LES MESURES ET LES BONNES PRATIQUES DE SECURITE

QU'EST-CE QUE LA SECURITE D'UN SITE WEB ?

Internet est un endroit dangereux ! Fréquemment, nous entendons parler de sites web devenus indisponibles en raison d'attaques par déni de service, ou affichant des informations modifiées (et souvent préjudiciables) sur leurs pages d'accueil. Dans d'autres cas, très médiatisés, des millions de mots de passe, d'adresses électroniques et de détails sur des cartes de crédit sont divulgués au public, exposant les utilisateurs du site web à la fois à l'embarras personnel et au risque de pertes financières.

L'objectif de la sécurité des sites web est de prévenir ces types d'attaques. Plus formellement, la sécurité des sites web est l'acte de protéger les sites web contre l'accès, l'utilisation, la modification, la destruction ou la perturbation non autorisées.

La sécurité efficace d'un site web nécessite un effort de conception sur l'ensemble du site : dans votre application web, dans la configuration du serveur web, dans vos politiques de création et de renouvellement des mots de passe et dans le code côté-client. Bien que tout cela semble très inquiétant, la bonne nouvelle est que si vous utilisez un framework web côté serveur, il incluera certainement par défaut des mécanismes de défense solides et bien pensés contre un certain nombre des attaques les plus courantes. D'autres attaques peuvent être atténuées grâce à la configuration de votre serveur web, par exemple en activant HTTPS. Enfin, les outils d'analyse de vulnérabilité accessibles au public peuvent vous aider à découvrir d'éventuelles erreurs dans votre conception.

QUELLES SONT LES RISQUES MAJEURS DE SECURITES D`UNE APPLICATION WEB

1. VALIDATION DES FORMULAIRES

Allez sur n'importe quel site à la mode avec un formulaire d'inscription et vous remarquerez des retours si vous n'entrez pas les données dans le format attendu. Vous aurez des messages comme :

- Ce champ est obligatoire » (vous ne pouvez pas le laisser vide)
- Veuillez entrer votre numéro de téléphone au format xxx-xxxx » (il attend trois chiffres suivis d'un tiret, suivi de quatre chiffres).

- Veuillez entrer une adresse e-mail valide » (ce que vous avez saisi ne ressemble pas à une adresse e-mail valide).
- Votre mot de passe doit comporter entre 8 et 30 caractères et contenir une majuscule, un symbole et un chiffre » (sérieusement ?).

Bonnes pratiques de sécurité

La validation est une tâche très courante dans les applications Web. Les données saisies dans les formulaires doivent être validées. Les données doivent également être validées avant d'être écrites dans une base de données ou transmises à un service Web.

Le validateur est conçu pour valider des objets par rapport à des contraintes. Dans la vraie vie, une contrainte pourrait être : « Le gâteau ne doit pas être brûlé ». Dans Symfony, les contraintes sont similaires : ce sont des assertions qu'une condition est vraie.

Remplir des formulaires web doit être aussi facile que possible. Alors pourquoi être tatillons et bloquer les utilisateurs à chaque fois ? Il y a trois raisons principales :

- **obtenir de bonnes données dans un bon format** — les applications ne tourneront pas correctement si les données utilisateur sont stockées dans un format fantaisiste, ou si les bonnes informations ne sont pas aux bons endroits ou totalement omises.

- **protéger nos utilisateurs** — s'ils entrent un mot de passe facile à deviner ou aucun, des utilisateurs malveillants peuvent aisément accéder à leurs comptes et voler leurs données.

- **nous protéger nous-mêmes** — il existe de nombreuses façons dont les utilisateurs malveillants peuvent utiliser les formulaires non protégés pour endommager l'application dans laquelle ils se trouvent (voir Sécurité du site Web).

Mesures mises en place

Installation et utilisation l'outil FORM de Symfony.

La création et le traitement de formulaires HTML sont difficiles et répétitifs. Vous devez vous occuper du rendu des champs de formulaire HTML, de la validation des données soumises, du mappage des données du formulaire dans des objets et bien plus encore. Symfony inclut une fonctionnalité de formulaire puissante qui fournit toutes ces fonctionnalités et bien d'autres pour des scénarios vraiment complexes.

