

Rapport Application web vulnérable

Groupe :

- Jules Poissonnet
- Ruddy Morel
- Lucas Damian-Picollet
- Romain Grange

Technologies utilisées :

Nuxt JS, Tailwind, Prisma

Présentation de l'app

L'application se veut être un twitter du pauvre avec trois pages: un login, un feed et un profil.

Le but est de pouvoir s'inscrire, se connecter, poster des tweets qui sont lisibles par tout le monde et visionner son profil avec uniquement ses tweets.

1. SQL injection :

Voici le code pour l'injection SQL que nous avons implémenté dans notre application vulnérable :

```
const [user] = await prisma.$queryRawUnsafe
(`SELECT * FROM "User" WHERE email = '${email}' AND password =
```

Le problème ici c'est que on laisse à l'utilisateur la possibilité de mettre ce qu'il veut dans l'email et on ne fait aucune vérification sur ce qu'il rentre, donc il peut directement agir sur le SQL.

Exemple :

On peut voir si l'on met un email aléatoire et un mot de passe aléatoire, on rajoute juste ceci dans le champ email :

```
' OR 1=1; --
```

': Cette apostrophe permet de fermer une chaîne de caractères dans une requête SQL.

': Une condition toujours vraie. Si elle est injectée dans une clause conditionnelle (comme dans un), elle force la condition à être vraie pour toutes les lignes d'une table.

': Indique la fin de la requête SQL

': C'est un commentaire en SQL. Tout ce qui suit dans la ligne sera ignoré par le serveur SQL.



Sign in to your account

Email address

jhlhlqlqdzjlkdzqj@gmail.com ' OR 1=1; --

Password

.....

Sign in

If you don't have an account? [Sign up](#)

Et on accède directement au site :

Full-stack developer passionate about creating amazing web experiences. Building the future of the web, one commit at a time. 🚀

Joined September 16, 2021

89 Following 12 Followers

Edit profile

En tant qu'admin vu que c'est la première ligne de notre table en base de donnée :

| | id | name | password | email | tweets |
|--|--------------------|----------------------|-----------------------------|-----------------------|------------------------|
| | 1 | admin | \$2b\$10\$x5KY014WpRLvn0... | a@a.fr | 0 Tweet |

Passons à la correction il y a plusieurs possibilité pour les corrigées ont peut utilisé les requêtes paramétrées la ici avec des placeholders :

```
const [user] = await prisma.$queryRaw(`SELECT * FROM "User" WHERE email = $1 AND password = $2` ,
```

```
    email,  
    encrypted  
);
```

Ici les placeholders \$1 et \$2 signifie que le moteur SQL traitera ces valeurs comme des données brutes et non comme du code SQL.

Nous nous avons choisi d'utiliser les méthodes ORM natives de Prisma comme ci dessous car elles intègrent nativement des protections contre les injections SQL :

```
const user = await prisma.user.findUnique({  
  where: {  
    email  
  }  
});
```

2. XSS stockée sur la page index

La faille XSS stockée est due à l'aspect communautaire qui fait que l'input de l'utilisateur se retrouve sur le site final de tous les autres utilisateurs. Le risque est donc qu'un utilisateur malveillant dépose dans la base un post qui ait des effets de bords tels que du code.

La faille est présente dans le composant Tweet qui affiche ce qu'il reçoit de la base de donnée:

Tweet.vue

```
<p class="text-gray-900 mt-1" v-html="content"></p>
```

Le souci ici réside dans l'utilisation de v-html qui injecte directement le HTML dans le innerHTML du DOM. Heureusement le framework Vue ne permet pas l'exécution des balises `<script>` directement mais ça n'est pas la seule façon de faire exécuter du code. En effet, on peut par exemple ajouter une balise `` avec un onerror.

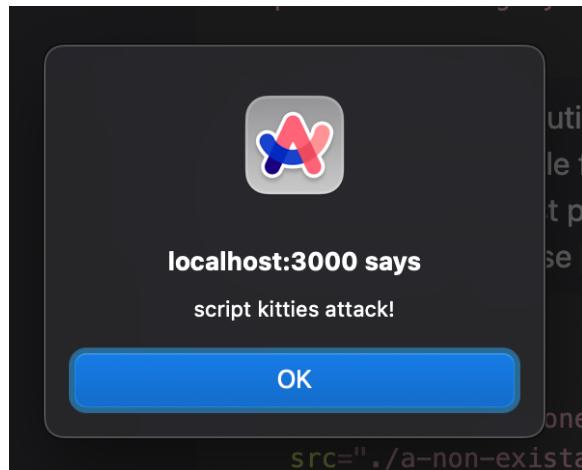
```
>
```

Home My Profile

 >

Post

En postant ce message, à chaque refresh l'utilisateur vera apparaitre une alerte:



Il est bon de noter que les balises `<style>` marchent également, donc on pourrait imaginer changer complètement la face du site.

