Semestre P24

SR10

Analyse de Sécurité

Etudiants :
Raphael Nguyen
Raphael Chauvier
Raphael Chauvier Chauvier

Vulnérabilité : Violation de contrôle d’Accès

Exemple d’attaque :

Un utilisateur malveillant a connaissance d’un ensemble d’urls, de noms de fichiers, d’organisations, de tout un ensemble de données sauf le mot de passe de l’utilisateur qui a réellement les droits d’accès.

Il ne peut pas s’authentifier, mais peut tenter de simuler des requêtes pour effectuer une action particulière.

Il accède par exemple à la route “/cancel-application” qui pour l’exemple n’est pas sécurisée (**elle l’est en pratique dans notre site**) et envoie un identifiant arbitraire d’un utilisateur et d’une offre d’emploi, imaginons qu’il répète la requête avec chaque couple d’identifiant possible pour détruire toutes les candidatures. **Puisque la route n’est théoriquement pas sécurisée, et qu’aucune vérification de la légitimité de l’utilisateur n’a été effectuée, il parviendra à ses fins.**

Ne pas procurer de lien à ses pages dans l’interface des utilisateurs concernés n’est pas suffisant, il faut prévenir l’accès aux URLs interdites à certains utilisateurs.

Nous allons l’empêcher d’atteindre ces pages en établissant un critère d’évaluation stocké dans sa session côté serveur, que nous confrontons à des modalités d’accès définis pour chaque page ou groupe de page, ainsi que des **vérifications dans les routes** qui portent sur des informations de sessions générées lors du **processus d’authentification**.

Par exemple, seuls des utilisateurs connectés à une session recruteur avec un identifiant d’organisation peuvent accéder à la page “add_job”, et ceux qui peuvent accéder à **/cancel-application** doivent avoir un identifiant de session en accord avec les données fournies dans le corps de leur POST.

Express-session :

Les données d’authentification nécessaires pour contrôler l’accès des utilisateurs aux différentes pages sont stockées dans des express-session.

Ces sessions gérées côté serveur ne peuvent pas être modifiées en dehors du cadre permis par les opérations côté serveur, potentiellement en réaction à un input client mais très facilement contrôlable.

Nous nous en servons en particulier pour stocker une information userEmail qui permet d'identifier l'utilisateur en cours, un userType pour gérer son type d'utilisateur, et enfin un userAffiliation qui stocke éventuellement l'identifiant de l'organisation à laquelle il appartient. Ces informations sont tirées des données de la base de données au moment de l'authentification et ne sont donc jamais saisies directement (ni simplement envoyées) par l'utilisateur. Cela permet d'isoler les valeurs de session d'inputs utilisateur.

Si nous contrôlons correctement le processus de création d'utilisateur (ce qui est normalement le cas), nous protégeons efficacement les données de session.

Niveaux de contrôle d'accès :

Nous avons défini un ensemble de chemins d'accès "nonSecure" c'est à dire accessibles sans se connecter, l'ensemble des autres pages ont un comportement par défaut qui redirige l'utilisateur vers la page de connexion.

Nous avons ensuite établi une série de middlewares que nous ajoutons manuellement à certaines pages pour gérer leur niveau d'accès. Ces middlewares s'assurent de rediriger correctement les utilisateurs non compatibles vers les pages d'erreur.

```
async function requireAffiliation(req, res, next) : Promise<void>  {  Show usages  Raph +1
    try {
        if (req.session.userAffiliation) {
            next();
        } else {
            if (! req.session.userEmail) {
                req.session.returnTo = req.originalUrl || '/'; // save current page
                res.redirect('/login');
            }
            else if (req.session.userType === 'recruteur') {
                let error : Error  = new Error();
                error.status = 500;
                error.message = 'Erreur interne du serveur : recruteur sans email';
                next(error);
            } else {
                let error : Error  = new Error();
                error.status = 403;
                error.message = 'Vous n\'avez pas le droit d\'accéder à cette page.';
                next(error);
            }
        }
    } catch (e) {
        next(e);
    }
```

