Analyse sécurité

Leonardoken D'ALESSIO et Karl ENGELBRECHT

Injection

Non sécurisé

Imaginons qu'un utilisateur malveillant entre ceci : http://localhost:3000/job-application/34;%20SELECT%20email,%20pwd,%20lastname,%20firstname,%20phonenumber,%20dateofcreation,%20status%20FROM%20User;%20–

Ceci pourrait conduire à recevoir les données de tous les utilisateurs de la base de données à cause de l'injection.

Comment nous avons sécurisé

Nous avons implémenté une sécurisation en 2 étapes. La première étape est de sécuriser toutes les requêtes via des requêtes paramétrées, comme ceci ;

```
try {
  const [results, ...] = await db.query(
    `
    SELECT * FROM Organisation
    WHERE Organisation.status = ?
    AND Organisation.SIREN NOT IN
    (SELECT RecruiterOrganisation.idorganisation
    FROM RecruiterOrganisation
    WHERE RecruiterOrganisation.idrecruiter = ?)`,
    [status, idRecruiter]
  );
```

La deuxième étape se trouve dans la couche applicative. En effet, tous les paramètres rentables dans l'URL sont, dans le cas de notre application, automatiquement convertis en Number en Javascript. Ainsi, même si la première protection ne fonctionne pas, au niveau

applicatif, le texte "dangereux"est converti en chiffre, ce qui donne une erreur de conversion, et donc crash le site au niveau client.

Exposition de données sensibles

Non sécurisé

Dans la version non sécurisée, imaginons qu'un utilisateur malveillant (ou même un développeur qui a les droits d'accès) entre dans la base de données.

Sur la table des User, il pourrait alors voir ceci :

33	non-securise@gmail.com	monmotdepassesupersecurise	Dupont	Jean	0677777777	2024-06-21	1

Bien que le mail, le nom, prénom et numéro de téléphone ne soient pas forcément des données sensibles, le simple fait que le mot de passe soit écrit tel quel laisse à l'utilisateur malveillant la possibilité d'accéder au compte et de réaliser des dégâts.

Comment nous avons sécurisé

Nous avons alors implémenté un hachage du mot de passe à l'aide de l'algorithme BCrypt. Voici un exemple d'entrée sur la table User :

18	admin@gmail.com	$2b$10$mRNrqzyOY/P..2zbZ1IWgeWHcGjHYief9h341McvtCb...	Lê	Taline	06666666	2024-06-05	1

Ainsi, même avec un accès à la base de données, il devient impossible de savoir le mot de passe des utilisateurs.

Il y a alors 2 étapes en plus :
Lors de la création de compte, il devient nécessaire de hacher le mot de passe :

```
bcrypt.hash(password, saltRounds, cb: async (err, hash) : Promise<void>  => {
    await User.create(email, hash, lastName, firstName, phoneNumber);
    const userId : ...  = await User.getIdByEmail(email)
    await Applicant.create(userId);
});
```

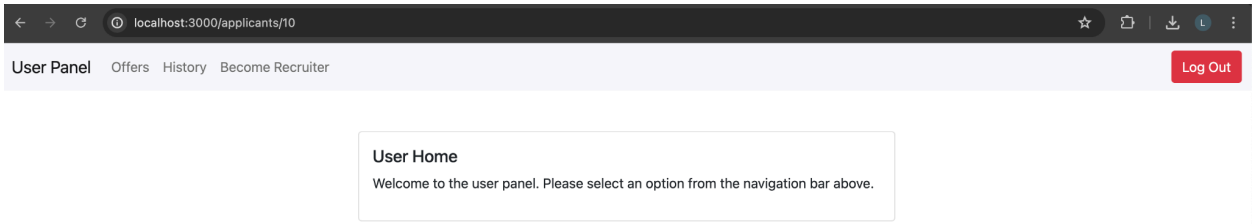
Et lors du login, il devient nécessaire de comparer avec le hash :

```
const match = await bcrypt.compare(pwd, results[0].pwd);
```

