l'EPSI, l'école d'ingénierie informatique
Expert en informatique et sytémes d'information -Fast Track

ÉVALUATION FINALE

Sécurité by design

Réalisé par :

— HAMZA Lydia

Présenté pour Mr : Mme Amina MARIE

Date : 11 mai 2025

Table des matières

Introduction

Dans un contexte où la cybersécurité est devenue une préoccupation majeure pour toutes les organisations, il est essentiel d'intégrer les bonnes pratiques de sécurité dès les premières phases de développement d'une application. Ce projet s'inscrit dans cette logique : concevoir une application web simple – un formulaire de contact – tout en appliquant des principes stricts de sécurisation, conformément aux recommandations de l'OWASP et aux standards modernes du développement sécurisé.

L'objectif était de construire un environnement de travail local (sans base de données externe), permettant de manipuler les concepts de sécurité tout en assurant la confidentialité, l'intégrité et la disponibilité des données utilisateurs.

Objectifs pédagogiques

Ce projet a pour finalité pédagogique de permettre aux étudiants de :

- **Comprendre les principales vulnérabilités** d'une application web : XSS, injections, attaques par session, etc.
- **Mettre en œuvre des contre-mesures concrètes**, notamment à l'aide de bibliothèques et outils de l'écosystème Node.js.
- **Appliquer les principes de sécurité dans l'architecture même du code** : depuis le stockage des données jusqu'à la transmission HTTPS.
- **Tester et analyser une application via des outils spécialisés**, tel que Chrome DevTools.

En résumé, il ne s'agit pas uniquement de coder un formulaire, mais bien de le concevoir comme un système sécurisé à part entière.

1 Présentation fonctionnelle du projet

1.1 Description des pages développées

1.1.1 Page d'inscription (/register.html)

Cette page permet à un nouvel utilisateur de créer un compte. Le formulaire comporte les champs suivants :

— Nom d'utilisateur

— Adresse e-mail

— Mot de passe

— reCAPTCHA v2 (Google)

Le mot de passe est immédiatement haché côté serveur à l'aide de bcrypt avant d'être stocké localement dans un fichier users.json. Une vérification du reCAPTCHA est également effectuée côté serveur via une requête à l'API Google.

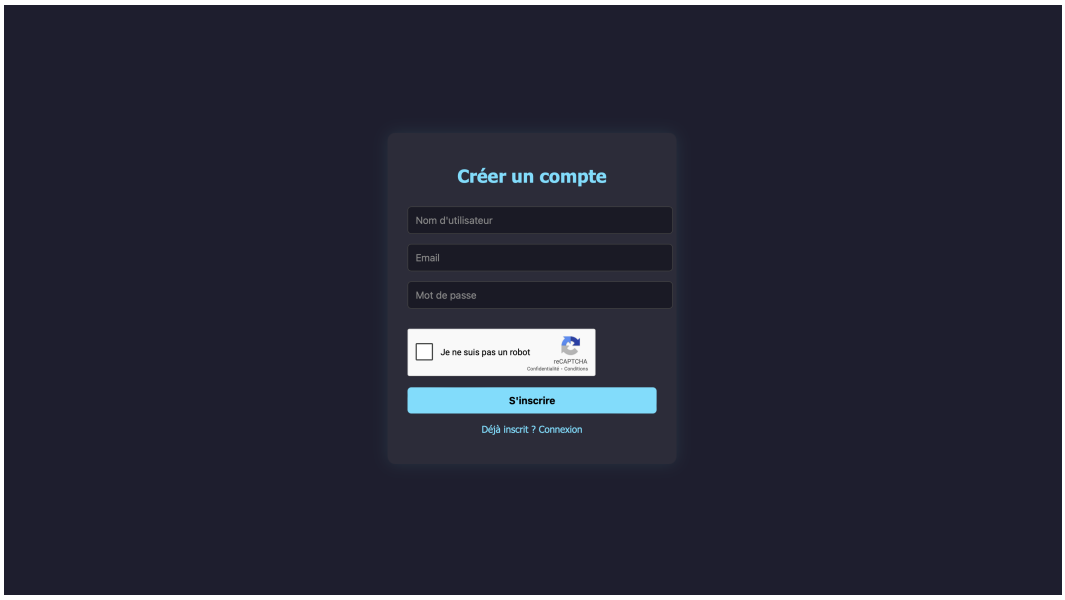


FIGURE 1 – Inscription

1.1.2 Page de connexion (/login.html)

Cette page permet aux utilisateurs enregistrés de se connecter. Le formulaire comprend :

— **Email**

— **Mot de passe**

— **reCAPTCHA v2**

Une session sécurisée est créée à la suite d'une authentification réussie à l'aide du module `express-session`. Si les identifiants sont valides et que le reCAPTCHA est vérifié, l'utilisateur est redirigé automatiquement vers la page de contact sécurisée.

