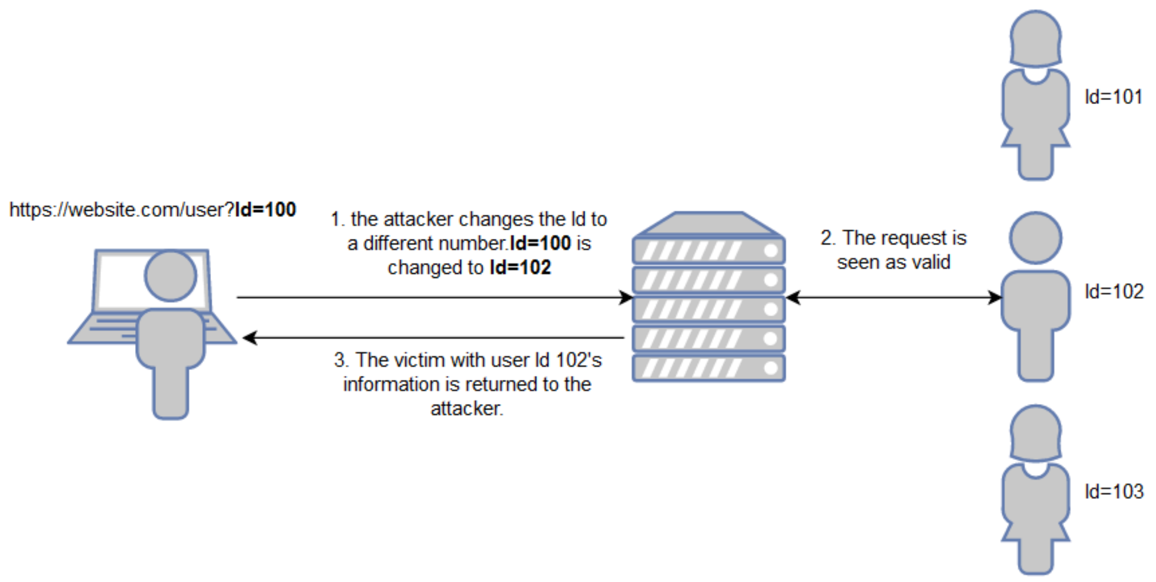
Comprehensive Analysis of IDOR Vulnerabilities in C# Applications: Identification, Exploitation, and Remediation


https://website.com/user?**Id=100**
1. the attacker changes the Id to a different number.**Id=100** is changed to **Id=102**
2. The request is seen as valid
3. The victim with user Id 102's information is returned to the attacker.
Id=101
Id=102
Id=103

Okan YILDIZ

Senior Security Engineer / Software Developer

31.07.2024

Introduction

Insecure Direct Object References (IDOR) remain one of the most prevalent and dangerous security vulnerabilities in modern web applications. Ranked consistently in the OWASP Top 10 (under the broader category of Broken Access Control), IDOR vulnerabilities continue to plague applications across all technology stacks, including those built with C# and the .NET framework.

This comprehensive technical article explores IDOR vulnerabilities specifically in the context of C# applications, providing detailed code examples of vulnerable implementations, exploitation techniques, and secure remediation strategies. We will examine how these vulnerabilities manifest across different .NET architectures including ASP.NET MVC, ASP.NET Core, Web API, and various data access layers.

IDOR vulnerabilities arise when an application exposes references to internal implementation objects, such as database primary keys, file names, or directories, without sufficient authorization checks. This allows attackers to manipulate these references to access unauthorized data or functionality. In C# applications, these vulnerabilities can be particularly subtle due to the framework's powerful features and abstractions.

By the end of this article, you will have a deep technical understanding of IDOR vulnerabilities in C# applications, how they can be exploited, and most importantly, how to implement robust defenses against them through proper authorization controls and secure coding practices.

Understanding IDOR Vulnerabilities

The Core Problem

At its heart, an IDOR vulnerability occurs when an application uses client-supplied input to directly access objects without verifying the user's authorization to access those specific resources. This fundamental security flaw bypasses the application's intended access controls, allowing attackers to manipulate references and access unauthorized resources.

In technical terms, an IDOR vulnerability exists when these three conditions are met:

1. The application uses client-provided identifiers to access server-side objects or resources directly
2. The application fails to verify the user's access rights to the requested resource

3. The user can discover or guess valid identifiers for resources they should not access

IDOR in the Security Landscape

IDOR vulnerabilities are part of the broader "Broken Access Control" category, which has consistently ranked at the top of the OWASP Top 10 Web Application Security Risks. According to the OWASP 2021 list, Broken Access Control moved from the fifth position to the number one spot, indicating the growing prevalence and impact of these vulnerabilities.

The impact of IDOR vulnerabilities can be severe, potentially leading to:

- Unauthorized access to sensitive data
- Privilege escalation
- Account takeover
- Data tampering
- Information disclosure
- Regulatory compliance violations

IDOR vs. Other Access Control Issues

It's important to distinguish IDOR from other access control vulnerabilities:

- IDOR focuses specifically on the manipulation of direct references to objects
- Missing Function-Level Authorization occurs when sensitive functionality isn't protected
- Forced Browsing involves accessing resources by URL manipulation
- Insecure Direct Object Reference focuses on the direct exposure of internal implementation objects

While these issues are related, this article will focus specifically on IDOR vulnerabilities and their manifestation in C# applications.

IDOR in C# Web Applications

C# and the .NET framework provide several architectural models for web applications, each with its own potential IDOR vulnerabilities:

ASP.NET MVC and IDOR

ASP.NET MVC applications often expose controller actions that accept resource identifiers as parameters. Without proper authorization checks, these endpoints can be vulnerable to IDOR attacks. For example:

```
// Vulnerable controller action
[HttpGet]
public ActionResult ViewDocument(int documentId)
{
    var document = _documentRepository.GetById(documentId);
    return View(document);
}
```

This code retrieves a document by ID without verifying if the current user is authorized to access it.

ASP.NET Core and Authorization

ASP.NET Core introduced a more robust authorization framework, but it must be implemented correctly. IDOR vulnerabilities can still occur if authorization policies are incorrectly configured or circumvented.

Web API and Resource Access

RESTful APIs built with ASP.NET Web API or ASP.NET Core are particularly susceptible to IDOR vulnerabilities due to their resource-centric nature. API endpoints that accept resource identifiers need proper authorization checks to prevent unauthorized access.

Entity Framework and Data Access

Many C# applications use Entity Framework for data access. Without appropriate filtering of query results based on user context, IDOR vulnerabilities can expose data across tenant boundaries or security contexts.

Architectural Considerations

Modern C# applications often use tiered architectures with separate API layers, service layers, and data access layers. IDOR vulnerabilities can occur at any of these layers, and a comprehensive security approach must address access control at each level.

Types of IDOR Vulnerabilities

Simple ID Manipulation

The most basic form of IDOR vulnerability involves direct manipulation of resource identifiers, typically numeric IDs or simple strings.

Vulnerable C# Example:

```
[HttpGet("api/orders/{orderId}")]
public IActionResult GetOrder(int orderId)
{
    var order = _context.Orders.FirstOrDefault(o => o.Id == orderId);
    if (order == null)
        return NotFound();



return Ok(<span class="hljs-name">order</span>)<span class="hljs-comment">;</span>






}
```

This endpoint retrieves an order by its ID without verifying if the current user has permission to access it. An attacker could simply change the `orderId` parameter to access other users' orders.

Exploitation:

1. Attacker logs in and accesses their own order: `GET /api/orders/1234`
2. Attacker observes the numeric ID pattern
3. Attacker changes the ID: `GET /api/orders/1235`
4. If vulnerable, the application returns another user's order details

Predictable Resource Locations

Some applications use predictable patterns for resource identifiers or locations, making it easy for attackers to guess valid references.

Vulnerable C# Example:

```
[HttpGet("download")]
public IActionResult DownloadFile(string fileName)
{
    var filePath = Path.Combine(_environment.WebRootPath, "userfiles",
fileName);
    if (!System.IO.File.Exists(filePath))
        return NotFound();



return PhysicalFile(<span class="hljs-name">filePath</span>, <span
class="hljs-string">"application/octet-stream"</span>, fileName)<span
class="hljs-comment">;</span>
}
```

This code directly uses a user-supplied filename to access files on the server without any access control checks.

Exploitation:

1. Attacker notices downloads use URLs like `/download?fileName=invoice_user123.pdf`
2. Attacker tries `/download?fileName=invoice_user456.pdf`
3. If vulnerable, the server returns another user's file

Insecure UUID/GUID Implementations

Many developers believe that using GUIDs (Globally Unique Identifiers) instead of sequential IDs prevents IDOR vulnerabilities. However, if these GUIDs are exposed to users and no authorization checks exist, IDOR vulnerabilities remain.

Vulnerable C# Example:

```
[HttpGet("api/documents/{documentId}")]
public IActionResult GetDocument(Guid documentId)
{
    var document = _context.Documents.FirstOrDefault(d => d.Id ==
documentId);
    if (document == null)
        return NotFound();


return Ok(<span class="hljs-name">document</span>)<span class="hljs-comment">;</span>

}
```

Despite using GUIDs, this code still fails to check if the user is authorized to access the requested document.

