Beveiliging van MVC core web applicaties

ASP.NET Core MVC is a web development framework, dat veel wordt gebruikt door ontwikkelaars over de hele wereld om webtoepassingen te ontwikkelen. Deze webapplicaties zijn echter kwetsbaar gebleken voor aanvallen uit verschillende bronnen, en het is onze verantwoordelijkheid om onze gegevens te beschermen. Hier worden de voornaamste aanvallen en hoe je je ASP.NET Core MVC-webtoepassing ertegen kan beveiligen. Hieronder volgen de aanbevolen procedures om kwetsbaarheden in uw toepassingen te voorkomen:

1. Cross-Site Scripting (XSS)
2. SQL Injection
3. Cross-Site Request Forgery (CSRF)
4. Custom Error Page voor Error Handling
5. Version Discloser
6. SSL (Secure Sockets Layer) en HSTS
7. XXE (XML External Entity) Aanval
8. Improper Authentication en Session Management
9. Sensitive Data Exposure and Audit Trail
10. File Upload Validatie

1. Cross-Site Scripting (XSS)

Wat is Cross-Site Scripting?

Het injecteren van een kwaadaardige (javascipt)script via het invoer- / formulierveld van een webpagina met de bedoeling vertrouwelijke informatie te stelen, zoals inloggegevens of andere authenticatiegegevens, cookies en sessiewaarden, wordt een cross-site scripting (XSS) attack of aanval genoemd.

Scriptinjectie kan op de volgende manieren gebeuren:

a. Form Inputs
b. URL Query Strings
c. HTTP Headers

Hoe beveiligen tegen Cross-Site Scripting aanvallen?

Cross-site scripting-aanvallen kunnen op de volgende manieren worden voorkomen:

a. Reguliere Expressie Attributen
b. HTML Encoding
c. URL Encoding

a. Reguliere Expressie Attributen gebruiken

Je kan reguliere expressies gebruiken om de formulierinvoer van de gebruiker te valideren en Hiermee schadelijke tekens of symbolen weigeren en/of alleen acceptabele vereiste tekens in het invoerveld toestaan.

Meer informatie over reguliere expressie attributen en RegEx vind je hier: RegularExpressionAttribute Class en The Regular Expression Object Model.

b. HTML Encoding

De MVC Razor engine encodeert automatisch alle invoer, zodat scripts die geïnjecteerd worden via een form-field nooit kunnen worden uitgevoerd.

Opmerking: Indien je 'raw HTML text' zonder encoding wil plaatsen in een razor view, gebruik dan de syntax **@Html.Raw()**

URL Encoding

Meestal gebruiken we 'plain text' in URL-querystrings. Deze kunnen worden misbruikt voor XSS-aanvallen. Je kan deze soort aanvallen voorkomen door de invoer van de queryparameter(s) in de URL coderen.

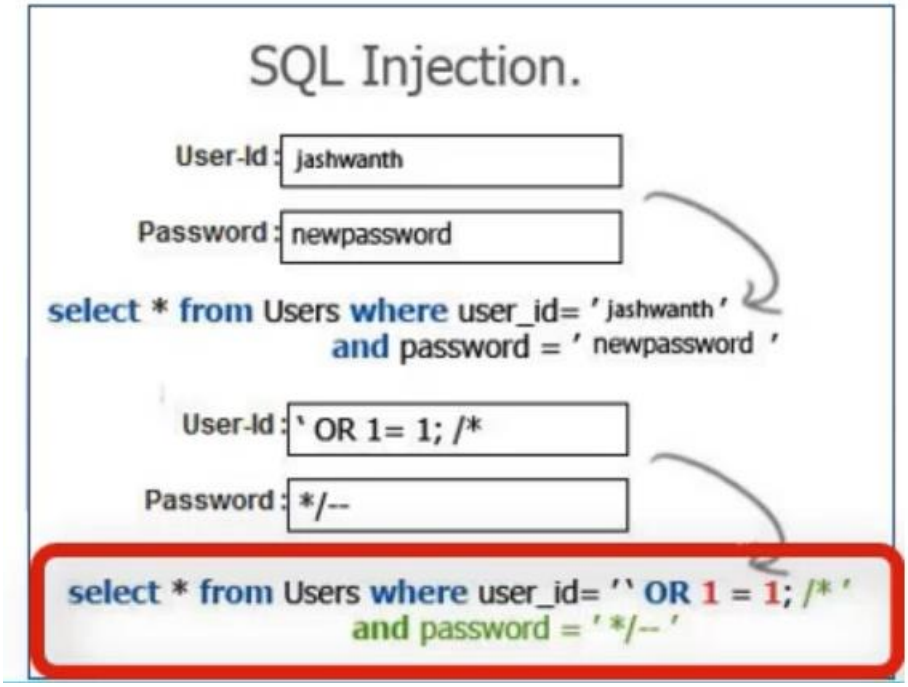
Er bestaat een ingebouwde library in NuGet voor het coderen en decoderen van tekst:

```
1  string encodedValue = System.Net.WebUtility.UrlEncode("raw-string-text");
2  string decodedValue = System.Net.WebUtility.UrlDecode(encodedValue);
```

