OWASP Top Ten and Mitigations in .NET

Roman Kuroptev

11.12.24

What is OWASP

- **OWASP** stands for the **Open Web Application Security Project**.
- It is a **non-profit organization** focused on improving the **security of software**.
- Provides freely available resources, tools, and **guidance for secure software development**.
- Community-driven, widely recognized as a **trusted authority** in web application security.

OWASP Top Ten

- The **OWASP Top Ten** is a **list of the most critical security risks** for web applications.
- Compiled based on data collected from industry experts, vulnerability reports, and real-world attack scenarios.
- Acts as a **best practice guide** to help developers and organizations understand the biggest threats and secure their applications.
- **Revised regularly** to adapt to evolving security threats

Why Security Matters in Software Development

Financial Impact:

- Average cost of a data breach in 2023: $4.45M USD (IBM Security).

Reputation Damage:

- Breaches erode customer trust and brand reputation (e.g., Equifax, Marriott).

Regulatory Compliance:

- GDPR fines up to 4% of global revenue for non-compliance.

Rising Threat Landscape:

- 15% increase in web application attacks in 2024 (Verizon).

Real-Life Examples:

- SolarWinds hack and Log4j vulnerability exposed critical supply chain risks.

1. Broken Access Control

- Broken Access Control occurs when users are able to **access resources or perform actions** that they **should not be authorized** for. This typically arises from improper checks on user permissions.

  Example Scenario: An attacker simply **forces browsing to target URLs**. Admin rights are required for access to the admin page.

  If an unauthenticated user can access either page, it's a flaw. If a non-admin can access the admin page, this is a flaw.

- Vulnerable Code Example:

```
0 references
public IActionResult AdminSettings()
{
    // Admin-only code here
    return View();
}
```

- Mitigation in .NET: Use ASP.NET's `Authorize` attribute to restrict access.

```
[Authorize(Roles = "Admin")]
0 references
public IActionResult AdminSettings()
{
// Admin-only code here
}
```

1. Broken Access Control

- Insecure Direct Object References (IDOR):
  - Customer ID or an Order ID in request parameters which can be guessed to access other users data
    - GET /orders/12345
- Relying Solely on Client-Side Checks or Hidden Fields
  - Bypassed by calling API directly

2. Cryptographic Failures

- Cryptographic failures happen when sensitive data is not properly encrypted, or weak algorithms are used, making data vulnerable to attackers.

  Example Scenario: Storing user passwords in plain text in the database.

- Vulnerable Code Example:

```
var query = "INSERT INTO Users (Username, Password) VALUES (@username, @password)";
using (var command = new SqlCommand(query, connection))
{
    command.Parameters.AddWithValue("@username", username);
    command.Parameters.AddWithValue("@password", password); // Vulnerable: Storing plain text password
    command.ExecuteNonQuery();
}
```

Mitigation in .NET: Always hash passwords before storing them, using a strong algorithm like BCrypt or PBKDF2.

```
string hashedPassword = BCrypt.Net.BCrypt.HashPassword(password);
```

2. Cryptographic Failures

Site lacks TLS enforcement or uses weak encryption.

- Attacker intercepts traffic on an insecure network (e.g., public Wi-Fi).
- Downgrades HTTPS to HTTP, steals session cookies.
- Replays stolen cookies to hijack authenticated sessions.