Pour remédier cette faille, il faut utiliser une interpolation sécurisée pour éviter l'injection de script. Avec Vue.js, on peut tout simplement insérer le contenu dans le template comme suit:

```
<p class="text-gray-900 mt-1">{{ content }}</p>
```

3. IDOR (Insecure Direct Object Reference)

L'IDOR est une faille sensible, car n'importe quel utilisateur pourrait avoir accès à un objet auquel il ne devrait pas avoir accès (par exemple, des données

utilisateur, des fichiers, etc.). Ces failles sont principalement causées par des manipulations d'ID dans l'URL, par exemple.

Voici l'url de l'utilisateur 1 : <http://localhost:3000/profile/1>

Voici l'url de l'utilisateur 2 : <http://localhost:3000/profile/2>

Chacune de ces URL est accessible par n'importe qui, mais une vérification des autorisations est nécessaire afin d'éviter d'ouvrir l'accès à des données non autorisées.

La faille est dans le serveur 

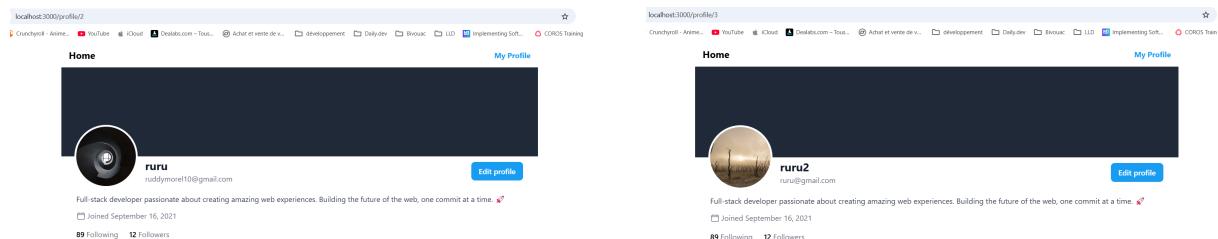
```
const id = getRouterParam(event, 'id')
const user = await prisma.user.findUnique({
  select: {
    id: true,
    name: true,
    email: true
  },
  where: {
    id: parseInt(id)
  }
});
```

La récupération de l'utilisateur s'effectue via un paramètre qui peut être manipulé par l'utilisateur, sans qu'aucune vérification d'autorisation ne soit effectuée.

Pour remédier à cette faille, il est possible d'utiliser le JWT pour récupérer l'ID directement depuis celui-ci, garantissant ainsi qu'il est propre à l'utilisateur connecté.

Voici l'exemple de code sécurisé dans le projet  [secure](#)

 me.get.js



```

const token = getHeader(event, 'Authorization').replace('Bear');

const decoded = jwt.verify(token, 'pouetpouetpouet');

const user = await prisma.user.findUnique({
  select: {
    id: true,
    name: true,
    email: true
  },
  where: {
    id: decoded.userId
  }
});

```

Ici le JWT est vérifier et décodé afin d'extraire l'ID de l'utilisateur et de retourné une donnée dont il a l'autorisation.

4. Hashage des mots de passe MD5

Une autre faille serait de hasher les mots de passe en MD5. Bien que le hashage soit une bonne pratique, l'utilisation de MD5 est inutile, car cet algorithme de hashage est obsolète et a été compromis.

Voici l'exemple de code avec la faille

```

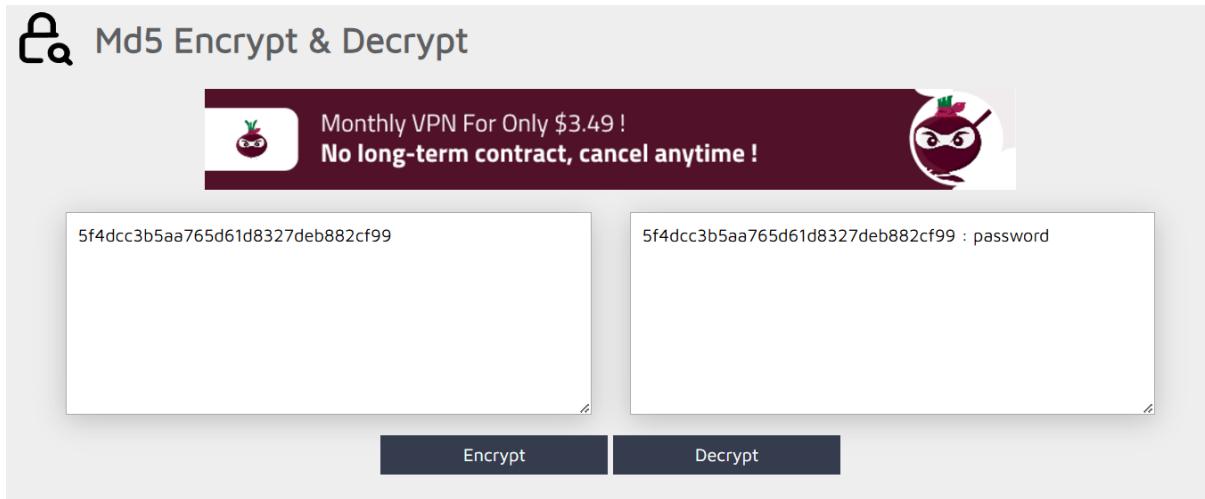
const hashPassword = async (plainPassword) => {
  try {
    // Utilisation de MD5 pour le hashage du mot de passe
    return crypto.createHash('md5').update(plainPassword)
  } catch (error) {
    throw error;
  }
};

