Mettre en place les recommandations de sécurité de la partie front

Table des matières

I. Contexte

Durée (en minutes) : 60

Environnement de travail :

- Ordinateur avec Windows, MacOS ou Linux
- Visual studio Code
- Avoir l'api de quai antique installée sur la machine

Pré requis *:* connaître Git, et avoir des bases de JS, et Bootstrap. Savoir faire des requêtes fetch

Objectifs

- Savoir utiliser un linter
- Savoir parer une faille XSS
- Savoir authentifier une requête FETCH

Contexte

À mesure que les applications deviennent de plus en plus complexes et que la quantité de données manipulées augmente, il est essentiel de veiller à ce que le code source de ces applications soit sécurisé. Un aspect majeur de cette sécurité est la prévention des failles et vulnérabilités potentielles, telles que les failles XSS (Cross-Site Scripting) qui peuvent compromettre la sécurité du système. Il nous faut aussi chercher à pallier à nos éventuelles étourderies de codage. Pour atteindre cet objectif, ce cours propose d'utiliser un outil d'analyse statique du code source, appelé linter, pour détecter et corriger les erreurs de programmation, les violations de conventions de codage et les problèmes de sécurité. Nous verrons aussi l'importance d'une gestion appropriée des jetons d'authentification pour sécuriser l'accès aux ressources et fonctionnalités de l'API.

II. Sécuriser notre code

A. Utilisation d'un linter

Définition **LINTER**

Un linter est un outil d'analyse statique du code source utilisé dans le développement de logiciels. Son rôle principal est d'inspecter le code source d'un programme informatique pour détecter les erreurs de programmation, les violations de conventions de codage et d'autres problèmes potentiels. Les linters sont particulièrement utiles pour améliorer la qualité du code, renforcer la cohérence du style de codage et réduire les bogues.

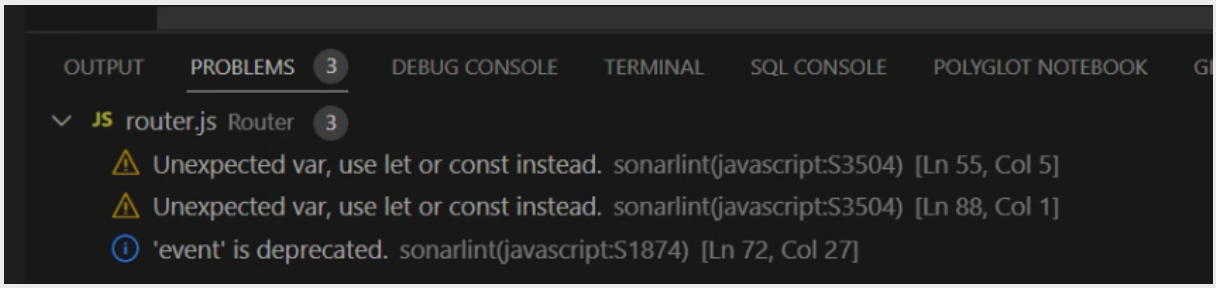
On peut distinguer quelques grandes actions effectuées par le linter : la détection d'erreurs de syntaxe, l'analyse statique du code, la conformité aux conventions de codage, l'analyse de la qualité du code ou bien la détection de vulnérabilités de sécurité.

Les linters sont généralement configurables pour s'adapter aux normes de codage spécifiques d'un projet ou d'une équipe de développement. Ils peuvent être utilisés en tant qu'extension d'éditeurs de code (comme ESLint pour Visual Studio Code) ou intégrés dans des pipelines de développement pour automatiser la détection des problèmes de code.

Exemple **Sonarlint**

SonarLint est un outil d'analyse statique du code open source développé par SonarSource. C'est un linter que vous pouvez avoir sur votre environnement de travail, comme VSCODE.

C'est une extension, que vous pouvez installer. Une fois installé, il vous montrera les différentes erreurs présentes sur votre code.



