Security and Identity

Asp.net core

Overview

- authentication
- authorization
- data protection
- HTTPS enforcement
- app secrets
- anti-request forgery protection
- CORS management

Authentication vs. Authorization

Authentication is a process in which a user provides credentials that are then compared to those stored in an operating system, database, app or resource. If they match, users authenticate successfully, and can then perform actions that they're authorized for, during an authorization process. The authorization refers to the process that determines what a user is allowed to do.

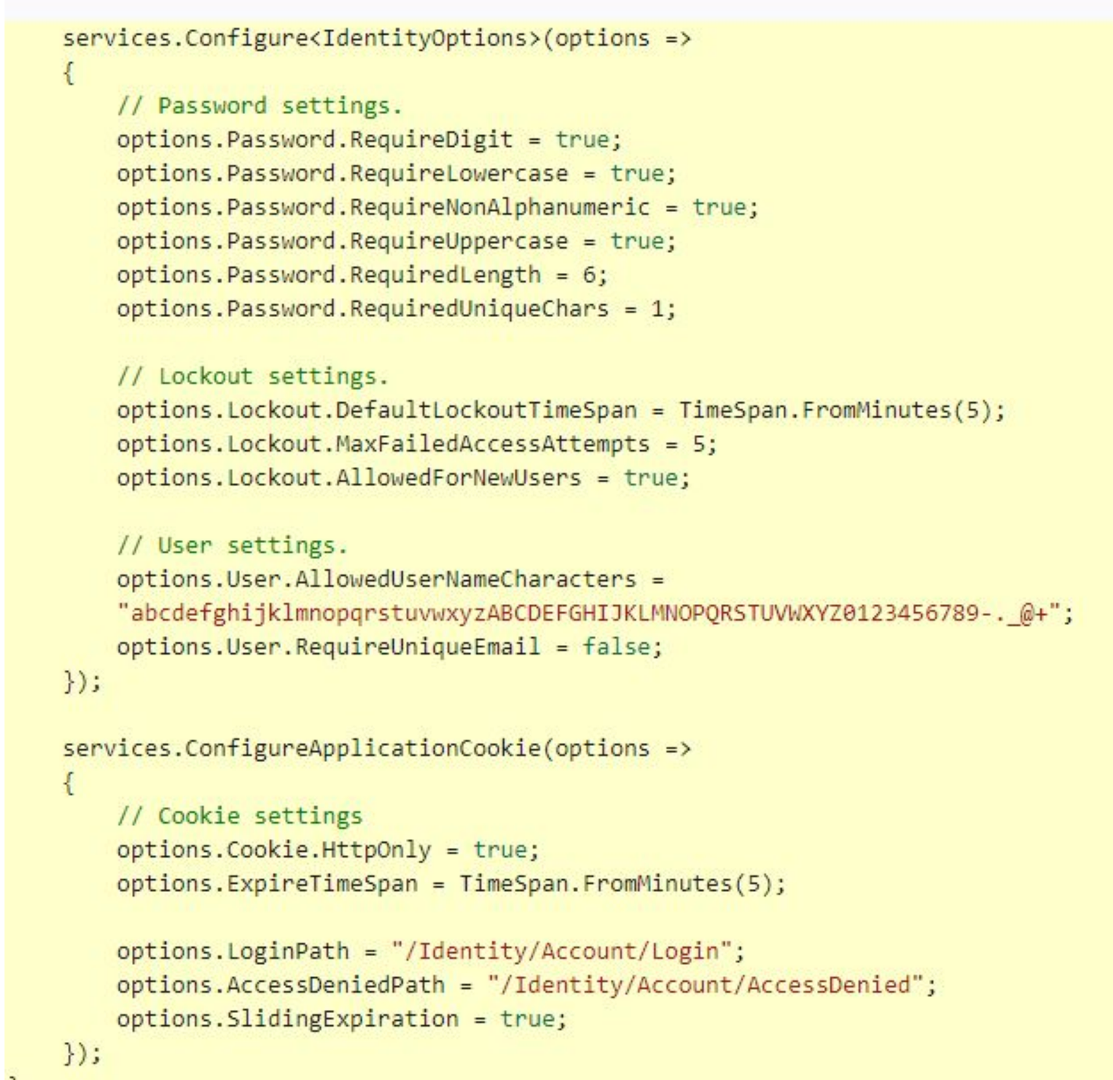
Another way to think of authentication is to consider it as a way to enter a space, such as a server, database, app or resource, while authorization is which actions the user can perform to which objects inside that space (server, database, or app).

