Unit 7
Securing in Asp.Net Core Application

Authentication

- Authentication is the process of recognizing a user's identity by verifying the provided security credentials like password, finger prints etc. It is the mechanism of associating an incoming request with a set of identifying credentials. The credentials provided are compared to those on a file in a database of the authorized user's information on a local operating system or within an authentication server.
- Authentication process can be described in two distinct phases - identification and actual authentication.
- Identification phase provides a user identity to the security system. This identity is provided in the form of a user ID. The security system will search all the abstract objects that it knows and find the specific one of which the actual user is currently applying. Once this is done, the user has been identified. The fact that the user claims does not necessarily mean that this is true. An actual user can be mapped to other abstract user object in the system, and therefore be granted rights and permissions to the user and user must give evidence to prove his identity to the system. The process of determining claimed user identity by checking user-provided evidence is called authentication and the evidence which is provided by the user during process of authentication is called a credential.

ASP.NET Identity Framework

- ASP.NET Core Identity:
  - Is an API that supports user interface (UI) login functionality.
  - Manages users, passwords, profile data, roles, claims, tokens, email confirmation, and more.
- ASP.NET Core Identity is an API with which you can create Authorization and Authentication features in your web application. Users can create an account and login with a user name and password. This also includes Roles and Roles Management. It provides the necessary functionality to manage users, roles, claims etc. It also provides options for two-factor authentication, password reset, email confirmation etc.
- ASP.NET Core Identity uses a SQL Server Database to store user names, passwords, roles, and profile data.

Features

1. One ASP.NET
   It can be used across all of the ASP.NET frameworks, such as ASP.NET MVC, Web Forms and Web APIs.
2. User Management
   ASP.NET Identity comes with API to handle Creating, Editing and deleting of users.
3. Control over schema
   The Developer has now complete control over the user information. We can easily extend the user profile information to include more user information like date of birth or any other information. Identity uses the entity framework **code first** to store the user related information to the database.
4. Two-factor Authentications using email/SMS.
   Two-factor authentication is a process where the user is authenticated by a combination of two different methods. For Example, password and sending a security code to users registered email or Mobile.
5. Account Confirmation Mail
   The Account Confirmation email can be sent to the user to verify the email ID.
6. Account Lockout
   Account lockout feature disables the user **accounts** if the user enters the wrong password for a specified number of times over a short duration.
7. Roles Management
   These are stored in a separate table. We can Create, Edit and delete roles.
8. Claims Management
   The claim is a piece of information about the user. It could be his name, his ID, email or anything that is important to the application you build. They are stored as a name-value pair. Claims allow developers to add lots more information about the user than the simple username-password provided by the old membership system.
9. External Identity Providers
   Users can now log in using any of the social login providers like Facebook, google+, linked in etc.
10. Asynchronous
    Most of the APIs in the ASP.NET Core Identity are asynchronous.
11. Unit test
    You can unit test your login/Register actions using the unit test framework as it is now simple to mock the identity framework.

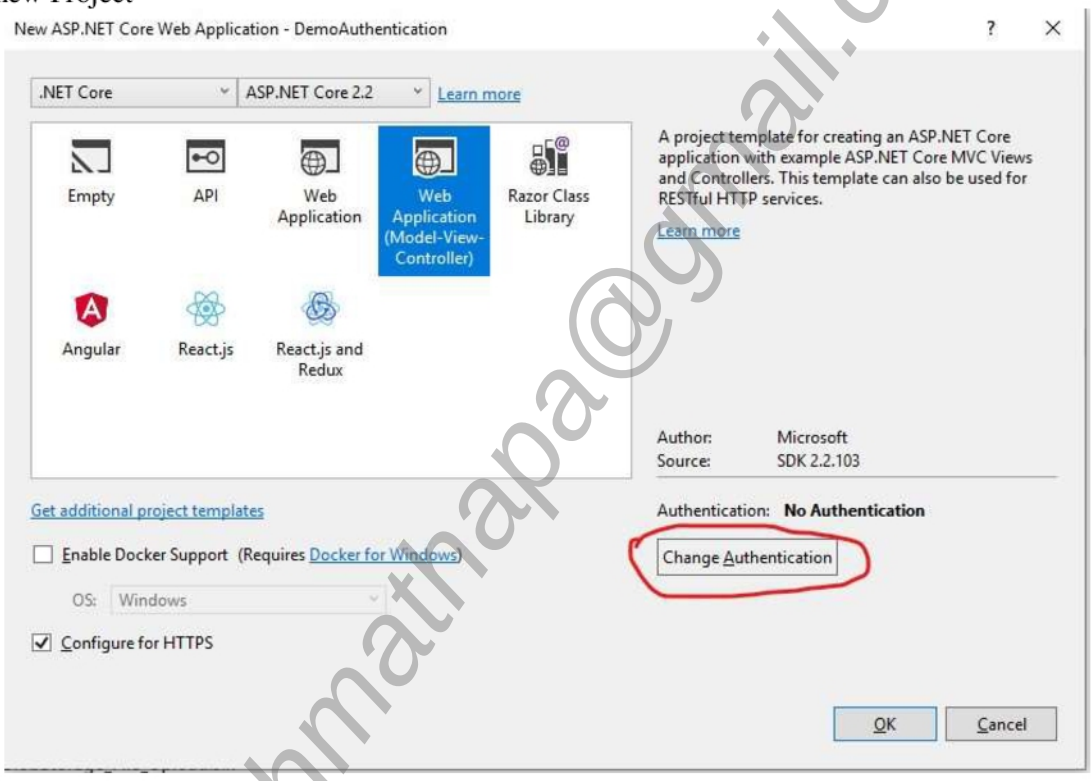
Creating Asp.Net MVC Core Application with Authentication(Identity)

- ASP.NET Core Identity needs 4 packages to be installed in your application. These packages are:
  1. Microsoft.AspNetCore.Identity
  2. Microsoft.AspNetCore.Identity.EntityFrameworkCore
  3. Microsoft.EntityFrameworkCore.Design
  4. Microsoft.EntityFrameworkCore.SqlServer
- These all packages can be installed using NuGet Package Manager.