SQL Injection

Wat is SQL Injection?

Het is een gevaarlijke aanval waarbij ongeautoriseerde gebruikers kwaadaardige SQL-code injecteren via een webpagina die vervolgens in uw database wordt uitgevoerd, waardoor de aanvallers toegang krijgen tot vertrouwelijke informatie die erin is opgeslagen of databasegegevens kunnen wijzigen of verwijderen.



zeer gevaarlijk om deze soort query te gebruiken: String Sqlquery= “select * from users where user_id =” + txtuser + “ and password=’” +txtpwd

Meer informatie: [SQL injection]

Hoe SQL-injectie voorkomen

Een SQL injection aanval kan worden voorkomen op de volgende manieren:

a. valideer invoer
b. gebruik stored procedures
c. gebruik geparametriseerde queries
d. gebruik Entity Framework of een andere ORM (bv Dapper)
e. gebruik de minst geprivileerde DB toegang vanaf websites
f. Bewaar gegevens geëncrypteerd

Valideer invoer

Defending against SQL injection by validating inputs requires the following actions:

Om u te beschermen tegen SQL-injectie door invoer te valideren, zijn de volgende acties vereist:

• Sta geen speciale tekens toe die betrokken zijn bij SQL-scripts.

• Gebruik reguliere expressies en gegevensannotaties om invoer te valideren.

- Valideer de gebruikersinvoer aan zowel de clientzijde serverzijde
- Sta geen speciale tekens toe die gebruikt worden in een SQL-script
- Gebruik reguliere expressies en gegevensannotaties om invoer te valideren

Gebruik Stored Procedures

Het gebruik van opgeslagen procedures voorkomt SQL-injectie, maar we moeten de invoerparameters die aan de opgeslagen procedures zijn doorgegeven, nog steeds valideren.

Gebruik Geparametriseerde Queries

Als je inline-query's wilt gebruiken, moet je query's met parameters gebruiken om SQL-injectie te voorkomen.

In het volgende codevoorbeeld ziet je een query met parameters:

```
SqlConnection sqlConnection = new SqlConnection();
SqlCommand cmd = new SqlCommand("select * from Users where id = @id", sqlConnection);
SqlParameter param = new SqlParameter();
param.ParameterName = "@id";   // Here using id value as parameter.
param.Value = id;
cmd.Parameters.Add(param);
```

Gebruik Entity Framework of een andere ORM (bv Dapper)

ORM staat voor **object-relationele mapper,** die SQL-objecten mapt op uw toepassings-klasseobjecten. Als je Entity Framework correct gebruikt, ben je niet vatbaar voor SQL-injectie-aanvallen omdat Entity Framework intern gebruikmaakt van geparametriseerde query's.

Gebruik de minst geprivileerde DB toegang vanaf websites

We moeten de machtigingen voor DB-gebruikers beperken voor tabellen met vertrouwelijke gegevens. We moeten bijvoorbeeld de machtigingen voor invoegen, bijwerken en verwijderen beperken voor tabellen met betrekking tot betalingen en transacties, en we moeten ook de machtigingen beperken voor tabellen waarin de persoonlijke gegevens van een gebruiker worden opgeslagen.

Als een gebruiker alleen met Select-query's werkt, moeten we alleen toestemming geven voor de Select-instructie en mogen we geen toestemming geven voor Insert-, Update- en Delete-instructies.

Bewaar enkel geëncrypteerde Data

Bewaar geen confidentiële informatie zoals wachtwoorden in gewone tekst in een database. Bewaar wachtwoorden steeds in een geëncrypteerd vorm.

Cross-Site Request Forgery (CSRF)

Wat is Cross-Site Request Forgery?