Introduction to Identity on ASP.NET Core

ASP.NET Core Identity is a membership system that supports user interface (UI) login functionality. Users can create an account with the login information stored in Identity or they can use an external login provider. Supported external login providers include Facebook, Google, Microsoft Account, and Twitter.

Identity can be configured using a SQL Server database to store user names, passwords, and profile data. Alternatively, another persistent store can be used, for example, MySql.

Identity configuration



```
services.Configure<IdentityOptions>(options =>
{
    // Password settings.
    options.Password.RequireDigit = true;
    options.Password.RequireLowercase = true;
    options.Password.RequireNonAlphanumeric = true;
    options.Password.RequireUppercase = true;
    options.Password.RequiredLength = 6;
    options.Password.RequiredUniqueChars = 1;

    // Lockout settings.
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
    options.Lockout.MaxFailedAccessAttempts = 5;
    options.Lockout.AllowedForNewUsers = true;

    // User settings.
    options.User.AllowedUserNameCharacters =
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
    options.User.RequireUniqueEmail = false;
});

services.ConfigureApplicationCookie(options =>
{
    // Cookie settings
    options.Cookie.HttpOnly = true;
    options.ExpireTimeSpan = TimeSpan.FromMinutes(5);

    options.LoginPath = "/Identity/Account/Login";
    options.AccessDeniedPath = "/Identity/Account/AccessDenied";
    options.SlidingExpiration = true;
});
```

Demo

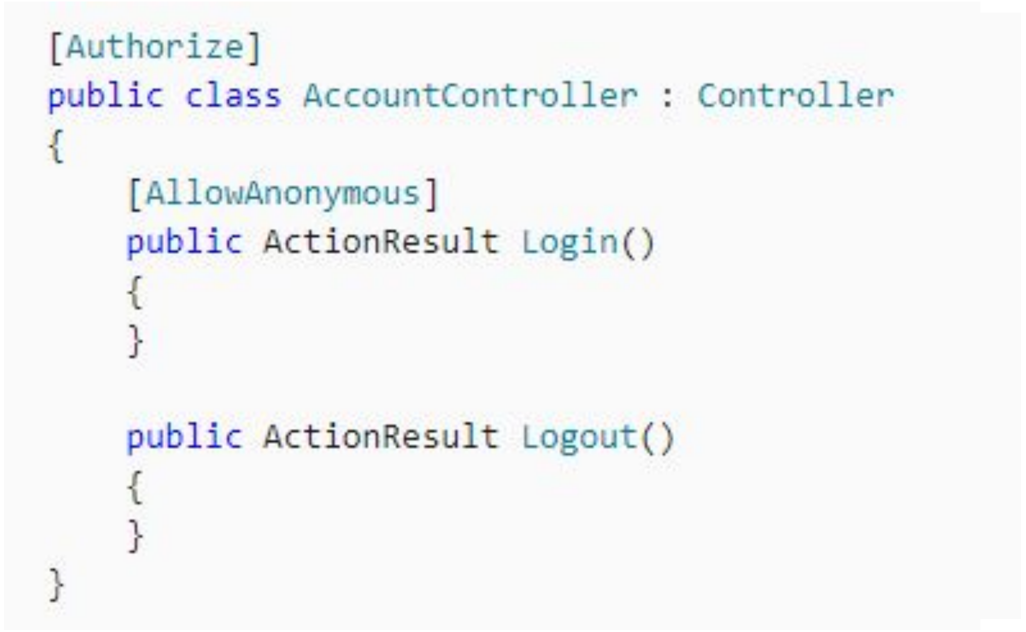
authorization

Authorization refers to the process that determines what a user is able to do. For example, an administrative user is allowed to create a document library, add documents, edit documents, and delete them. A non-administrative user working with the library is only authorized to read the documents.