Exploitation:

1. Attacker accesses their own document and observes the GUID in the URL
2. Through an information leak elsewhere in the application, the attacker learns GUIDs of other documents
3. Attacker replaces their document's GUID with another document's GUID
4. If vulnerable, the application returns the unauthorized document

Horizontal vs. Vertical Privilege Escalation

IDOR vulnerabilities can enable two types of privilege escalation:

1. Horizontal Privilege Escalation: Accessing resources belonging to other users at the same privilege level
2. Vertical Privilege Escalation: Accessing resources requiring a higher privilege level

Vulnerable C# Example (Horizontal):

```
[Authorize]
[HttpGet("api/users/{userId}/profile")]
public IActionResult GetUserProfile(int userId)
{
    var profile = _context.UserProfiles.FirstOrDefault(p => p.UserId ==
userId);
    if (profile == null)
        return NotFound();



return Ok(<span class="hljs-name">profile</span>)<span class="hljs-comment">;</span>
}
```

This endpoint allows any authenticated user to access any user's profile by changing the userId parameter.

Vulnerable C# Example (Vertical):

```
[Authorize]
[HttpGet("api/reports/{reportId}")]
public IActionResult GetReport(int reportId)
{
    var report = _context.Reports.FirstOrDefault(r => r.Id == reportId);
   if (report == null)
       return NotFound();

// Missing<span class="hljs-built_in"> check </span>for administrative access to certain report types

 return Ok(report);
}
```

This endpoint fails to check if certain report types require administrative privileges.

Indirect References Through Non-Primary Keys

IDOR vulnerabilities can also occur when applications use non-primary key fields as reference points.

Vulnerable C# Example:

```
[HttpGet("api/invoices/by-number/{invoiceNumber}")]
public IActionResult GetInvoiceByNumber(string invoiceNumber)
{
    var invoice = _context.Invoices.FirstOrDefault(i => i.InvoiceNumber ==
invoiceNumber);
   if (invoice == null)
       return NotFound();
// Missing<span class="hljs-built_in"> check </span>if the current user owns this invoice
  return Ok(invoice);
}
```

This endpoint uses invoice numbers instead of IDs, but still fails to verify ownership.

Vulnerable Code Patterns in C

ASP.NET MVC Controllers With IDOR Issues

ASP.NET MVC applications frequently expose IDOR vulnerabilities through controller actions that retrieve data based on user input without proper authorization.

Vulnerable Profile Management:

```
public class ProfileController : Controller
{
   private readonly ApplicationDbContext _context;

  public ProfileController(ApplicationDbContext context)
    {
        _context = context;
    }



// VULNERABLE: No authorization<span class="hljs-built_in"> check
   [HttpGet]
   public ActionResult Details(int userId)
    {
        var userProfile = _context.UserProfiles
            .Include(p => p.PersonalInformation)
            .FirstOrDefault(p => p.UserId == userId);



  <span class="hljs-built_in"> if </span>(userProfile == null)
       <span class="hljs-built_in"> return </span>NotFound();

   <span class="hljs-built_in"> return </span>View(userProfile);
}

// VULNERABLE: No authorization<span class="hljs-built_in"> check </span>on edit
[HttpPost]
[ValidateAntiForgeryToken]
  public ActionResult Edit(int userId, UserProfileViewModel model)
    {
        var profile = _context.UserProfiles.FirstOrDefault(p => p.UserId ==
userId);
       if (profile == null)
           return NotFound();



  <span class="hljs-built_in"> if </span>(ModelState.IsValid)
    {
        profile.FirstName = model.FirstName;
        profile.<span class="hljs-class">LastName = model.LastName;</span>
        profile.PhoneNumber = model.PhoneNumber;
        profile.DateOfBirth = model.DateOfBirth;

        _context.SaveChanges();
       <span class="hljs-built_in"> return </span>RedirectToAction(<span
class="hljs-string">"Details"</span>,<span class="hljs-built_in"> new </span>{ userId });
    }

   <span class="hljs-built_in"> return </span>View(model);
}



}
```

This controller allows any authenticated user to view and edit any user's profile by manipulating the `userId` parameter.

Exploitation Methods:

1. An attacker can increment or decrement the userId parameter to access other users' profiles.
2. Form submissions can be intercepted and modified to change the targeted userId.

ASP.NET Core API Endpoints With IDOR Issues

ASP.NET Core Web API endpoints are particularly prone to IDOR vulnerabilities due to their resource-centric design.

Vulnerable Order Management API:

```
[Route("api/[controller]")]
[ApiController]
[Authorize]
public class OrdersController : ControllerBase
{
    private readonly OrderDbContext _context;
public OrdersController(<span class="hljs-name">OrderDbContext</span> context)
{
    _context = context<span class="hljs-comment">;</span>
}

// VULNERABLE: No authorization check on order access
[<span class="hljs-name">HttpGet</span>(<span class="hljs-string">"{id}"</span>)]
public async Task&lt;ActionResult&lt;Order&gt;&gt; GetOrder(<span
class="hljs-name">int</span> id)
{
    var order = await _context.Orders
        .Include(<span class="hljs-name">o</span> =&gt; o.Items)
        .Include(<span class="hljs-name">o</span> =&gt; o.PaymentDetails)
        .FirstOrDefaultAsync(<span class="hljs-name">o</span> =&gt; o.Id == id)<span
class="hljs-comment">;</span>

    if (<span class="hljs-name">order</span> == null)
        return NotFound()<span class="hljs-comment">;</span>

    return order<span class="hljs-comment">;</span>
}

// VULNERABLE: No authorization check on order cancellation
[<span class="hljs-name">HttpPost</span>(<span class="hljs-string">"{id}/cancel"</span>)]
public async Task&lt;IActionResult&gt; CancelOrder(<span class="hljs-name">int</span> id)
{
    var order = await _context.Orders.FindAsync(<span class="hljs-name">id</span>)<span
class="hljs-comment">;</span>
    if (<span class="hljs-name">order</span> == null)
        return NotFound()<span class="hljs-comment">;</span>
    order.Status = OrderStatus.Cancelled<span class="hljs-comment">;</span>
    order.CancellationDate = DateTime.UtcNow<span class="hljs-comment">;</span>

    await _context.SaveChangesAsync()<span class="hljs-comment">;</span>
    return NoContent()<span class="hljs-comment">;</span>
}
}
```

This API allows any authenticated user to access, view, and cancel any order by manipulating the order ID.

Entity Framework Access Control Issues

Entity Framework makes data access easy in C# applications, but its abstraction can lead to IDOR vulnerabilities when queries don't include appropriate user context filters.

Vulnerable Repository Implementation:

```
public class DocumentRepository : IDocumentRepository
{
    private readonly ApplicationDbContext _context;
```

```
<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-title">DocumentRepository</span>(<span class="hljs-params">ApplicationDbContext
context</span>)
</span>{
    _context = context;
}

<span class="hljs-comment">// VULNERABLE: No filtering based on user context</span>
<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-keyword">async</span> Task&lt;Document&gt; <span
class="hljs-title">GetDocumentByIdAsync</span>(<span class="hljs-params"><span
class="hljs-keyword">int</span> documentId</span>)
</span>{
    <span class="hljs-keyword">return</span> <span class="hljs-keyword">await</span>
_context.Documents
        .Include(d =&gt; d.Metadata)
        .FirstOrDefaultAsync(d =&gt; d.Id == documentId);
}

<span class="hljs-comment">// VULNERABLE: No filtering based on user context</span>
<span class="hljs-keyword">public</span> <span class="hljs-keyword">async</span>
Task&lt;List&lt;Document&gt;&gt; GetDocumentsByFolderIdAsync(<span
class="hljs-keyword">int</span> folderId)
{
    <span class="hljs-keyword">return</span> <span class="hljs-keyword">await</span>
_context.Documents
        .Where(d =&gt; d.FolderId == folderId)
        .ToListAsync();
}
}
```

This repository class retrieves documents without filtering based on the current user's context, allowing access to any document if the ID is known.

File Access IDOR Vulnerabilities

File operations in C# applications can be vulnerable to IDOR if they rely on user-supplied input without proper access control.

Vulnerable File Manager:

```
[Authorize]
public class FileController : Controller
{
    private readonly string _uploadsFolder;
```

```
public FileController(<span class="hljs-name">IWebHostEnvironment</span> env)
{
    _uploadsFolder = Path.Combine(<span class="hljs-name">env.WebRootPath</span>, <span
class="hljs-string">"uploads"</span>)<span class="hljs-comment">;</span>
}

// VULNERABLE: Direct file access without authorization
[<span class="hljs-name">HttpGet</span>(<span class="hljs-string">"download"</span>)]
public IActionResult DownloadFile(<span class="hljs-name"><span
class="hljs-builtin-name">string</span></span> fileName)
{
    var filePath = Path.Combine(<span class="hljs-name">_uploadsFolder</span>,
fileName)<span class="hljs-comment">;</span>

    if (<span class="hljs-name">!System.IO.File.Exists</span>(<span
class="hljs-name">filePath</span>))
        return NotFound()<span class="hljs-comment">;</span>

    var contentType = <span class="hljs-string">"application/octet-stream"</span><span
class="hljs-comment">;</span>
    return PhysicalFile(<span class="hljs-name">filePath</span>, contentType,
Path.GetFileName(<span class="hljs-name">filePath</span>))<span
class="hljs-comment">;</span>
}

// VULNERABLE: File deletion without authorization check
[<span class="hljs-name">HttpPost</span>(<span class="hljs-string">"delete"</span>)]
public IActionResult DeleteFile(<span class="hljs-name"><span
class="hljs-builtin-name">string</span></span> fileName)
{
    var filePath = Path.Combine(<span class="hljs-name">_uploadsFolder</span>,
fileName)<span class="hljs-comment">;</span>

    if (<span class="hljs-name">!System.IO.File.Exists</span>(<span
class="hljs-name">filePath</span>))
        return NotFound()<span class="hljs-comment">;</span>
    System.IO.File.Delete(<span class="hljs-name">filePath</span>)<span
class="hljs-comment">;</span>
    return Ok()<span class="hljs-comment">;</span>
}
}
```

This controller allows any authenticated user to download or delete any file by knowing or guessing the filename.

