Exercice d’injection SQL

# Création du projet

Démarrer Visual Studio

Créer un projet de type « Blazor Web Assembly » (en fonction de votre version de Visual Studio, vous verrez peut-être juste le modèle « Application Blazor », et le choix entre le mode Web Assembly ou Server Side sera fait un peu plus loin dans l’assistant)

Une image contenant texte

Description générée automatiquement

Décocher « Placer la solution et le projet dans le même répertoire »

Choisir les options .NET 6.0, Web Assembly si nécessaire, décocher le HTTPS qui ne sera pas utile et activer l’option d’hébergement sur ASP.NET :

Une image contenant texte

Description générée automatiquement

Lancer l’application par défaut avec la touche F5 pour vérifier, avant de faire quoi que ce soit de plus, que tout fonctionne bien. Si nécessaire, changez dans les options de lancement le navigateur pour pointer sur Chrome ou Edge :

Une image contenant texte

Description générée automatiquement

# Ajout des dépendances nécessaires

Utiliser les dépendances Nuget sur le projet Server :

Une image contenant texte

Description générée automatiquement

## Depuis octobre 2022

Le package Microsoft.EntityFrameworkCore.Sqlite.Core contient désormais toutes les dépendances nécessaires ; il suffit donc de l’installer et ceci remplace les deux ajouts de packages ci-dessous, gardés uniquement pour référence.

## Avant octobre 2022

Ajouter une dépendance sur le paquet Microsoft.Data.SQLite.Core :

Une image contenant texte

Description générée automatiquement

Puis, rajouter une dépendance sur le paquet SQLitePCLRaw.bundle\_e\_sqlite3 :

Une image contenant texte, capture d’écran, moniteur

Description générée automatiquement

Ces dépendances vont nous permettre d’avoir accès à une base de données locale et très légère pour l’exercice, avec très peu de paramétrage.

# Initialisation de la base de données

Afin de rendre sans risque toutes les expériences, le contenu complet de la base de données sera réinitialisé à chaque démarrage du serveur.

Dans le Startup.cs du projet Server, rajouter l’exemple de code ci-dessous au début de la fonction Configure:

using (var conn = new SqliteConnection("Data Source=test.db"))

{

conn.Open();

var commande = conn.CreateCommand();

commande.CommandText = "DROP TABLE IF EXISTS PERSONNES; CREATE TABLE PERSONNES (nom VARCHAR(32), prenom VARCHAR(32), age INT)";

commande.ExecuteNonQuery();

}

Ceci aura pour effet de provoquer une erreur de compilation et il faudra rajouter en haut du fichier l’instruction suivante :

using Microsoft.Data.Sqlite;

A partir de là, vous avez toute latitude pour créer de nouvelles tables, nouvelles colonnes si vous le souhaitez. Si toutefois vous préférez partir d’une structure déjà fournie, voici un exemple de code qui vous permettra de tester une injection SQL, comme vu lors de la première partie de la formation :

using (var conn = new SqliteConnection("Data Source=test.db"))

{

conn.Open();

var commande = conn.CreateCommand();

commande.CommandText = "DROP TABLE IF EXISTS PERSONNES; CREATE TABLE PERSONNES (nom VARCHAR(32), prenom VARCHAR(32), age INT)";

commande.ExecuteNonQuery();

commande = conn.CreateCommand();

commande.CommandText = "DROP TABLE IF EXISTS CONTRATS; CREATE TABLE CONTRATS (entreprise VARCHAR(32), sujet VARCHAR(32), montant INT)";

commande.ExecuteNonQuery();

commande = conn.CreateCommand();

commande.CommandText = "DROP TABLE IF EXISTS USERS; CREATE TABLE USERS (login VARCHAR(32), hash VARCHAR(32))";

commande.ExecuteNonQuery();

commande = conn.CreateCommand();

commande.CommandText = "INSERT INTO PERSONNES (nom, prenom, age) VALUES ('Lagaffe', 'Gaston', 63)";

commande.ExecuteNonQuery();

commande = conn.CreateCommand();

commande.CommandText = "INSERT INTO CONTRATS (entreprise, sujet, montant) VALUES ('Ministère', 'Contrat ultra-secret', 1000000)";

commande.ExecuteNonQuery();

}

Bien sûr, il reste préférable que vous créiez votre propre structure pour expérimenter, mais si vous ne connaissez pas le SQL, mieux vaut ne pas perdre de temps sur ce point.