Authorization is orthogonal and independent from authentication. However, authorization requires an authentication mechanism. Authentication is the process of ascertaining who a user is. Authentication may create one or more identities for the current user.

Simple authorization

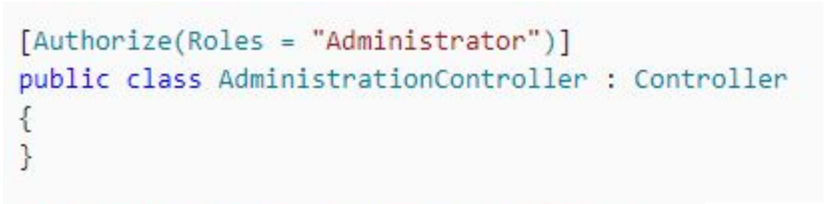
Authorization in MVC is controlled through the `AuthorizeAttribute`attribute and its various parameters. At its simplest, applying the `AuthorizeAttribute`attribute to a controller or action limits access to the controller or action to any authenticated user.



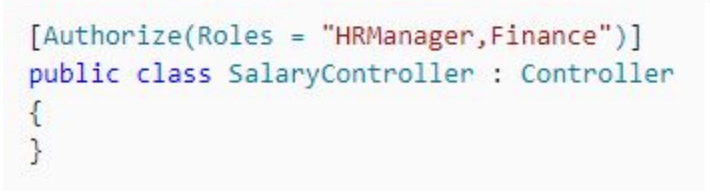
```
[Authorize]
public class AccountController : Controller
{
    [AllowAnonymous]
    public ActionResult Login()
    {
    }

    public ActionResult Logout()
    {
    }
}
```

Role-based authorization

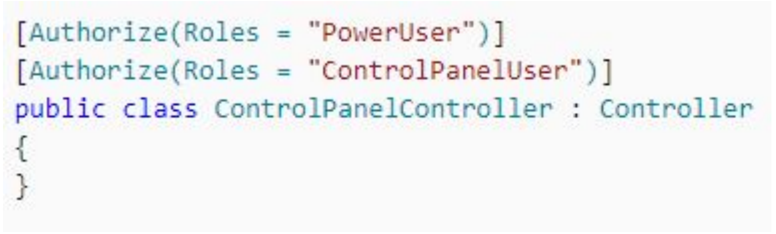
When an identity is created it may belong to one or more roles. For example, Tracy may belong to the Administrator and User roles whilst Scott may only belong to the User role. How these roles are created and managed depends on the backing store of the authorization process. Roles are exposed to the developer through the IsInRole method on the ClaimsPrincipal class.



```
[Authorize(Roles = "Administrator")]
public class AdministrationController : Controller
{
}
```



```
[Authorize(Roles = "HRManager,Finance")]
public class SalaryController : Controller
{
}
```

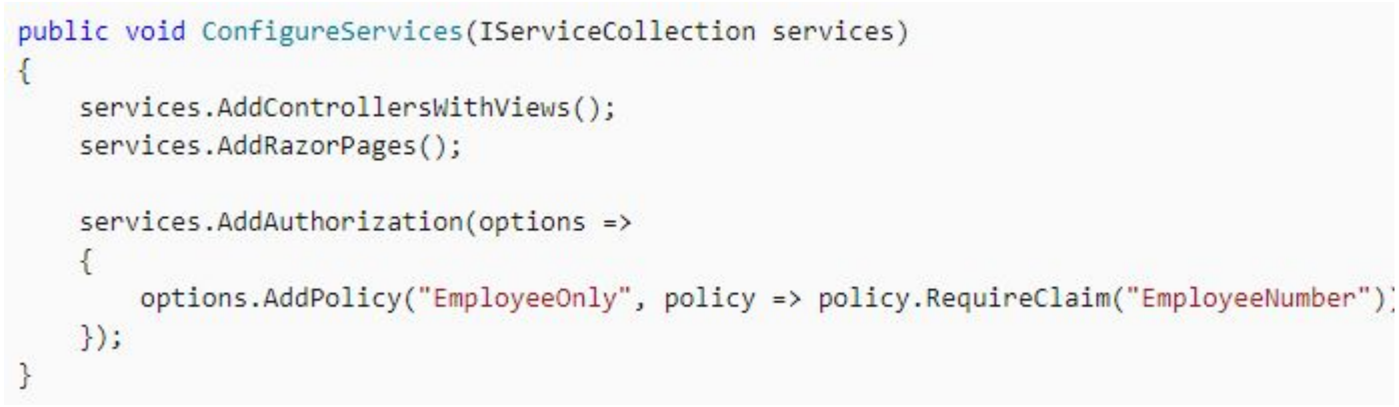


```
[Authorize(Roles = "PowerUser")]
[Authorize(Roles = "ControlPanelUser")]
public class ControlPanelController : Controller
{
}
```

