Exercice

Objectif de l'exercice :Concevoir et développer un formulaire de contact sécurisé qui permet aux utilisateurs de soumettre leurs informations de contact de manière sécurisée.

Résultat attendu : Un formulaire de contact web fonctionnel et sécurisé qui permet aux utilisateurs de soumettre leurs informations de contact de manière sécurisé.

L'application est conçue et développée en suivant les principes de la sécurité by désigne donc garantir une résistance aux attaques.

Compte rendu

Politique de Sécurité du Contenu (CSP)

La Politique de Sécurité du Contenu (CSP) est un ajout important à la sécurité de notre application web. Elle aide à détecter et atténuer certains types d'attaques, notamment les Cross-Site Scripting (XSS) et les injections de données. Dans le cadre de ce projet, une CSP stricte a été mise en place via l'utilisation du middleware helmet dans Express. Voici un exemple de configuration CSP utilisée :

```
app.use(
  helmet.contentSecurityPolicy({
    directives: {
      defaultSrc: ["'self'"],
      scriptSrc: ["'self'", "https://trusted.cdn.com"],
      styleSrc: ["'self'", "https://trusted.cdn.com"],
      imgSrc: ["'self'", "data:", "https://trusted.cdn.com"],
      connectSrc: ["'self'", "https://api.trusted.com"],
    },
  })
);
```

Cette configuration assure que seuls les scripts, styles, images, et requêtes AJAX provenant des sources spécifiées (le propre domaine de l'application et des CDN/ APIs de confiance) peuvent être exécutés ou inclus, renforçant ainsi la sécurité contre les injections malveillantes.

Mesures de Sécurité au Niveau du Code

Plusieurs mesures de sécurité ont été implémentées au niveau du code pour assurer la protection de l'application :

- Validation et Assainissement des Entrées : Utilisation de express-validator pour valider et nettoyer les données reçues des utilisateurs, prévenant ainsi les injections SQL et XSS.
- Cryptage des Données : Les messages des utilisateurs sont cryptés avant d'être stockés dans la base de données, utilisant crypto pour le cryptage, garantissant la confidentialité des données sensibles.
- HTTPS : Mise en place d'une communication sécurisée via HTTPS, en générant et utilisant des certificats SSL/TLS pour le développement local et en s'assurant que la production utilise également HTTPS, protégeant ainsi les données en transit.
- Utilisation de Helmet : Configuration de divers en-têtes de sécurité HTTP via helmet pour protéger l'application contre des vulnérabilités communes, comme le sniffing de type MIME, le clickjacking, et d'autres.

Attaques Prévenues

- Cross-Site Scripting (XSS) : Grâce à l'assainissement des entrées utilisateur et à une CSP stricte, l'application est protégée contre les tentatives d'injection de scripts malveillants.

```
app.post('/send-message', [
    body('name').trim().escape(),
    body('email').trim().escape(),
    body('message').trim().escape(),
], (req, res) => {
    // Logique de traitement...
});
```

  Ici, .trim().escape() assure que les entrées utilisateur sont nettoyées de tout code potentiellement malveillant, empêchant ainsi les attaques XSS.
- Injection SQL : Bien que l'application utilise MongoDB (qui n'est pas vulnérable aux attaques SQL classiques), la validation et l'assainissement des entrées protègent contre des injections potentielles dans d'autres parties de l'application ou dans des futures intégrations de bases de données relationnelles.

```
body('email').isEmail().normalizeEmail(),
```

  isEmail() et normalizeEmail() aident à s'assurer que l'entrée est bien formatée comme un email, réduisant le risque d'injections.

- Man-in-the-Middle (MitM) : L'utilisation de HTTPS avec des certificats SSL/TLS assure que les données échangées entre le client et le serveur sont cryptées, empêchant les attaques MitM.

```
const server = https.createServer({
  key: pem.private,
  cert: pem.cert
}, app);

server.listen(3000, () => {
  console.log('Server running on https://localhost:3000');
});
```

```
// Générer une paire de clés et un certificat auto-signé
const pki = forge.pki;
const keys = pki.rsa.generateKeyPair(2048);
const cert = pki.createCertificate();
cert.publicKey = keys.publicKey;
cert.serialNumber = '01';
cert.validity.notBefore = new Date();
cert.validity.notAfter = new Date();
cert.validity.notAfter.setFullYear(cert.validity.notBefore.getFullYear() + 1);
const attrs = [{
  name: 'commonName',
  value: 'localhost'
}, {
  name: 'countryName',
  value: 'US'
}, {
  shortName: 'ST',
  value: 'Virginia'
}, {
  name: 'localityName',
  value: 'Blacksburg'
}, {
  name: 'organizationName',
  value: 'Test'
}, {
  shortName: 'OU',
  value: 'Test'
}];
cert.setSubject(attrs);
cert.setIssuer(attrs);
cert.sign(keys.privateKey);

const pem = {
  private: pki.privateKeyToPem(keys.privateKey),
  cert: pki.certificateToPem(cert)
};
```

La mise en place d'HTTPS chiffre la communication entre le client et le serveur, protégeant ainsi contre les écoutes clandestines et les attaques MitM.

- Clickjacking : L'en-tête X-Frame-Options configuré via helmet empêche le contenu de l'application d'être embarqué dans des iframes sur des sites tiers, protégeant contre les attaques de clickjacking.

```
app.use(helmet.frameguard({ action: 'deny' }));
```

  Cet en-tête informe les navigateurs de ne pas embarquer les pages de l'application dans des iframes, ce qui empêche les attaques de clickjacking où un attaquant pourrait tromper un utilisateur en cliquant sur quelque chose de différent de ce que l'utilisateur croit.

Ces mesures, combinées à une approche globale de sécurité dès la conception, assurent une protection solide de l'application contre une variété d'attaques courantes, tout en garantissant une expérience utilisateur sûre et fiable.

Tests Réalisés

Tests de vulnérabilité automatisés (entre autres) :

- Intégration d'ESLint et du Plugin de Sécurité dans le Projet de Formulaire de Contact

Objectif :

L'objectif de l'intégration d'ESLint avec un plugin de sécurité est d'identifier et de corriger proactivement les vulnérabilités de sécurité potentielles dans le code source de notre application de formulaire de contact. Cela aide à garantir que l'application est développée avec les meilleures pratiques de sécurité dès le départ, réduisant ainsi le risque d'exploitations malveillantes.

Lorsque l'on utilise ESLint au sein du projet, il détecte les failles dans le code, et on peut programmer une automatisation à chaque action comme la modification de fichier, l'ouverture de nouveaux programmes / leur exécution, ce qui a permis de pallier à certains problèmes et palliera surement à beaucoup d'autres dans le futur.

- Injections manuelles

Nous simulons des attaques d'injections SQL et XSS en injectant délibérément du code malveillant dans les champs d'entrée, afin de tester la robustesse de notre validation et de notre assainissement des données. Cette méthode nous permet de détecter et de corriger les failles potentielles, assurant ainsi que notre application peut efficacement contrer les tentatives d'injections malveillantes.