information like credentials in environment variables or secure vaults.

Example (Environment Variables):

```
var adminPassword = Environment.GetEnvironmentVariable("ADMIN_PASSWORD");
```

2. Password Hashing:

Use hashing libraries like BCrypt.Net-Next to hash passwords.

Example:

```
using BCrypt.Net;
string hashedPassword = BCrypt.HashPassword("password123");
bool isValid = BCrypt.Verify("password123", hashedPassword);
```

3. Secure File Uploads:

Sanitize filenames using a library like Path.GetFileName to prevent directory traversal.

Restrict uploads to specific file types and enforce size limits.

Example:

```
var allowedExtensions = new[] { ".jpg", ".png", ".pdf" };
var fileExtension = Path.GetExtension(file.FileName);
if (!allowedExtensions.Contains(fileExtension))
{
   return BadRequest(new { Message = "Invalid file type" });
}
```

4. Input Validation:

Use FluentValidation or custom logic to validate inputs.

Example:

```
if (string.IsNullOrWhiteSpace(request.Username) || request.Username.Length > 50)
{
   return BadRequest(new { Message = "Invalid username" });
}
```

5. General Error Handling:

Use a global error-handling middleware to standardize error responses.

```
app.UseExceptionHandler(errorApp =>
{
    errorApp.Run(async context =>
    {
        context.Response.StatusCode = 500;
        await context.Response.WriteAsync("An unexpected error occurred.");
    });
});
```

Step 4: Code After Recommendations

C#

```
using Microsoft.AspNetCore.Mvc;
using System.IO;
using BCrypt.Net;

namespace SecureApp.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class UserController : ControllerBase
    {
        private static readonly Dictionary<string, string> Users = new();

        public UserController()
        {
            // Load credentials from environment variables
```

```
            Users["admin"] =
Environment.GetEnvironmentVariable("ADMIN_PASSWORD_HASH") ??
BCrypt.HashPassword("default");
        }

        [HttpPost("login")]
        public IActionResult Login([FromBody] LoginRequest request)
        {
            if (Users.ContainsKey(request.Username) &&
BCrypt.Verify(request.Password, Users[request.Username]))
            {
                return Ok(new { Message = "Login successful" });
            }
            return Unauthorized(new { Message = "Invalid credentials" });
        }

        [HttpPost("upload")]
        public IActionResult UploadFile(IFormFile file)
        {
            var allowedExtensions = new[] { ".jpg", ".png", ".pdf" };
            var fileExtension = Path.GetExtension(file.FileName);

            if (!allowedExtensions.Contains(fileExtension))
            {
                return BadRequest(new { Message = "Invalid file type" });
            }

            var uploadsFolder =
Path.Combine(Directory.GetCurrentDirectory(), "uploads");
            Directory.CreateDirectory(uploadsFolder);

            var filePath = Path.Combine(uploadsFolder,
Path.GetFileName(file.FileName));
```

```
            using var stream = new FileStream(filePath, FileMode.Create);
            file.CopyTo(stream);

            return Ok(new { Message = "File uploaded successfully" });
        }
    }

    public class LoginRequest
    {
        public string Username { get; set; }
        public string Password { get; set; }
    }
}
```

Conclusion

By combining manual code review with static code analysis tools, you can effectively identify and mitigate security vulnerabilities. For ongoing security:

Integrate static analysis tools like SonarQube or Fortify into the CI/CD pipeline.

Regularly update dependencies and frameworks.

Perform periodic security audits.