IDOR in SignalR Communications

SignalR, a real-time communications library for ASP.NET, can also be vulnerable to IDOR issues.

Vulnerable Chat Hub:

```
[Authorize]
public class ChatHub : Hub
{
    private readonly IChatRepository _chatRepository;
```

```
<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-title">ChatHub</span>(<span class="hljs-params">IChatRepository
chatRepository</span>)
</span>{
    _chatRepository = chatRepository;
}

<span class="hljs-comment">// VULNERABLE: No authorization check for accessing chat
history</span>
<span class="hljs-keyword">public</span> <span class="hljs-keyword">async</span>
Task&lt;List&lt;ChatMessage&gt;&gt; GetChatHistory(<span class="hljs-keyword">int</span>
chatRoomId)
{
    <span class="hljs-keyword">return</span> <span class="hljs-keyword">await</span>
_chatRepository.GetChatHistoryAsync(chatRoomId);
}

<span class="hljs-comment">// VULNERABLE: No verification if user is in the room</span>
<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-keyword">async</span> Task <span
class="hljs-title">SendMessageToRoom</span>(<span class="hljs-params"><span
class="hljs-keyword">int</span> chatRoomId, <span class="hljs-keyword">string</span>
message</span>)
</span>{
    <span class="hljs-keyword">var</span> username = Context.User.Identity.Name;

    <span class="hljs-keyword">await</span> _chatRepository.SaveMessageAsync(chatRoomId,
username, message);
    <span class="hljs-keyword">await</span> Clients.Group(<span
class="hljs-string">$"room_<span
class="hljs-subst">{chatRoomId}</span>"</span>).SendAsync(<span
class="hljs-string">"ReceiveMessage"</span>, username, message);
}
}
```

This SignalR hub allows any user to access chat history and send messages to any room by providing the room ID.

Exploitation Techniques

Basic Parameter Tampering

The simplest IDOR exploitation technique involves modifying request parameters to access unauthorized resources.

Exploitation Process:

1. Identify Parameter Patterns: Examine URLs, form fields, and JSON payloads for resource identifiers.
2. Manual Parameter Modification: Change identified parameters to target different resources.
3. Sequential Testing: For numeric IDs, try incrementing or decrementing values.
4. Pattern-Based Guessing: Look for patterns in IDs and attempt to predict valid values.
5. Automation: Use scripts to test multiple parameter values systematically.

C# Context Exploitation:

In ASP.NET applications, parameter tampering can target:

- Route parameters (`/users/{id}`)
- Query string parameters (`?userId=123`)
- Form fields (`<input type="hidden" name="orderId" value="456">`)
- JSON request bodies (`{"documentId": 789}`)

Forced Browsing Attack Methodology

Forced browsing involves directly accessing URLs with predictable resource identifiers.

Exploitation Steps:

1. URL Pattern Analysis: Identify how the application structures URLs for resource access.
2. Directory and File Enumeration: Attempt to access resources by guessing file and directory names.
3. Predictable Resource Naming: Test access to resources with predictable names (e.g., `invoice_2023_1234.pdf`).
4. Bypassing Client-Side Controls: Directly access URLs that might be hidden by client-side restrictions.

Tools for Forced Browsing:

- Burp Suite's Intruder: For systematic testing of URL patterns
- OWASP ZAP's Forced Browse: For directory and file discovery
- Custom scripts: For testing application-specific patterns

API Endpoint Manipulation

Modern C# applications often use RESTful APIs, which can be vulnerable to IDOR through endpoint manipulation.

Exploitation Process:

1. API Documentation Review: Look for endpoints that accept resource identifiers.
2. HTTP Method Testing: Test different HTTP methods (GET, POST, PUT, DELETE) on identified endpoints.
3. Parameter Manipulation: Modify resource identifiers in API requests.
4. Request Header Tampering: Modify headers that might influence resource selection or access control.

Example C# API Exploitation:

```
// Original authorized request
GET /api/reports/user/123
```

```
// Manipulated request to access another user's reports
GET /api/reports/user/456
```

HTTP Request Method Switching

Some C# applications implement different authorization checks for different HTTP methods on the same endpoint.

Exploitation Steps:

1. Method Identification: Identify endpoints that respond to multiple HTTP methods.
2. Method Switching: Change the HTTP method (e.g., from GET to POST) while keeping the same resource identifier.
3. Content-Type Manipulation: Modify the Content-Type header to bypass method-specific filtering.

C# MVC Example:

An application might have different authorization checks in different action methods:

```
// Properly secured GET method
[Authorize(Roles = "Administrator")]
[HttpGet("api/users/{id}")]
public IActionResult GetUser(int id) { ... }


// Vulnerable POST method
[Authorize] // Missing the role requirement
[HttpPost("api/users/{id}")]
public IActionResult UpdateUser(int id, UserModel model) { ... }
```

An attacker might bypass the stricter GET authorization by using the POST method.

Race Conditions in IDOR Contexts

Race conditions can exacerbate IDOR vulnerabilities by exploiting timing windows in authorization checks.

Exploitation Process:

1. Identify Multi-Step Processes: Look for operations that involve multiple requests.
2. Time-of-Check to Time-of-Use (TOCTOU): Identify gaps between when authorization is checked and when the resource is accessed.
3. Parallel Request Execution: Send multiple requests simultaneously to exploit timing windows.

C# Vulnerability Example:

```
// Step 1: Check access (proper authorization)
[HttpGet("api/documents/{id}/can-access")]
public IActionResult CanAccessDocument(int id)
{
    var userId = User.FindFirst(ClaimTypes.NameIdentifier).Value;
    var canAccess = _documentService.UserCanAccess(id, userId);
    return Ok(new { canAccess });
}
// Step 2: Actually get the document (vulnerable to race condition)
[HttpGet("api/documents/{id}/download")]
public IActionResult DownloadDocument(int id)
{
    // Assumes authorization was already checked in a previous request
    var document = _documentService.GetDocumentById(id);
    return File(document.Content, document.ContentType, document.FileName);
}
```

An attacker could exploit this by sending a valid access check for their own document, then quickly switching to another document ID in the download request.

Mass Assignment Exploitation

Mass assignment vulnerabilities often accompany IDOR issues, allowing attackers to modify unexpected properties.

Exploitation Process:

1. Model Analysis: Identify C# model classes with sensitive properties.
2. Property Discovery: Find endpoints that use model binding to update objects.
3. Property Injection: Add unexpected properties to request payloads.

Vulnerable C# Example:

```
// Vulnerable model class
public class UserProfile
{
    public int Id { get; set; }
    public string UserId { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public bool IsAdmin { get; set; } // Sensitive property
}
// Vulnerable controller action
[HttpPut("api/profile/{id}")]
public async Task<IActionResult> UpdateProfile(int id, UserProfile profile)
{
    if (id != profile.Id)
```

```
         return BadRequest();




<span class="hljs-comment">// Missing check if the current user owns this profile</span>
<span class="hljs-comment">// Missing protection against setting IsAdmin property</span>

_context.Entry(profile).State = EntityState.Modified;
<span class="hljs-keyword">await</span> _context.SaveChangesAsync();

<span class="hljs-keyword">return</span> NoContent();
}
```

An attacker could exploit this by including `"IsAdmin": true` in their request payload.

Using Developer Tools for IDOR Testing

Browser developer tools are essential for discovering and testing IDOR vulnerabilities.

Testing Process:

1. Network Monitoring: Use the Network tab to observe API calls and resource identifiers.
2. Request Modification: Modify requests using the Edit and Resend feature in developer tools.
3. Local Storage Analysis: Check for stored IDs or tokens that might enable IDOR attacks.
4. JavaScript Analysis: Examine client-side code for hidden resource identifiers or API endpoints.

Example Testing Process:

1. Access a legitimate resource and capture the request
2. Identify the resource identifier pattern
3. Modify the identifier to target a different resource
4. Compare responses to detect successful unauthorized access

Request Interception with Proxy Tools

Proxy tools provide more advanced capabilities for IDOR testing.

Key Tools:

- Burp Suite: Industry-standard web security testing tool
- OWASP ZAP: Free alternative with similar capabilities
- Fiddler: .NET-specific proxy with .NET Framework inspection capabilities

Testing Process:

1. Proxy Configuration: Set up the interception proxy to capture all application traffic.
2. Request Capture: Navigate the application normally to capture legitimate requests.
3. Request Manipulation: Use the proxy to modify resource identifiers before they reach the server.
4. Response Analysis: Examine responses for evidence of successful unauthorized access.
5. Sequence Testing: Test resource access at different stages of business processes.