```

On peut noter que dans cette implémentation du hash, aucun sel n'est utilisé, ce qui rend le mot de passe presque équivalent à un texte en clair dans la base de données.

Résultat du hash de password : **5f4dcc3b5aa765d61d8327deb882cf99**

Son simple passage dans le premier site de décryptage nous donnera le mot de passe.



The screenshot shows a web application titled "Md5 Encrypt & Decrypt". At the top, there is a banner for a monthly VPN service with the text "Monthly VPN For Only \$3.49!" and "No long-term contract, cancel anytime!". Below the banner, there are two input fields. The left field contains the MD5 hash "5f4dcc3b5aa765d61d8327deb882cf99". The right field contains the decrypted password "5f4dcc3b5aa765d61d8327deb882cf99 : password". At the bottom of the page are two buttons: "Encrypt" and "Decrypt".

Et voici le code sécurisé

```
const hashPassword = async (plainPassword) => {
  try {
    const salt = bcrypt.genSaltSync(saltRounds);
    return bcrypt.hashSync(plainPassword, salt);
  } catch (error) {
    throw error;
  }
};
```

Ici nous utilisons bcrypt, une bibliothèque de hashage sécurisée qui propose des solutions simples pour le hashage et l'ajout de sel.

Résultat du hash password :

\$2b\$10\$ffJvYKGc01pr5hScw95X0u2215dQgQcUdqRShgTtKfJ5vFc0uAbri

Beaucoup plus complexe à décrypter.

5. Exposition de données sensibles

Une faille courante consiste à retourner des informations sensibles non nécessaires au client. Ces données peuvent inclure des mots de passe hashés,

des tokens, ou d'autres champs internes qui ne devraient pas être accessibles par un utilisateur.

Voici l'exemple de code avec la faille

```
const user = await prisma.user.findUnique({  
  where: {  
    id: parseInt(id)  
  }  
});
```

Dans cet exemple, l'API retourne toutes les informations de l'utilisateur, y compris le mot de passe hashé. Cela expose inutilement des données sensibles dans la réponse de l'API.

Exemple de réponse de l'API avec la faille

```
{  
  "user": {  
    "id": 2,  
    "name": "ruru",  
    "password": "$2b$10$TA3iNkq/3t8RuiwBRTR7M0Yxb5jIda/wpx0ix:  
    "email": "ruddymorel10@gmail.com"  
  }  
}
```

Dans cette réponse, bien que le mot de passe soit hashé, son inclusion dans la réponse est une faille de sécurité. Si l'API est compromise ou si des journaux capturent cette réponse, des attaques par force brute ou par dictionnaire peuvent être effectuées sur les hash.

Voici le code sécurisé

```
const user = await prisma.user.findUnique({  
  select: {  
    id: true,  
    name: true,  
    email: true  
  },  
  where: {
```

```
    id: decoded.userId  
  }  
});
```

Dans cette version, seuls les champs nécessaires sont retournés (id, name, email). Les informations sensibles comme le mot de passe ou d'autres champs internes ne sont pas incluses dans la réponse.

Exemple de réponse de l'API corrigée

```
{  
  "user": {  
    "id": 2,  
    "name": "ruru",  
    "email": "ruddymorel10@gmail.com"  
  }  
}
```

6. Absence de limitation des requêtes

Une autre faille courante dans les applications web est l'absence de mécanismes pour limiter le nombre de requêtes effectuées par un utilisateur ou une adresse IP sur une période donnée. Cette faille peut permettre à un attaquant de :

- **Saturer le serveur** : En envoyant un grand nombre de requêtes en un temps court, un attaquant peut provoquer une surcharge du serveur, rendant l'application indisponible pour les autres utilisateurs (**attaque par déni de service, DoS**).
- **Exploiter des fonctionnalités critiques** : Par exemple, tester un grand nombre de combinaisons pour des attaques par force brute (comme deviner des mots de passe ou des jetons d'authentification).
- **Augmenter la consommation des ressources** : Cela peut impacter les coûts en cloud computing si les ressources sont facturées à l'utilisation.

Solution pour corriger cette faille

Implémenter un système de limitation des requêtes