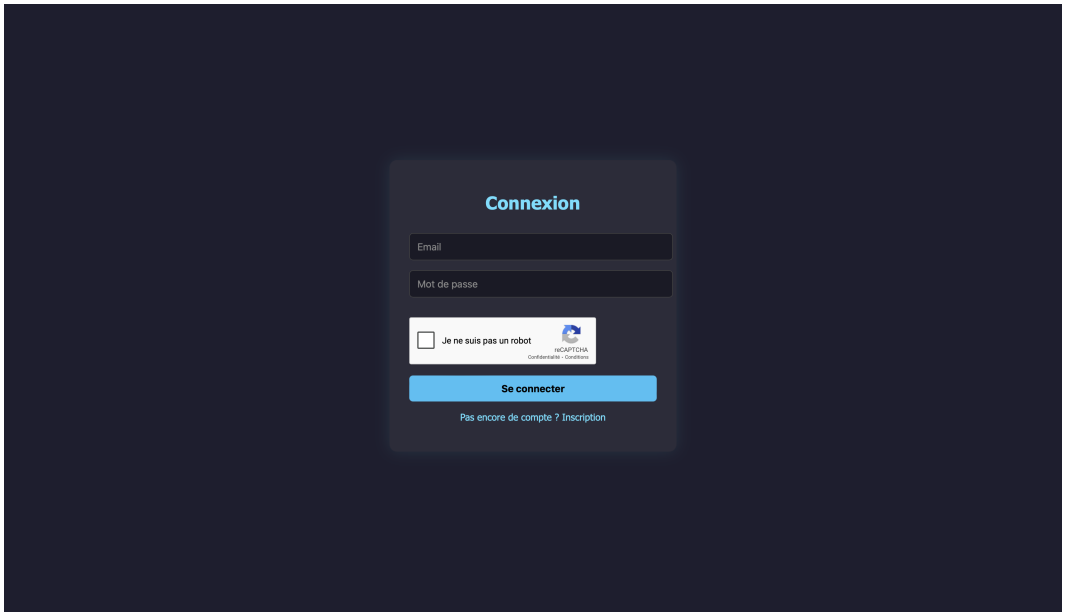


FIGURE 2 – Connexion

1.1.3 Formulaire de contact (/index.html)

Accessible uniquement après authentification, ce formulaire sécurisé permet à l'utilisateur connecté d'envoyer un message.

Il contient les champs suivants :