C# Application-Specific Techniques:

- Inspect for `.NET-specific` serialization patterns
- Look for ASP.NET-specific request headers or cookies
- Test ViewState tampering in legacy WebForms applications
- Check for ASP.NET Core `Antiforgery` token bypass opportunities

Automated IDOR Discovery

Several tools can assist in automated discovery of IDOR vulnerabilities.

Automation Approaches:

1. Session Comparison: Test the same endpoints with different user sessions.
2. ID Crawling and Fuzzing: Automatically discover and test resource identifiers.
3. Response Difference Analysis: Compare responses to detect unauthorized access.

Tools and Frameworks:

- Burp Suite's Autorize extension: Compares requests across different authorization contexts
- OWASP ZAP's Access Control Testing: Automated access control testing

- Custom scripts using tools like Selenium or Puppeteer combined with proxy interception

Secure Code Implementations

Authorization Frameworks in C

ASP.NET Core provides robust authorization frameworks that, when properly implemented, mitigate IDOR vulnerabilities.

Basic Role-Based Authorization:

```
[Authorize(Roles = "Administrator")]
public class AdminController : Controller
{
    // Controller actions...
}
```

While simple, role-based authorization alone is insufficient for preventing IDOR vulnerabilities as it doesn't account for resource ownership.

Claims-Based Authorization

Claims-based authorization provides more flexibility for implementing fine-grained access controls.

Secure C# Implementation:

```
// Claims-based policy registration in Startup.cs
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options =>
    {
        options.AddPolicy("DocumentOwnerOnly", policy =>
            policy.RequireAssertion(context =>
            {
                var documentId = context.Resource as int?;
                if (!documentId.HasValue)
                    return false;
           <span class="hljs-keyword">var</span> userId =
context.User.FindFirstValue(ClaimTypes.NameIdentifier);
            <span class="hljs-keyword">var</span> documentService = context.Resource <span
class="hljs-keyword">as</span> IDocumentService;

            <span class="hljs-keyword">return</span>
documentService.IsDocumentOwnedByUser(documentId.Value, userId);
```

```
        }));
});
}
```

```
// Secure controller using the policy
[HttpGet("api/documents/{id}")]
[Authorize("DocumentOwnerOnly")]
public async Task<ActionResult<Document>> GetDocument(int id)
{
    var document = await _context.Documents.FindAsync(id);
    if (document == null)
        return NotFound();




<span class="hljs-keyword">return</span> <span class="hljs-built_in">document</span>;
}
```

This approach ensures that users can only access documents they own.

Resource-Based Authorization

Resource-based authorization is the most effective approach for preventing IDOR vulnerabilities, as it explicitly checks authorization against specific resources.

Secure C# Implementation:

```
[HttpGet("api/orders/{id}")]
public async Task<ActionResult<Order>> GetOrder(int id)
{
    var order = await _context.Orders.FindAsync(id);
    if (order == null)
        return NotFound();



<span class="hljs-comment">// Check if the current user is authorized to access this
specific order</span>
<span class="hljs-keyword">var</span> authorizationResult = <span
class="hljs-keyword">await</span> _authorizationService.AuthorizeAsync(
    User, order, <span class="hljs-string">"OrderAccessPolicy"</span>);

<span class="hljs-keyword">if</span> (!authorizationResult.Succeeded)
    <span class="hljs-keyword">return</span> Forbid();

<span class="hljs-keyword">return</span> order;
}
```

The `_authorizationService.AuthorizeAsync` method checks if the current user is authorized to access the specific order resource.

Implementing IAuthorizationHandler in C

Custom authorization handlers provide the most flexible way to implement resource-specific authorization.

Secure C# Implementation:

```
// Define the requirement
public class OrderOwnerRequirement : IAuthorizationRequirement { }



// Implement the handler
public class OrderOwnerAuthorizationHandler :
AuthorizationHandler<OrderOwnerRequirement, Order>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        OrderOwnerRequirement requirement,
        Order resource)
    {
        var userId = context.User.FindFirstValue(ClaimTypes.NameIdentifier);


   <span class="hljs-comment">// Check if the user owns the order or is an admin</span>
    <span class="hljs-keyword">if</span> (resource.UserId == userId ||
        context.User.IsInRole(<span class="hljs-string">"Administrator"</span>))
    {
        context.Succeed(requirement);
    }

    <span class="hljs-keyword">return</span> Task.CompletedTask;
}
}


// Register the handler in Startup.cs
services.AddSingleton<IAuthorizationHandler,
OrderOwnerAuthorizationHandler>();
services.AddAuthorization(options =>
{
    options.AddPolicy("OrderAccessPolicy", policy =>
        policy.Requirements.Add(new OrderOwnerRequirement()));
});
```

This implementation ensures that only the order's owner or administrators can access it.

Secure API Design Patterns

Beyond authorization, certain API design patterns can reduce the risk of IDOR vulnerabilities.

User-Scoped Endpoints:

```
// Instead of this vulnerable pattern:
[HttpGet("api/orders/{id}")]
// Use this more secure pattern:
[HttpGet("api/users/{userId}/orders/{orderId}")]
public async Task<ActionResult<Order>> GetUserOrder(string userId, int
orderId)
{
    // Verify the current user matches the requested userId
    if (User.FindFirstValue(ClaimTypes.NameIdentifier) != userId)
        return Forbid();


<span class="hljs-keyword">var</span> order = <span class="hljs-keyword">await</span>
_context.Orders
    .FirstOrDefaultAsync(o =&gt; o.Id == orderId &amp;&amp; o.UserId == userId);

<span class="hljs-keyword">if</span> (order == <span class="hljs-literal">null</span>)
    <span class="hljs-keyword">return</span> NotFound();

<span class="hljs-keyword">return</span> order;
}
```

This pattern makes the user context explicit in the URL and enforces it in the query.

Indirect Reference Maps

One of the most effective techniques for preventing IDOR vulnerabilities is to use indirect reference maps, which translate between public tokens and internal resource identifiers.

Secure C# Implementation:

```
public class SecureReferenceMap
{
    private readonly IMemoryCache _cache;
    private readonly string _currentUserId;



<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-title">SecureReferenceMap</span>(<span class="hljs-params">IMemoryCache cache,
IHttpContextAccessor contextAccessor</span>)
</span>{
    _cache = cache;
    _currentUserId =
contextAccessor.HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier);
}

<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-keyword">string</span> <span class="hljs-title">CreateReference</span>(<span
class="hljs-params"><span class="hljs-keyword">int</span> resourceId</span>)
</span>{
    <span class="hljs-comment">// Create a random token</span>
    <span class="hljs-keyword">var</span> token = Guid.NewGuid().ToString(<span
class="hljs-string">"N"</span>);

    <span class="hljs-comment">// Store the mapping between token and resource ID for this
user</span>
    <span class="hljs-keyword">var</span> cacheKey = <span class="hljs-string">$"<span
class="hljs-subst">{_currentUserId}</span>:<span class="hljs-subst">{token}</span>"</span>;
    _cache.Set(cacheKey, resourceId, TimeSpan.FromHours(<span
class="hljs-number">24</span>));

    <span class="hljs-keyword">return</span> token;
}

<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-keyword">bool</span> <span class="hljs-title">TryGetResourceId</span>(<span
class="hljs-params"><span class="hljs-keyword">string</span> token, <span
class="hljs-keyword">out</span> <span class="hljs-keyword">int</span> resourceId</span>)
</span>{
    <span class="hljs-keyword">var</span> cacheKey = <span class="hljs-string">$"<span
class="hljs-subst">{_currentUserId}</span>:<span class="hljs-subst">{token}</span>"</span>;
    <span class="hljs-keyword">return</span> _cache.TryGetValue(cacheKey, <span
class="hljs-keyword">out</span> resourceId);
}
}
// Using the secure reference map in a controller


[HttpGet("api/documents/{reference}")]
public async Task<ActionResult<Document>> GetDocument(string reference)
{
    if (!_referenceMap.TryGetResourceId(reference, out var documentId))
        return NotFound();


<span class="hljs-keyword">var</span> document = <span class="hljs-keyword">await</span>
_context.Documents.FindAsync(documentId);
<span class="hljs-keyword">if</span> (document == <span class="hljs-literal">null</span>)
    <span class="hljs-keyword">return</span> NotFound();

<span class="hljs-keyword">return</span> document;


}
```

With this approach, the client never sees the actual resource IDs, only per-user temporary tokens that map to those IDs.

Advanced Security Techniques

Implementing Custom Authorization Filters

Authorization filters provide a way to apply access control logic across multiple actions or controllers.