Or

We can directly create project with Identity authentication as below

- Open a New Project in Visual Studio 2017.
- Choose ASP.NET Core Web Application Template and name the Project as DemoAuthentication Click OK
- Choose the Platform as .NET Core and select the Version ASP.NET Core 2.2. Choose the template as Web Application (Model-View-Controller). Select Change Authentication as "Individual User Account". Click OK to create the new Project



Migrating Identity Database

- Migration is a technique that helps us to create, update and sync the database with your model classes
- Before performing migration operation we have to configure the database connection string in the appsetting.json file as below.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=<server name>;Database=<database name>
                                    Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

The above Json file provides the database name, database server name, and related credentials (Window authentication or SQL server authentication) to establish the connection with the database.

- Now, to automate the migrations & create a database for Identity we need to run the following commands in the package manager console.

Pm>add-migration MyIdentity
pm>update-database

- The first command will create the SQL Statements, necessary to create the database, in a few files. These files are created inside the **/Data/Migrations** folder.

- After successfully running the second commands, it will create Identity related table from the file created by first command in the SQL Server database specified by the default connection in appsettings.json.

Note: We can user pm>remove-migration to revert the previous migration operation

Scaffolding Views (login, Register, Logout and others)

- If we check in our existing project code, we **do not find** any Login or Register and other user UI. We can generate them via Scaffolding technique in Asp.Net Core.
- ASP.NET Scaffolding is a code generation framework for ASP.NET Core Web applications.
- Visual Studio IDE includes pre-installed code generators for MVC and Web API projects.
- We add scaffolding to your project when you want to quickly add code that interacts with data models.
- Using scaffolding can reduce the amount of time to develop standard data operations in your project.

Steps to Scaffold

1. To add identity right click on project in solution explorer and select Add=>New Scaffolded Items from the context menu.
2. On selecting Add New Scaffolded Item we need to select Identity under Installed and click on Add Button.

3. After clicking Identity, then we have to select the layout page, required controller/views like login, logout, register etc and DataContext Class for Identy and proceed to Ok. This will add new folder **Area and subfolder Identity** with all the necessary model, views and controller files necessary for authentication and authorization.

Area

- Areas are an ASP.NET feature used to organize related functionality into a group as a separate:
  i. Namespace for routing.
  ii. Folder structure for views and Razor Pages.
- Using areas creates a hierarchy for the purpose of routing by adding another route parameter, area, to controller and action or a Razor Page page.
- Areas provide a way to partition an ASP.NET Core Web app into smaller functional groups, each with its own set of Razor Pages, controllers, views, and models. An area is effectively a structure inside an app. In an ASP.NET Core web project, logical components like Pages, Model, Controller, and View are kept in different folders. The ASP.NET Core runtime uses naming conventions to create the relationship between these components. For a large app, it may be advantageous to partition the app into separate high level areas of functionality. For instance, an e-commerce app with multiple business units, such as checkout, billing, and search. Each of these units have their own area to contain views, controllers, Razor Pages, and models.
- To add area, right click on solution explorer, then click Add and then Area. Give suitable Area name, then this will be added in the Area folder having sub folder Model, View, Controller and Data.

Configuring Identity Services

- IdentityDbContext provides all the DbSet properties needed to manage the identity tables in SQL Server so our application DbContext class must inherit from IdentityDbContext class instead of DbContext class.

```
public class AppDbContext : IdentityDbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options): base(options)
    {
    }
    // Rest of the code
}
```

- In ConfigureServices() method of the Startup class, include the following identity service.

```
public void ConfigureServices(IServiceCollection services)
{
      services.AddDbContext<ApplicationDbContext>(options =>
                  options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
```

```
        services.AddIdentity<IdentityUser,IdentityRole> ()
                                                    .AddEntityFrameworkStores<ApplicationDbContext>();
    }
```

- In the configure() Method of Startup class, add the authentication middleware.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseAuthentication();
}
```

Write Asp.Net Core MVC program to demonstrate the concept or user registration using Identity Framework.

Model:RegisterModel.cs

```
namespace IdentityDemo1.Models
{
    public class RegisterModel
    {
            public int Id { get; set; }
            public string Email { get; set; }
            public string Password { get; set; }
    }
}
```

Controller: HomeController.cs

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Identity;
namespace IdentityDemo1.Controllers
{
    public class HomeController : Controller
    {
        UserManager<IdentityUser> _usermanager;
        public HomeController(UserManager<IdentityUser> usermanager)
        {
            _usermanager = usermanager;
        }
        public IActionResult ShowRegisterPage()
        {
            return View(null);
        }
        public async Task< IActionResult> SubmitRegister(RegisterModel r)
        {
            var user = new IdentityUser { UserName = r.Email, Email = r.Email };
            var result = await _usermanager.CreateAsync(user, r.Password);
            if (result.Succeeded)
            {
                return Content("User Registered Successfully");
            }
            else
            {
                string msg="";
                foreach (var er in result.Errors)
                {
                    msg = msg + er.Description+"\n";
                }
                return Content(msg);
            }
        }
    }
}
```

View: ShowRegistrationPage.cshtml

```
@model IdentityDemo1.Models.RegisterModel
<html>
<body>
    <form method="post" action="SubmitRegister">
        <label asp-for="Email"></label>
        <input asp-for="Email" />
        <br />
        <label asp-for="Password"></label>
        <input asp-for="Password"/>
        <br />
        <input type="submit" value="Submit" />
    </form>
</body>
</html>
```

Write Asp.Net Core MVC program to demonstrate the concept or user login/authentication using identity framework.