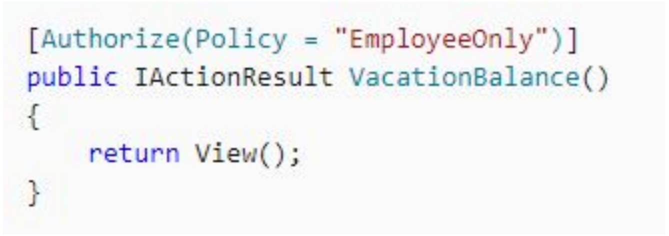
Claims-based authorization

When an identity is created it may be assigned one or more claims issued by a trusted party. A claim is a name value pair that represents what the subject is, not what the subject can do. For example, you may have a driver's license, issued by a local driving license authority. Your driver's license has your date of birth on it. In this case the claim name would be `DateOfBirth`, the claim value would be your date of birth, for example `8th June 1970` and the issuer would be the driving license authority. Claims based authorization, at its simplest, checks the value of a claim and allows access to a resource based upon that value.

Claims-based authorization



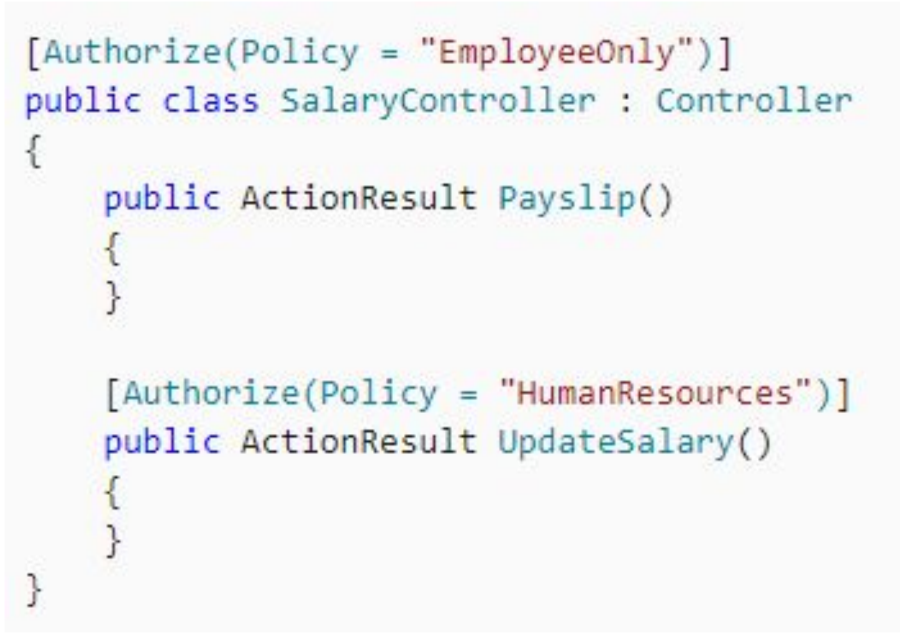
```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    services.AddRazorPages();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("EmployeeOnly", policy => policy.RequireClaim("EmployeeNumber"))
    });
}
```



```
[Authorize(Policy = "EmployeeOnly")]
public IActionResult VacationBalance()
{
    return View();
}
```

Multiple Policy Evaluation

If you apply multiple policies to a controller or action, then all policies must pass before access is granted.



```
[Authorize(Policy = "EmployeeOnly")]
public class SalaryController : Controller
{
    public ActionResult Payslip()
    {
    }

    [Authorize(Policy = "HumanResources")]
    public ActionResult UpdateSalary()
    {
    }
}
```

Authentication fundamentals

Cookie-based authentication is a popular form of authentication. Token-based authentication systems are growing in popularity, especially for Single Page Applications (SPAs).