Een aanvaller doet zich voor als “trusted source” en stuurt sends some vervalste gegevens naar een website. De website verwerkt de vervalste gegevens omdat het gelooft dat het uit betrouwbare bron komt.

Als je wil weten hoe zo’n aanval in het werk gaat, kan je hier meer informatie vinden: Cross-Site Request Forgery (CSRF).

Voorbeeld van een aanval: wanneer een gebruiker geld overschrijft van één bankrekening naar een andere, is er een “trusted connection” gelegd tussen de gebruikere en de website van de bank nadat de gebruiker is ingelogd.

De gebruiker klikt bv terzelfdertijd op een kwaadaardige link in bv een email van een aanvaller.

De aanvaller maakt misbruik van de beveiligde connective tussen de gebruikere en de website van de bank en kan zelf bv geld transferten verrichten.

Voor dit soort aanvallen ligt een risico bij de server side (web applicatie), niet de kant van de eindgebruiker.

Hoe Cross-Site Request Forgery voorkomen?

Dit soort aanvallen kan worden voorkomen dmv een **AntiForgeryToken**.

We gebruiken hievoor de HTML tag helper **asp-antiforgery** in een HTML /Razor attribuut en zetten de waarde op **true**, er wordt dan een anti-forgery token gegenereerd. We moeten eveneens een attribuut **[ValidateAntiForgeryToken]** plaatsen boven de form post action methode die nagaat of een geldig token is gegenereerd.

```
<div class="col-md-4">
    <section>
        <form asp-route-returnurl="@ViewData["ReturnUrl"]" method="post" asp-antiforgery="true">
            <h4>Use a local account to log in.</h4>
            <hr />
            <div asp-validation-summary="All" class="text-danger"></div>
            <div class="form-group">...</div>
            <div class="form-group">...</div>
            <div class="form-group">...</div>
            <div class="form-group">...</div>
            <div class="form-group">...</div>
        </form>
    </section>
</div>
```

Zet de asp-antiforgery Tag Helper op true

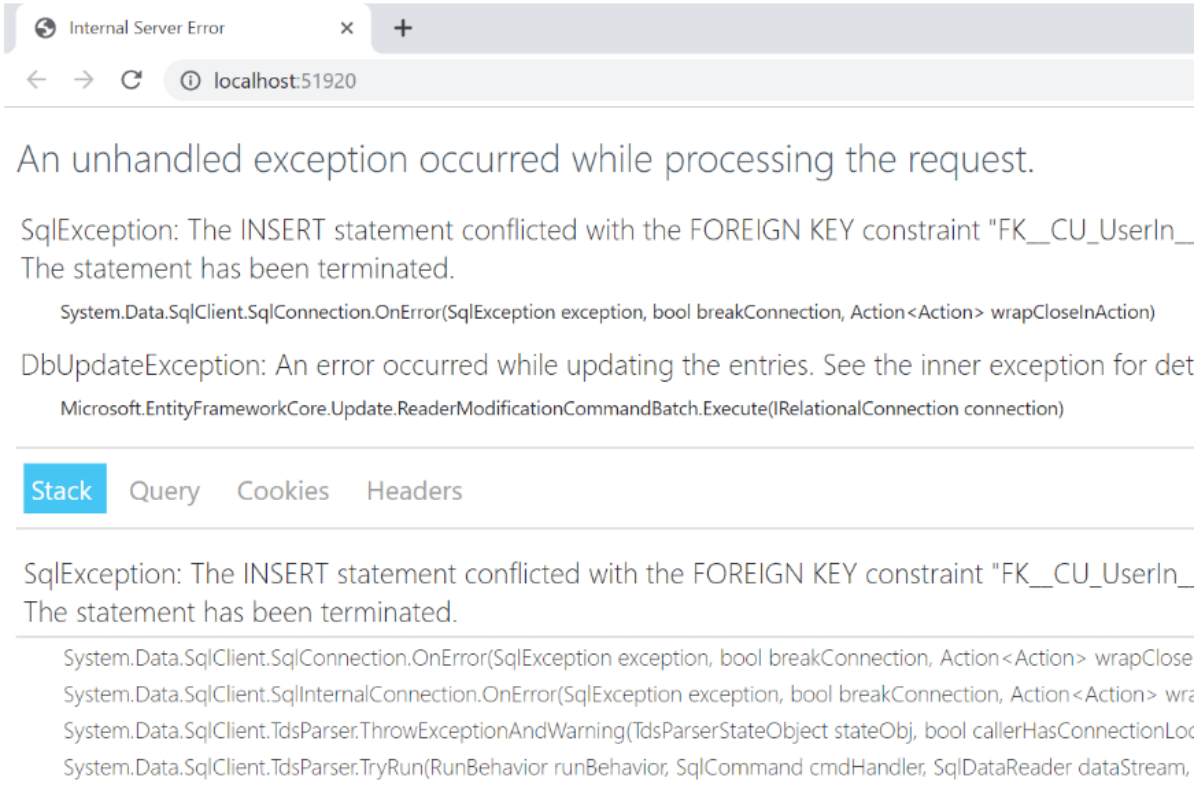
```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)...

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

Voeg het [ValidateAntiForgeryToken] attribuut toe aan de [HttpPost] Methode van de Form Action.

Gecustomiseerde Error Page voor Error Handling

Bij foutmeldingen kan er gevoelige informatie worden getoond in de foutboodschap aan eindgebruikers, zoals bv database configuratie info, tabel namen, stored procedures, data structures en programmeer code structuren.



Voorbeeld van gevoelige foutmelding informatie die wordt getoond

Hoe to Custom Error Handling toevoegen

Er zijn twee manieren om custom error-handling pagina's te voorzien in een ASP.NET Core applicatie.

1) De eerste aanpak is om een custom error-handling attribuut te voorzien dmv using **ExceptionFilterAttribute**. Dit attribuut handelt te error af.

Je kan de **OnException** methode overridden en exceptions in text-bestanden schrijven bv of de foutmelding in de database bewaren. Via de volgende code kan dan geredirect worden naar de custom error pagina:

```
01
02
03
04
05
06
07
08
09
10
```

```
  public class CustomExceptionFilterAttribute: ExceptionFilterAttribute
  {
     //write the code logic to store the error here

var result = new RedirectToRouteResult(
     new RouteValueDictionary
     {
        {"controller", "Error"}, {"action", "CustomError"}
     });
  }
```

We moeten deze filter registreren in de **ConfigureServices** methode van de **Startup.cs** file.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
    //Registering CustomExceptionFilterAttribute
    options.Filters.Add(typeof(CustomExceptionFilterAttribute));

    ...
    ...
    )
}
```

2) De 2de methode is het gebruiken van **UseExceptionHandler** in de productie omgeving:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseHttpsRedirection();
    }
    else if (env.IsProduction())
    {
        app.UseExceptionHandler("/error/customerror");
    }
```

We tonen dan de developer error page met gevoelige informatie dan enkel in development omgeving en er wordt een custom error pagina (zonder gevoelige info) getoond in te browser in een productie-omgeving.

Gebruik SSL (Secure Sockets Layer) en HSTS

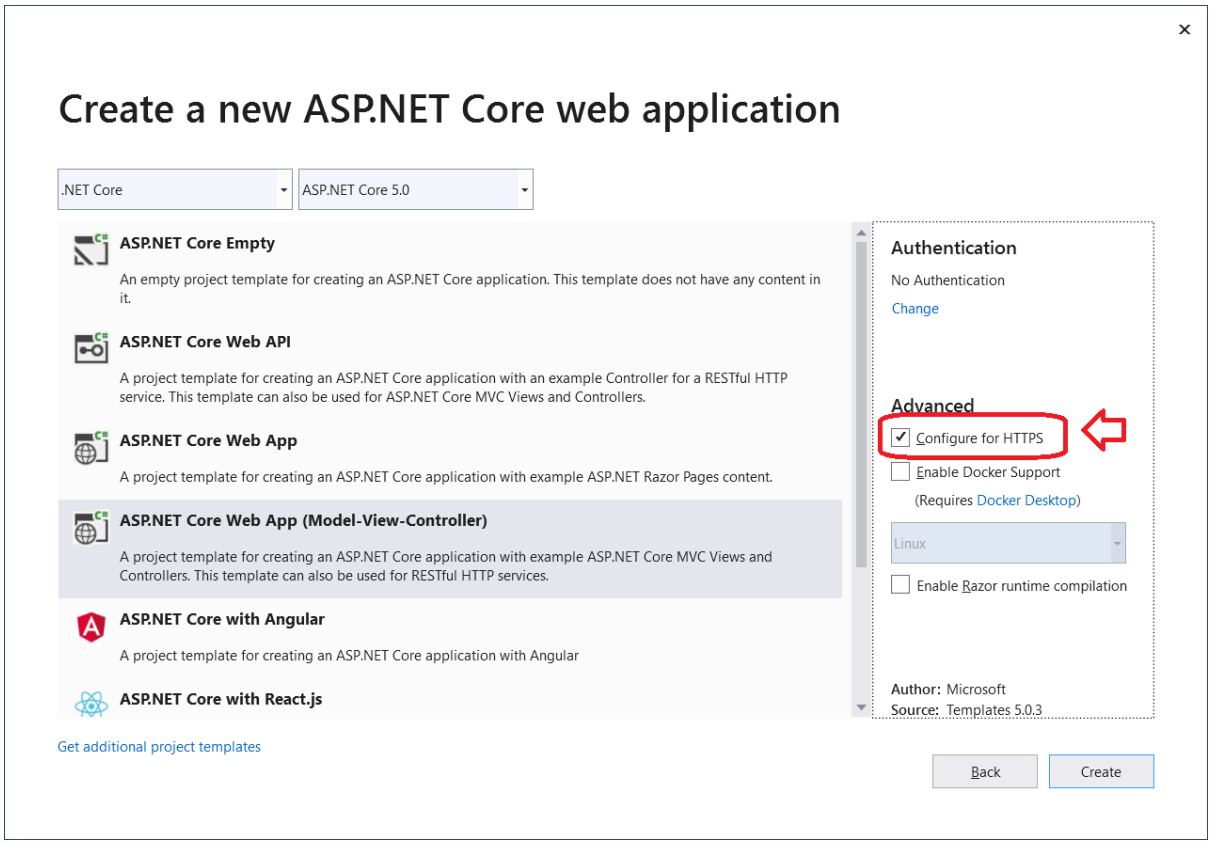
Wat is SSL?

SSL staat voor Secure Sockets Layer, deze verzekert een beveiligd geëncrypteerde connectie tussen client en server. Bij SSL worden de HTTP requests vanuit de client browser en HTTP Responses van de server geëncrypteerd.

We kunnen HTTPS (HyperText Transfer Protocol Secure) gebruiken om een ASP.NET Core applicatie te beveiligen.

In ASP.NET Core 2.1 en ook latere versies kunnen we gemakkelijk een web applicatie aanmaken die voor HTTPS is geconfigureerd.

Wanneer je een nieuwe web applicatie aanmaakt met Visual Studio heb je een optie om HTTPS te gebruiken:



Configureren van een ASP.NET Core Web App voor HTTPS.

Wat is HSTS (HTTP Strict Transport Security)?