— **Nom**

— **Adresse e-mail**

— **Message**

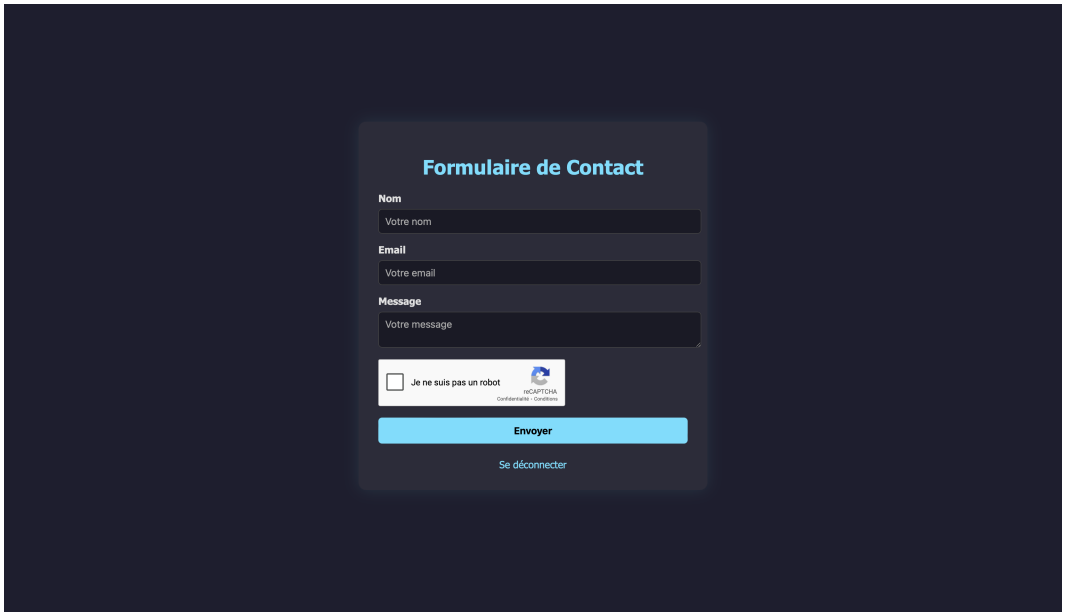


FIGURE 3 – Formulaire

— **reCAPTCHA v2**

Les données saisies par l'utilisateur sont ensuite :

— **Filtrées** à l'aide de `sanitize-html` pour bloquer les attaques XSS

— **Stockées localement** dans le fichier `messages.json`

— **Horodatées automatiquement** pour assurer une traçabilité

1.2 Parcours utilisateur

Voici le cheminement classique d'un utilisateur dans l'application :

1. L'utilisateur arrive sur la page d'inscription s'il n'a pas encore de compte.
2. Il s'enregistre avec ses informations ainsi qu'un reCAPTCHA.
3. Il se connecte via la page de connexion, avec vérification des identifiants et du reCAPTCHA.
4. Une fois connecté, il accède au formulaire de contact sécurisé.
5. Il peut envoyer un message, puis se déconnecter.

Chaque étape est protégée contre les usages malveillants, assurant un parcours fluide mais sécurisé.

2 Architecture technique du projet

2.1 Schéma de l'architecture

Le projet repose sur une architecture simple mais structurée :

— Le **frontend** est composé de fichiers statiques HTML/CSS/JS dans le dossier `public/`

— Le **backend** est un serveur Node.js utilisant Express, dans `src/`

— Les **certificats SSL** sont générés localement et stockés dans `config/ssl/`

— Les **logs** des requêtes HTTP sont enregistrés dans `logs/access.log`

— Les données utilisateurs et messages sont conservées dans des fichiers JSON

2.2 Technologies utilisées

Technologie / Outil	**Rôle**
Node.js / Express.js	Création du serveur backend local
HTTPS (module `https`)	Chiffrement de la communication client-serveur
Helmet	Sécurisation des en-têtes HTTP
bcrypt	Hachage sécurisé des mots de passe
express-session	Gestion des sessions avec cookies sécurisés
sanitize-html	Nettoyage des champs utilisateur contre le XSS
Google reCAPTCHA v2	Protection contre les bots automatisés

2.3 Organisation des fichiers

```
formulaire_securise/
 public/           → Pages HTML, CSS, JS
 src/              → Code serveur, JSON utilisateurs/messages
 config/ssl/       → Certificats SSL
 logs/             → Fichier access.log
 .env              → Variables d'environnement (non versionné)
 package.json      → Dépendances Node.js
```

Cette organisation respecte les standards d'un projet Node.js sécurisé, avec une séparation claire entre code, données, configurations et interfaces.

3 Principes de sécurité appliqués

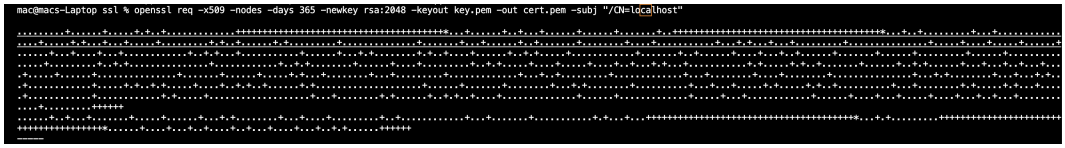
Cette partie détaille l'ensemble des mesures de sécurité mises en œuvre selon les principes du **Security by Design**. Chaque mesure répond à une vulnérabilité web classique identifiée par les guides OWASP.