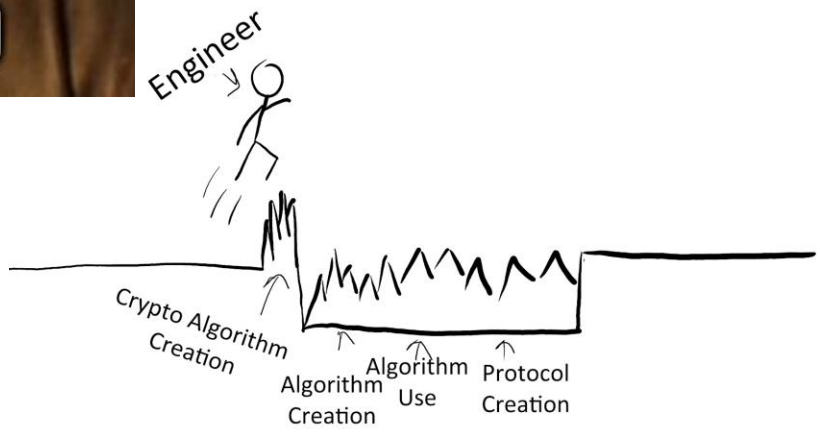
- **Vulnerable Code Example**

```
public void Configure(IApplicationBuilder app)
{
// Not enforcing HTTPS
app.UseForwardedHeaders();
app.UseStaticFiles();
}
```

Mitigation in .NET

```
public void Configure(IApplicationBuilder app)
{
    app.UseHttpsRedirection();
    app.UseForwardedHeaders();
    app.UseStaticFiles();
}
```

2. Cryptographic Failures



2. Cryptographic Failures

Use of `System.Random` for Cryptographic Operations:

- `System.Random` is not cryptographically secure

Insecure or Obsolete Algorithms

- MD5 and SHA-1 insecure
  - considered weak and collision-prone
  - SHA-256 or SHA-3 secure

Hard-Coded Keys in Source Code

- Storing encryption keys, passwords, or private keys directly in the source code

Lack of Key Rotation

3. Injection

- Injection flaws occur when untrusted data is sent to an interpreter as part of a command or query, leading to code execution.

  Example Scenario: SQL Injection through insecure queries.

```
using (var command = new SqlCommand("SELECT * FROM Users WHERE Username = '" + username + "'", connection))
{
    var reader = command.ExecuteReader();
    // Process results
}
```

- Mitigation in .NET: Always use parameterized queries.

```
var query = "SELECT * FROM Users WHERE Username = @username";
using (var command = new SqlCommand(query, connection))
{
    command.Parameters.AddWithValue("@username", username);
    // Execute the command
}
```

4. Insecure Design

- **Insecure design refers to weaknesses in the application due to design flaws that make it inherently insecure.**

  Example Scenario 1: A credential recovery workflow might include “questions and answers,” which is prohibited by NIST 800-63b, the OWASP ASVS, and the OWASP Top 10.

4. Insecure Design

```
public IActionResult RecoverAccount(string question, string answer)
{
    var user = dbContext.Users.FirstOrDefault(
        u => u.SecurityQuestion == question && u.SecurityAnswer == answer
        );
    if (user != null)
    {
        // Grant account access
        return View("AccountRecoverySuccess");
    }
    return View("AccountRecoveryFailed");
}
```

Vulnerable Code Example:

```
public IActionResult RecoverAccount(string email)
{
    // Generate a one-time passcode (OTP)
    var otp = GenerateOTP();
    SendOTPToUser(email, otp);
    return View("OTPVerification");
}
```

Mitigation in .NET: Replace the insecure recovery method with more secure mechanisms such as MFA or OTP.

4. Insecure Design

- **Insufficient Threat Modeling and Security Requirements**:
  - Missing or incomplete security controls
- **Lack of Defense-in-Depth for Sensitive Operations**:
  - Perimeter approach where security past a certain point is neglected.

4. Insecure Design

- **Scenario 1: Cinema Chain Booking Abuse**
- **Problem**: Attackers book 600 seats across all cinemas in a few requests, causing income loss.
- **Mitigation**:
  - Enforce validation and business rules.
  - Require deposits for large bookings.
  - **Scenario 2: Retail Chain Bot Scalping**
- **Problem**: Scalpers use bots to buy high-end video cards, causing bad publicity and frustrated customers.
- **Mitigation**:
  - Implement CAPTCHA or rate limiting to block bots.
  - Monitor purchasing patterns to detect and block suspicious activities.