Cookie-based authentication

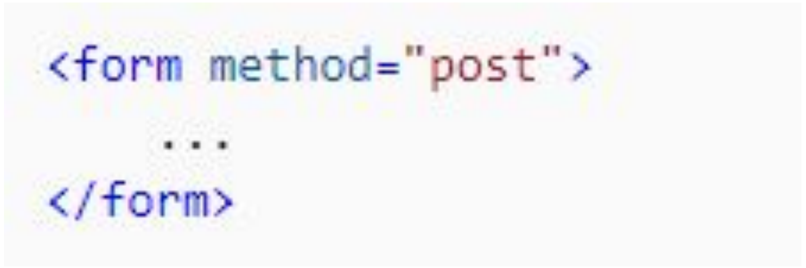
When a user authenticates using their username and password, they're issued a token, containing an authentication ticket that can be used for authentication and authorization. The token is stored as a cookie that accompanies every request the client makes. Generating and validating this cookie is performed by the Cookie Authentication Middleware. The middleware serializes a user principal into an encrypted cookie. On subsequent requests, the middleware validates the cookie, recreates the principal, and assigns the principal to the User property of HttpContext.

Token-based authentication

When a user is authenticated, they're issued a token (not an antiforgery token). The token contains user information in the form of claims or a reference token that points the app to user state maintained in the app. When a user attempts to access a resource requiring authentication, the token is sent to the app with an additional authorization header in form of Bearer token. This makes the app stateless. In each subsequent request, the token is passed in the request for server-side validation. This token isn't *encrypted*; it's *encoded*. On the server, the token is decoded to access its information. To send the token on subsequent requests, store the token in the browser's local storage

antiforgery tokens

The FormTagHelper injects antiforgery tokens into HTML form elements. The following markup in a Razor file automatically generates antiforgery tokens