Middleware de vérification de statut recruteur et affiliation à middlewares/requireAffiliation.js

Vérifications dans la base de données :

Pour certaines situations comme **/cancel-application** ou **/download-attachment** (toutes deux dans routes/applications.js), nous avons établi des tests additionnels. Ces tests sont implémentés dans les routes en termes de logique booléenne, et dans le modèle associé en termes de logique SQL.

```
} else if (req.session.userType === 'recruteur') {
    if (await OffreEmploi.isUserInOrganisation(idOffre, req.session.userEmail)) {
        // ne supprime que si le recruteur est membre de l'organisation de l'offre
        await Candidature.delete(idCandidat, idOffre);
        req.session.message = `Vous avez refusé la candidature de ${await Utilisateur.getN
        req.session.messageType = 'notification';
    } else {
        req.session.message = 'Vous n\'êtes pas autorisé à annuler cette candidature';
        req.session.messageType = 'error';
    }
```

Test dans la route /cancel-application du fichier routes/applications.js

Par exemple, dans la route /cancel-application, nous testons si l'utilisateur recruteur qui tente d'effectuer l'action appartient bien à l'organisation dont la fiche poste liée à l'offre est issue. Pour cela nous faisons appel à la méthode **isUserInOrganisation** que nous avons implémenté au niveau du modèle offre d'emploi, et nous lui fournissons l'identifiant de l'utilisateur **issu de la session sécurisée** (**et non de son input via son POST.body**) pour vérification dans la base de données.

Ainsi, la vérification est systématiquement effectuée avec des données sécurisées de session, et ne peuvent donc pas être falsifiées à moins que l'utilisateur malveillant possède le **mot de passe** de sa cible.

```
async isUserInOrganisation(idOffre, userEmail) : Promise<boolean>  {
    const query : string  = `
        SELECT COUNT(*) FROM OffreEmploi AS O
        JOIN FichePoste AS F ON O.IdFiche = F.IdFiche
        JOIN Organisation AS Org ON F.IdOrganisation = Org.NumeroSiren
        JOIN Utilisateur AS U ON Org.NumeroSiren = U.IdOrganisation
        WHERE O.IdOffre = ? AND U.Email = ?;
    `;
```

Requête SQL associée à l'exemple précédent, pour référence.

Test et prévention des erreurs de programmeur :

Finalement, afin de nous assurer que la sécurisation des routes soit correctement implémentée, malgré d'éventuels bugs qui pourraient survenir au cours du développement, nous avons implémenté des tests spécifiquement centrés sur la sécurité des pages dans nos tests de routing.

Au cours de ces tests, nous présentons différents états de sessions qui peuvent être construits par une diversité d'utilisateurs via le processus de login (sessions candidat, sessions recruteur, …) et nous vérifions que les pages répondent bien positivement ou négativement en fonction.

Plus de détails sont fournis dans notre document sur les tests.

```
it('candidates should have receive a 500 error code denying illegal access', async ()
    await agent.get('/test/mock-candidat');
    const res = await agent.get('/jobs/add_job');
    expect(res.statusCode).toBe( expected: 403);
});
```

Test de sécurité implémenté dans tests/routing/jobs.test.js associé à la route /jobs/add_job qui doit refuser les utilisateurs qui ne sont pas recruteurs.

Vulnérabilité : Stockage de mots de passe en clair

Exemple d'attaque :

Sans aucune protection, les mots de passe des utilisateurs peuvent être stockés en clair dans la base de données, ce qui rend votre application vulnérable à une compromission de la base de données.