Model: LoginModel.cs

```
namespace IdentityDemo1.Models
{
    public class LoginModel
    {
        public int Id { get; set; }
        public string email { get; set; }
        public string password { get; set; }
    }
}
```

Controller: HomeController.cs

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Identity;
namespace IdentityDemo1.Controllers
{
    public class HomeController : Controller
    {
        UserManager<IdentityUser> _usermanager;
        public HomeController(UserManager<IdentityUser> usermanager)
        {
            _usermanager = usermanager;
        }
        public IActionResult ShowLoginPage()
        {
            return View(null);
        }
        public async Task<IActionResult> SubmitLogin(Models.RegisterModel r)
        {
            var user = await _usermanager.FindByEmailAsync(r.Email);
            if (user != null && await _usermanager.CheckPasswordAsync(user, r.Password))
            {
                return Content("Welcome " + r.Email);
            }
            else
            {
                return Content("Invalid Username or Password");
            }
        }
    }
}
```

View:ShowLoginPage.cshtml

```
@model IdentityDemo1.Models.LoginModel
<html>
<body>
    <form asp-action="SubmitLogin">
        <label asp-for="email"></label>
        <input asp-for="email"/>
        <br />
        <label asp-for="password"></label>
        <input asp-for="password"/>
        <br />
        <input type="submit" value="Login" />
    </form>
</body>
</html>
```

Authorization: Roles, Claims and Policies

- **Authorization refers to the process that determines what a user is able to do.** i.e. Authorization is the process of determining what the logged-in user can and cannot do to a specific resource. Or is granted to perform a particular action. In an organization you may have the following roles
    a. Employee
    b. Manager
    c. HR
- Depending on the role or roles the logged-in user belong to, you may or may not authorize access to certain resources in the application. Since we are using Roles to make authorization checks, this is commonly called Role Based Access Control (RBAC) or Role Based Authorization. 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 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.
- ASP.NET Core authorization provides a simple, declarative role and a rich policy-based model.
- Authorization is expressed in requirements, and handlers evaluate a user's claims against requirements.
- Imperative checks can be based on simple policies or policies which evaluate both the user identity and properties of the resource that the user is attempting to access.
- Authorization components, including the **AuthorizeAttribute** and **AllowAnonymousAttribute** attributes, are found in the **Microsoft.AspNetCore.Authorization** namespace.

Simple authorization in ASP.NET Core

- Authorization in ASP.NET Core is controlled with **AuthorizeAttribute** and its various parameters.
- In its simplest form, applying the [Authorize] attribute to a controller or action, limits access to that component to any authenticated user.
- For example

1. **Securing Controller**
   The following code limits access to the **AccountController** to any authenticated user.

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

   In the above example, only the authenticated user is allowed to access the actions methods defined in Account Controller as the controller is decorated with Authrize Attribute.

2. **Securing Action**
   The following code limits access to Logout action method of the **AccountController** to any authenticated user.

```
public class AccountController : Controller
{
    public ActionResult Login()
    {
    }

    [Authorize]
    public ActionResult Logout()
    {
    }
}
```

In the above example, only the authenticated user is allowed to access the actions methods Logout() defined in Account Controller as the controller is decorated with Authrize Attribute.

Note:
We can also lock down a controller but allow anonymous, unauthenticated access to individual actions.
For this we use the **AllowAnonymous** attribute to allow access by non-authenticated users to individual actions.
For example:

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

    public ActionResult Logout()
    {
    }
}
```

This would allow only authenticated users to the AccountController, except for the Login action, which is accessible by everyone, regardless of their authenticated or unauthenticated as it is decorated with **AllowAnonymous attribute.**

Role Based Authorization

- When an identity is created it may belong to one or more roles.
- For example, Ram may belong to the Administrator and User roles whilst Hari 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.

Adding Role Check

- Role-based authorization checks are declarative—the developer embeds them within their code, against a controller or an action within a controller, specifying roles which the current user must be a member of to access the requested resource.
- For example, the following code limits access to any actions on the AdministrationController to users who are a member of the Administrator role:

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

We can specify multiple roles as a comma separated list:

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

```
}
```

This controller would be only accessible by users who are members of the HRManager role or the Finance role.

Similarly, if we apply multiple attributes then an accessing user must be a member of all the roles specified; the following sample requires that a user must be a member of both the PowerUser and ControlPanelUser role.

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

We can further limit access by applying additional role authorization attributes at the action level:

```
[Authorize(Roles = "Administrator, PowerUser")]
public class ControlPanelController : Controller
{
        public ActionResult SetTime()
        {
        }

        [Authorize(Roles = "Administrator")]
        public ActionResult ShutDown()
        {
        }
}
```

In the previous code snippet members of the Administrator role or the PowerUser role can access the controller and the SetTime action, but only members of the Administrator role can access the ShutDown action.