# Ajout de l’API avec la vulnérabilité

Créer un nouveau contrôleur dans le projet Server, nommé par exemple PersonnesController. Voici un exemple de code présentant une faille de type injection SQL, dans la lecture comme dans l’écriture (l’espace de nommage utilisé ci-dessous en exemple devra bien sûr être adapté) :

using BlazorApp4.Shared;

using Microsoft.AspNetCore.Mvc;

using Microsoft.Data.Sqlite;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace BlazorApp4.Server.Controllers

{

[ApiController]

[Route("api/[controller]")]

public class PersonnesController : Controller

{

[HttpPost]

public IActionResult CreationPersonne([FromBody] Personne personne)

{

try

{

using (var conn = new SqliteConnection("Data Source=test.db"))

{

conn.Open();

var commande = conn.CreateCommand();

commande.CommandText = "INSERT INTO PERSONNES (nom, prenom, age) VALUES ('" + personne.Nom + "', '" + personne.Prenom + "', " + personne.Age.ToString() + ")";

commande.ExecuteNonQuery();

}

return new CreatedResult("#", personne);

}

catch (Exception ex)

{

return new UnprocessableEntityObjectResult(ex.ToString());

}

}

[HttpGet]

public Tuple<List<Personne>, string> GetAll([FromQuery] string? IndicationNom)

{

var donnees = new List<Personne>();

string erreur = string.Empty;

try

{

using (var conn = new SqliteConnection("Data Source=test.db"))

{

conn.Open();

var commande = conn.CreateCommand();

commande.CommandText = "SELECT nom, prenom, age FROM PERSONNES WHERE nom LIKE '%" + IndicationNom + "%'";

using (var reader = commande.ExecuteReader())

{

while (reader.Read())

{

donnees.Add(

new Personne()

{

Nom = reader.GetString(0),

Prenom = reader.GetString(1),

Age = reader.GetInt32(2)

});

}

}

}

}

catch (Exception ex)

{

erreur = ex.ToString();

}

return new Tuple<List<Personne>, string>(donnees, erreur);

}

}

}

La classe Personne à rajouter dans le projet Shared se présente comme suit :

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace BlazorApp4.Shared

{

public class Personne

{

public string? Nom { get; set; }

public string? Prenom { get; set; }

public int Age { get; set; }

}

}

Encore une fois, vous n’êtes pas obligé de prendre ce code tel quel et, si vous êtes à l’aise, le mieux est de créer votre propre implémentation avec une faille pour la tester.

# Créer une page d’accès web (option)

Si vous le souhaitez, vous pouvez créer dans le projet Client un composant Razor qui vous permettra de piloter les API du serveur et de tirer parti de la vulnérabilité à partir d’une GUI. Si vous le préférez, vous pouvez bien sûr utiliser un utilitaire comme Postman ou autre pour accéder directement aux API.

Le code proposé est le suivant, à adapter selon vos goûts (et bien sûr, à votre nom de projet) :

@page "/personnes"

@using Microsoft.AspNetCore.WebUtilities

@using System.Web

@using BlazorApp4.Shared

@inject HttpClient Http

@inject NavigationManager Nav

<h1>Liste des personnes</h1>

<p>Filtre : <input type="text" id="zonefiltre" **@bind**="filtre" /><button **@onclick**="Navigate">Rafraîchir liste</button></p>

@if (personnes == null)

{

<p><em>Loading...</em></p>

}

else

{

<table class="table">

<thead>

<tr>

<th>Nom</th>

<th>Prénom</th>

<th>Age</th>

</tr>

</thead>

<tbody>

@foreach (var personne in personnes)