5. Security Misconfiguration

- **Example Scenario 1**: Leaving **detailed error messages** available to the public, **exposing stack traces**.
- This occurs when security settings are not implemented correctly, or default configurations are left unchanged.

Vulnerable Code Example:

```
app.UseDeveloperExceptionPage();
```

Mitigation in .NET: Use custom error handling in production.

```
app.UseExceptionHandler("/Home/Error");
```

- Additionally, disable verbose error messages by setting `customErrors` in `web.config` or using `Environment.IsDevelopment()` checks. Ensure sample applications are removed from the production server and default credentials are changed to prevent unauthorized access.

5. Security Misconfiguration

Vulnerabilities in Sample Applications

- Mixing Dev and Production config
- Sample applications left on production servers have known flaws.
- Attackers exploit these vulnerabilities (e.g., default admin accounts).
- Potential for server compromise and full takeover.

Mitigation:

- Remove sample applications from production servers.
- Change default credentials; disable/delete unused accounts.
- Don’t mix configuration for test / dev / prod enviroments
- Regularly audit servers for outdated or unnecessary components, default config.

6. Vulnerable and Outdated Components

- **Using components that are outdated or have known vulnerabilities can put the application at risk.**
- **Example Scenario: Using an old version of Newtonsoft.Json with known vulnerabilities. Older versions, like version 6.0.1, may contain deserialization vulnerabilities that allow attackers to execute arbitrary code during the deserialization process if untrusted data is used.**
- **Vulnerable Code Example:**

```
<PackageReference Include="Newtonsoft.Json" Version="6.0.1" />
```

- **Mitigation in .NET: Regularly update NuGet packages and dependencies. Use tools like Dependabot or GitHub security alerts to keep dependencies up to date.**

7. Identification and Authentication Failures

- **Explanation: Flaws in identification or authentication can allow unauthorized access to user accounts or sensitive areas.**
- **Example Scenario: Allowing weak passwords like '123456'.**
- **Vulnerable Code Example:**
- ```
  options.Password.RequiredLength = 3;
  options.Password.RequireDigit = false;
  ```
- **Mitigation in .NET: Enforce strong password policies using ASP.NET Identity:**
- ```
  options.Password.RequireDigit = true;
  options.Password.RequiredLength = 8;
  options.Password.RequireNonAlphanumeric = true;
  ```
- Additionally, consider implementing multi-factor authentication (MFA).

7. Identification and Authentication Failures

- **Credential Stuffing**
- Attackers use lists of known passwords to test credentials.
- Applications without protections can act as password oracles, validating credentials for attackers.
- **Mitigation:**
- Implement rate limiting and account lockout mechanisms.
- Monitor for multiple failed login attempts.

```
0 references
public IActionResult Login(string username, string password)
{
    var user = dbContext.Users.FirstOrDefault(u => u.Username == username);
    if (user != null && user.Password == password)
    {
        // Successful login
        return View("Dashboard");

    // Failed login
    return View("LoginFailed");
}
```

7. Identification and Authentication Failures

- **Mitigation Code Example:**

```
0 references
public IActionResult Login(string username, string password)
{
    if (IsAccountLocked(username))
    {
        0 references
        return View("AccountLocked");
    }

    var user = dbContext.Users.FirstOrDefault(u => u.Username == username);
    if (user != null && VerifyPassword(user.Password, password))
    {
        ResetFailedAttempts(username);
        return View("Dashboard");
    }

    IncrementFailedAttempts(username);
    if (GetFailedAttempts(username) >= 5)
    {
        LockAccount(username);
    }
    return View("LoginFailed");
}
}
```

8. Software and Data Integrity Failures

Scenario: Compromised NuGet Package

- Blind trust in third-party packages introduces security risks.
- Attackers may compromise popular packages to inject malicious code.