3.1 HTTPS : Sécurisation des échanges

Pourquoi ? Le protocole HTTP ne chiffre pas les données. Les informations sensibles peuvent donc être interceptées sur un réseau local. **HTTPS** chiffre les communications grâce à SSL/TLS.

Mise en œuvre :

— Génération d'un certificat SSL local avec `OpenSSL`



— Utilisation du module `https` de Node.js

— Mise en place d'un serveur HTTPS à la place d'un serveur HTTP

3.2 Hachage des mots de passe

Pourquoi ? Un mot de passe en clair est facilement exploitable en cas de fuite. Le hachage permet de le rendre illisible et non réversible.

Mise en œuvre :

— Utilisation de `bcrypt`

— Application de 10 tours de salage (*salt rounds*)

— Stockage dans `users.json` uniquement après hachage

```
    const hashedPassword = await bcrypt.hash(password, 10);
    users.push({ username, email, password: hashedPassword });
    saveUsers(users);

    res.send('Inscription réussie ! <a href="/login.html">Connectez-vous</a>');
});
```

3.3 Sessions et cookies sécurisés

Pourquoi ? Pour se protéger contre les attaques de type *session hijacking* ou *fixation*.

Mise en œuvre :

— Gestion des sessions via `express-session`

— Cookies configurés avec :

  — `HttpOnly` : inaccessible depuis JavaScript

  — `Secure` : transmis uniquement en HTTPS

  — `SameSite=Strict` : pas de requête inter-site

```
app.use(session({
  secret: process.env.SESSION_SECRET || 'default-secret',
  resave: false,
  saveUninitialized: false,
  cookie: {
    httpOnly: true,
    secure: true, // fonctionne uniquement avec HTTPS
    sameSite: 'strict'
  }
}));
```

3.4 Protection anti-bot : Google reCAPTCHA v2

Pourquoi ? Un formulaire sans protection peut être soumis automatiquement par des scripts malveillants.

Mise en œuvre :

— Intégration du script reCAPTCHA v2 dans les pages HTML

```
<div class="g-recaptcha" data-sitekey="6LeGozYrAAAAAHN59bYagtwltrQm-bnWwUSJq0JK"></div>
```

— Vérification côté serveur via une requête à l'API Google

— Application sur l'inscription, la connexion et l'envoi de messages

.env

```
RECAPTCHA_SECRET=6LeGozYrAAAAAATIHMnjm4ADRsTzOKp5-adUAnZE
SESSION_SECRET=FirTtt8LKoPfMHi6OXJkrsiZY8SgDjkt
```

```
app.post('/register', async (req, res) => {
  const { username, email, password, 'g-recaptcha-response': captcha } = req.body;

  if (!captcha) return res.status(400).send('Captcha manquant');

  const verifyURL = `https://www.google.com/recaptcha/api/siteverify?secret=${process.env.RECAPTCHA_SECRET}&response=${captcha}`;
  try {
    const captchaRes = await fetch(verifyURL, { method: 'POST' });
    const data = await captchaRes.json();
    if (!data.success) return res.status(403).send('Captcha invalide');
  } catch (err) {
    return res.status(500).send('Erreur de vérification Captcha');
  }
```

3.5 Filtrage anti-XSS

Pourquoi ? Le XSS permet d'injecter du JavaScript malveillant dans la page, menaçant les sessions ou redirigeant l'utilisateur.

Mise en œuvre :

— Utilisation de `sanitize-html`

— Suppression des balises HTML et scripts dans les champs `name`, `email`, `message`