Violation de contrôle d’accès

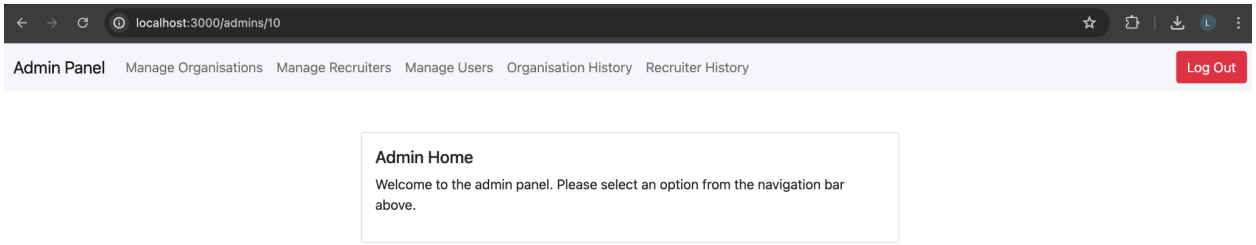
Non sécurisé

Imaginons qu'un utilisateur "Applicant" (candidat) aille sur notre site, il sera mené vers l'interface basique candidat suivante :



Elle pourrait alors se dire qu'il existe forcément un rôle "admin", et que l'interface d'un administrateur aura pour structure :
admins/numeroAdmin

Il s'exécute, et tombe sur la page suivante :



Ainsi, il peut usurper le rôle de l'administrateur d'identifiant 10, et avoir accès à toutes les fonctionnalités administrateur, ce qui est très dangereux.

Comment nous avons sécurisé

Rappelons que dans notre application, la session d'un utilisateur aura la structure suivante :

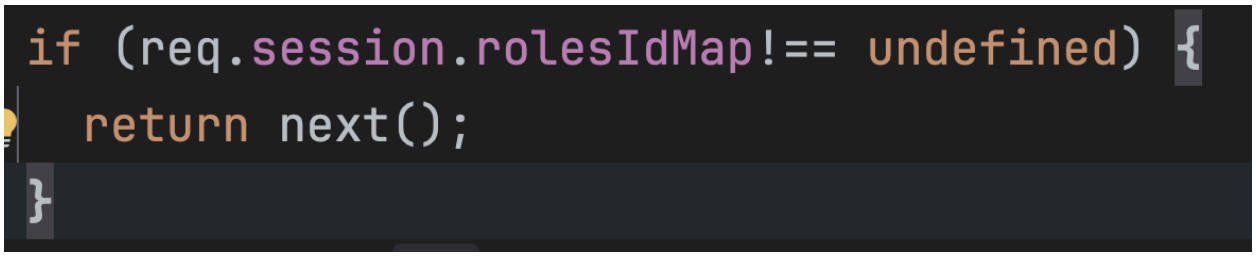
```
{
        "email" : …      ,
        "rolesIdMap": {
                "adminId' : …  ,
                "recruiterId": …          ,
                "applicantId": …
        }
}
```

Ainsi, pour chaque utilisateur on lui associe à sa session son id associé à son rôle.

Dans le cas précédent, la session ressemblerait à ceci :

```
{
        "email" : "ronaldo@gmail.com",
        "rolesIdMap": {
                "adminId' : null,
                "recruiterId": null,
                "applicantId": 10
        }
}
```

Dans notre version sécurisée, il y a une vérification sur 2 niveaux pour les routes sécurisées. Premièrement, on vérifie si l'utilisateur a bien login :



```
if (req.session.rolesIdMap!== undefined) {
  return next();
}
```

Deuxièmement, pour chaque route nécessitant un rôle, on vérifie la présence du rôle et de la correspondance entre l'id entré dans la barre de navigateur et l'id du rôle associé dans la session. Voici un exemple pour admin :

```
Leonardo +1
app.all( path: "/admins/:idAdmin*",  handlers: (req : Request<P, ResBody, ReqBody, R
  if (Number(req.params.idAdmin) === req.session.rolesIdMap.adminId) {
    return next();
  }
  res.redirect( url: "/login");
});
```

Ceci permet d'éviter à un utilisateur qui a bien un rôle admin d'accéder au compte admin d'un autre utilisateur.
Au cas où une des conditions n'est pas remplie, l'utilisateur est automatiquement redirigé vers la page de login.