{

<tr>

<td>@personne.Nom</td>

<td>@personne.Prenom</td>

<td>@personne.Age.ToString()</td>

</tr>

}

</tbody>

</table>

}

<h1>Ajouter une personne</h1>

<div>

<**EditForm** **OnSubmit**="Creer" **Model**="PersonneEnCreation">

<div>

Nom : <input type="text" **@bind-value**="@PersonneEnCreation.Nom" />

</div>

<div>

Prénom : <input type="text" **@bind-value**="@PersonneEnCreation.Prenom" />

</div>

<div>

Age : <input type="text" **@bind-value**="@PersonneEnCreation.Age" />

</div>

<button type="submit" class="btn btn-primary">Créer nouvelle demande</button>

</**EditForm**>

</div>

@code {

private Personne PersonneEnCreation = new Personne() { Age = 20 };

async void Creer()

{

try

{

var resultat = await Http.PostAsJsonAsync<Personne>("api/Personnes", PersonneEnCreation);

resultat.EnsureSuccessStatusCode();

}

catch

{

Console.WriteLine("Problème lors de la création");

}

this.StateHasChanged();

}

private Personne[] personnes;

private string filtre { get; set; } = string.Empty;

void Navigate()

{

// Juste pour faire voir le force reload ; normalement, on changerait plutôt les données par un appel d'API réalisé par le client WASM

Nav.NavigateTo("/personnes?indicationNom=" + HttpUtility.UrlEncode(filtre), forceLoad: true);

}

protected override async Task OnInitializedAsync()

{

string indicationNom = null;

var uri = Nav.ToAbsoluteUri(Nav.Uri);

if (QueryHelpers.ParseQuery(uri.Query).TryGetValue("indicationNom", out var valeurs))

indicationNom = valeurs[0];

string filtre = indicationNom is null ? string.Empty : "?IndicationNom=" + indicationNom;

var resultat = await Http.GetFromJsonAsync<Tuple<Personne[], string>>("api/Personnes" + filtre);

personnes = resultat.Item1.ToArray();

this.StateHasChanged();

}

}

Pour que cette nouvelle page soit visible, il faut modifier le fichier NavMenu.razor qui se trouve dans le dossier Shared, en ajoutant le code ci-dessous :

<li class="nav-item px-3">

<**NavLink** class="nav-link" href="personnes">

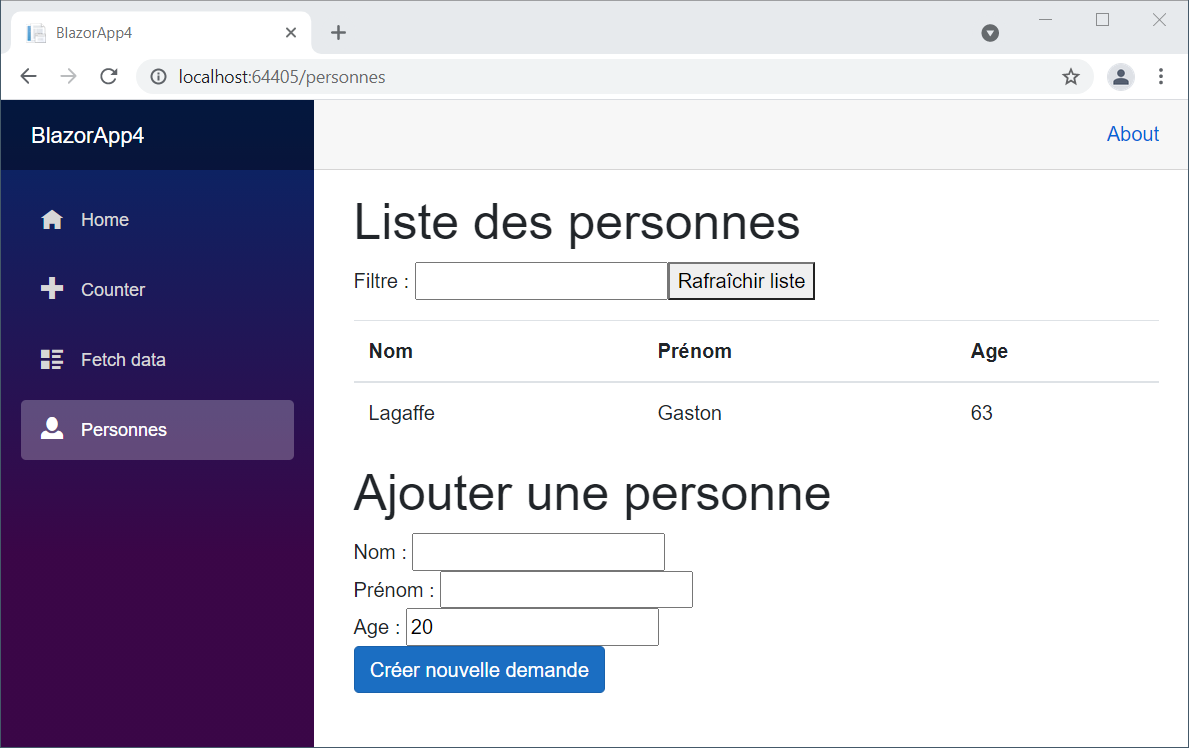
<span class="oi oi-person" aria-hidden="true"></span> Personnes