Claim 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. So, it is really a piece of information about the user, not what the user can and cannot 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 are policy based. We create a policy and include one or more claims in that policy. The policy is then used along with the policy parameter of the Authorize attribute to implement claims based authorization.
- Claims based authorization, at its simplest, checks the value of a claim and allows access to a resource based upon that value. For example, if you want access to a night club the authorization process might be: The door security officer would evaluate the value of your date of birth claim and whether they trust the issuer (the driving license authority) before granting you access.
- An identity can contain multiple claims with multiple values and can contain multiple claims of the same type.

Adding claims checks

- Claim based authorization checks are declarative - the developer embeds them within their code, against a controller or an action within a controller, specifying claims which the current user must possess, and optionally the value the claim must hold to access the requested resource.
- Claims requirements are policy based, the developer must build and register a policy expressing the claims requirements.
- The simplest type of claim policy looks for the presence of a claim and doesn't check the value.
- First you need to build and register the policy. This takes place as part of the Authorization service configuration, which normally takes part in ConfigureServices() in your *Startup.cs* file.

```
public void ConfigureServices(IServiceCollection services)
{
```

```
        options.AddPolicy("DeleteRolePolicy",
policy=>policy.RequireClaim("DeleteRole"));
}
```

To satisfy above policy requirements, the logged-in user must have Delete Role claim

Similarly, when we have to add more than one claim for a policy it can be done by chaining the claim as below

```
services.AddAuthorization(options =>
            {
                options.AddPolicy("DeleteRolePolicy",

policy=>policy.RequireClaim("DeleteRole").RequireClaim("CreateRole"));
            });
```

To satisfy this policy requirements, the loggedin user must have both the claims

We then apply the policy using the Policy property on the AuthorizeAttribute attribute to specify the policy name;

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

In this case the EmployeeOnly policy checks for the presence of both DeleteRole and CreateRole claim on the current identity.

The AuthorizeAttribute attribute can be applied to an entire controller, in this instance only identities matching the policy will be allowed access to any Action on the controller.

If you have a controller that's protected by the AuthorizeAttribute attribute, but want to allow anonymous access to particular actions you apply the AllowAnonymousAttribute attribute.

```
[Authorize(Policy = "DeleteRolePolicy")]
public class HomeController : Controller
{
    public ActionResult InsertRecord ()
    {
    }

    [AllowAnonymous]
    public ActionResult ViewRecord ()
    {
    }
}
```

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

```
[Authorize(Policy = "DeleteRolePolicy")]
public class HomeController : Controller
{
    public ActionResult InsertRecord()
    {
    }

    [Authorize(Policy = "UpdateRolePolicy")]
```

```
    public ActionResult UpdateRecord()
    {
    }
}
```

In the above example any identity which fulfills the DeleteRolePolicy policy can access the InsertRecord action method as that policy is enforced on the controller. However, in order to call the UpdateRecord action method the identity must fulfill *both* the DeleteRolePolicy policy and the UpdateRolePolicy policy.

Most claims come with a value. You can specify a list of allowed values when creating the policy. The following example would only succeed for employees whose employee number was 1, 2, 3, 4 or 5.

```
public void ConfigureServices(IServiceCollection services)
{
        services.AddAuthorization(options =>
        {
            options.AddPolicy("Founders", policy =>
                              policy.RequireClaim("EmployeeNumber", "1",
    "2", "3", "4", "5"));
        });
}
```

To satisfy this policy requirements, the loggedin user must have EmployeeNumber claim with any value 1, 2, 3, 4 or 5. Any other logged user user with EmployeeNumber claim but not with this claim value are not allowed to access.

Write ASP.net Core MVC program using Identity Framework to create a login page with email and password input field and remember me checkbox. When user presses the submit button, validate the user login and redirect to welcome page that shows the welcome message if the user claim role as "Admin". If the user directly enters the URL of welcome page, the `access must be denied and redirected to login page. Similarly, if the user logged in by checking the remember me option, the next request to login page must be automatically redirect to welcome page bypassing the authentication.

Models: LoginModel.cs

```
namespace IdentityDemo1.Models
{
    public class LoginModel
    {
        public int Id { get; set; }
        public string email { get; set; }
        public string password { get; set; }
        public bool Rememberme { get; set; }
    }
}
```

Controller: Home Controller.cs

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;
using IdentityDemo1.Models;
namespace IdentityDemo1.Controllers
{
    public class Home1 : Controller
    {
        SignInManager<IdentityUser> _signinmanager;
        public Home1(SignInManager<IdentityUser> signinmanager)
        {
            _signinmanager = signinmanager;
        }
```

```
        [HttpGet]
        public IActionResult showloginpage()
        {
            if (_signinmanager.IsSignedIn(User))
            {
                return Redirect("welcome");
            }
            else
            {
                return View(null);
            }
        }


        [HttpPost]
        public async Task<IActionResult> SubmitLogin(LoginModel r)
        {
            var result = await _signinmanager.PasswordSignInAsync(r.email, r.password,
                                r.Rememberme, false);//signing in with persistent cookie if
                                                     remember be checked in other wise session cookie
            if (result.Succeeded)
            {
                return Redirect("welcome");
            }
            else
            {
                return Redirect("showloginpage");
            }
        }


       [Authorize (Roles ="Admin")]
        public IActionResult welcome()
        {
            return Content("Welcome To My Site");
        }
    }
}
```

View: showloginpage.cshtml