B. Faille XSS

Assainir le code HTML injecté dans notre page

Une faille XSS (Cross-Site Scripting) est une vulnérabilité de sécurité courante dans les applications web. Elle permet à des attaquants d'injecter des scripts malveillants (souvent du code JavaScript) dans des pages web vues par d'autres utilisateurs. Ces scripts peuvent être exécutés dans le navigateur des victimes, ce qui peut avoir des conséquences néfastes.

Dans notre cas, une faille XSS pourrait arriver si un utilisateur envoie une donnée en base de données. Imaginons qu'un utilisateur veuille ajouter une image dans la galerie d'image. Au lieu d'envoyer le titre de l'image, il tape du code HTML dans le formulaire pour inclure cette image. Lorsque nous allons afficher cette image, si nous affichons le HTML donné par l'utilisateur malveillant, nous allons afficher des informations que nous ne voulons pas afficher.

Avant d'injecter le titre de l'image, il faut être sûr que cette chaîne de caractère n'est pas corrompue.

Méthode **Assainir le code HTML injecté dans notre page**

Chaque fois que nous allons injecter du code HTML dans notre page avec des données récupérées depuis notre API, nous allons utiliser cette méthode pour assainir le texte :

```
function sanitizeHtml(text){
    // Créez un élément HTML temporaire de type "div"
    const tempHtml = document.createElement('div');

    // Affectez le texte reçu en tant que contenu texte de l'élément "tempHtml"
    tempHtml.textContent = text;

    // Utilisez .innerHTML pour récupérer le contenu de "tempHtml"
    // Cela va "neutraliser" ou "échapper" tout code HTML potentiellement malveillant
    return tempHtml.innerHTML;
}
```

Explication du code

- function sanitizeHtml(text) : c'est une fonction nommée sanitizeHtml qui prend en entrée une chaîne de texte potentiellement dangereuse, c'est-à-dire une chaîne qui pourrait contenir du code JavaScript malveillant.
- const tempHtml = document.createElement('div'); : cette ligne crée un élément HTML temporaire. Cet élément div sera utilisé pour manipuler le texte en tant qu'élément HTML au lieu de simplement le traiter comme une chaîne de caractères.

- tempHtml.textContent = text; : ici, le contenu textuel de l'élément div temporaire est défini comme étant la chaîne de texte fournie en entrée à la fonction. Cela signifie que le texte est inséré dans l'élément div, mais en tant que texte brut, pas en tant que code HTML.
- return tempHtml.innerHTML; : finalement, cette ligne renvoie le contenu de l'élément div temporaire en utilisant la propriété .innerHTML. Cela semble contre-intuitif, car nous aurions pu simplement retourner tempHtml.textContent. Cependant, en retournant .innerHTML, nous nous assurons que tout code HTML potentiellement malveillant est échappé ou neutralisé.

En quoi cela aide-t-il à prévenir les failles XSS ?

L'objectif principal de cette fonction est de neutraliser tout code HTML potentiellement malveillant qui pourrait être injecté dans le texte par un utilisateur. L'utilisation de textContent lors de l'insertion du texte dans l'élément temporaire garantit que le texte est traité comme du texte brut, sans être interprété comme du code HTML. Ensuite, en utilisant .innerHTML pour récupérer le contenu de l'élément temporaire, nous nous assurons que tout contenu potentiellement dangereux est échappé, ce qui signifie qu'il est converti en une représentation inoffensive qui ne sera pas interprétée comme du code HTML.

Cela empêche efficacement les attaquants de pouvoir injecter du code JavaScript malveillant ou des balises HTML dans le texte, réduisant ainsi considérablement le risque de failles XSS. En fin de compte, l'utilisation de cette fonction avant d'afficher du contenu utilisateur dans votre application web contribue à renforcer la sécurité en protégeant contre les attaques XSS.

III. Envoyer des requêtes authentifiées

A. Les jetons

Définition **Jeton d'authentification**

Les jetons d'authentification servent à sécuriser l'accès aux ressources et aux fonctionnalités de l'API en vérifiant l'identité de l'utilisateur ou de l'application qui effectue la demande. Il existe plusieurs types de jetons d'authentification, chacun ayant des caractéristiques et des utilisations spécifiques.