Secure C# Implementation:

```
public class ResourceOwnershipFilter : IAsyncAuthorizationFilter
{
    private readonly IResourceAccessVerifier _accessVerifier;




<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-title">ResourceOwnershipFilter</span>(<span
class="hljs-params">IResourceAccessVerifier accessVerifier</span>)
</span>{
    _accessVerifier = accessVerifier;
}

<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-keyword">async</span> Task <span
class="hljs-title">OnAuthorizationAsync</span>(<span
class="hljs-params">AuthorizationFilterContext context</span>)
</span>{
    <span class="hljs-comment">// Extract resource ID from route data</span>
    <span class="hljs-keyword">if</span> (!context.RouteData.Values.TryGetValue(<span
class="hljs-string">"id"</span>, <span class="hljs-keyword">out</span> <span
class="hljs-keyword">var</span> resourceIdObj))
        <span class="hljs-keyword">return</span>;

    <span class="hljs-keyword">if</span> (!<span
class="hljs-keyword">int</span>.TryParse(resourceIdObj.ToString(), <span
class="hljs-keyword">out</span> <span class="hljs-keyword">var</span> resourceId))
        <span class="hljs-keyword">return</span>;

    <span class="hljs-comment">// Get current user ID</span>
    <span class="hljs-keyword">var</span> userId =
context.HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier);
    <span class="hljs-keyword">if</span> (<span
class="hljs-keyword">string</span>.IsNullOrEmpty(userId))
    {
        context.Result = <span class="hljs-keyword">new</span> ForbidResult();
        <span class="hljs-keyword">return</span>;
    }

    <span class="hljs-comment">// Verify ownership</span>
    <span class="hljs-keyword">bool</span> hasAccess = <span
class="hljs-keyword">await</span> _accessVerifier.UserCanAccessResourceAsync(
        userId, resourceId, context.HttpContext.Request.Method);

    <span class="hljs-keyword">if</span> (!hasAccess)
    {
        context.Result = <span class="hljs-keyword">new</span> ForbidResult();
    }
}
}



// Resource verifier implementation
public class ResourceAccessVerifier : IResourceAccessVerifier
{
    private readonly ApplicationDbContext _context;



<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-title">ResourceAccessVerifier</span>(<span
class="hljs-params">ApplicationDbContext context</span>)
</span>{
    _context = context;
}

<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-keyword">async</span> Task&lt;<span class="hljs-keyword">bool</span>&gt; <span
class="hljs-title">UserCanAccessResourceAsync</span>(<span class="hljs-params"><span
class="hljs-keyword">string</span> userId, <span class="hljs-keyword">int</span>
resourceId, <span class="hljs-keyword">string</span> httpMethod</span>)
</span>{
    <span class="hljs-comment">// Check if the user is an admin</span>
    <span class="hljs-keyword">var</span> user = <span class="hljs-keyword">await</span>
_context.Users
        .FirstOrDefaultAsync(u =&gt; u.Id == userId);

    <span class="hljs-keyword">if</span> (user.IsAdmin)
        <span class="hljs-keyword">return</span> <span class="hljs-literal">true</span>;

    <span class="hljs-comment">// Check resource ownership based on resource type</span>
    <span class="hljs-comment">// This example assumes resourceId refers to a
Document</span>
    <span class="hljs-keyword">var</span> document = <span
class="hljs-keyword">await</span> _context.Documents
        .FirstOrDefaultAsync(d =&gt; d.Id == resourceId);

    <span class="hljs-keyword">if</span> (document == <span
class="hljs-literal">null</span>)
        <span class="hljs-keyword">return</span> <span class="hljs-literal">false</span>;

    <span class="hljs-comment">// For GET requests, check read access</span>
    <span class="hljs-keyword">if</span> (httpMethod == <span
class="hljs-string">"GET"</span>)
    {
        <span class="hljs-keyword">return</span> document.OwnerId == userId ||
document.SharedWithUserIds.Contains(userId);
    }

    <span class="hljs-comment">// For modification requests, require ownership</span>
    <span class="hljs-keyword">return</span> document.OwnerId == userId;
} }
```

This filter automatically checks resource ownership for all endpoints where it's applied.

Dynamic Permission Verification

For applications with complex permission models, dynamic verification provides flexible access control.

Secure C# Implementation:

```
public class PermissionVerificationService : IPermissionVerificationService
{
    private readonly ApplicationDbContext _context;




<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-title">PermissionVerificationService</span>(<span
class="hljs-params">ApplicationDbContext context</span>)
</span>{
    _context = context;
}

<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-keyword">async</span> Task&lt;<span class="hljs-keyword">bool</span>&gt; <span
class="hljs-title">HasPermissionAsync</span>(<span class="hljs-params">
    <span class="hljs-keyword">string</span> userId, <span
class="hljs-keyword">string</span> resourceType, <span class="hljs-keyword">int</span>
resourceId, <span class="hljs-keyword">string</span> permission</span>)
</span>{
    <span class="hljs-comment">// Check direct permissions</span>
    <span class="hljs-keyword">var</span> directPermission = <span
class="hljs-keyword">await</span> _context.UserPermissions
        .AnyAsync(p =&gt; p.UserId == userId &amp;&amp;
                 p.ResourceType == resourceType &amp;&amp;
                 p.ResourceId == resourceId &amp;&amp;
                 p.Permission == permission);

    <span class="hljs-keyword">if</span> (directPermission)
        <span class="hljs-keyword">return</span> <span class="hljs-literal">true</span>;

    <span class="hljs-comment">// Check role-based permissions</span>
    <span class="hljs-keyword">var</span> userRoles = <span
class="hljs-keyword">await</span> _context.UserRoles
        .Where(ur =&gt; ur.UserId == userId)
        .Select(ur =&gt; ur.RoleId)
        .ToListAsync();

    <span class="hljs-keyword">if</span> (!userRoles.Any())
        <span class="hljs-keyword">return</span> <span class="hljs-literal">false</span>;

    <span class="hljs-keyword">var</span> rolePermission = <span
class="hljs-keyword">await</span> _context.RolePermissions
        .AnyAsync(p =&gt; userRoles.Contains(p.RoleId) &amp;&amp;
                 p.ResourceType == resourceType &amp;&amp;
                 (p.ResourceId == resourceId || p.ResourceId == <span
class="hljs-literal">null</span>) &amp;&amp;
                 p.Permission == permission);

    <span class="hljs-keyword">return</span> rolePermission;
}
}
```

```
// Using the service in a controller


[HttpGet("api/resources/{type}/{id}")]
public async Task<IActionResult> GetResource(string type, int id)
{
    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);


<span class="hljs-comment">// Check if the user has 'read' permission for this
resource</span>
<span class="hljs-keyword">bool</span> hasPermission = <span
class="hljs-keyword">await</span> _permissionService.HasPermissionAsync(
    userId, type, id, <span class="hljs-string">"read"</span>);

<span class="hljs-keyword">if</span> (!hasPermission)
    <span class="hljs-keyword">return</span> Forbid();

<span class="hljs-comment">// Retrieve and return the resource</span>
<span class="hljs-keyword">var</span> resource = <span class="hljs-keyword">await</span>
_resourceService.GetResourceAsync(type, id);
<span class="hljs-keyword">if</span> (resource == <span class="hljs-literal">null</span>)
    <span class="hljs-keyword">return</span> NotFound();

<span class="hljs-keyword">return</span> Ok(resource);
}
```

This service verifies permissions based on resource type, resource ID, and the specific permission required.

Audit Logging for Access Control

Implementing comprehensive audit logging can help detect and investigate potential IDOR exploitation attempts.

Secure C# Implementation:

```
public class AccessAuditLogger : IAccessAuditLogger
{
    private readonly ApplicationDbContext _context;
    private readonly IHttpContextAccessor _httpContextAccessor;



<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-title">AccessAuditLogger</span>(<span class="hljs-params">
    ApplicationDbContext context,
    IHttpContextAccessor httpContextAccessor</span>)
</span>{
    _context = context;
    _httpContextAccessor = httpContextAccessor;
}
<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-keyword">async</span> Task <span
class="hljs-title">LogAccessAttemptAsync</span>(<span class="hljs-params">
    <span class="hljs-keyword">string</span> resourceType, <span
class="hljs-keyword">int</span> resourceId, <span class="hljs-keyword">bool</span>
wasSuccessful</span>)
</span>{
    <span class="hljs-keyword">var</span> context = _httpContextAccessor.HttpContext;
    <span class="hljs-keyword">var</span> userId =
context.User.FindFirstValue(ClaimTypes.NameIdentifier) ?? <span
class="hljs-string">"anonymous"</span>;

    <span class="hljs-keyword">var</span> log = <span class="hljs-keyword">new</span>
AccessLog
    {
        Timestamp = DateTime.UtcNow,
        UserId = userId,
        IPAddress = context.Connection.RemoteIpAddress.ToString(),
        UserAgent = context.Request.Headers[<span
class="hljs-string">"User-Agent"</span>].ToString(),
        ResourceType = resourceType,
        ResourceId = resourceId,
        WasSuccessful = wasSuccessful,
        HttpMethod = context.Request.Method,
        Path = context.Request.Path,
        Query = context.Request.QueryString.ToString()
    };

    _context.AccessLogs.Add(log);
    <span class="hljs-keyword">await</span> _context.SaveChangesAsync();
}
}
```

```
// Using the logger in a controller
```

```
[HttpGet("api/documents/{id}")]
public async Task<ActionResult<Document>> GetDocument(int id)
{
    var document = await _context.Documents.FindAsync(id);
```

```
<span class="hljs-keyword">if</span> (document == <span class="hljs-literal">null</span>)
{
    <span class="hljs-keyword">await</span> _auditLogger.LogAccessAttemptAsync(<span
class="hljs-string">"Document"</span>, id, <span class="hljs-literal">false</span>);
    <span class="hljs-keyword">return</span> NotFound();
}

<span class="hljs-keyword">var</span> userId =
User.FindFirstValue(ClaimTypes.NameIdentifier);
<span class="hljs-keyword">bool</span> isAuthorized = document.OwnerId == userId ||
                     document.SharedWithUserIds.Contains(userId);
```

```
<span class="hljs-keyword">await</span> _auditLogger.LogAccessAttemptAsync(<span
class="hljs-string">"Document"</span>, id, isAuthorized);

<span class="hljs-keyword">if</span> (!isAuthorized)
    <span class="hljs-keyword">return</span> Forbid();

<span class="hljs-keyword">return</span> document;
}
```

This logger records all access attempts, including successful and failed ones, providing valuable data for security analysis.

