AXE D'AMÉLIORATIONS

Recommandations

Protocoles d’authentification 4IW2

Étudiants : Philippe DELENTE, Camille GIRARD, Catalina DANILA

1. Sécurisation des cookies

Pour renforcer la sécurité des cookies utilisés dans l'application, il est essentiel d'appliquer les drapeaux `Secure` et `HttpOnly` lors de leur création. Ces paramètres permettent de limiter les risques d'attaques telles que le vol de session ou les attaques XSS (Cross-Site Scripting).

```php
setcookie("session_token", $sessionToken, [
    'expires' => time() + 3600, // Durée de validité du cookie (1 heure)
    'path' => '/',              // Accessible sur tout le domaine
    'domain' => 'example.com', // Domaine spécifique pour le cookie
    'secure' => true,           // Transmet uniquement via HTTPS
    'httponly' => true,         // Inaccessible par le JavaScript client
    'samesite' => 'Strict',     // Empêche les requêtes intersites
]);
```

1. Flag `Secure` :
    - Ce drapeau garantit que les cookies sont uniquement transmis via des connexions HTTPS sécurisées. Cela empêche leur interception sur des connexions HTTP non sécurisées.
2. Flag `HttpOnly` :
    - Ce paramètre empêche l'accès au cookie depuis les scripts JavaScript du côté client. Cela réduit significativement le risque d'attaques XSS.
3. Option SameSite :
    - La configuration en Strict empêche les cookies d'être envoyés lors des requêtes provenant de domaines tiers. Cela renforce la protection contre les attaques CSRF (Cross-Site Request Forgery).

2. Gestion des environnements de variables

Le fichier `.env` contient des informations sensibles, telles que les identifiants de base de données et d'autres secrets. Il est crucial de le protéger pour éviter son exposition à des utilisateurs non autorisés ou à des systèmes publics, comme les dépôts Git.

Améliorations possibles :

1. Le fichier `.env` doit être protégé par un `.gitignore`. Le fichier `.env` doit inclure toutes les informations sensibles.

```
#fichier .gitignore

mysql-data2/
.DS_Store
.idea/
www/Views/styles/src/css/main.css
www/Views/styles/src/css/main.css.map
vendor/
node_modules/

#rajoute .env
.env
```

```
# fichier .env
DB_HOST=postgres
DB_PORT=5432
DB_NAME=gfm
DB_USER=gofindme
DB_PASSWORD=Gofindme.2024
TABLE_PREFIX=gfm
```

- Dans notre projet, le fichier `.env` est déjà dans le dépôt, il faut le retirer avec la commande suivante :

```
git rm --cached .env
git commit -m "Ajout du fichier .env dans le .gitignore"
```

2. Ajouter un contrôle pour vérifier l'existence du fichier `.env` et utiliser `phpdotenv` pour charger les variables

```
// fichier config.php
require 'vendor/autoload.php';

if (file_exists( filename: __DIR__ . '/.env')) {
    $dotenv = Dotenv\Dotenv::createImmutable(__DIR__);
    $dotenv->load();
}

define('DB_HOST', getenv( varname: 'DB_HOST') ?: 'localhost');
define('DB_PORT', getenv( varname: 'DB_PORT') ?: '3306');
define('DB_NAME', getenv( varname: 'DB_NAME') ?: 'database');
define('DB_USER', getenv( varname: 'DB_USER') ?: 'root');
define('DB_PASSWORD', getenv( varname: 'DB_PASSWORD') ?: '');
define('TABLE_PREFIX', getenv( varname: 'TABLE_PREFIX') ?: '');
```

3. Restreindre les permissions du fichier .env (Linux) :

```
chmod 600 .env
chown www-data:www-data .env
```

4. Pour plus de sécurité, il faut également empêcher l'accès au fichier `.env` via le serveur web :

```
<Files .env>
    Order Allow,Deny
    Deny from all
</Files>
```

3. Prévention des failles XSS

Les failles XSS (Cross-Site Scripting) permettent aux attaquants d'injecter du code malveillant dans des pages web, ce qui peut compromettre les données des utilisateurs et la sécurité de l'application. Pour prévenir ces failles, voici les améliorations à faire.