3.6 Journalisation des activités

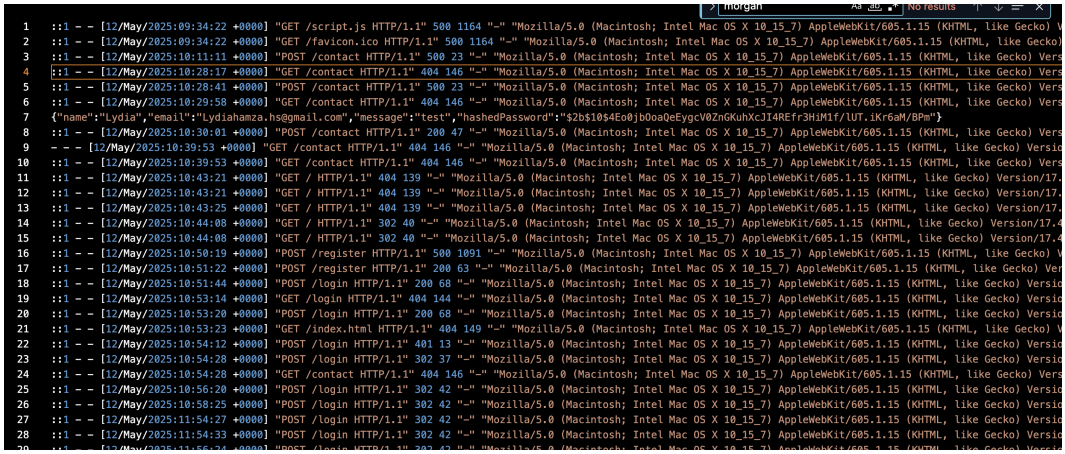
Pourquoi ? Pour suivre les requêtes, les tentatives de connexion et identifier les comportements suspects.

Mise en œuvre :

— Utilisation du module `morgan`

— Stockage dans le fichier `logs/access.log`

— Format de log standard avec date, méthode, route, statut HTTP



4 Tests de sécurité réalisés

Cette section présente les différents tests effectués afin de valider la robustesse du formulaire face aux principales vulnérabilités web.

4.1 Scénarios de test

1. Injection XSS dans le champ `message`

But : Tester si un script JavaScript est exécuté lors de la soumission du formulaire.
Test : Insertion de `<script>alert(1)</script>`
ETAPE 1 : avant utilisation sanitizeHtml


```
{
  "name": "lydia",
  "email": "lydiahamza.hs@gmail.com",
  "message": "<script>alert('XSS')</script>\n",
  "date": "2025-05-12T12:46:42.889Z"
},
```


ETAPE 2 : Après utilisation de sanitizeHtml


```
{
  "name": "test",
  "email": "test@gmail.com",
  "message": "\n",
  "date": "2025-05-12T12:50:35.314Z"
}
```


