Exercice de « Path Traversal »

# Création du projet

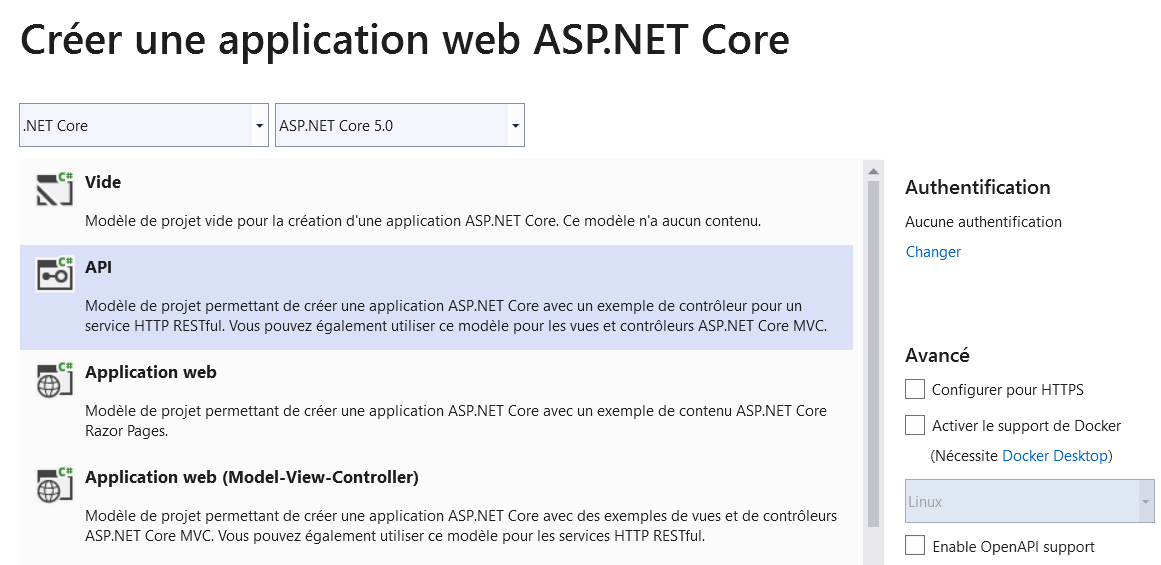
Dans ce document, on rentrera moins dans le détail des manipulations techniques que dans celui sur l’injection SQL.

Créez pour cet exercice un projet .NET de type Application web ASP.NET Core :

Une image contenant texte

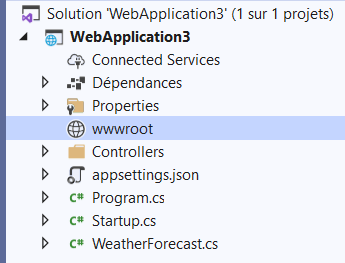
Description générée automatiquement

Paramétrer les options de l’assistant tel que ci-dessous :



# Ajout du contenu de fichiers

Créez un dossier nommé wwwroot dans le projet. Ce nom est reconnu comme un dossier spécial (vérifiez que l’icône s’ajuste effectivement) qui portera les fichiers exposés par le serveur web qui soutient également l’API.



Dans ce répertoire, créez des sous-répertoires, au moins deux, avec le premier qui contient des images légitimes pour l’accès par un identifiant utilisateur autorisé, et un autre dossier qui simule des données confidentielles d’un autre utilisateur. Par exemple :

Une image contenant texte

Description générée automatiquement

# Mise en place du code avec vulnérabilité

Dans le contrôleur d’API exemple créé automatiquement (WeatherForecastController.cs), supprimez le code proposé par défaut (ainsi que la classe WeatherForecast qui ne sert alors plus à rien) et introduisez le code ci-dessous :

[HttpGet]

public ContentResult Get()

{

string CustomerCode = "00000032";

string CustomerName = "SALVIA";

string CustomerAgent = "JP Gouigoux";

StringBuilder sb = new StringBuilder();

sb.Append("<h1>Bienvenue, " + CustomerName + " !</h1>");

sb.Append("<h3>Votre correspondant</h3>");

sb.Append("<p>" + CustomerAgent + "</p>");

sb.Append("<h3>Vos documents</h3>");

sb.Append("<iframe src='" + Request.Scheme + "://" + Request.Host + "/" + CustomerCode + "/' width='100%' height='100%' frameborder='0'/>");

return new ContentResult {

ContentType = "text/html",

Content = sb.ToString(),

StatusCode = (int)HttpStatusCode.OK

};

}

Les ajouts d’imports ci-dessus en entête de classe seront nécessaires pour que la solution compile correctement :

using System.Net;