Voici les principaux types de jetons d'authentification :

- Token JWT (JSON Web Token) : les tokens JWT sont des jetons auto-contenus au format JSON. Ils sont largement utilisés pour l'authentification et l'autorisation dans les applications web et les services web. Un token JWT contient généralement des informations sur l'utilisateur et des claims (revendications) telles que les autorisations, la date d'expiration, etc. Les tokens JWT sont signés numériquement, ce qui les rend sécurisés et vérifiables.
- Token OAuth : les tokens OAuth sont utilisés dans le cadre du protocole OAuth pour autoriser des applications tierces à accéder aux ressources d'un utilisateur sans exposer les identifiants de l'utilisateur. Les tokens OAuth peuvent être de différents types, notamment les tokens d'accès, les tokens de rafraîchissement, les tokens de demande, etc.
- Token API : les tokens API, également appelés clés d'API, sont utilisés pour authentifier des applications ou des services tiers lorsqu'ils accèdent à une API. Ces tokens sont souvent inclus dans les en-têtes HTTP des requêtes API pour vérifier l'authenticité de l'appelant.

Méthode

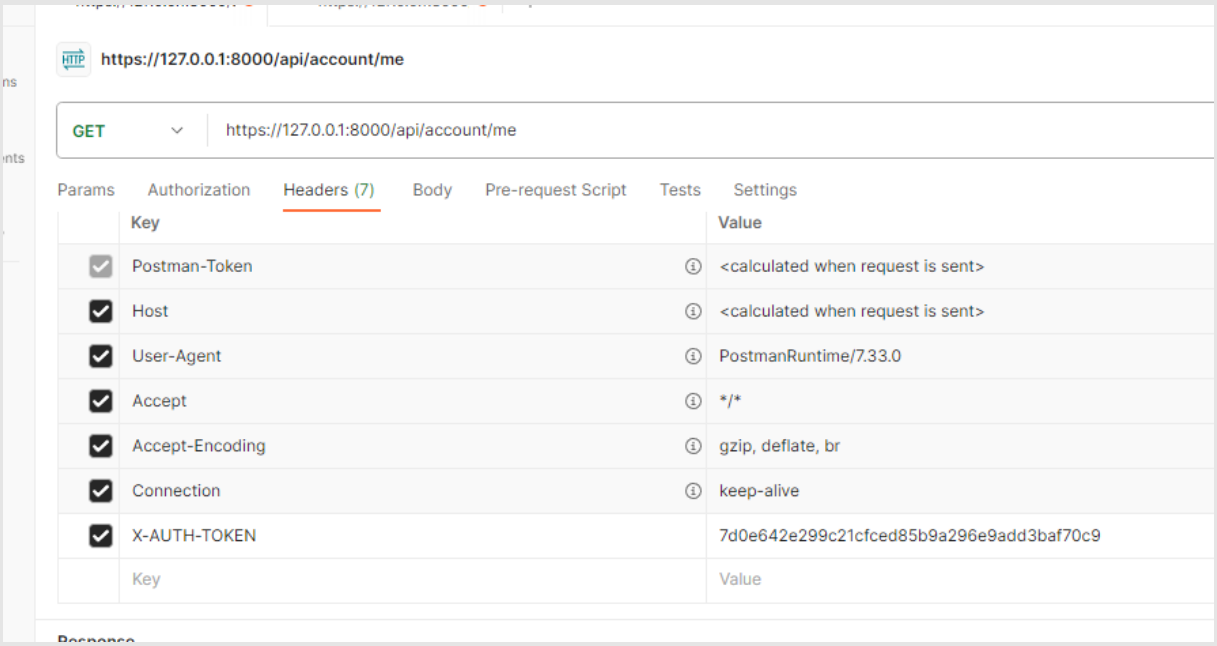
Dans notre application, nous allons utiliser les token API. Comment cela fonctionne ? Lorsque nous effectuons une connexion, notre API nous renvoie un token. Actuellement, nous le stockons en cookie. Ce token nous permet de valider notre identité. Nous le voyons ici dans l'attribut 'apiToken'. Lorsque nous enverrons une requête à notre API, nous devrons passer ce token dans le header de la requête. Ainsi l'API verra que nous possédons un token. Grâce à ce token, elle pourra nous identifier, et donc nous donner ou refuser l'accès à certaines routes.