FIGURE 4 – Enter Caption

Résultat : Le contenu est filtré et non exécuté. **XSS bloqué.**

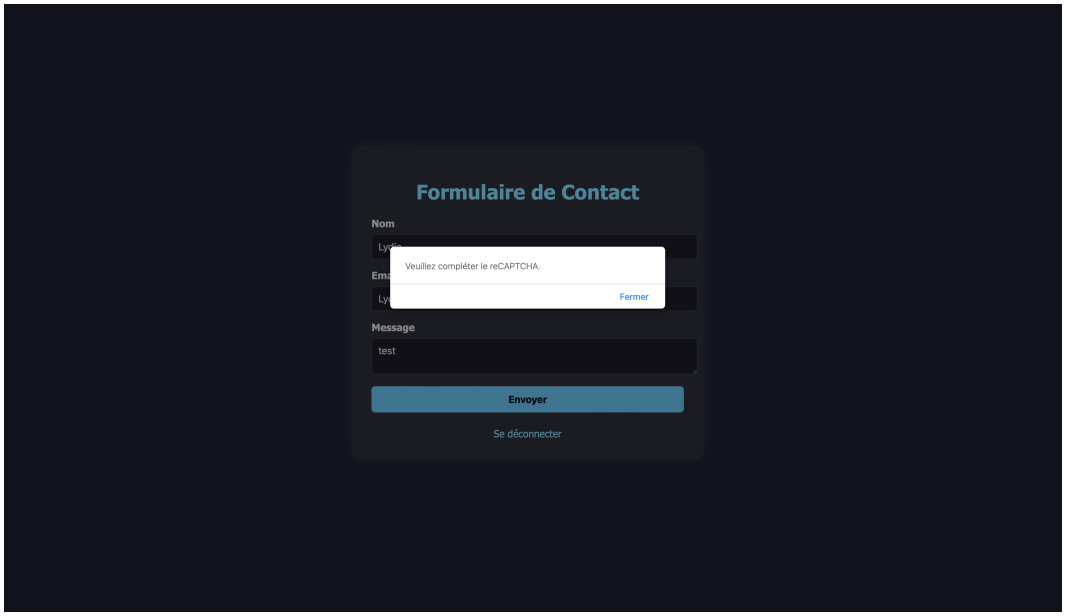
2. Soumission sans reCAPTCHA

But : Tester le comportement du serveur si un bot tente de contourner le reCAPTCHA.
Test : Soumission sans le champ `g-recaptcha-response`

```
150       <div class="input-group">
151         <label for="message">Message</label>
152         <textarea id="message" name="message" required placeholder="Votre message"></textarea>
153       </div>
154
155       <!-- <div class="g-recaptcha" data-sitekey="6LeGozYrAAAAAHN59bYagtwltrQm-bnWwUSJq0JK"></div> -->
156
157       <button type="submit" class="submit-btn">Envoyer</button>
158     </form>
159
160     <div class="logout-link">
161       <a href="/logout">Se déconnecter</a>
162     </div>
163   </div>
164
165   <!-- ✅ MODAL -->
166   <div class="modal" id="confirmationModal">
```

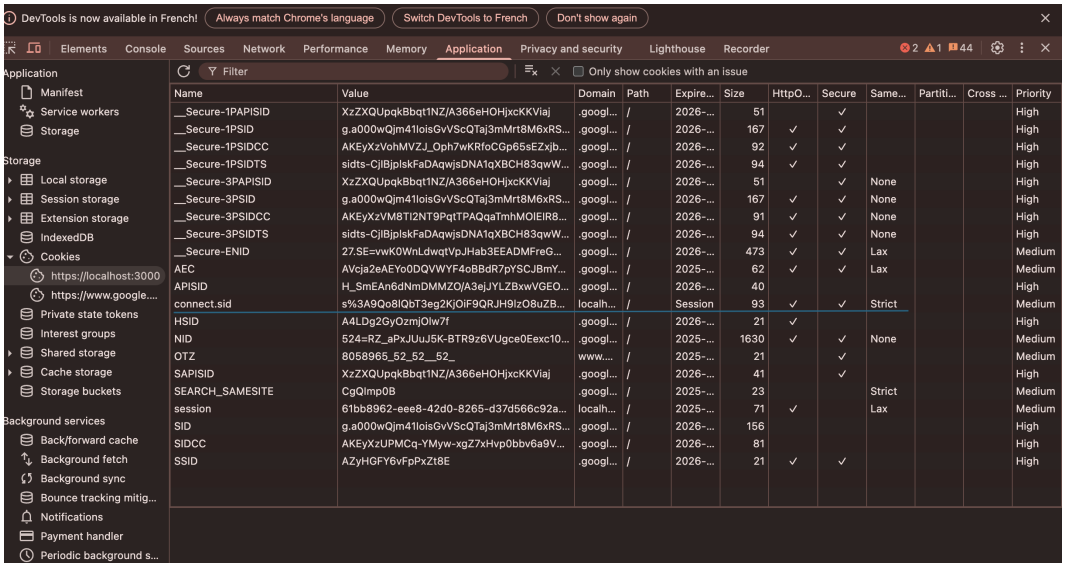
FIGURE 5 – Enter Caption

Résultat : Le serveur retourne une fenêtre pop up pour nous demander de faire le captcha. **Protection active.**



3. Analyse des cookies avec Chrome DevTools

But : Vérifier la présence des attributs de sécurité sur les cookies de session.
Test : Inspection des cookies via l'onglet *"Application" > "Cookies"* dans Chrome DevTools.
Résultat : Le cookie de session est correctement marqué avec les attributs `HttpOnly` et `Secure`. Le flag `SameSite=Strict` est également actif. **Protection conforme.**