```
@model IdentityDemo1.Models.LoginModel
<html>
<body>
    <form asp-action="SubmitLogin">
        <label asp-for="email"></label>
        <input asp-for="email"/>
        <br />
        <label asp-for="password"></label>
        <input asp-for="password"/>
        <br />
        Remember Me:<input asp-for="Rememberme" />
        <input type="submit" value="Login" />
    </form>
</body>
</html>
```


Common Vulnerabilities:

- Vulnerability is a cyber-security term that refers to a flaw in a system that can leave it open to attack.
- A website vulnerability is a weakness or misconfiguration in a website or web application code that leaves security exposed to a thread and allows an attacker to gain some level of control of the site, and possibly the hosting server.
- Most vulnerabilities are exploited through automated means, such as vulnerability scanners and botnets.
- Cybercriminals create specialized tools that scour the internet for common and publicized vulnerabilities.
- Once found, these vulnerabilities are then exploited to steal data, distribute malicious content, or inject defacement and spam content into the vulnerable site.

- Some of the Security vulnerability that are frequently exploited by attackers includes
  1. **Cross-site Request Forgery attacks (XSRF or CSRF)**
     - Cross-site request forgery (also known as XSRF or CSRF) is an attack against web-hosted apps whereby a malicious web app can influence the interaction between a client browser and a web app that trusts that browser.
     - These attacks are possible because web browsers send some types of authentication tokens automatically with every request to a website. This form of exploit is also known as a *one-click attack* or *session riding* because the attack takes advantage of the user's previously authenticated session.

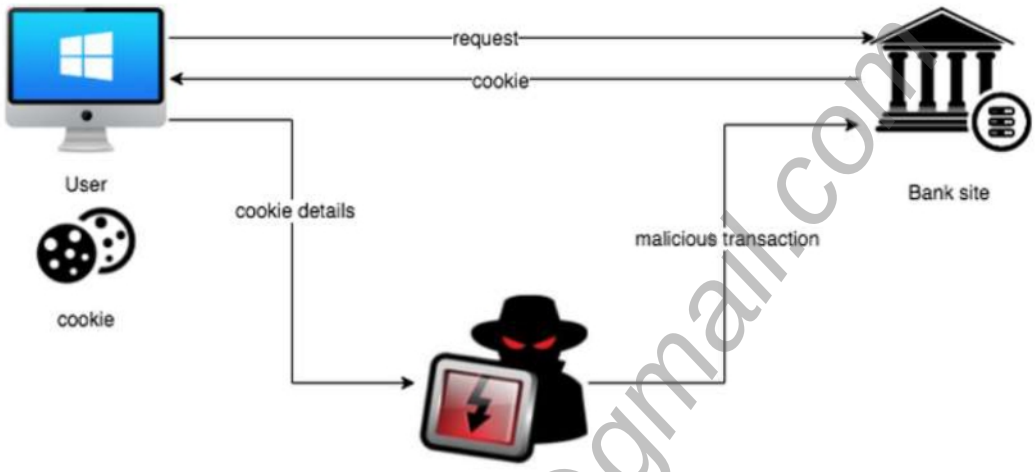


Fig: Middleman intercepting the session and cookies detail

     - An example of a CSRF attack:
       - A user signs into some website www.good-banking-site.com. The server authenticates the user and issues a response that includes an authentication cookie. The site is vulnerable to attack because it trusts any request that it receives with a valid authentication cookie.
       - The user visits a malicious site, www.bad-crook-site.com.
       - The malicious site, www.bad-crook-site.com, contains an HTML form similar to the following:

```
<h1>Congratulations! You're a Winner!</h1>
<form action="http://good-banking-site.com/api/account" method="post">
    <input type="hidden" name="Transaction" value="withdraw">
    <input type="hidden" name="Amount" value="1000000">
    <input type="submit" value="Click to collect your prize!">
</form>
```

       - Notice that the form's action posts to the vulnerable site, not to the malicious site. This is the "cross-site" part of CSRF.
       - The user selects the submit button. The browser makes the request and automatically includes the authentication cookie for the requested domain, www.good-banking-site.com.
       - The request runs on the www.good-banking-site.com server with the user's authentication context and can perform any action that an authenticated user is allowed to perform.
       - In addition to the scenario where the user selects the button to submit the form, the malicious site could:
         - Run a script that automatically submits the form.
         - Send the form submission as an AJAX request.
         - Hide the form using CSS.
     - CSRF attacks are possible against web apps that use cookies for authentication because:
       - Browsers store cookies issued by a web app.
       - Stored cookies include session cookies for authenticated users.
       - Browsers send all of the cookies associated with a domain to the web app every request regardless of how the request to app was generated within the browser.
     - Users can guard against CSRF vulnerabilities by taking precautions:

1. Sign off of web apps when finished using them.
2. Clear browser cookies periodically.

- However, CSRF vulnerabilities are fundamentally a problem with the web app, not the end user. The most common approach to defending against CSRF attacks is to use the *Synchronizer Token Pattern* (STP). STP is used when the user requests a page with form data:
  - The server sends a token associated with the current user's identity to the client.
  - The client sends back the token to the server for verification.
  - If the server receives a token that doesn't match the authenticated user's identity, the request is rejected.
- The token is unique and unpredictable. All of the forms in ASP.NET Core MVC and Razor Pages templates generate antiforgery tokens. Razor Pages are automatically protected from XSRF/CSRF.
- The following pair of .cshtml view example generates ant forgery token implicitely/automatically.

