**Description**

Broken Authorization (also known as Broken Access Control or Privilege Escalation) is the hypernym for a range of flaws that arise due to the ineffective implementation of authorization checks used to designate user access privileges.

Different users are permitted or denied access to various content and functions in adequately designed and implemented authorization frameworks depending on the user's designated role and corresponding privileges. For example, in a web application, authorization is subject to authentication and session management. However, designing authorization across dynamic systems is complex, and may result in inconsistent mechanisms being written as the applications evolve: authentication libraries and protocols change, user roles do as well, more users come, users go, some users are (not) removed when gone... access control design decisions are made not by technology, but by humans, so the potential for error is high and ever-present.

Vulnerabilities of this nature may affect any modern software present in web applications, databases, operating systems, and other technological infrastructure reliant on authorization controls.

**Impact**

In a successful attack, a malicious actor may be able to access unauthorized content, change or delete content, perform functions, and even assume full control of site administration. Once this level of compromise has been achieved, the damage of the attack is limited only by the privileges granted to the impersonated victim.

Indeed, the [breach of a toy-making company](https://www.troyhunt.com/data-from-connected-cloudpets-teddy-bears-leaked-and-ransomed-exposing-kids-voice-messages/) called CloudPets illustrates the aforementioned human error. In this case, the result of non-existent authorization controls led to the leak and subsequent ransom of children's voice messages, which had been recorded via, and stored (badly) in, the cloud.

**Scenarios**

From the perspective of the user, there are two main categories of authorization controls to consider:

* Horizontal Authorization Controls
* Vertical Authorization Controls

**Horizontal Authorization Control Bypass**

Bypassing Horizontal Authorization Controls describes the act of an unprivileged user gaining access to other users' accounts that possess equal privileges.

For example, imagine an application accepting unverified data in a method call downstream to retrieve account information. An attacker could easily modify the accountId parameter in the HTTP Request to retrieve data from one or even multiple other users' accounts.

The application uses unverified data in a method call downstream to retrieve account information:

http://vulnerableapp.com/user/account?accountId=7800001

http://vulnerableapp.com/user/account?accountId=7800002

etc.

Insecure Direct Object References, or IDOR, is a related scenario involving user-supplied input being utilized to access objects directly.

**Vertical Authorization Control Bypass**

Vertical Authorization Control bypasses describe the *upwards* use of access. That is, when a user with a certain level of privilege can indicate that they possess some higher level of access, like administrative level access, to the application.

In this example, an attacker has browsed to an administrative URL, where admin rights *should* be required to access the admin page.

http://vulnerableapp.com/user/account

http://vulnerableapp.com/admin/panel

If the application does not check whether the role of the session user matches the role required to access the resource, a user without admin privileges will be able to access the page by simply knowing/guessing the target URL and browsing to it.

**Prevention**

There are numerous basic steps developers can take to prevent Broken Authorization Control attacks, starting with an appraisal of an application's access control requirements and then the formation of an appropriate security policy congruent with the findings. This should include an access control matrix, a clear policy of what types of users can access the system, and what these users can and can't do with their access. Use role-based access control (RBAC) to enforce the roles at the appropriate boundaries.

For web applications, make sure that the access control mechanism is enforced correctly at the server side on every page and API endpoint. Users should not be able to access any unauthorized functionality or information by simply requesting direct access to the page or object.

**Testing**

Verify that the principle of least privilege exists - users should only be able to access functions, data files, URLs, controllers, services, and other resources, for which they possess specific authorization.

* **OWASP ASVS**: [4.1, 4.2](https://github.com/OWASP/ASVS/releases/download/v4.0.2_release/OWASP.Application.Security.Verification.Standard.4.0.2-en.pdf)
* **OWASP Testing Guide**: [Authorization Testing](https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/05-Authorization_Testing/README.html)

**.NET**

**Vulnerable example #1**

The following ASP.NET Core snippet shows a MVC controller that does not protect an administrative action.

**public** **class** **AccountController** : **Controller** {

**public** ActionResult **AdminFunctionality**() {

}

}

Simple authentication and authorization controls can be enforced by adding the [Authorize] attribute either on the controller or on the action.

**Vulnerable example #2**

The following ASP.NET Core snippet shows an incorrect mix of [AllowAnonymous] and [Authorize] attributes, resulting in a non-protected action.

[**AllowAnonymous**]

**public** **class** **AccountController** : **Controller** {

[**Authorize**]

**public** ActionResult **AdminFunctionality**() {

}

**public** actionResult **PublicFunctionality**() {

}

}

[AllowAnonymous] bypasses all authorization statements, hence the [Authorize] attribute is ignored and does not protect the administrative functionality.

**Prevention**

Apply the recommended authentication and authorization mechanisms depending on the framework of choice.

**ASP.NET Core**

ASP.NET Core authorization provides different models; a simple, declarative role, and a rich policy-based model. At its simplest, applying the [Authorize] attribute to a controller or action limits access to any authenticated user. Once the user is authorized, controllers expose the ControllerBase.HttpContext that can be leveraged to fetch the current user details.

[**Authorize**]

**public** **class** **AccountController** : **Controller**

{

**public** ActionResult **AdminFunctionality**()

{

**var** userName = HttpContext.User.FindFirst(ClaimTypes.Name).Value

*// Do something based on the current user details*

}

}

The [Authorize] attribute can limit access to users who are members of a specific role, e.g. Administrator in the following example.

[**Authorize(Roles = "Administrator")**]

**public** **class** **AccountController** : **Controller**

{

}

If policies have been configured at startup, they can be applied using the Policy property on the same attribute, e.g. [Authorize(Policy = "RequireAdministratorRole")]. More complex authorization strategies can be enforced by using claims-based and policy-based authorizations, or by defining custom authorization methods AuthorizationHandler. See the references for further information.

**References**

[Microsoft - Simple authorization in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/simple)  
[Microsoft - Role-based authorization in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/roles)  
[Microsoft - Claims-based authorization in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/claims)  
[Microsoft - Policy-based authorization in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/policies)