4. Test de configuration HTTPS avec SSL Labs

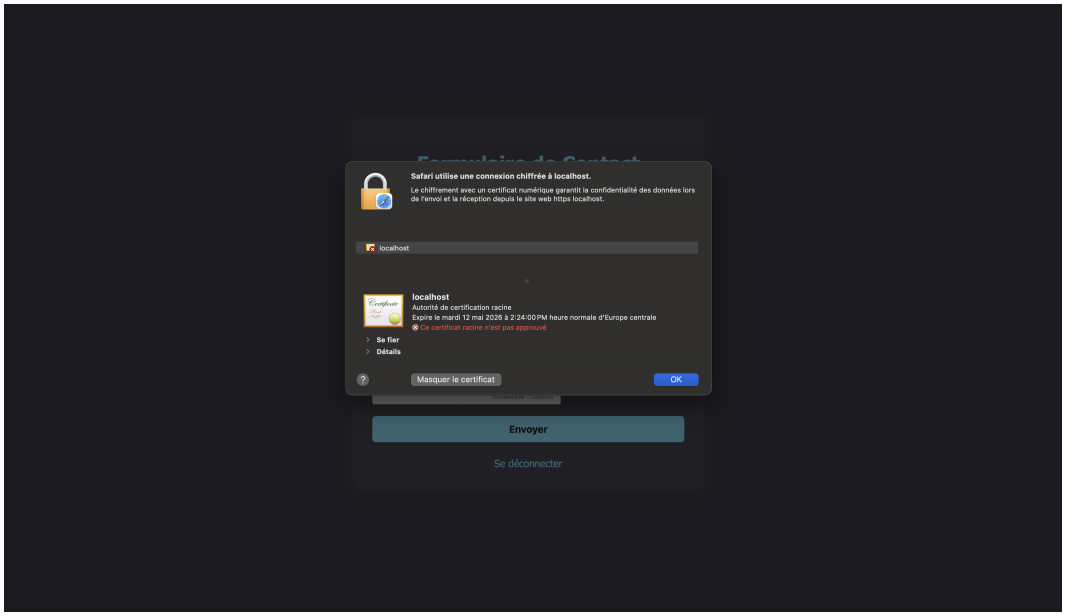
But : Contrôler la validité et la qualité de la configuration SSL/TLS de l'application via un outil externe.

Outil : Analyseur SSL Labs de Qualys, permettant d'effectuer un audit en ligne de la configuration HTTPS.

Test : Tentative d'analyse du serveur local via `https://localhost:3000`.

Résultat : L'analyse échoue car SSL Labs ne peut scanner que des domaines publics accessibles sur Internet. Le certificat utilisé a été généré localement avec `OpenSSL`, uniquement pour un usage interne.

Remarque à l'évaluateur : Ce comportement est attendu. Les certificats auto-signés créés pour le développement local (`localhost`) ne sont pas reconnus par les autorités de certification publiques. C'est pourquoi les navigateurs affichent un avertissement, et les outils comme SSL Labs ne peuvent pas vérifier leur configuration.



4.2 Résumé des résultats

Scénario de test	Résultat
Injection XSS	Bloqué
Soumission sans reCAPTCHA	Rejetée
Analyse des cookies avec DevTools	tout est ok
Génération du certificat	Ok

Ces tests confirment l'efficacité des protections mises en place et démontrent que l'application respecte les bonnes pratiques fondamentales en matière de sécurité web.

5 Conclusion

Ce projet a permis de mettre en œuvre de manière concrète les principes du développement sécurisé dès la conception, en suivant la démarche du **Security by Design**. L'application développée – un formulaire de contact sécurisé – intègre des mécanismes essentiels de protection des données utilisateurs, de lutte contre les attaques et de renforcement de la confiance.

Chaque fonctionnalité a été pensée dans une optique de sécurité :

- **HTTPS** pour assurer la confidentialité des échanges
- **Hachage des mots de passe** avec bcrypt pour éviter tout stockage sensible
- **Protection contre les scripts malveillants (XSS)** grâce à sanitize-html
- **Sessions sécurisées et cookies protégés** avec les bons flags
- **Contrôle humain** via Google reCAPTCHA pour stopper les bots
- **Tests de sécurité automatisés et manuels** validant les protections mises en œuvre

Au-delà de l'aspect technique, ce projet a également permis d'adopter une **réflexion structurée autour de la sécurité**, en anticipant les vulnérabilités potentielles et en implémentant des solutions adaptées dès la conception de l'application.

Perspectives : Ce projet pourrait être étendu avec une base de données réelle, un système de journalisation plus avancé, un système de rôles utilisateurs, ou encore des tests automatisés d'intrusion.

En conclusion, ce travail m'a permis d'acquérir des compétences solides sur la sécurisation des applications web, tout en découvrant les outils, techniques et réflexes indispensables à tout développeur soucieux de la qualité et de la robustesse de ses applications.