La validation en Symfony est effectuée en ajoutant un ensemble de règles, appelées contraintes (de validation) à une classe. Vous pouvez les ajouter soit à la classe d'entité, soit à la classe de formulaire.

Contraintes mis en place

Des contraintes ont été ajouté pour la validation de chaque champ qui le nécessite.
Par exemple le champs 'Name' possède les contraintes suivantes :

- Pas de champs vide
- Pas de champs nul
- Longueur Champs entre 2 et 50 caractères

Le champs email comporte les contraintes suivantes :

- Pas de champs vide
- Champs de type Email obligatoire
- Longueur Champs entre 2 et 50 caractères

Le champs Password comporte les contraintes suivantes :

- Pas de champs vide
- Champs de type Email obligatoire
- Longueur Champs entre 6 et 15 caractères
- Majuscule obligatoire

Exemple de syntaxe pour les contraintes pour un Email :

2. INJECTION

Les failles d'injection, telles que l'injection SQL, NoSQL, OS et LDAP, se produisent lorsque des données non fiables sont envoyées à un interpréteur dans le cadre d'une commande ou d'une requête. Les données hostiles de l'attaquant peuvent inciter l'interpréteur à exécuter des commandes involontaires ou à accéder à des données sans autorisation appropriée.

Les données non fiables peuvent être des données de la requête HTTP (y compris le chemin d'URL, la chaîne de requête, les données POST, la chaîne de l'agent utilisateur et d'autres en-têtes de requête HTTP), mais également d'autres sources telles que la base de données, les services Web internes et externes ou les variables d'environnement. .

Il est important de noter que si l'injection SQL est probablement la forme d'injection la plus connue (et la plus répandue), l'injection n'est en fait pas limitée aux bases de données SQL ou relationnelles. Tout type d'interpréteur peut être vulnérable à l'injection s'il ne peut pas faire la différence entre la commande ou la requête que vous avez écrite et les données non fiables qui y sont collées. J'utiliserai l'injection SQL à des fins de démonstration car SQL sera familier à la plupart des développeurs, mais gardez à l'esprit que les mêmes principes s'appliquent à un concept plus large.

Les bonnes pratiques

Doctrine est l'ORM utilisé dans le projet et selon la documentation :

L'ORM protège bien mieux contre l'injection SQL que le DBAL seul. Vous pouvez considérer les API suivantes comme étant à l'abri de l'injection SQL :

- \Doctrine\ORM\EntityManager#find()et getReference().
- Toutes les valeurs sur les objets insérés et mis à jour viaDoctrine\ORM\EntityManager#persist()
- Toutes les méthodes de recherche sur Doctrine\ORM\EntityRepository.
- Entrée utilisateur définie sur les requêtes DQL ou les méthodes QueryBuilder via
    - setParameter()ou variantes
    - setMaxResults()
    - setFirstResult()

- Requêtes via l'API Criteria sur Doctrine\ORM\PersistentCollectionet Doctrine\ORM\EntityRepository.

Vous n'êtes PAS à l'abri d'une injection SQL lorsque vous utilisez une entrée utilisateur avec :

- API d'expression deDoctrine\ORM\QueryBuilder
- Concaténation des entrées utilisateur dans les instructions DQL SELECT, UPDATE ou DELETE ou en SQL natif.

Cela signifie que les injections SQL ne peuvent se produire qu'avec Doctrine ORM lorsque vous travaillez avec des objets de requête de tout type. La règle de sécurité consiste à toujours utiliser des paramètres d'instruction préparés pour les objets utilisateur lors de l'utilisation d'un objet Query.

Mesures mises en place

Les requêtes par défaut que propose doctrine ne suffisaient pas pour développer la fonctionnalité de filtrage de recherche.

Il m'a fallu écrire deux requêtes Doctrine pour filtrer mes recherches.

J’ai bien veillé qu’il n’y ai pas de concaténation pour les entrées utilisateur et lors de l'écriture des requêtes, j’ai bien veillé à passer mes variables en tant que paramètres à l'aide de la méthode setParameter().

3. AUTHENTIFICATION BROKEN

Les fonctions d'application liées à l'authentification et à la gestion de session sont souvent implémentées de manière incorrecte, permettant aux attaquants de compromettre des mots de passe, des clés ou des jetons de session, ou d'exploiter d'autres défauts d'implémentation pour assumer temporairement ou définitivement l'identité d'autres utilisateurs.