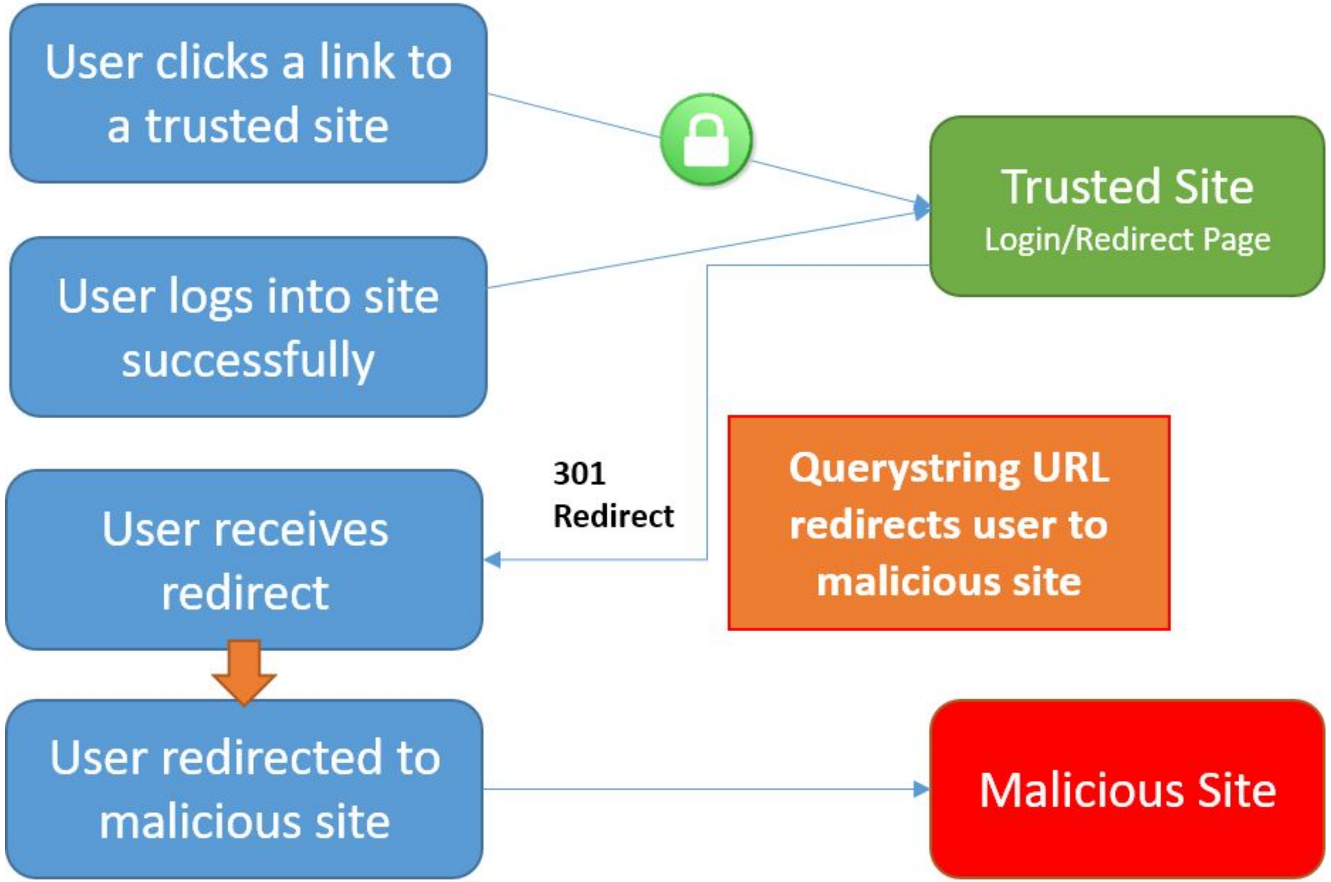


```
<form method="post">
    ...
</form>
```

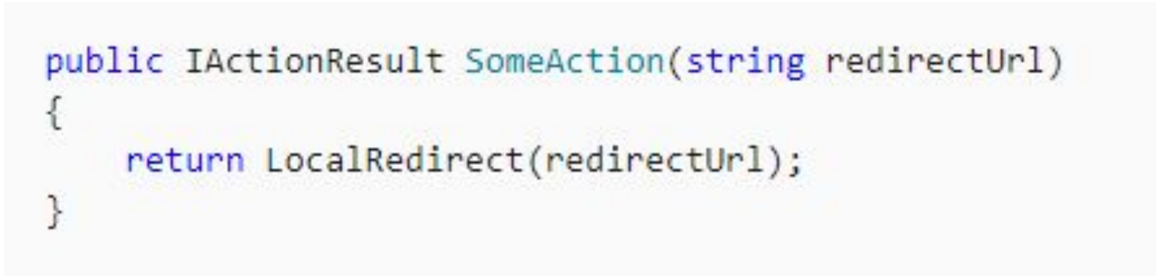
The automatic generation of antiforgery tokens for HTML form elements happens when the `<form>` tag contains the `method="post"` attribute and either of the following are true:

- The action attribute is empty (`action=""`).
- The action attribute isn't supplied (`<form method="post">`).

