BEYRIES Corentin - DIAS Vincent - CHAMINADE Pierre

Compte Rendu de Sécurité

Mesures de sécurité mises en place dans le code

Voici un résumé des principales mesures de sécurité que nous avons mises en place dans le code pour protéger l'application contre les attaques courantes :

1. **Utilisation de `Flask-Limiter` pour la protection contre les attaques par déni de service (DoS)**:
    a. **Rate limiting**: Nous avons intégré la bibliothèque `Flask-Limiter` pour restreindre le nombre de soumissions par minute, afin de limiter les attaques par déni de service (DoS) basées sur l'envoi massif de requêtes (flooding).
    b. Exemple : `"5 per minute"` permet à un utilisateur de soumettre le formulaire un maximum de 5 fois par minute depuis une même adresse IP.

```
limiter = Limiter(key_func=get_remote_address)
limiter.init_app(app)
```

Protection contre les attaques CSRF (Cross-Site Request Forgery) :

- **Token CSRF**: Un token CSRF unique est généré pour chaque session utilisateur et est inclus dans chaque formulaire. Lors de la soumission du formulaire, le serveur vérifie que le token envoyé correspond à celui stocké dans la session.
- Exemple de génération et de vérification des tokens :

```python
def generate_csrf_token():  1 usage  Beyou
    if "_csrf_token" not in session:
        token = secrets.token_urlsafe(16)
        session['_csrf_token'] = token
    return session['_csrf_token']


def verify_csrf_token(token):  1 usage  Beyou
    return session.get('_csrf_token') == token
```

Cryptographie des messages sensibles :

- **Cryptographie des messages :** Les messages soumis par les utilisateurs sont chiffrés avec la bibliothèque `cryptography.fernet` avant d'être enregistrés dans la base de données, ce qui protège les données sensibles.

```python
KEY = Fernet.generate_key()
cipher_suite = Fernet(KEY)
```

Validation des entrées utilisateurs (Input Validation):

- **Validation de l'email :** Vérification de la validité du format de l'adresse email avec une expression régulière.
- **Validation des champs :** Vérification de la présence des champs `name`, `email`, et `message`, et des limites de longueur pour le `name` et `message`.

```python
def validate_input(name, email, message):  1 usage  Beyou
    if not name or not email or not message:
        return False, "All fields are required."
    if len(name) > 100 or len(message) > 500:
        return False, "Name or message is too long."
    email_regex = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$'
    if not re.match(email_regex, email):
        return False, "Invalid email address."
    return True, "Valid input."
```

Utilisation de `secret_key` sécurisé :

- Le `secret_key` est stocké dans une variable d'environnement (via `os.getenv`), ce qui permet de sécuriser les sessions et les tokens CSRF. En production, le serveur lèvera une erreur si la clé secrète n'est pas définie.

```python
app.secret_key = os.getenv('ENCRYPTION_KEY')
```

Attaques dont nous sommes protégés

Voici une liste d'attaques contre lesquelles le code met en place des mesures de protection :

1. **Attaque par déni de service (DoS) :**
    a. Grâce à `Flask-Limiter`, les requêtes excessives provenant d'une même adresse IP sont limitées. Cela empêche une seule source d'inonder le serveur de requêtes.
2. **Attaque par Cross-Site Request Forgery (CSRF)** :
    a. L'application génère un token CSRF unique pour chaque utilisateur et vérifie sa validité lors de la soumission des formulaires, ce qui protège contre les attaques où un utilisateur malveillant pourrait envoyer des requêtes de l'extérieur au nom d'un autre utilisateur.
3. **Attaque par injection SQL** :

    a. L'utilisation de requêtes SQL paramétrées (`c.execute("INSERT INTO contacts (name, email, message) VALUES (?, ?, ?)", ...)`) empêche les attaques par injection SQL, car les données de l'utilisateur sont échappées de manière sécurisée.
4. **Attaque par Cross-Site Scripting (XSS)** :
    a. Bien que la protection XSS ne soit pas directement implémentée dans ce code (par exemple en échappant les données dans le template HTML), il est recommandé de toujours échapper les données dans les templates pour éviter l'injection de scripts malveillants dans le HTML.

Résumé des attaques et des résultats des tests effectués

- **Test de l'attaque DoS** : En envoyant des requêtes excessives à l'API, nous avons constaté que l'application limite le nombre de requêtes à 5 par minute. Toute requête au-delà de cette limite retourne un code d'erreur 429 (trop de requêtes), ce qui protège contre les attaques par déni de service (DoS).

- **Test de l'attaque CSRF** : Nous avons essayé de soumettre un formulaire sans le token CSRF ou avec un token incorrect. Dans les deux cas, le serveur a rejeté la requête avec un message d'erreur indiquant que le token CSRF était invalide.

- **Test de l'attaque par injection SQL** : En tentant d'injecter des commandes SQL dans les champs `name` ou `message`, l'application a correctement échappé les valeurs, empêchant toute manipulation malveillante de la base de données.

- **Test de l'attaque XSS** : En insérant des scripts JavaScript malveillants dans les champs de formulaire, ces scripts ne se sont pas exécutés dans le navigateur grâce à la validation des entrées et au fait que Flask échappe automatiquement les variables dans les templates.