  Real-World Case:

  TypoSquatting

  `EntityFramework` instead of `EntityFrameworkCore`

  Key Point:
- Validate the origin and authenticity of third-party packages.

8. Software and Data Integrity Failures

- **Mitigations and Secure Implementation**
  - **Use Signed Packages:** Enforce signature validation to ensure package authenticity.
  - **Restrict Package Sources:** Only use verified internal or well-known repositories.
  - **Monitor Dependencies:** Regularly check for vulnerabilities in dependencies.
  - **Secure CI/CD Pipeline:** Implement automated dependency checks during builds.
  - **Dependency Pinning:** Pin dependencies to specific versions to avoid unexpected updates.
  - **Key Point:** Proper controls around package management are crucial for software and data integrity. Validate, restrict, and audit dependencies consistently.

9. Security Logging and Monitoring Failures

- **Importance: Detecting and responding to incidents effectively.**
- **Common Issue: Insufficient logging or monitoring can lead to delayed detection of attacks like brute-force.**

```
public async Task<IActionResult> Login(UserLoginModel model)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }

    var user = await _userManager.FindByNameAsync(model.Username);
    if (user == null || !await _userManager.CheckPasswordAsync(user, model.Password))
    {
        // No logging for failed login attempt
        ModelState.AddModelError(string.Empty, "Invalid login attempt.");
        return View(model);
    }

    // No logging for successful login
    await _signInManager.SignInAsync(user, isPersistent: false);
    return RedirectToAction("Index", "Home");
}
```

```
public async Task<IActionResult> Login(UserLoginModel model)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }

    var user = await _userManager.FindByNameAsync(model.Username);
    if (user == null || !await _userManager.CheckPasswordAsync(user, model.Password))
    {
        // Logging the failed login attempt
        _logger.LogWarning("Failed login attempt for user: {Username} from IP: {IP}",
                           model.Username,
                           HttpContext.Connection.RemoteIpAddress);
        ModelState.AddModelError(string.Empty, "Invalid login attempt.");
        return View(model);
    }

    // Logging the successful login attempt
    _logger.LogInformation("User {Username} successfully logged in from IP: {IP}",
                           model.Username,
                           HttpContext.Connection.RemoteIpAddress);
    await _signInManager.SignInAsync(user, isPersistent: false);
    return RedirectToAction("Index", "Home");
}
```

10. Server-Side Request Forgery (SSRF)

- **What is SSRF?**
- **Server-Side Request Forgery (SSRF) occurs when an attacker tricks a server into making a request to an unintended destination.**
- **Attackers use SSRF to gain unauthorized access to internal services or exfiltrate sensitive data by exploiting trusted server privileges.**

```
[HttpGet("fetch")]
public async Task<IActionResult> FetchUrl([FromQuery] string targetUrl)
{
    var response = await _httpClient.GetAsync(targetUrl);
    var content = await response.Content.ReadAsStringAsync();
    return Content(content);
}
```

- **Mitigation Strategies**
- **Input Validation**
  - Allow only trusted URLs/domains (use an allowlist).
  - Validate and sanitize URL input.
- **Network Restrictions**
  - Block outbound requests to internal IP ranges that should not be accessible.
- **Safer Implementation Example**

```
[HttpGet("fetch")]
public async Task<IActionResult> FetchUrl([FromQuery] string targetUrl)
{
    if (!IsValidDomain(targetUrl))
    {
        return BadRequest("Invalid URL domain.");
    }
    var response = await _httpClient.GetAsync(targetUrl);
    return Content(await response.Content.ReadAsStringAsync());
}
private bool IsValidDomain(string url)
{
    Uri uriResult;
    return Uri.TryCreate(url, UriKind.Absolute, out uriResult) && allowedDomains.Contains(uriResult.Host);
}
```

Honorable mentions:

- https://owasp.org/www-project-top-10-for-large-language-model-applications/