HSTS is een web security policy die je web applicatie beschermt tegen downgrade protocol aanvallen en cookie hijacking. Het dwingt de web server om te communiceren over een HTTPS connective en weigert onveilige HTTP connecties.

De ASP.NET Core template voegt standaard middleware toe voor HSTS. Het is niet aangeraden om dit te gebruiken in development environment, Aangezien de browser de HSTS header cachet.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseHttpsRedirection();
    }
    else
    {
        app.UseExceptionHandler("/error");
        var options = new RewriteOptions().AddRedirectToHttps();

        app.UseRewriter(options);
        app.UseHsts();
    }
```

Zetten van HSTS Policy in ASP.NET Core Code

In ASP.NET Core kan je eveneens middleware toevoegen die requests redirect van niet-beveiligde HTTP naar de beveiligde HTTPS:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpsRedirection(options =>
    {
        options.RedirectStatusCode = StatusCodes.Status307TemporaryRedirect;
        options.HttpsPort = 8080;
    });
}
```

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

    app.UseHttpsRedirection();
```

Zetten van HTTPS Redirection in ASP.NET Core Code

XXE (XML External Entity) Attack

Indien je web applicatie code XML files parst die geüploaded is door een eindgebruiker, loop je het risico voor een XXE (XML External Entity) aanval. Een slecht geconfigureerde XML parser verwerkt een XML input met kwaadaardige XML code of een referentie naar een externe entiteit. Deze soort aanvallen kunnen een denial-of-service(DOS) aanval veroorzaken door entiteiten te injecteren binnen entiteiten, waardoor de server zodanig belast wordt dat de server platgaat.

Hoe deze aanval voorkomen?

Indien je **XmlTextReader** om XML files te parsen, moet je de **DtdProcessing** property op **Prohibit** of **Ignore** zetten.

Indien je deze property op **Parse** zet, zal het de DTD specificaties in het XML document verwerken.Bij default staat deze property **DtdProcessing** op **Parse** . Het is dus kwetsbaar voor aanvallen.