Voici les données renvoyées par la méthode login :

```
{
    "user": "testdepuisPostman@email.com",
    "apiToken": "e4aba280522d9ab6721a3b92f89377a3108373a9",
    "roles": [
        "ROLE_USER"
    ]
}
```

Méthode

Si nous voulons accéder à une ressource dont nous n'avons pas l'accès, nous recevrons une réponse avec le code 401 'Unauthorized'. Pour accéder à une route qui nécessite une authentification, nous avons besoin d'ajouter l'en-tête 'X-AUTH-TOKEN', en lui spécifiant la valeur du token précédemment reçu lors du login.



B. Authentifier une requête

Méthode

Nous voulons maintenant créer une méthode permettant de récupérer les informations de l'utilisateur. Nous voulons exécuter cette requête HTTP depuis notre code JavaScript, via la méthode fetch.

Nous voulons ici faire un GET. Habituellement, pour faire un GET, nous aurions produit ce code :

```
let myHeaders = new Headers();
    let requestOptions = {
        method: 'GET',
        headers: myHeaders,
        redirect: 'follow'
    };

    fetch(apiUrl+"account/me", requestOptions)
    .then(response => return response.json();)
    .then(result => console.log(result))
    .catch(error =>{
```

```
        console.error("erreur lors de la récupération des données utilisateur", error);
    });
```

Or ce code ne gère pas l'authentification, tout comme dans postman, nous devons ajouter le header 'X-AUTH-TOKEN', grâce à cette ligne de code :

```
    myHeaders.append("X-AUTH-TOKEN", getToken());
```

Notre méthode pour récupérer les informations utilisateurs ressemble donc à cela :

```
function getInfosUser(){
    let myHeaders = new Headers();
    myHeaders.append("X-AUTH-TOKEN", getToken());

    let requestOptions = {
        method: 'GET',
        headers: myHeaders,
        redirect: 'follow'
    };

    fetch(apiUrl+"account/me", requestOptions)
    .then(response =>{
        if(response.ok){
            return response.json();
        }
        else{
            console.log("Impossible de récupérer les informations utilisateur");
        }
    })
    .then(result => {
        return result;
    })
    .catch(error =>{
        console.error("erreur lors de la récupération des données utilisateur", error);
    });
}
```

IV. L'essentiel

Ce cours nous a montré plusieurs aspects fondamentaux de la sécurité du code source dans le développement logiciel. Nous avons vu comment éviter les failles XSS sur notre site, en évitant qu'un utilisateur puisse enregistrer des données qui seraient exécutées sur notre site au final.

Nous avons vu aussi comment configurer un linter, qui est un outil d'analyse statique du code source qui inspecte le code pour détecter les erreurs de programmation, les violations de conventions de codage et autres problèmes potentiels. Enfin, nous avons vu comment authentifier de façon sécurisée nos requêtes grâce aux jetons d'authentification. Ces jetons permettent de vérifier l'identité de l'utilisateur ou de l'application qui effectue une requête. Lors de l'interaction avec une API, l'envoi du jeton d'authentification dans le header de la requête (souvent sous le nom 'X-AUTH-TOKEN') permet à l'API de valider l'identité de l'appelant et d'accorder ou refuser l'accès aux routes en conséquence, renforçant ainsi la sécurité de l'application.

V. Pour aller plus loin

A. Pour aller plus loin

- Corriger toutes les erreurs remontées par le linter
- Vérifier que toutes nos inclusions de HTML depuis le JavaScript utilisent bien la méthode sanitizeHtml