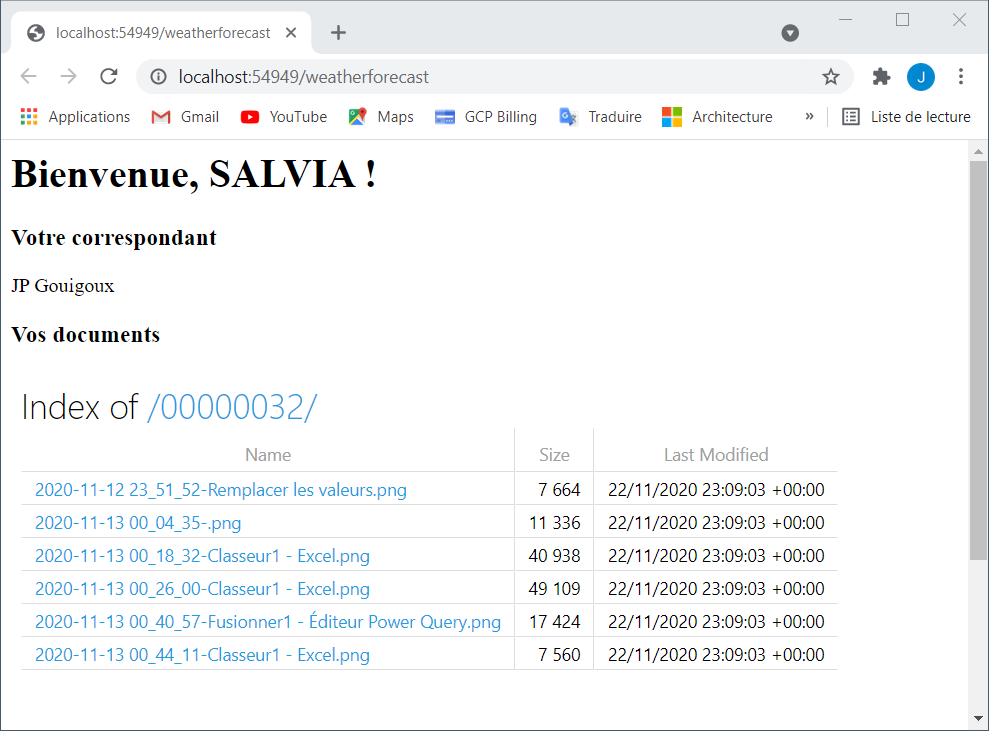
using System.Text;

Pour que la solution fonctionne, il est également nécessaire d’activer l’exposition des fichiers contenus dans le serveur en rajoutant les instructions suivantes dans la fonction Configure du Startup.cs :

app.UseStaticFiles();

app.UseDirectoryBrowser();

L’application est alors prête et un appui sur F5 devrait aboutir à un affichage comme suit :



# Mise en place d’une authentification réaliste (option)

Si vous le souhaitez, au lieu de déclarer une identification arbitraire en dur (les trois premières lignes de code ci-dessus), vous pouvez vous brancher sur votre système d’authentification existant si vous en avez un facile à mettre en place (par exemple si vous savez déjà brancher une application sur AzureAD et que vous ou quelqu’un de votre équipe avez les droits pour créer une nouvelle inscription d’application dans votre organisation).

# Exploitation de la vulnérabilité

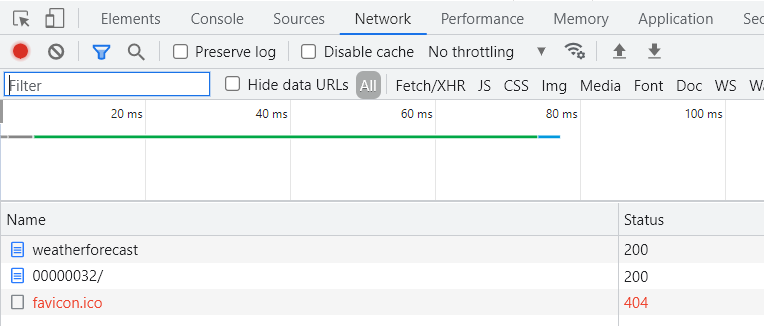
Le premier but de l’exercice est de trouver comment, en restant à la place de l’utilisateur identifié en dur, on peut, par une simple manipulation du navigateur, accéder aux données de l’autre utilisateur.

Pour cela, vous aurez peut-être besoin d’activer les outils de développeur dans le navigateur (raccourci F12), et de naviguer dans l’arborescence HTML de la page :

Une image contenant texte

Description générée automatiquement

Un coup d’œil sur l’onglet Network pourra également aider à comprendre le mode de fonctionnement du système et comment il peut être contourné.



A ce stade, l’objectif n’est pas seulement de montrer l’accès aux fichiers normalement non-autorisés, mais d’être en mesure de détailler le chemin de l’attaque et d’expliciter précisément la faille.

# Correction de la vulnérabilité

L’objectif suivant est (logiquement) de supprimer ou au moins de contenir la vulnérabilité détectée (en analyse de risque, on parle de mitigation). Vous pouvez réaliser cette étape dès la semaine 2 si vous avancez rapidement dans l’exercice. Sinon, ce sera le sujet de la session suivante.

Toute correction est bonne du moment qu’elle est argumentée et que vous avez un regard critique sur sa qualité, et le niveau de risque résiduel (le risque restant après l’application de la sécurité supplémentaire que vous aurez mise en place pour contrer la vulnérabilité observée).