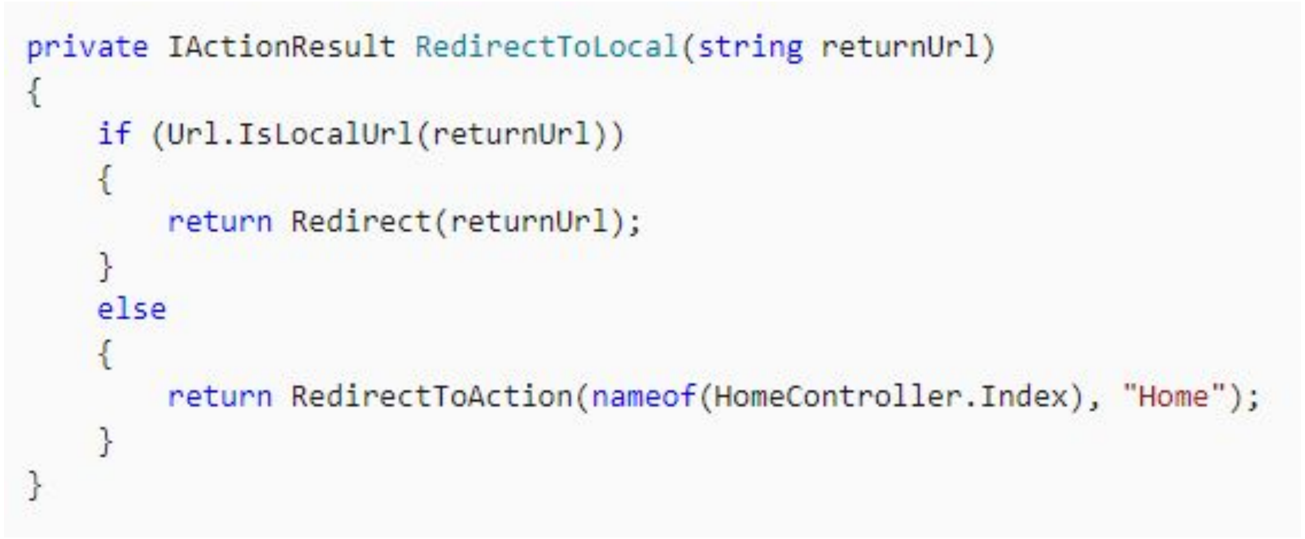
Open Redirection Attack Process


User clicks a link to a trusted site
Trusted Site
Login/Redirect Page
User logs into site successfully
301 Redirect
Querystring URL redirects user to malicious site
User receives redirect
User redirected to malicious site
Malicious Site

Protecting against open redirect attacks



```
public IActionResult SomeAction(string redirectUrl)
{
    return LocalRedirect(redirectUrl);
}
```



```
private IActionResult RedirectToLocal(string returnUrl)
{
    if (Url.IsLocalUrl(returnUrl))
    {
        return Redirect(returnUrl);
    }
    else
    {
        return RedirectToAction(nameof(HomeController.Index), "Home");
    }
}
```

Prevent Cross-Site Scripting (XSS)