```
<form action="Account/WithDrawAmount" method="post">
    ...
</form>
                                        or
<form asp-controller="Account" asp-action="WithDrawAmount" method="post">
    ...
</form>
                                        or
@using (Html.BeginForm("WithDrawAmount", "Acount", FormMethod.Post))
{
    ...
}
```

- We also can add an antiforgery token to a <form> element explicitely also as below

```
<form action="Account/WithDrawAmount" method="post">
    @Html.AntiForgeryToken()
</form>

Or
<form asp-controller="Home" asp-action="WithDrawAmount" method="post" asp-antiforgery="true">
    ...
</form>
```

- In each of the preceding cases, ASP.NET Core adds a hidden form field similar to the following in the generated HTML.

```
<input name="__RequestVerificationToken" type="hidden" value="CfDJ8NrAkS ... s2-m9Yw">
```

- ASP.NET Core includes these filters for working with antiforgery tokens:

1. ValidateAntiForgeryToken
   - ValidateAntiForgeryToken is an action filter that can be applied to an individual action, a controller, or globally.
   - Requests made to actions that have this filter applied are blocked unless the request includes a valid antiforgery token.

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult WithDrawAmount()
{
}
```

   So the action method WithDrawAmount() is secured from anti forgery attack. If the antiforgery token is not available or if the token is invalid, the validation will fail and action method WithDrawAmount() will not be allowed to access i.e. Access Denied

2. IgnoreAntiforgeryToken

   If the ValidateAntiForgeryToken attribute is applied across the app's controllers, it can be overridden with the IgnoreAntiforgeryToken attribute.

```
[HttpPost]
```

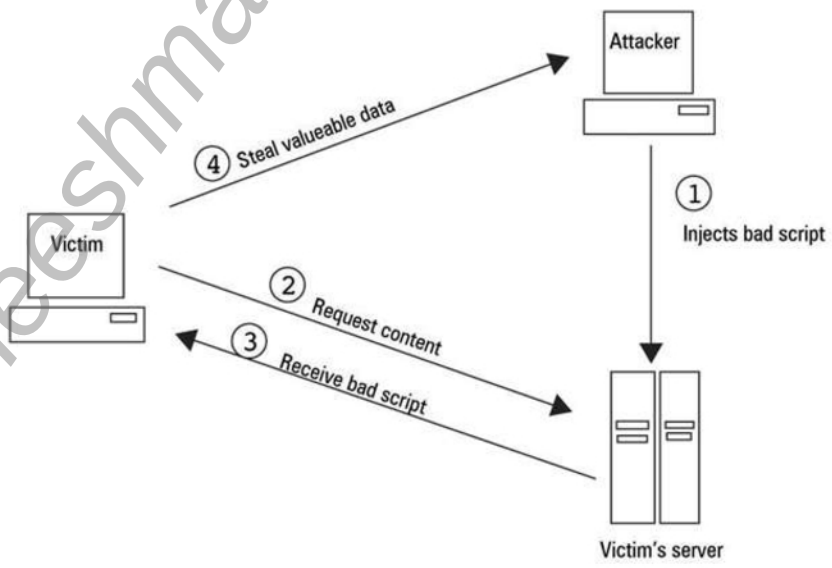
```
[ValidaAntiforgeryToken]
public class AccountController : Controller
{
    [IgnoreAntiforgeryToken]
    public IActionResult Login()
    {
    }
    public IActionResult WithDrawAmount()
    {
    }
}
```

So the action method WithDrawAmount() is secured from anti forgery attack but for the action method Login() the anti forgery validation is ignored as it is decorated with IgnoreAntiForgeryToken filter.

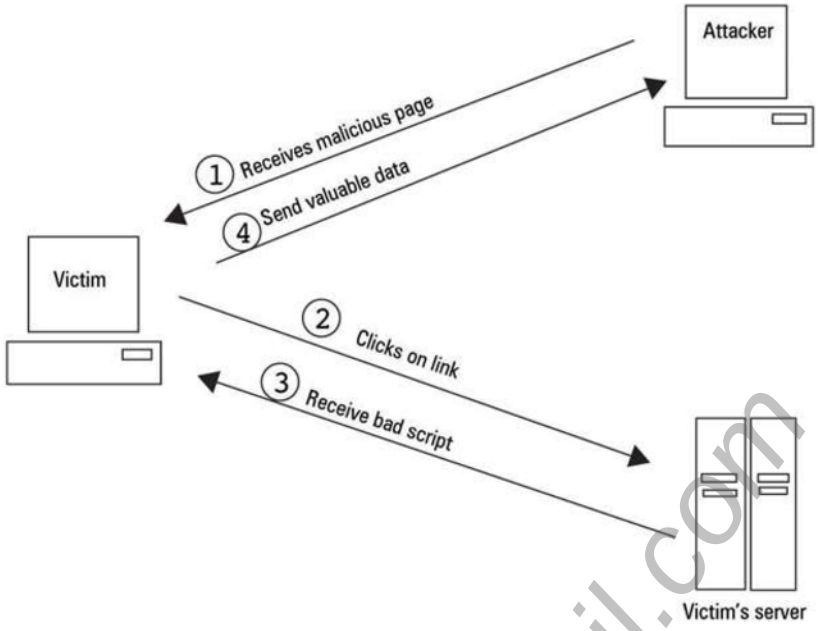
2. **Cross-site Scripting(XSS) Attacks**
   - Cross-Site Scripting (XSS) is a security vulnerability which enables an attacker to place client side scripts (usually JavaScript) into web pages.
   - When other users load affected pages the attacker's scripts will run, enabling the attacker to steal cookies and session tokens, change the contents of the web page through DOM manipulation or redirect the browser to another page.
   - Cross-site scripting is used to target website visitors, rather than the website or server itself.
   - Browsers are unable to discern whether or not the script is intended to be part of the website, resulting in malicious actions, including:
     - Session hijacking
     - Spam content being distributed to unsuspecting visitors
     - Stealing session data
   - +XSS vulnerabilities generally occur when an application takes user input and outputs it to a page without validating, encoding or escaping it.