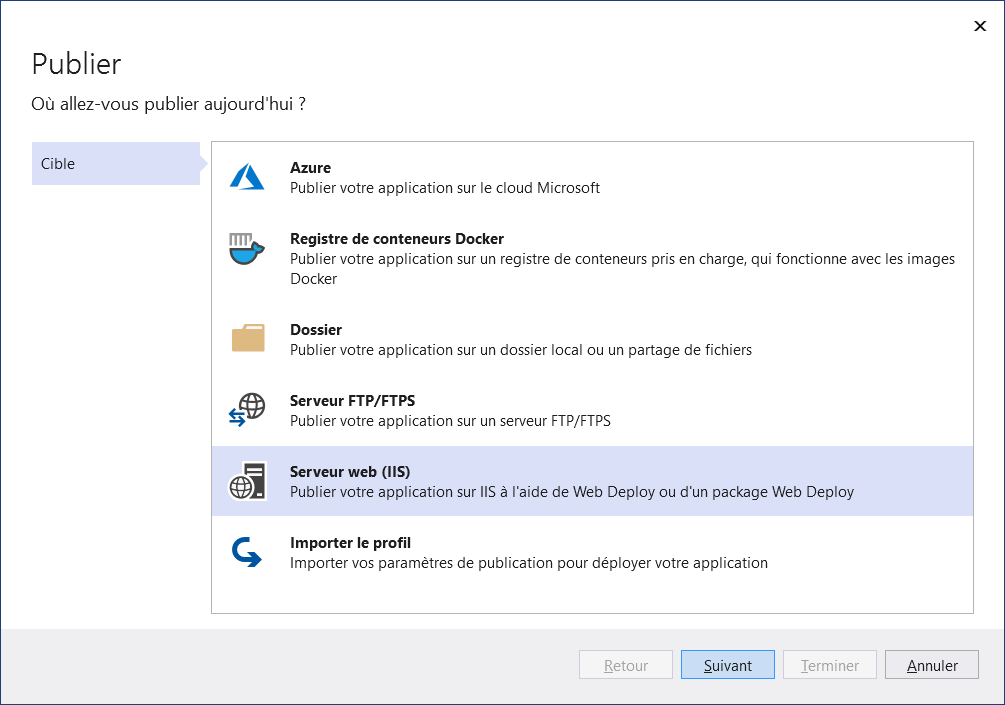
# Création d’un vrai Path Traversal

La vulnérabilité montrée ci-dessus n’est pas réellement un Path Traversal au sens complet du terme, car ASP.NET ne permet pas d’afficher des fichiers qui sont en-dehors du périmètre de l’application. Le but de cette troisième partie de l’exercice est de contourner cette protection intégrée à ASP.NET en créant un code encore pire du point de vue de la sécurité que celui montré plus haut.

Pour cela, on donne juste quelques pistes, à savoir qu’il faudra au minimum :

* coder une vulnérabilité qui travaille directement au niveau des accès fichiers (using System.IO) ;
* modifier les autorisations pour rendre le fonctionnement possible.

Autre astuce : IIS Express, qui est utilisé par défaut, ne gère que très peu la sécurité. Quelles sont les différences constatées si vous déployez sur un « vrai » serveur web ?



L’approche sur votre serveur web IIS local sera plus simple (il faut bien penser à activer la fonctionnalité Windows), mais si vous avez un abonnement Azure (même un compte gratuit suffit), le plus simple est de créer directement un App Service :

Une image contenant texte

Description générée automatiquement

Le déploiement se fait alors directement depuis Visual Studio :

Une image contenant texte

Description générée automatiquement

Et l’URL fournie permet d’accéder à la version déployée sur Azure… et qui sera bien moins perméable (la découverte de pourquoi et comment est le but de cette partie de l’exercice).

# Remarque

Cette partie de l’exercice peut paraître très artificielle car on doit lever des sécurités de la plateforme pour aboutir à notre objectif de créer une faille. Le réflexe est de se dire « mais qui ferait ça en vrai ? ». En fait, malheureusement énormément de développeurs, car la sécurité n’est souvent pas prise en compte dans les estimations de temps. Que celui qui n’a jamais donné les droits administrateurs à un compte pour régler un problème d’accès à une ressource me jette la première pierre 😊.

L’objectif pédagogique est donc double :

* Faire se rendre compte que ce genre de « levée de barrière » est très souvent tentant, mais qu’il faut prendre l’habitude de ne plus l’accepter, faire comprendre au PO que la sécurité prend du temps (quitte à désormais ajouter X% à toutes les estimations de temps de développement) et utiliser ce temps pour comprendre en profondeur les mécanismes de sécurité intégrée et les implémenter correctement.
* Montrer que les plateformes fournissent déjà de nombreuses protections de sécurité pour nous faciliter la vie et qu’on gagne énormément de temps au final à investir sur ces sécurités plutôt que d’essayer de coder l’équivalent par nous-mêmes. A terme, il n’est pas impossible que l’état de l’art considère comme une faute professionnelle d’avoir levé une protection système, alors autant s’y mettre tout de suite.