Indien je deze property op **Prohibit** zet, zal een exception worden opgeworpen indien een DTD (Document Type Definition) is gespecifieerd.

Indien je deze property op **Ignore** zet, zal een DTD specificatie in het document genegeerd worden en het verwerken van het XML document blijft mogelijk.

```
public IActionResult ReadXML()
{
    XmlTextReader reader = new XmlTextReader("fileName");
    reader.DtdProcessing = DtdProcessing.Ignore;
    while (reader.Read())
    {
        var data = reader.Value;
    }
    return this.View();
}
```

Zetten van DtdProcessing property op Ignore.

https://web-in-security.blogspot.com/2016/03/xxe-cheat-sheet.html

https://medium.com/@klose7/xxe-attacks-part-2-xml-dtd-related-attacks-a572e8deb478

Onveilige Authenticatie en Session Management

De meeste web applicaties hebben een authentication module, deze zou met de nodige voorzichtigheid moeten worden geschreven. Let op dat de authenticatie cookies worden verwijderd na een logout, anders kunnen aanvallers user-credentials stelen.

De volgende fouten maken aanvallers het mogelijk om gegevens te stelen:

a. Onveilige connectie (zonder SSL).
b. voorspelbare login credentials.
c. Bewaren van ongeëncrypteerde credentials.
d. Niet goed geprogrammeerde logouts.

Hoe deze fouten voorkomen?

a. Verwijder cookies na een logout.
b. Beveilig cookies en sessions dmv SSL.
c. Beveilig cookies door ze HttpOnly te zetten.

De volgende code verwijdert session waarden en authentication cookies (.AspNetCore.Session) nadat een gebruiker uitlogt :

```
//Remove session
HttpContext.Session.Clear();

// Removing Cookies
CookieOptions option = new CookieOptions();
if (Request.Cookies[".AspNetCore.Session"] != null)
{
    option.Expires = DateTime.Now.AddDays(-1);
    Response.Cookies.Append("Myapplication.Session", "", option);
}
```

Om **HttpOnly** op cookies in ASP.NET Core te zetten:

```
1   CookieOptions option = new CookieOptions {Expires = DateTime.Now.AddHours(24), HttpOnly = true};
```

HttpOnly zorgt dat de cookie niet kan worden gelezen vanaf client-side scripts.

Om **HttpOnly** te kunnen gebruiken, zet je eerst de **UseCookiePolicy** methode in **Startup.cs** :

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())...
    else...
    app.UseCookiePolicy(new CookiePolicyOptions
    {
        HttpOnly = HttpOnlyPolicy.Always,
        Secure = CookieSecurePolicy.Always,
        MinimumSameSitePolicy = SameSiteMode.None,
    });
```

Configuratie van UseCookiePolicy.

Gevoelige gegevens beveiligen

Bij het beheren van gevoelige gegevens, bv persoonlijke informatie van eindgebruikers, moeten we zorgen dat hackers niet aan deze gegevens kunnen.

Hoe gevoelige gegevens beveiligen?

a. Stuur gevoelige gegevens steeds in geëncrypteerd formaat dmv een encryptie algoritme. Bv een adres van een klant
b. Gebruik SSL en access web applicaties in productie in HTTPS mode.
c. Bewaar geen gevoelige gegevens in database, paswoorden,…Indien je dit toch nodig hebt, gebruik een sterk encryptiealgoritme om de gegevens te bewaren.

Audit Trail

Zorg voor monitoring van je web applicatie in productie omgeving op regelmatige tijdsstippen. Je kan logs opzetten, bv IIS logs, of je kan in text-files of database loggen. Behalve aanvallen kan je via de logs eveneens fouten of performantieproblemen detecteren.

File Upload Validatie

Indien je web app een file upload form control voorziet, krijgen aanvallers de mogelijkheid attackers om kwaadaardige script files te uploaden.

Eerst valideer je de file extensie, maar dit is niet voldoende.

Welke validaties zijn nodig?

a. Controlleer de file extensie.
b. Controleer de content type en file bytes.
c. Scan de file met een anti-virus

https://www.syncfusion.com/blogs/post/10-practices-secure-asp-net-core-mvc-app.aspx

https://www.red-gate.com/simple-talk/dotnet/net-development/using-auth-cookies-in-asp-net-core/

https://www.codeproject.com/Tips/5265736/Scan-File-s-for-the-Virus-before-Uploading-to-Serv