1. Utilisation de `htmlspecialchars()` pour échapper les entrées utilisateur

La fonction `htmlspecialchars()` transforme les caractères spéciaux en entités HTML, empêchant ainsi l'exécution de scripts injectés.

```php
$userInput = $_POST['user_input'] ?? '';
echo htmlspecialchars($userInput, flags: ENT_QUOTES, encoding: 'UTF-8');
```

2. Validation et filtrage des données utilisateur

Avant de stocker ou d'utiliser les données utilisateur, il faut appliquer des règles de validation strictes pour garantir qu'elles respectent le format attendu. Cela réduit les risques de failles XSS et d'injections SQL.

Exemple 1 avec validation d'un nom d'utilisateur :

```php
if (preg_match( pattern: '/^[a-zA-Z0-9_]+$/', $_POST['username'])) {
    $username = $_POST['username'];
} else {
    echo "Nom d'utilisateur invalide.";
}
```

Exemple 2 avec validation d'un email :

```php
$email = filter_input( type: INPUT_POST, var_name: 'email', filter: FILTER_VALIDATE_EMAIL);
if (!$email) {
    die('Email invalide');
}
```

3. Définir des en-têtes HTTP de sécurité

Ajouter des en-têtes HTTP pour renforcer la sécurité contre les failles XSS et empêcher le chargement de contenu non sécurisé.

```
Header set Content-Security-Policy "default-src 'self'; script-src 'self';"
Header set X-XSS-Protection "1; mode=block"
```

4. Utilisation sécurisée de PDO pour prévenir les injections SQL

Les injections SQL constituent une menace majeure pour la sécurité des bases de données. En utilisant les méthodes `prepare()` et `execute()` de PDO, nous pouvons sécuriser les requêtes et empêcher l'exécution de commandes malveillantes.

Les requêtes préparées permettent de séparer le code SQL des données utilisateur, garantissant que les entrées utilisateur ne sont pas interprétées comme des commandes SQL.

```php
$stmt = $pdo->prepare('SELECT * FROM users WHERE email = :email');
$stmt->execute(['email' => $email]);
$user = $stmt->fetch();
```

De plus, nous pouvons aussi empêcher la divulgation d'erreurs SQL. Les erreurs SQL ne doivent jamais être affichées à l'utilisateur final, car elles peuvent révéler des informations sensibles sur la structure de la base de données. En cas d'erreur, il faut enregistrer les détails dans un fichier de journalisation sécurisé et afficher un message générique à l'utilisateur.

```php
try {
    $stmt = $pdo->prepare("SELECT * FROM users WHERE username = :username");
    $stmt->execute([':username' => $username]);
} catch (PDOException $e) {
    // Logguez l'erreur sans l'afficher à l'utilisateur
    error_log($e->getMessage());
    die("Une erreur est survenue. Veuillez réessayer plus tard.");
}
```

5. Ajout des en-têtes de sécurité http

La configuration des en-têtes HTTP est essentielle pour renforcer la sécurité de l'application web. En complément des mesures prises pour prévenir les failles XSS (voir étape 3), voici une liste des en-têtes HTTP importants à configurer pour protéger l'application contre divers types d'attaques.

En-Têtes HTTP à Configurer :

1. Strict-Transport-Security (HSTS) : Force l'utilisation du HTTPS et empêche les attaques de type "downgrade".

```
Header always set Strict-Transport-Security "max-age=31536000;
includeSubDomains"
```

2. Content-Security-Policy (CSP) : Empêche l'exécution de scripts non autorisés et limite les sources de contenu.

```
Header always set Content-Security-Policy "default-src 'self';
script-src 'self' https://trusted-cdn.com; style-src 'self';"
```

3. X-Content-Type-Options : Empêche le navigateur de deviner le type de contenu pour prévenir certains types d'attaques.

```
Header always set X-Content-Type-Options "nosniff"
```

4. X-Frame-Options : Empêche le site d'être intégré dans des iframes, prévenant ainsi les attaques de type clickjacking.

```
Header always set X-Frame-Options "DENY"
```

5. Referrer-Policy : Contrôle quelles informations sont envoyées dans l'en-tête Referer.

```
Header always set Referrer-Policy "no-referrer-when-downgrade"
```

6. Permissions-Policy : Restreint l'accès aux fonctionnalités sensibles comme la caméra, le micro, ou la géolocalisation.

```
Header always set Permissions-Policy "geolocation=(), microphone=(),
camera=()"
```

