Mesures et bonnes pratiques de sécurité

Hashage de mot de passe :

Les bases de données sont les cibles d'attaques permanentes par les pirates informatiques. En effet, elles représentent une mine d'informations, notamment les mots de passe des utilisateurs, c'est pourquoi elles doivent faire l'objet d'une surveillance accrue. Il est donc indispensable de hasher les mots de passe stockés dans les bases de données.

Pour cela, Symfony met à notre disposition le UserPasswordHasherInterface. Ce dernier nous permet d'appeler la fonction hashPassword() à laquelle il faut passer en premier paramètre l'entité concernée et, en second paramètre, le mot de passe. Symfony va alors regarder l'étiquette password_hashers dans le fichier security.yaml et procéder à l'encodage. Dans mon application, l'algorithme « auto » est utilisé. Ce hasher sélectionne automatiquement l'algorithme le plus sécurisé disponible.

Le rôle des entités et les routes des contrôleurs :

Chaque utilisateur de l'application se voit attribuer un rôle leur de sa création. Ce procédé permet de s'assurer que seuls les utilisateurs bénéficiant de ce rôle ne seront autorisés à consulter certaines ressources. Ainsi, en préfixant les routes des contrôleurs par des attributs PHP 8 (par ex : #[Routes('/admin/partner']), l'accès à ces pages est bloqué pour les entités ne disposant pas du rôle requis.

Les formulaires :

Pour m'assurer que les données saisies par les utilisateurs soient bien interprétées par Symfony, j'ai typé tous les champs des formulaires. Ainsi, dans un champ de mot de passe, les caractères seront remplacés par de points ; les champs de mail attendront une chaine de caractères contenant un @.

Suivant les recommandations de la documentation de Symfony, j'ai également ajouté des contraintes pour les champs de formulaires. Celles-ci se trouvent dans les différents formType. Par exemple, pour un mot de passe, j'ai précisé que le champ ne peut être vide, avec la contrainte « NotBlank », et qu'il doit contenir un certain nombre de caractères, avec la contrainte « Length », qui permet de définir une limite minimum et une limite maximum.

Dans les contrôleurs, on retrouve une condition « if » qui permet de vérifier si le formulaire est bien soumis et valide avant d'effectuer la sauvegarde en base de données.

Les contrôleurs :

Selon les recommandations de SensioLabs, créateur du Framework Symfony et du moteur de templating Twig, j'ai fait en sorte que les contrôleurs soient les plus concis possible pour permettre une meilleure maintenabilité du code. Il est plus facile de s'y retrouver avec un grand nombre de

petits contrôleurs nommés de façon explicite plutôt qu'avec peu de contrôleurs contenant trop de code.

Les tests unitaires :

Un test unitaire est une procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel ou d'une portion de programme. IL vise le plus petit élément identifiable de l'application. Ces tests sont isolés les uns des autres, ils peuvent être lancés dans n'importe quel ordre sans affecter le résultat des prochains tests.

Afin de m'assurer que les entités de mon projet soient correctement codées, j'ai effectué des tests unitaires sur chacune d'elles. Pour chacun d'eux, je suis assuré que, lors de la création d'un nouvel objet, les données insérées correspondaient bien à ce qui est attendues. Et à l'inverse, le test échoue dans le cas contraire.

Failles XSS :

Pour protéger mon application des failles XSS et des injection SQL, j'ai choisi d'utiliser l'ORM Doctrine qui se sert de requêtes préparées.

Les failles XSS, pour cross scripting, permettent d'exécuter des scripts malveillant côté client. Les cybercriminels ciblent des sites web avec des fonctions vulnérables qui acceptent les entrées utilisateur, comme les barres de recherches, les zones de commentaires ou les formulaires de connexion. Les criminels joignent leur code malveillant au site Web légitime, trompant ainsi les navigateurs pour qu'ils exécutent leur malware chaque fois que le site est visité. Les hackers peuvent aussi se servirent du XSS pour diffuser un malware, réécrire le contenu du site, perturber des réseaux sociaux et les identifiants d'un utilisateur.

Les failles SQLi, pour SQL injection, sont des attaques consistant à modifier une requête SQL en cours par l'injection d'un morceau de requête non prévu, souvent par le biais d'un formulaire. La personne à l'origine de cette attaque peut ainsi accéder à la base de données et en modifier son contenu et donc compromettre la sécurité du système.

J'ai donc choisi d'utiliser ORM Doctrine car il embarque le mécanisme des requêtes préparées. L'exécution de ces dernières se déroule en deux temps : la préparation et l'exécution. D'abord, lors de la préparation, un template de requête est envoyé au serveur de base de données. Le serveur effectue une vérification, et initialise les ressources interne du serveur pour une utilisation ultérieure. Le serveur MySQL supporte le mode anonyme, avec des marqueurs de position utilisant le caractères « ? ». Ensuite, pendant l'exécution, le client lie les valeurs de paramètres et les envoie au serveur. Celui-ci exécute l'instruction avec les valeurs liées en utilisant les ressources internes précédemment créées.

Ainsi, Doctrine et Symfony vont s'assurer que la requête de m'utilisateur ne contient pas de caractère dangereux susceptible de compromettre la base de données. Tous les éventuels caractères spéciaux seront échappés.

Le moteur de template Twig :

J'utilise le moteur de template Twig qui me permet de lier facilement le HTML et le PHP tout en les séparant afin de garder un code clair et simple à manipuler. Il présente un autre avantage en avantage en termes de sécurité : par défaut, le texte est échappé. Cela signifie que, si un utilisateur tente d'écrire du code Javascript ou html dans un formulaire, celui-ci sera considéré comme une chaîne de caractère et ne sera donc pas exécuté. Il s'agit donc là une protection supplémentaire contre les failles XSS.

Les entités :

Lors de la création des entités de l'application, je me suis assuré que la visibilité des propriétés était bien à « private ». Cela permet de déclarer des attributs ou des méthodes qui ne seront visibles et accessibles directement que depuis l'intérieur même de la classe.

Protocole https :

L'application est hébergée sur le PAAS Heroku. J'ai pris soin de m'assurer qu'elle bénéficie du protocole https. La présence du cadenas à gauche de l'adresse en atteste. HTTPS est un protocole HyperText Transfert Protocol sécurisé, ce qui signifie que les requêtes et réponses http sont chiffrées.

Cependant, je me suis rendu compte qu'au moment de la connexion d'un utilisateur, ce dernier est renvoyé vers la page de login mais qui a perdu le certificat de sécurité.

Pour corriger cela, je me suis rendu sur la documentation de Heroku. A cette adresse, https://help.heroku.com/J2R1S4T8/can-heroku-force-an-application-to-use-ssl-tls, la plateforme indique la marche à suivre, en consultant cette page https://stackoverflow.com/questions/1329647/force-ssl-https-with-mod-rewrite/34065445#34065445. Ainsi, au modifiant le fichier « .htaccess » présent dans le dossier « public » du projet, l'application bénéficie tout le temps de protocole https.