Role-Based vs. Resource-Based Access Control

Understanding the difference between role-based access control (RBAC) and resource-based access control (ReBAC) is crucial for preventing IDOR vulnerabilities.

RBAC Limitations:

```
// Role-based approach - vulnerable to IDOR
[Authorize(Roles = "User")]
public class DocumentsController : Controller
{
    [HttpGet("{id}")]
    public async Task<ActionResult<Document>> GetDocument(int id)
    {
        var document = await _context.Documents.FindAsync(id);
        if (document == null)
            return NotFound();



   <span class="hljs-comment">// Any user can access any document</span>
    <span class="hljs-keyword">return</span> <span class="hljs-built_in">document</span>;
}
}
```

This code ensures only authenticated users with the "User" role can access documents, but doesn't verify if the user should access a specific document.

ReBAC Implementation:

```
// Resource-based approach - protected against IDOR
public class DocumentsController : Controller
{
    private readonly IAuthorizationService _authorizationService;
<span class="hljs-comment">// Inject the authorization service</span>
<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-title">DocumentsController</span>(<span
class="hljs-params">IAuthorizationService authorizationService</span>)
</span>{
    _authorizationService = authorizationService;
}

[Authorize]
[HttpGet(<span class="hljs-string">"{id}"</span>)]
<span class="hljs-keyword">public</span> <span class="hljs-keyword">async</span>
Task&lt;ActionResult&lt;Document&gt;&gt; GetDocument(<span class="hljs-keyword">int</span>
id)
{
    <span class="hljs-keyword">var</span> document = <span
class="hljs-keyword">await</span> _context.Documents.FindAsync(id);
    <span class="hljs-keyword">if</span> (document == <span
class="hljs-literal">null</span>)
        <span class="hljs-keyword">return</span> NotFound();

    <span class="hljs-comment">// Check if the user can access this specific
document</span>
    <span class="hljs-keyword">var</span> authResult = <span
class="hljs-keyword">await</span> _authorizationService.AuthorizeAsync(
        User, document, <span class="hljs-string">"DocumentAccess"</span>);

    <span class="hljs-keyword">if</span> (!authResult.Succeeded)
        <span class="hljs-keyword">return</span> Forbid();

    <span class="hljs-keyword">return</span> document;
}
}
```

This approach checks if the current user is authorized to access the specific document requested.

Detection and Testing Methodologies

Manual Testing Approaches

Manual testing for IDOR vulnerabilities requires a systematic approach.

Step-by-Step Testing Process:

1. Identify Entry Points: Catalog all endpoints that accept resource identifiers.
2. Resource Mapping: Document the types of resources and their identifier patterns.
3. Access Control Matrix: Create a matrix of user roles and their expected access rights.
4. Multi-User Testing: Use multiple accounts with different permission levels.
5. Parameter Manipulation: Systematically test each endpoint with unauthorized resource IDs.
6. Boundary Testing: Test edge cases like:

    - Self-references (current user's own resources)
    - References to non-existent resources
    - References to resources at permission boundaries
7. HTTP Method Testing: Test all supported HTTP methods for each endpoint.

Key Test Scenarios:

- Can User A access User B's resources?
- Can a regular user access admin-only resources?
- Can an unauthenticated user access authenticated resources?
- Do access controls persist across the entire application?

Automated IDOR Testing

Automated tools can assist in discovering IDOR vulnerabilities at scale.

Automation Approach:

1. Crawling and Mapping: Automatically discover and map application endpoints.
2. Session Comparison: Test endpoints with different user sessions.
3. Parameter Fuzzing: Automatically manipulate resource identifiers.
4. Response Analysis: Analyze responses to detect successful unauthorized access.

C# Specific Testing Tools:

- Burp Suite with Autorize extension: For automating session comparison tests
- OWASP ZAP: For access control testing
- Custom test scripts: Using frameworks like Selenium or Playwright

Static Code Analysis for IDOR

Static analysis tools can help identify potential IDOR vulnerabilities in C# code.

Key Static Analysis Approaches:

1. Authorization Pattern Detection: Identify endpoints missing authorization checks.
2. Data Flow Analysis: Track how resource identifiers flow through the application.
3. Framework-Specific Rules: Apply rules specific to ASP.NET and ASP.NET Core.

Tools for C# Applications:

- SonarQube: With custom rules for IDOR detection
- Roslynator: For custom C# code analysis
- Security Code Scan: .NET security analyzer
- PREfast/FxCop: Classic .NET code analysis tools

Common Code Patterns to Detect:

- Controller actions with resource ID parameters but no authorization checks
- Entity Framework queries without user context filters
- Direct resource access methods without permission verification

Dynamic Testing with Proxies

Proxy tools enable detailed inspection and manipulation of HTTP traffic.

Testing Process:

1. Proxy Setup: Configure an interception proxy like Burp Suite or OWASP ZAP.
2. Application Mapping: Navigate the application normally to build a site map.
3. Request Analysis: Identify patterns in resource identification.
4. Authorization Testing: Compare responses between different user accounts.
5. Parameter Tampering: Systematically modify resource identifiers.
6. Session Handling: Test with different authentication tokens or cookies.

Proxy-Assisted Testing Techniques:

- Use Burp Suite's Comparer to identify differences in responses
- Use OWASP ZAP's Access Control Testing add-on
- Set up proxy rules to automatically modify certain parameters
- Record and replay requests with modified identifiers

Testing Authorization with Authenticated Sessions

Testing IDOR vulnerabilities requires multiple authenticated sessions.

Multi-User Testing Approach:

1. Account Preparation: Create test accounts with different permission levels.
2. Session Management: Maintain separate browser sessions for each account.
3. Cross-Session Testing: Use resource identifiers from one session in another.
4. Cookie and Token Analysis: Examine authentication and session tokens.

5. Horizontal Privilege Testing: Test access between accounts at the same level.
6. Vertical Privilege Testing: Test access between accounts at different levels.

C# Specific Testing Considerations:

- Test ASP.NET Core Identity authentication mechanisms
- Examine JWT token usage for API authentication
- Check for cookie-based session persistence
- Test claims-based and policy-based authorization

Real-World Case Studies

Case Study 1: E-commerce Platform Order Information Disclosure

An e-commerce platform built with ASP.NET Core had a vulnerability in its order tracking system.

Vulnerable Code:

```
[Authorize]
[HttpGet("api/orders/{orderNumber}/details")]
public async Task<ActionResult<OrderDetails>> GetOrderDetails(string
orderNumber)
{
    var order = await _context.Orders
        .Include(o => o.Items)
        .Include(o => o.ShippingDetails)
        .Include(o => o.PaymentSummary)
        .FirstOrDefaultAsync(o => o.OrderNumber == orderNumber);
```

```
<span class="hljs-keyword">if</span> (<span class="hljs-keyword">order</span> == null)
    <span class="hljs-keyword">return</span> NotFound();

<span class="hljs-comment">// Missing authorization check</span>

<span class="hljs-keyword">return</span> new OrderDetails
{
    OrderNumber = <span class="hljs-keyword">order</span>.OrderNumber,
    Status = <span class="hljs-keyword">order</span>.Status,
    PlacedDate = <span class="hljs-keyword">order</span>.PlacedDate,
    Items = <span class="hljs-keyword">order</span>.Items,
    ShippingAddress = <span class="hljs-keyword">order</span>.ShippingDetails.Address,
    PaymentMethod = <span class="hljs-keyword">order</span>.PaymentSummary.Method,
    LastFourDigits = <span class="hljs-keyword">order</span>.PaymentSummary.LastFourDigits,
    <span class="hljs-keyword">Total</span> = <span class="hljs-keyword">order</span>.<span
class="hljs-keyword">Total</span>
};
}
```

Exploitation:

An attacker logged in with their own account, made a purchase, and obtained their own order number (e.g., "ORD-12345"). They then simply changed the order number in the URL to access other customers' orders (e.g., "ORD-12346", "ORD-12347").

This exposed other customers' names, addresses, purchase history, and partial payment information.