Types

1. Stored XSS
   - Stored XSS occurs when user input is stored on the database server: such as in message field, comment field (the entire vulnerability is in server-site code)
   - And then victim is able to retrieve the stored data from the web app.
   - Script code execute on victim's browser.



2. REFLECTED XSS
   - Reflected XSS occurs when data provided by a web client is used immediately by server-site scripts scripts to generate a page of results for that user. if unvalidated user-supplied data is included in the resulting page without HTML encoding, this will allow client-site code to be injected into the dynamic page.



Protecting your application against XSS

- At a basic level XSS works by tricking your application into inserting a <script> tag into your rendered page. Developers should use the following prevention steps to avoid introducing XSS into their application.
  - 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. Untrusted data includes 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).
  - Before putting untrusted data inside an HTML element ensure it's HTML **encoded.** The HtmlEncode method applies HTML encoding to a string to prevent a special character to be interpreted as an HTML tag. For example, HTML encoding takes characters such as < and changes them into a safe form like <
  - 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 '.
  - 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.
  - Before putting untrusted data into a URL query string ensure its URL encoded
  - Validation can also be a useful tool in limiting XSS attacks. For example, a numeric string containing only the characters 0-9 won't trigger an XSS attack. Validation becomes more complicated when accepting HTML in user input. Parsing HTML input is difficult, if not impossible. Markdown, coupled with a parser that strips embedded HTML, is a safer option for accepting rich input. Never rely on validation alone. Always encode untrusted input before output, no matter what validation or sanitization has been performed.

Example:

- 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.

```
@str //will show encoded text
```

- **Tag helpers** will also encode input you use in tag parameters.
- If we have need to display the content without encoding, we can use following html helper method.

```
@Html.Raw(str)//the browser now can interpret  tag elements/scripting elements in str
```

```
@model IEnumerable< string>;
<html>
<body>
        @foreach (var str in Model)
        {
           @str
```

```
                    <br />
                    @Html.Raw(str)
                }
            </body>
            </html>
```

- In the above example, if we pass a string like
  1. Hello
  2. <scipt>alert("hy")</script>

  then, the first one @str will show these string as it is because The Razor engine used in MVC automatically encodes all output sourced from variables. But in case of @Html.Raw(str), this will display Hello in bold form for the first one and for the second it will display pop up javascript message.

3. SQL Injection attacks

- SQL injection vulnerabilities refer to areas in website code where direct user input is passed to a database.
- So here code puts invalidated input directly into an SQL statement, instead of using a parameterized query (i.e. one with placeholders.)
- Bad actors utilize these forms to inject malicious code, sometimes called *payloads*, into a website's database. This allows the cybercriminal to access the website in a variety of ways, including:
  1. Injecting malicious/spam posts into a site
  2. Stealing customer information
  3. Bypassing authentication to gain full control of the website

- Due to its versatility, SQL injection is one of the most commonly exploited website vulnerabilities.
- Fortunately, SQL injection attacks are one of the easiest to mitigate, because parameterized query support is built into every database access library.

- For example, Consider the following action method that validate user login.

```
[HttpPost]
public IActionResult SubmitLogin1(string uname, string pwd)
{
            SqlConnection con = new SqlConnection(@"Data Source=.\SQLEXPRESS;Initial
                                    Catalog=db_Mac1; Integrated Security=True");
            con.Open();
            SqlCommand cmd = new SqlCommand("select * from tbl_login where uname='"+uname+"' and
                                    password='"+pwd+"'", con);
            SqlDataReader dr = cmd.ExecuteReader();
            if (dr.Read())
            {
                return Content("Login Successful");
            }
            else
            {
                return Content("Login Inuccessful");
            }
}
```

- The above action method is vulnerable to SQL injection attack. It is because **we've used the form input values** *name and pwd* **with no data validation at all including Empty form validations**. Nothing bad will happen if we're sure that this value will only come from trusted sources, but this is not always.
- If the attacker doesn't know what the username is then he/she simply provides a 'or 1=1 for the username, So, when the user presses the submit button then the resulting query will be formed as

```
select * from tbl_login where Username='' or 1=1--'and Password=''
```

True Always Commented

- The above query will return entire rows from table tbl_login if there is atleast one row in the table thereby displays "Login Successful" message.
- Anything placed into that TextBox control will be added to your SQL string. This situation invites a hacker to replace that string with something malicious.

Preventing SQL injection:

- Don't accept the user input as it is. Do the sanitization.
- If you are using an ORM, an Object Relational Mapper like Entity Framework, you are normally covered.
- Using parameterized query will prevent such injection. Using parameterized queries is a three-step process:
  1. Construct the SqlCommand command string with parameters.
  2. Declare a SqlParameter object, assigning values as appropriate.
  3. Assign the SqlParameter object to the SqlCommand object's Parameters property.