Les attaquants disposent de plusieurs techniques pour attaquer l'authentification des utilisateurs. Beaucoup d'entre eux sont basés sur le fait que les utilisateurs utilisent couramment des mots de passe très faibles et réutilisent leurs mots de passe pour plusieurs services. Les attaques courantes incluent le Credential stuffing et les attaques par force brute . Les attaquants peuvent également tenter de voler le cookie de session d'un utilisateur authentifié via des attaques XSS ou man-in-the-middle.

Définition

Credential stuffing :
Le credential stuffing est l'injection automatisée de paires de nom d'utilisateur et de mot de passe volées ("identifiants") dans les formulaires de connexion du site Web, afin d'accéder frauduleusement aux comptes d'utilisateurs.

Attaques par force brute :
On parle d’attaque par force brute lorsqu’un pirate essaie de trouver (« craquer ») un mot de passe via un processus intensif d’essais et d’erreurs assisté par ordinateur.

Mesures mises en place

SecurityBundle

Définition et fonctionnalités
Symfony fournit de nombreux outils pour sécuriser votre application. Certains outils de sécurité liés à HTTP, tels que les cookies de session sécurisée et la protection CSRF, sont fournis par défaut. Le SecurityBundle, fournit toutes les fonctionnalités d'authentification et d'autorisation nécessaires pour sécuriser votre application.

Le contrôle de la sécurité sous Symfony est très avancé mais également très simple. Pour cela Symfony distingue :

- l'authentification,
- l'autorisation.

Prenons quelques minutes pour définir ces mécanismes.

L'authentification, c'est le procédé qui permet de déterminer qui est votre visiteur. Il y a deux cas possibles :

- le visiteur est anonyme car il ne s'est pas identifié,
- le visiteur est membre de votre site car s'est identifié.

Sous Symfony, c'est le firewall qui prend en charge l'authentification.

Régler les paramètres du firewall va vous permettre de sécuriser le site. En effet, vous pouvez restreindre l'accès à certaines parties du site uniquement aux visiteurs qui sont membres. Autrement dit, il faudra que le visiteur soit authentifié pour que le firewall l'autorise à passer.

L'autorisation intervient après l'authentification. Comme son nom l'indique, c'est la procédure qui va accorder les droits d'accès à un contenu. Sous Symfony, c'est l'access control qui prend en charge l'autorisation.

Prenons l'exemple de différentes catégories de membres. Tous les visiteurs authentifiés ont le droit de poster des messages sur le forum mais uniquement les membres administrateurs ont des droits de modération et peuvent les supprimer. C'est l'access control qui permet de faire cela.

Installation et paramétrage

C’est la première brique que j’ai installé et configuré pour sécuriser l’authentification de l’application.

Ce bundle nous a permis de paramétrer les outils suivants :

- Hashage de mot de passe
- Activation du firewall
- Activation de l’acces control
- Activation de la protection CSRF
- Activation d’une politique de longueur, de complexité et de rotation des mots de passe
- Création d’un formulaire de connexion associé à un login Controller
- Limitation des tentatives de connexion (paramètre `login_throttling`)

J’ai créé aussi un rôle dans l’acces controller pour chaque interface

- ROLE_ADMIN pour accéder à l’interface Admin
- ROLE_STRUCTURE accéder à l’interface structure
- ROLE_PARTENAIRE accéder à l’interface partenaire

Cela aura pour but de permettre aux utilisateurs d’accéder à leur espace selon le rôle qui leur a été attribuer.

Pour sécuriser les espaces respectifs de chaque utilisateur un contrôle supplémentaire a été mis en place pour les utilisateurs non admin.
En récupérant l'utilisateur courant via le getUser(), je peux vérifier si l'espace auxquelles l'utilisateur veux accéder lui appartient.

4. EXPOSITION DES DONNEES SENSIBLES

Les données envoyées sur Internet non cryptées peuvent être reniflées ou interceptées à l'aide d'une attaque de type "man-in-the-middle". Par exemple, en écoutant des réseaux sans fil publics non sécurisés, en installant des renifleurs de paquets sur des routeurs non sécurisés, en usurpant le DNS de la victime afin qu'il résolve les noms de domaine en une adresse IP contrôlée par l'attaquant, ou en utilisant un Wi-Fi Pineapple qui trompe les appareils pour qu'ils s'y connectent en se faisant passer pour un réseau sans fil avec lequel ils se sont déjà connectés. Il existe de nombreux autres scénarios possibles, mais le fait est que nous devons supposer que notre trafic peut être intercepté et que nous devons le chiffrer pour protéger nos données.