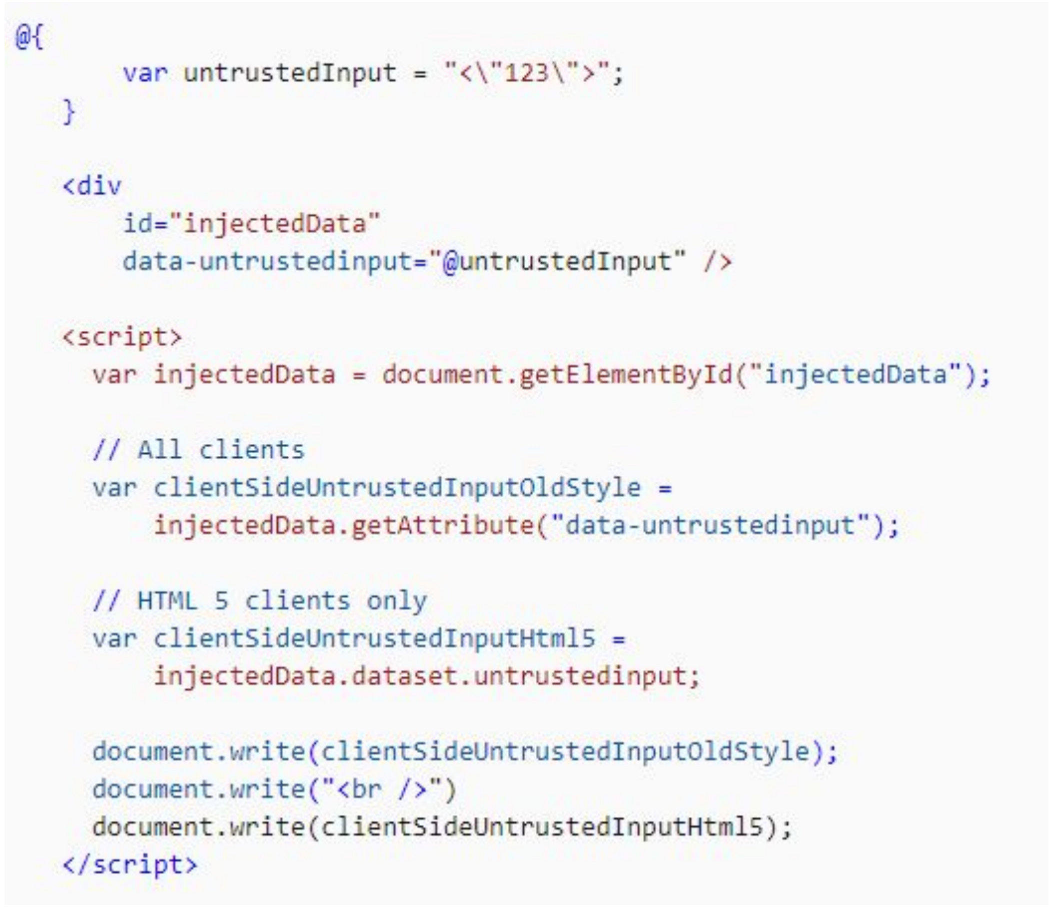
1. Never put untrusted data into your HTML input, unless you follow the rest of the steps below. Untrusted data is any data that may be controlled by an attacker, HTML form inputs, query strings, HTTP headers, even data sourced from a database as an attacker may be able to breach your database even if they cannot breach your application.
2. Before putting untrusted data inside an HTML element ensure it's HTML encoded. HTML encoding takes characters such as < and changes them into a safe form like &lt;
3. Before putting untrusted data into an HTML attribute ensure it's HTML encoded. HTML attribute encoding is a superset of HTML encoding and encodes additional characters such as " and '.
4. Before putting untrusted data into JavaScript place the data in an HTML element whose contents you retrieve at runtime. If this isn't possible, then ensure the data is JavaScript encoded. JavaScript encoding takes dangerous characters for JavaScript and replaces them with their hex, for example < would be encoded as `\u003C`.
5. Before putting untrusted data into a URL query string ensure it's URL encoded.

HTML Encoding using Razor

The Razor engine used in MVC automatically encodes all output sourced from variables, unless you work really hard to prevent it doing so. It uses HTML attribute encoding rules whenever you use the @ directive. As HTML attribute encoding is a superset of HTML encoding this means you don't have to concern yourself with whether you should use HTML encoding or HTML attribute encoding. You must ensure that you only use @ in an HTML context, not when attempting to insert untrusted input directly into JavaScript. Tag helpers will also encode input you use in tag parameters.

JavaScript Encoding

y be times you want to insert a
 JavaScript to process in your view.
 two ways to do this. The safest way
alues is to place the value in a data
f a tag and retrieve it in your
t



```
@{
       var untrustedInput = "<\"123\">";
   }

   <div
       id="injectedData"
       data-untrustedinput="@untrustedInput" />

   <script>
     var injectedData = document.getElementById("injectedData");

     // All clients
     var clientSideUntrustedInputOldStyle =
         injectedData.getAttribute("data-untrustedinput");

     // HTML 5 clients only
     var clientSideUntrustedInputHtml5 =
         injectedData.dataset.untrustedinput;

     document.write(clientSideUntrustedInputOldStyle);
     document.write("<br />")
     document.write(clientSideUntrustedInputHtml5);
   </script>
```

```
@using System.Text.Encodings.Web;
   @inject JavaScriptEncoder encoder;

   @{
       var untrustedInput = "<\"123\">";
   }

   <script>
       document.write("@encoder.Encode(untrustedInput)");
   </script>
```

```
<script>
       document.write("\u003C\u0022123\u0022\u003E");
   </script>
```