</**NavLink**>

</li>

La dépendance Microsoft.AspNetCore.WebUtilities doit être rajoutée sur le projet Client.

Lorsque l’application est lancée et que le menu Personnes est choisie, l’interface doit ressembler à ceci :



# Tester l’injection SQL

Arrivé à ce stade, vous avez tout ce qu’il faut pour essayer de supprimer des tables, accéder à des colonnes non prévues pour la lecture, etc.

Pour rappel, les doubles-tirets servent de commentaires SQL, la grammaire pour supprimer une table est DROP TABLE [nom de la table], et vous pouvez également utiliser DELETE FROM [nom de la table] avec une condition WHERE si vous le souhaitez. Pour séparer plusieurs ordres SQL, ce sont les points-virgules qui doivent être utilisés.

Soyez créatifs et n’oubliez pas de regarder les messages d’erreur pour vous guider vers la bonne syntaxe d’injection SQL qui vous permettra d’arriver à vos fins. Et ne vous inquiétez pas si vous cassez quelque chose, c’est le but et l’application se réinitialise complètement, y compris le contenu de base de données, quand vous la relancez…

Vous pouvez injecter sur la lecture comme sur l’écriture, le but étant d’arriver à trouver par vous-même le texte à envoyer pour exploiter la vulnérabilité qui a été mise en place. Pensez à bien noter vos attaques ayant fonctionné pour la suite de l’exercice.

# Protéger l’application

La parade privilégiée pour les injections SQL est d’utiliser les SqlParameter. L’exercice consiste à mettre en place cette protection et vérifier que les attaques qui avaient été trouvées la semaine précédente ne fonctionnent plus.

Il est également possible d’utiliser un Object/Relationnal Mapper (comme Microsoft EntityFramework) afin de déléguer la génération, et donc la sécurisation, des requêtes à une librairie dédiée spécifiquement au pilotage de la base de données. Dans ce cas, toutes les requêtes seront écrites en Linq avec des paramètres C# et seront converties en SQL par le compilateur. Ainsi, on évite les oublis de paramètres dans les requêtes SQL. Attention cependant à la performance, qui peut être mise en défaut si cette approche n’est pas bien maitrisée.

Pour la suite de l’exercice, il est demandé de trouver une modification du code qui introduit une faille d’injection SQL qui ne soit pas protégeable par ces approches (les paramètres SQL, en particulier, ne servent que pour les paramètres de filtre, alors qu’une injection peut venir de n’importe quelle construction par de l’assemblage de texte, donc partout dans la chaîne SQL).

Et, bien sûr, vous devez juste après une autre façon de sécuriser cette injection SQL, typiquement par une validation des entrées. Utilisez une approche exclusive ainsi qu’une approche inclusive et détaillez les avantages et inconvénients de chaque.