7. X-XSS-Protection : Active une protection contre les attaques XSS dans les navigateurs qui le supportent.

```
Header always set X-XSS-Protection "1; mode=block"
```

Implémentation dans le .htaccess:

```
<IfModule mod_headers.c>
    Header always set Strict-Transport-Security "max-age=31536000; includeSubDomains"
    Header always set Content-Security-Policy "default-src 'self'; script-src 'self';"
    Header always set X-Content-Type-Options "nosniff"
    Header always set X-Frame-Options "DENY"
    Header always set Referrer-Policy "no-referrer-when-downgrade"
    Header always set Permissions-Policy "geolocation=(), microphone=(), camera=()"
    Header always set X-XSS-Protection "1; mode=block"
</IfModule>
```

Ajout Direct dans l'Application PHP :

```
header( header: "Strict-Transport-Security: max-age=31536000; includeSubDomains");
header( header: "Content-Security-Policy: default-src 'self'; script-src 'self';");
header( header: "X-Content-Type-Options: nosniff");
header( header: "X-Frame-Options: DENY");
header( header: "Referrer-Policy: no-referrer-when-downgrade");
header( header: "Permissions-Policy: geolocation=(), microphone=(), camera=()");
header( header: "X-XSS-Protection: 1; mode=block");
```

6. Implémentation de JWT sécurisé (si nécessaire)

L'utilisation des JSON Web Tokens (JWT) est une méthode efficace pour gérer l'authentification et l'échange sécurisé d'informations. Cependant, leur implémentation nécessite des pratiques spécifiques pour garantir leur sécurité. Voici les étapes essentielles pour implémenter un JWT sécurisé.

1. **Utiliser un algorithme de signature robuste** comme **HS256** (HMAC avec SHA-256) ou **RS256** (RSA avec SHA-256). Il faut éviter les algorithmes faibles, tels que none, qui rendent le JWT vulnérable.
2. **Ne jamais inclure d'informations sensibles dans le payload du JWT**, car ce dernier est lisible par quiconque en a possession, même s'il est signé. Ne pas stocker de mots de passe, d'informations bancaires ou de données personnelles dans le JWT.
3. **Définir une durée d'expiration pour le JWT** afin de limiter son utilisation en cas de vol. Une durée raisonnable est généralement comprise entre 15 minutes et 1 heure. Pour des sessions prolongées, utiliser des refresh tokens pour renouveler les JWT expirés.
4. **Ajouter les champs** `aud` **(audience) et** `iss` **(issuer)** pour restreindre l'utilisation du JWT à une application ou audience spécifique et vérifier l'émetteur du token.
5. Si l'on utilise des cookies pour stocker les JWT, il faut **activer les options** `SameSite=Strict` **et** `HttpOnly` pour empêcher leur envoi à des sites tiers et limiter leur accès depuis JavaScript.

```php
use Firebase\JWT\JWT;
use Firebase\JWT\Key;

$key = 'secure-key';

// Création du payload
$payload = [
    'iss' => 'example.com',       // Émetteur (issuer)
    'aud' => 'example.com',       // Audience cible
    'iat' => time(),              // Date de création (issued at)
    'exp' => time() + 3600,       // Expiration : 1 heure
    'user_id' => 123              // Données utilisateur (exemple)
];

$jwt = JWT::encode($payload, $key, 'HS256');

try {
    $decoded = JWT::decode($jwt, new Key($key, 'HS256'));
    print_r($decoded);
} catch (Exception $e) {
    echo "Erreur de validation : " . $e->getMessage();
}
```

7. Validation et renforcement des secrets

Pour garantir la sécurité des secrets (comme les mots de passe), il est essentiel d'appliquer des règles strictes de validation et de renforcer leur complexité. Cela permet de réduire le risque d'attaques par force brute ou par dictionnaire.

```php
$password = $_POST['password'] ?? '';

if (!preg_match( pattern: '/^(?=.*[A-Z])(?=.*[a-z])(?=.*[0-9])(?=.*[\W]).{8,}$/', $password)) {
    die('Le mot de passe doit contenir au moins 8 caractères, une majuscule,
    une minuscule, un chiffre et un caractère spécial.');
}
```