RAPPORT DE PROJET

Développement d'une Architecture Web Sécurisée (E2E Encryption)

Projet : SecureCrypt **Auteur :** Taguianas **Date :** 2025 **Technologie :** SERN Stack (SQL/NoSQL omitted, Express, React, Node)

1. Introduction et Contexte

Dans un contexte numérique où la confidentialité des données est devenue critique (RGPD, cybersécurité), ce projet vise à concevoir et développer une application web de **chiffrement de bout en bout**.

L'objectif principal n'est pas seulement de manipuler du texte, mais de démontrer la mise en œuvre d'une architecture sécurisée respectant les standards cryptographiques modernes (AES-GCM, RSA-OAEP) et les bonnes pratiques de développement Web (Architecture découplée, CI/CD).

2. Architecture Technique

Le projet repose sur une architecture découplée (**Decoupled Architecture**) séparant distinctement la logique métier de l'interface utilisateur.

2.1 Stack Technologique

- **Frontend (Client) :** React.js 18.
    - *Choix technique :* Utilisation d'une SPA (Single Page Application) pour une fluidité optimale et une gestion d'état réactive.
    - *Styling :* Tailwind CSS pour une conception "Mobile First" et un mode sombre natif.
- **Backend (API) :** Node.js & Express.
    - *Choix technique :* Node.js permet l'utilisation du module natif crypto (basé sur OpenSSL), offrant des performances élevées sans dépendances lourdes.
- **Communication :** API RESTful via HTTPS.

2.2 Flux de Données

Le client envoie les données brutes et les clés via des requêtes POST sécurisées. Le serveur effectue les opérations mathématiques en mémoire (sans stockage persistant des données sensibles) et renvoie le résultat.

3. Implémentation Cryptographique

La sécurité du projet repose sur deux piliers majeurs, implémentés sans utiliser de bibliothèques tierces "boîte noire", mais via les primitives natives de Node.js.

3.1 Chiffrement Symétrique : AES-256-GCM

Nous avons choisi l'algorithme **AES (Advanced Encryption Standard)** avec une clé de 256 bits.

- **Évolution technique :** Initialement prévu en mode CBC (Cipher Block Chaining), le projet a migré vers le mode **GCM (Galois/Counter Mode)**.
- **Justification :** Le mode CBC assure la confidentialité mais pas l'intégrité. Le mode GCM est un chiffrement authentifié (AEAD). Il génère un **Tag d'authentification** qui garantit que le message chiffré n'a pas été altéré (attaque "bit-flipping") durant le transport.
- **Sécurité :** Un IV (Vecteur d'Initialisation) de 12 octets unique est généré pour chaque opération.

3.2 Chiffrement Asymétrique : RSA-2048-OAEP

Pour l'échange de clés ou de messages courts, nous utilisons RSA.

- **Padding :** Utilisation stricte du padding **OAEP** (Optimal Asymmetric Encryption Padding) avec hachage SHA-256.
- **Justification :** Protège contre les attaques par "oracle de remplissage" (Padding Oracle Attacks) qui compromettent les implémentations RSA plus anciennes (PKCS#1 v1.5).
- **Interopérabilité :** Les clés sont générées et exportées au format standard **PEM (PKCS#8)**, rendant les clés compatibles avec OpenSSL ou d'autres systèmes.

4. Sécurité Applicative et Backend

Au-delà de la cryptographie, l'application est durcie contre les attaques web courantes.

4.1 Protection des En-têtes HTTP (Helmet)

Intégration du middleware Helmet.js pour configurer automatiquement les en-têtes de sécurité :

- Protection contre le reniflage de type MIME (X-Content-Type-Options).
- Protection XSS basique (X-XSS-Protection).
- Interdiction d'iframe (X-Frame-Options) pour éviter le Clickjacking.

4.2 Gestion des Origines (CORS Strict)

L'API est configurée pour rejeter systématiquement toute requête ne provenant pas du domaine du Frontend authentifié (Access-Control-Allow-Origin). Cela empêche l'utilisation de l'API par des sites tiers malveillants.

4.3 Gestion des Secrets

Aucune configuration sensible (ports, clés API futures) n'est codée en dur. L'application utilise des variables d'environnement (dotenv) injectées au moment du runtime.

5. Expérience Utilisateur (UX/UI)

L'interface a été conçue pour rendre la cryptographie accessible :

- **Feedback visuel :** Indicateurs de chargement et messages d'erreur clairs.
- **Persistance locale :** Utilisation du localStorage pour conserver un historique des 10 dernières opérations côté client (sans envoi au serveur).

- **Fonctionnalités utilitaires :** Copie presse-papier automatique et téléchargement des clés RSA sous forme de fichiers physiques (.pem).

6. Déploiement et DevOps (CI/CD)

Le projet est déployé en environnement de production sur le cloud (PaaS Render).

- **Architecture de Déploiement :** Deux services distincts (Frontend Statique + Backend Web Service).
- **Environnements Dynamiques :** Le code détecte automatiquement l'environnement (development vs production) pour ajuster les URLs de l'API, facilitant le développement local sans casser la prod.
- **Gestion de version :** Utilisation de Git avec une stratégie stricte de .gitignore pour éviter la fuite de secrets ou de dépendances.

7. Conclusion et Perspectives

Ce projet a permis de valider la maîtrise du cycle de développement complet ("Full Stack Lifecycle"), de la conception de l'algorithme jusqu'à la mise en production sécurisée.

Améliorations futures envisagées :

1. **Base de données :** Intégration de MongoDB pour permettre aux utilisateurs de stocker des messages chiffrés persistants.
2. **Authentification :** Ajout de JWT (JSON Web Tokens) pour sécuriser l'accès à l'historique.
3. **Signature Numérique :** Implémentation de la signature de messages avec RSA pour prouver l'identité de l'expéditeur.

Rapport généré le 24/11/2025 pour le projet SecureCrypt.