Secure Implementation:

```
[Authorize]
[HttpGet("api/orders/{orderNumber}/details")]
public async Task<ActionResult<OrderDetails>> GetOrderDetails(string
orderNumber)
{
    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);

var <span class="hljs-keyword">order </span>= await _<span
class="hljs-built_in">context</span>.<span class="hljs-keyword">Orders
       .Include(o => o.Items)
        .Include(o => o.ShippingDetails)
        .Include(o => o.PaymentSummary)
        .FirstOrDefaultAsync(o => o.OrderNumber == orderNumber && o.UserId ==
userId);




if (<span class="hljs-keyword">order </span>== null)
    return NotFound()<span class="hljs-comment">;</span>

return new <span class="hljs-keyword">OrderDetails
    {
        OrderNumber = order.OrderNumber,
        Status = order.Status,
        PlacedDate = order.PlacedDate,
        Items = order.Items,
        ShippingAddress = order.ShippingDetails.Address,
        PaymentMethod = order.PaymentSummary.Method,
        LastFourDigits = order.PaymentSummary.LastFourDigits,
        Total = order.Total
    };
}
```

The fix adds a crucial filter to the database query, ensuring that users can only retrieve their own orders.

Case Study 2: Healthcare Patient Records Access

A healthcare management system built with ASP.NET MVC had a critical IDOR vulnerability allowing unauthorized access to patient records.

Vulnerable Code:

```
[Authorize(Roles = "Doctor,Nurse,Administrator")]
public class PatientController : Controller
{
    [HttpGet]
    public async Task<IActionResult> MedicalRecord(int patientId)
    {
        var record = await _patientService.GetMedicalRecordAsync(patientId);
        if (record == null)
            return NotFound();



   <span class="hljs-comment">// Missing check if the current staff member should have
access to this patient</span>

    <span class="hljs-keyword">return</span> View(record);
}
}
```

Exploitation:

Medical staff who were legitimately authorized to access some patient records could simply change the `patientId` parameter to access records of any patient in the system, regardless of whether they were assigned to that patient's care team.

Secure Implementation:

```
[Authorize(Roles = "Doctor,Nurse,Administrator")]
public class PatientController : Controller
{
    [HttpGet]
    public async Task<IActionResult> MedicalRecord(int patientId)
    {
        var currentUserId = User.FindFirstValue(ClaimTypes.NameIdentifier);



   <span class="hljs-comment">// First check if this is an administrator who has global
access</span>
```

```
    <span class="hljs-keyword">if</span> (User.IsInRole(<span
class="hljs-string">"Administrator"</span>))
    {
        <span class="hljs-keyword">var</span> record = <span
class="hljs-keyword">await</span> _patientService.GetMedicalRecordAsync(patientId);
        <span class="hljs-keyword">if</span> (record == <span
class="hljs-literal">null</span>)
            <span class="hljs-keyword">return</span> NotFound();

        <span class="hljs-comment">// Log the access for audit purposes</span>
        <span class="hljs-keyword">await</span>
_auditService.LogRecordAccessAsync(currentUserId, patientId, <span
class="hljs-string">"MedicalRecord"</span>);

        <span class="hljs-keyword">return</span> View(record);
    }

    <span class="hljs-comment">// For doctors and nurses, check if they're assigned to the
patient</span>
    <span class="hljs-keyword">bool</span> isAssignedToPatient = <span
class="hljs-keyword">await</span> _patientService.IsUserAssignedToPatientAsync(
        currentUserId, patientId);

    <span class="hljs-keyword">if</span> (!isAssignedToPatient)
        <span class="hljs-keyword">return</span> Forbid();

    <span class="hljs-keyword">var</span> patientRecord = <span
class="hljs-keyword">await</span> _patientService.GetMedicalRecordAsync(patientId);
    <span class="hljs-keyword">if</span> (patientRecord == <span
class="hljs-literal">null</span>)
        <span class="hljs-keyword">return</span> NotFound();

    <span class="hljs-comment">// Log the access for audit purposes</span>
    <span class="hljs-keyword">await</span>
_auditService.LogRecordAccessAsync(currentUserId, patientId, <span
class="hljs-string">"MedicalRecord"</span>);

    <span class="hljs-keyword">return</span> View(patientRecord);
}
}
```

The fix implements proper authorization checks based on the user's role and their assignment to the patient's care team, along with comprehensive audit logging.

Case Study 3: Financial Services API Vulnerability

A financial services company had an IDOR vulnerability in their account statement API.

Vulnerable Code:

```
[Authorize]
[Route("api/[controller]")]
[ApiController]
public class StatementsController : ControllerBase
{
    [HttpGet("download/{statementId}")]
    public async Task<IActionResult> DownloadStatement(Guid statementId)
    {
        var statement = await
_statementRepository.GetStatementByIdAsync(statementId);
        if (statement == null)
            return NotFound();



   // Missing authorization check

    var fileContent = await _documentService.GetStatementPdfAsync(<span
class="hljs-name">statementId</span>)<span class="hljs-comment">;</span>
    return File(<span class="hljs-name">fileContent</span>, <span
class="hljs-string">"application/pdf"</span>, $<span
class="hljs-string">"Statement_{statement.Date:yyyyMM}.pdf"</span>)<span
class="hljs-comment">;</span>
}
}
```

Exploitation:

Attackers discovered that statement IDs were GUIDs but were exposed in the HTML source of the account statement page. By collecting various statement IDs from their own account and observing the pattern, they could modify the GUID slightly to access other customers' financial statements.

Secure Implementation:

```
[Authorize]
[Route("api/[controller]")]
[ApiController]
public class StatementsController : ControllerBase
{
    [HttpGet("download/{referenceToken}")]
    public async Task<IActionResult> DownloadStatement(string referenceToken)
    {
        var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);



   // Convert the public token to an internal statement ID
    if (<span class="hljs-name">!_referenceMap.TryGetResourceId</span>(<span
class="hljs-name">userId</span>, referenceToken, out var statementId))
        return NotFound()<span class="hljs-comment">;</span>
    var statement = await _statementRepository.GetStatementByIdAsync(<span
class="hljs-name">statementId</span>)<span class="hljs-comment">;</span>
    if (<span class="hljs-name">statement</span> == null)
        return NotFound()<span class="hljs-comment">;</span>

    // Double-check ownership
    if (<span class="hljs-name">statement.AccountId</span> != userId)
    {
        _logger.LogWarning(<span class="hljs-string">"Possible IDOR attempt: User {UserId}
attempted to access statement {StatementId} belonging to account {OwnerId}"</span>,
            userId, statementId, statement.AccountId)<span class="hljs-comment">;</span>
        return Forbid()<span class="hljs-comment">;</span>
    }

    var fileContent = await _documentService.GetStatementPdfAsync(<span
class="hljs-name">statementId</span>)<span class="hljs-comment">;</span>
    return File(<span class="hljs-name">fileContent</span>, <span
class="hljs-string">"application/pdf"</span>, $<span
class="hljs-string">"Statement_{statement.Date:yyyyMM}.pdf"</span>)<span
class="hljs-comment">;</span>
}
}
```

The secure implementation uses indirect reference maps to prevent direct access to statement IDs, performs explicit ownership verification, and includes security logging.

Best Practices and Preventative Measures

Based on the vulnerabilities and secure implementations discussed, here are comprehensive best practices for preventing IDOR vulnerabilities in C# applications:

1. Implement Resource-Based Authorization

Always verify that the current user is authorized to access the specific resource being requested:

```
// Using ASP.NET Core's IAuthorizationService
[HttpGet("api/resources/{id}")]
public async Task<IActionResult> GetResource(int id)
{
    var resource = await _resourceRepository.GetByIdAsync(id);
    if (resource == null)
        return NotFound();
```

```
<span class="hljs-keyword">var</span> authResult = <span class="hljs-keyword">await</span>
_authorizationService.AuthorizeAsync(
    User, resource, <span class="hljs-string">"ResourceAccessPolicy"</span>);

<span class="hljs-keyword">if</span> (!authResult.Succeeded)
    <span class="hljs-keyword">return</span> Forbid();

<span class="hljs-keyword">return</span> Ok(resource);
```

```
}
```

2. Use Indirect Reference Maps

Avoid exposing internal identifiers directly to clients:

```
// Generate indirect references
public string GenerateReferenceToken(int resourceId)
{
    var token = Guid.NewGuid().ToString("N");
    _cache.Set($"{GetCurrentUserId()}:{token}", resourceId,
TimeSpan.FromHours(1));
    return token;
}
// Resolve references
public bool TryResolveReference(string token, out int resourceId)
{
    return _cache.TryGetValue($"{GetCurrentUserId()}:{token}", out
resourceId);
}
```

3. Apply User Context Filters in Data Access Layer

Filter data by user context at the repository/data access level:

```
// Secure repository implementation
public class DocumentRepository : IDocumentRepository
{
    private readonly ApplicationDbContext _context;
    private readonly IUserContextProvider _userContext;
```

```
<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-title">DocumentRepository</span>(<span class="hljs-params">
    ApplicationDbContext context,
    IUserContextProvider userContext</span>)
</span>{
    _context = context;
    _userContext = userContext;}

<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-keyword">async</span> Task&lt;Document&gt; <span
class="hljs-title">GetDocumentByIdAsync</span>(<span class="hljs-params"><span
class="hljs-keyword">int</span> documentId</span>)
</span>{
    <span class="hljs-keyword">var</span> userId = _userContext.GetCurrentUserId();

    <span class="hljs-keyword">return</span> <span class="hljs-keyword">await</span>
_context.Documents
        .Where(d =&gt; d.Id == documentId &amp;&amp; (
            d.OwnerId == userId ||
            d.SharedWithUsers.Any(u =&gt; u.UserId == userId)
        ))
        .FirstOrDefaultAsync();
}
}
```