Code vulnérable :

```
// Stockage des mots de passe en clair
function storePassword(username, password, callback) {
  const connection = getDbConnection();
  connection.query(
      "INSERT INTO users (username, password) VALUES (?, ?)",
      [username, password],
      function (err, results) {
      if (err) throw err;
      callback(results);
      }
  );
}

function checkPassword(username, password, callback) {
  const connection = getDbConnection();
  connection.query(
      "SELECT password FROM users WHERE username = ?",
      [username],
      function (err, results) {
      if (err) throw err;
      const storedPassword = results[0].password;
      callback(storedPassword === password);
      }
  );
}
```

Protection avec bcrypt :

Pour protéger les mots de passe, nous allons utiliser la bibliothèque bcrypt pour les hacher avant de les stocker dans la base de données. Le hachage des mots de passe garantit que même si la base de données est compromise, les mots de passe ne seront pas lisibles en clair.

```
const bcrypt = require("bcrypt");

// Fonction pour générer un hash de mot de passe et le stocker
function storePassword(username, plaintextPassword, callback) {
  bcrypt.hash(plaintextPassword, 10, function (err, hash) {
      if (err) throw err;
      const connection = getDbConnection();
      connection.query(
      "INSERT INTO users (username, password) VALUES (?, ?)",
      [username, hash],
      function (err, results) {
      if (err) throw err;
      callback(results);
      }
      );
  });
}

// Fonction pour vérifier le mot de passe
function checkPassword(username, plaintextPassword, callback) {
  const connection = getDbConnection();
  connection.query(
      "SELECT password FROM users WHERE username = ?",
      [username],
      function (err, results) {
      if (err) throw err;
      const storedHash = results[0].password;
      bcrypt.compare(plaintextPassword, storedHash, function (err, result)
{
      if (err) throw err;
      callback(result);
      });
      }
  );
}
```

Vulnérabilité : Injection SQL

L'injection SQL est une technique d'attaque où un attaquant peut insérer ou "injecter" du code SQL malveillant dans une requête. Cette attaque peut mener à des fuites de données, des modifications non autorisées ou même des suppressions de données critiques.

Exemple d'attaque :

Voici un exemple de code vulnérable à l'injection SQL. Ce code ne traite pas correctement l'entrée de l'utilisateur, permettant ainsi à un attaquant d'injecter des commandes SQL.

Code vulnérable :

```
async function getUserByEmail(email) {
      const query = `SELECT * FROM Utilisateur WHERE Email =
'${email}';`;
      const [results] = await pool.query(query);
      return results;
}

// Appel vulnérable
getUserByEmail("test@example.com' OR '1'='1");
```

Dans cet exemple, un attaquant pourrait passer `"test@example.com' OR '1'='1"` comme email, ce qui entraînerait l'exécution de la requête suivante :

```
SELECT * FROM Utilisateur WHERE Email = 'test@example.com' OR
'1'='1';
```

Cela renverrait toutes les lignes de la table `Utilisateur`, car la condition `OR '1'='1'` est toujours vraie.

Protection contre l'injection SQL :

Pour éviter l'injection SQL, il est essentiel d'utiliser des requêtes paramétrées. Cela permet de s'assurer que les entrées des utilisateurs sont traitées correctement et échappées avant d'être exécutées dans la base de données.

Code protégé :

```
async function getUserByEmail(email) {
      const query = `SELECT * FROM Utilisateur WHERE Email = ?;`;
      const [results] = await pool.query(query, [email]);
      return results[0];
}
```

Explication :

1. **Utilisation de paramètres** : Au lieu d'insérer directement l'email dans la requête, nous utilisons un paramètre (`?`) pour sécuriser la requête.
2. **Passage des valeurs séparément** : Les valeurs de paramètres sont passées dans un tableau à `pool.query`, ce qui garantit qu'elles sont correctement échappées et sécurisées par la bibliothèque de gestion de base de données.

En utilisant des requêtes paramétrées, nous nous assurons que les entrées des utilisateurs ne peuvent pas altérer la structure de la requête SQL, empêchant ainsi les injections SQL.