Mesures mises en place

- Installation et paramétrage de l'HTTPS

Le serveur web utilisé est l'outils NGINX
Un certificat SSL chez le provider ZEROSSL a été créé et téléchargé pour passer le serveur NGINX en HTTPS.

- Installation d'un pare-feu

J'ai installé le tandem Fail2ban le firewall UFW pour filtrer les paquets et limiter l'ouverture des ports du serveur.

Un pare-feu est essentiel lors de la configuration du VPS pour limiter le trafic indésirable sortant ou entrant dans votre VPS.

L'outil fail2ban permet de surveiller l'activité des logs de certains services, tel que SSH ou Apache. Lors d'un trop grand nombre d'authentifications ratées fail2ban va générer une règle IPTables, cette règle aura pour but d'interdire pendant une durée déterminée les connexions depuis l'adresse IP susceptible d'être un attaquant.
C'est L'outils NGINX que j'ai utilisé pour publier le site.
J'ai aussi installé un certificat SSL sur le serveur NGINX.

5. MAUVAISE CONFIGURATION DE LA SECURITE

La mauvaise configuration de la sécurité est le problème le plus fréquemment rencontré. Cela est généralement le résultat de configurations par défaut non sécurisées, de

configurations incomplètes ou ad hoc, d'un stockage cloud ouvert, d'en-têtes HTTP mal configurés et de messages d'erreur détaillés contenant des informations sensibles. Non seulement tous les systèmes d'exploitation, Framework, bibliothèques et applications doivent être configurés en toute sécurité, mais ils doivent également être corrigés/mis à niveau en temps opportun.

Mesures mises en place

- Déploiement de Symfony avec les variables d'environnements principal bien configuré
    - APP_ENV sur PROD.
    - APP_DEBUG désactivé
    - APP_SECRET définie sur une valeur aléatoire d'environ 32 caractères.
- Configuration des bases de données pour exiger l'authentification et accepter uniquement les connexions du réseau interne.
- Utilisation d'un proxy pour déployer le site en HTTPS
- Utilisation d'un certificat délivré par une autorité de certification.
- Grâce à Let's Encrypt, il est possible d'obtenir un certificat SSL gratuit et fiable !

6. LES FAILLES XSS

Les failles XSS se produisent chaque fois qu'une application inclut des données non fiables dans une nouvelle page Web sans validation ou échappement appropriés, ou met à jour une page Web existante avec des données fournies par l'utilisateur à l'aide d'une API de navigateur qui peut créer du HTML ou du JavaScript. XSS permet aux attaquants d'exécuter des scripts dans le navigateur de la victime qui peuvent détourner des sessions utilisateur, défigurer des sites Web ou rediriger l'utilisateur vers des sites malveillants.

Mesures mises en place

- Utilisation de Twig comme moteur de modèle, car il échappe automatiquement à la sortie par conception.

7. UTILISATION DE COMPOSANTS PRESENTANT DES VULNARIBILITES CONNUEES

Les composants, tels que les bibliothèques, les Framework et d'autres modules logiciels, s'exécutent avec les mêmes privilèges que l'application. Si un composant vulnérable est exploité, une telle attaque peut faciliter une grave perte de données ou une prise de contrôle du serveur. Les applications et les API utilisant des composants présentant des

vulnérabilités connues peuvent saper les défenses des applications et permettre diverses attaques et impacts.

Mesures mises en place

- Suppression des dépendances, fonctionnalités, composants, fichiers et documentations inutilisés ou inutiles. Utilisez ‘composer-unused’ pour trouver les packages Composer inutilisés et supprimez-les de votre liste de dépendances.

- Mise à jour de PHP et de Symfony

- Utilisez régulièrement la ‘composer outdatedcommande’ pour voir quels packages sont obsolètes.

8. TEST UNITAIRE

Quelques tests fonctionnels et unitaires ont été programmés pour vérifier la fiabilité des connexions utilisateurs, ainsi que la fiabilités des données lors de la création d’un utilisateur.