4. Implement Custom Authorization Filters

Apply consistent authorization logic across controllers:

```
public class ResourceOwnershipFilter : IAsyncAuthorizationFilter
{
    private readonly IResourceAccessVerifier _accessVerifier;




<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-title">ResourceOwnershipFilter</span>(<span
class="hljs-params">IResourceAccessVerifier accessVerifier</span>)
</span>{
    _accessVerifier = accessVerifier;
}

<span class="hljs-function"><span class="hljs-keyword">public</span> <span
class="hljs-keyword">async</span> Task <span
class="hljs-title">OnAuthorizationAsync</span>(<span
class="hljs-params">AuthorizationFilterContext context</span>)
</span>{
    <span class="hljs-keyword">if</span> (!context.RouteData.Values.TryGetValue(<span
class="hljs-string">"id"</span>, <span class="hljs-keyword">out</span> <span
class="hljs-keyword">var</span> idObj))
        <span class="hljs-keyword">return</span>;

    <span class="hljs-keyword">if</span> (!<span
class="hljs-keyword">int</span>.TryParse(idObj.ToString(), <span
class="hljs-keyword">out</span> <span class="hljs-keyword">var</span> resourceId))
        <span class="hljs-keyword">return</span>;

    <span class="hljs-keyword">var</span> userId =
context.HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier);
    <span class="hljs-keyword">if</span> (<span
class="hljs-keyword">string</span>.IsNullOrEmpty(userId))
    {
        context.Result = <span class="hljs-keyword">new</span> UnauthorizedResult();
        <span class="hljs-keyword">return</span>;
    }

    <span class="hljs-keyword">bool</span> hasAccess = <span
class="hljs-keyword">await</span> _accessVerifier.VerifyAccessAsync(
        userId, resourceId, context.HttpContext.Request.Method);

    <span class="hljs-keyword">if</span> (!hasAccess)
    {
        context.Result = <span class="hljs-keyword">new</span> ForbidResult();
    }
}
}
```

5. Use Scoped API Design Patterns

Design APIs to make the user context explicit:

```
// Instead of:
[HttpGet("api/documents/{id}")]
```

```
// Prefer:
[HttpGet("api/users/{userId}/documents/{documentId}")]
public async Task<IActionResult> GetUserDocument(string userId, int
documentId)
{
    // Verify the current user matches the requested userId
    if (User.FindFirstValue(ClaimTypes.NameIdentifier) != userId)
        return Forbid();
```

```
<span class="hljs-built_in">var</span> <span class="hljs-built_in">document</span> = <span
class="hljs-keyword">await</span> _documentRepository.GetByIdForUserAsync(userId,
documentId);
<span class="hljs-keyword">if</span> (<span class="hljs-built_in">document</span> == <span
class="hljs-literal">null</span>)
    <span class="hljs-keyword">return</span> NotFound();

<span class="hljs-keyword">return</span> Ok(<span class="hljs-built_in">document</span>);
}
```

6. Implement Comprehensive Audit Logging

Log all access attempts for security monitoring:

```
public async Task LogResourceAccessAsync(
    string userId, string resourceType, int resourceId, bool succeeded)
{
    var logEntry = new AccessLogEntry
    {
        Timestamp = DateTime.UtcNow,
        UserId = userId,
        IPAddress =
_httpContextAccessor.HttpContext.Connection.RemoteIpAddress.ToString(),
        ResourceType = resourceType,
        ResourceId = resourceId,
        WasSuccessful = succeeded,
        HttpMethod = _httpContextAccessor.HttpContext.Request.Method,
        UserAgent =
_httpContextAccessor.HttpContext.Request.Headers["User-Agent"].ToString()
    };
_context.AccessLogs.Add(logEntry);
await _context.SaveChangesAsync();
}
```

7. Avoid Mass Assignment Vulnerabilities

Use view models and explicit property mapping:

```
// Instead of binding directly to entity models:
public async Task<IActionResult> UpdateUser(int id, User user)
```

```
// Use view models and explicit mapping:
public async Task<IActionResult> UpdateUser(int id, UserUpdateModel model)
{
    var user = await _userRepository.GetByIdAsync(id);
    if (user == null)
        return NotFound();
```

```
<span class="hljs-comment">// Verify ownership</span>
<span class="hljs-keyword">if</span> (user.Id !=
User.FindFirstValue(ClaimTypes.NameIdentifier))
    <span class="hljs-keyword">return</span> Forbid();

<span class="hljs-comment">// Explicit property mapping</span>
user.Name = model.Name;
user.Email = model.Email;
user.PhoneNumber = model.PhoneNumber;
<span class="hljs-comment">// <span class="hljs-doctag">Note:</span> Sensitive properties
like IsAdmin are not mapped from the model</span>

<span class="hljs-keyword">await</span> _userRepository.UpdateAsync(user);
<span class="hljs-keyword">return</span> NoContent();}
```

8. Implement Proper Object-Level Authorization

Use authorization handlers for object-level access control:

```
public class DocumentAuthorizationHandler :
AuthorizationHandler<ResourceAccessRequirement, Document>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        ResourceAccessRequirement requirement,
        Document resource)
    {
        var userId = context.User.FindFirstValue(ClaimTypes.NameIdentifier);
```

```
   <span class="hljs-comment">// Check direct ownership</span>
    <span class="hljs-keyword">if</span> (resource.OwnerId == userId)
    {
        context.Succeed(requirement);
        <span class="hljs-keyword">return</span> Task.CompletedTask;
    }

    <span class="hljs-comment">// Check sharing permissions</span>
    <span class="hljs-keyword">if</span> (resource.SharedWith.Any(s =&gt; s.UserId ==
userId &amp;&amp; s.AccessLevel &gt;= requirement.MinimumAccessLevel))
    {
        context.Succeed(requirement);
        <span class="hljs-keyword">return</span> Task.CompletedTask;
    }

    <span class="hljs-comment">// Check administrative access</span>
    <span class="hljs-keyword">if</span> (context.User.IsInRole(<span
class="hljs-string">"Administrator"</span>))
    {
        context.Succeed(requirement);
        <span class="hljs-keyword">return</span> Task.CompletedTask;
    }

    <span class="hljs-keyword">return</span> Task.CompletedTask;
}
}
```

9. Conduct Regular Security Testing

Implement a comprehensive security testing program:

- Automated IDOR scanning with tools like Burp Suite
- Manual penetration testing specifically targeting access control
- Regular code reviews focusing on authorization logic
- Multi-user testing scenarios to verify proper access control

10. Follow Defense-in-Depth Principles

Implement multiple layers of protection:

- Proper authentication with MFA where possible
- Well-configured authorization at the application level
- Data access filtering at the repository level
- Database-level security with row-level security where appropriate
- Comprehensive input validation and sanitization
- Indirect reference mappings for sensitive resources
- Proper output encoding and escaping

Conclusion

IDOR vulnerabilities represent one of the most critical security risks in modern web applications, including those built with C# and the .NET framework. These vulnerabilities arise from a fundamental failure to properly verify that a user is authorized to access the specific resources they request.

Throughout this article, we've examined how IDOR vulnerabilities manifest in C# applications, explored various exploitation techniques, and presented comprehensive secure implementation patterns. The key lessons include:

1. Authorization Is Critical: Always implement proper authorization checks before allowing access to resources.
2. User Context Matters: Filter all data access based on the current user's context.
3. Indirect References Provide Protection: Avoid exposing internal identifiers directly to clients.
4. Defense-in-Depth Is Essential: Implement multiple layers of security controls.
5. Regular Testing Is Necessary: Continuously test for IDOR vulnerabilities with multiple user accounts.

By implementing the secure coding patterns and best practices outlined in this article, developers can significantly reduce the risk of IDOR vulnerabilities in their C# applications. Remember that security is an ongoing process, and staying informed about emerging threats and countermeasures is essential for maintaining robust application security.

References

1. OWASP Top 10 2021: Broken Access Control. https://owasp.org/Top10/A01_2021-Broken_Access_Control/
2. OWASP Testing Guide: Testing for Insecure Direct Object References. https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Applic

ation_Security_Testing/05-Authorization_Testing/04-Testing_for_Insecure_Direct_Object_References
3. Microsoft Documentation: ASP.NET Core Security Documentation. https://docs.microsoft.com/en-us/aspnet/core/security/
4. Microsoft Documentation: Resource-Based Authorization in ASP.NET Core. https://docs.microsoft.com/en-us/aspnet/core/security/authorization/resourcebased
5. Microsoft Documentation: Custom Authorization Policy Providers in ASP.NET Core. https://docs.microsoft.com/en-us/aspnet/core/security/authorization/iauthorizationpolicyprovider
6. PortSwigger Web Security Academy: Insecure Direct Object References. https://portswigger.net/web-security/access-control/idor
7. Troy Hunt: Understanding IDOR Vulnerabilities. https://www.troyhunt.com/understanding-idor-vulnerabilities/
8. Andrew Lock: ASP.NET Core Authentication and Authorization. https://andrewlock.net/introduction-to-authentication-with-asp-net-core/
9. Chsakell's Blog: ASP.NET Core Identity Series. https://chsakell.com/2018/04/28/asp-net-core-identity-series-getting-started/
10. NWebsec: Security Libraries for ASP.NET Core. https://docs.nwebsec.com/en/latest/