  The following program demonstrates the concept

```
[HttPost]
public IActionResult SubmitLogin(string uname, string pwd)
{
        SqlConnection con = new SqlConnection(@"Data Source=.\SQLEXPRESS;Initial Catalog=db_Mac1;
        Integrated Security=True");
        con.Open();
        SqlCommand cmd = new SqlCommand("select * from tbl_login where uname=@uname and
        password=@pwd",con);
        cmd.Parameters.AddWithValue("@uname", uname);
        cmd.Parameters.AddWithValue("@pwd", pwd);
        SqlDataReader dr = cmd.ExecuteReader();
        if (dr.Read())
        {
                return Content("Login Successful");
        }
        else
        {
                return Content("Login Inuccessful");
        }
}
```

4. Open Redirect Attacks

- A web app that redirects to a URL that's specified via the request such as the query string or form data can potentially be tampered with to redirect users to an external, malicious URL. This tampering is called an open redirection attack.
- Web applications frequently redirect users to a login page when they access resources that require authentication. The redirection typically includes a return URL query string parameter so that the user can be returned to the originally requested URL after they have successfully logged in. After the user authenticates, they're redirected to the URL they had originally requested.
- Because the destination URL is specified in the query string of the request, a malicious user could tamper with the query string. A tampered query string could allow the site to redirect the user to an external, malicious site. This technique is called an open redirect (or redirection) attack.
- So, whenever your application logic redirects to a specified URL, you must verify that the redirection URL hasn't been tampered with. **ASP.NET Core has built-in functionality to help protect apps from open redirect (also known as open redirection) attacks.**
- **The following figure illustrates this attack process.**

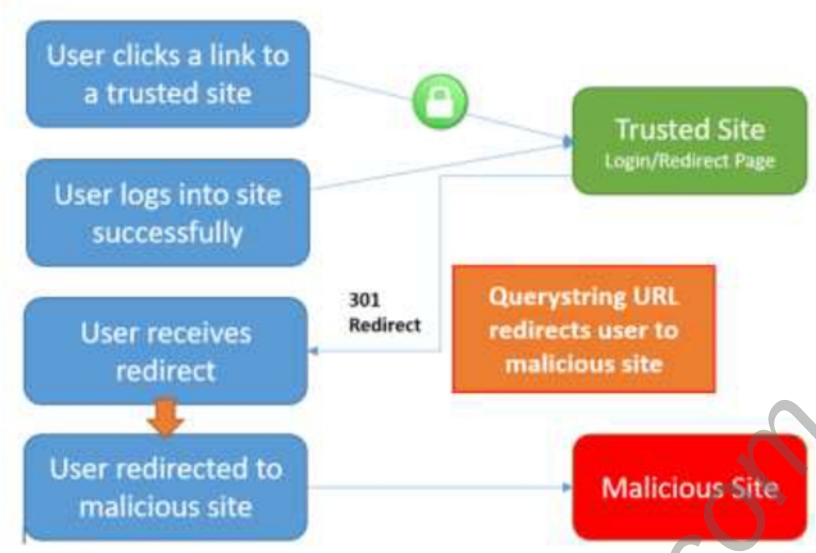


Fig: Open Redirect Process

- A malicious user can develop an attack intended to allow the malicious user access to a user's credentials or sensitive information. To begin the attack, the malicious user convinces the user to click a link to your site's login page with a returnUrl querystring value added to the URL. For example, consider an app at contoso.com that includes a login page at http://contoso.com/Account/LogOn?returnUrl=/Home/About. The attack follows these steps:
  1. The user clicks a malicious link to http://contoso.com/Account/LogOn?returnUrl=http://contoso1.com/Account/LogOn (the second URL is "contoso1.com", not "contoso.com").
  2. The user logs in successfully.
  3. The user is redirected (by the site) to http://contoso1.com/Account/LogOn (a malicious site that looks exactly like real site).
  4. The user logs in again (giving malicious site their credentials) and is redirected back to the real site.
- The user likely believes that their first attempt to log in failed and that their second attempt is successful. The user most likely remains unaware that their credentials are compromised.

Protecting against open redirect attacks

- If your application has functionality that redirects the user based on the contents of the URL, ensure that such redirects are only done locally within your app (or to a known URL, not any URL that may be supplied in the querystring).
  1. Use the LocalRedirect helper method from the base Controller class:

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

     LocalRedirect will throw an exception if a non-local URL is specified. Otherwise, it behaves just like the Redirect method.
  2. Use the IsLocalUrl method to test URLs before redirecting:

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

The IsLocalUrl method protects users from being inadvertently redirected to a malicious site. You can log the details of the URL that was provided when a non-local URL is supplied in a situation where you expected a local URL.

Assignment

1. **Role Based Authorization Vs Claim Based Authorization**

- Claims are a method of providing information about a user, and roles are a description of a user by way of which roles they belong.
- Claims are generally more useful because they can contain arbitrary data -- including role membership information. E.g. whatever is useful for the given application.
- Claim Based identities are more useful, but tend to be trickier to use because there's a lot of setup involved for acquiring the claims in the first place. RBAC identities are less useful because they are just a collection of roles, but they are generally easier to setup.
- In ASP.NET Core, a role is just a claim with type Role.
- Claims based authorization is relatively new and is the recommended approach. Role based authorization is still supported in asp.net core for backward compatibility but is not the recommended approach.

2. **Cookie-based authentication vs Token Based authentication**

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. 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. Don't be concerned about CSRF vulnerability if the token is stored in the browser's local storage. CSRF is a concern when the token is stored in a cookie.