Points Bonus

Mise en place des tests

Protocoles d’authentification 4IW2

Étudiants : Philippe DELENTE, Camille GIRARD, Catalina DANILA

1. Tests de Charge et Rapport avec Vegeta

Vegeta est un outil de test de charge open-source qui permet d'évaluer les performances d'une application web ou d'un service. Il génère des requêtes HTTP à une fréquence définie et mesure des indicateurs clés tels que la latence, le débit et les taux de réussite.

Configuration et Lancement des Tests

Script d'Automatisation

Pour simplifier et automatiser les tests de charge avec **Vegeta**, nous avons conçu un script bash. Ce script guide l'utilisateur à travers plusieurs étapes, permettant une personnalisation facile des paramètres des tests. Voici les principales fonctionnalités et étapes du script :

1. **Création du Dossier de Résultats :** Si le dossier `tests-vegeta` n'existe pas, il est automatiquement créé pour stocker les résultats.
2. **Demande de l'URL à Tester :** Le script demande l'utilisateur à saisir l'URL cible des tests.
3. **Paramétrage des Tests :** La durée des tests et le taux de requêtes par seconde ont des valeurs par défaut préconfigurées. L'utilisateur peut personnaliser ces paramètres en répondant aux prompts du script.
4. **Exécution des Commandes Vegeta :** Le script lance Vegeta pour attaquer l'URL cible et collecter les données de performance.
5. **Génération de Rapports :** À la fin des tests, le script produit :
    - Un fichier texte contenant un résumé des résultats.
    - Un fichier binaire des données brutes.
    - Un fichier HTML interactif permettant de visualiser un graphique des latences.

Exemple de commande pour lancer le script

```
./make-vegeta-tests.sh
```

Commandes Vegeta

Les commandes clés utilisées dans le script sont :

- **vegeta attack :** pour envoyer des requêtes.
- **vegeta report :** pour générer un rapport texte.
- **vegeta plot :** pour produire un fichier HTML contenant un graphique.

Résultats des Tests du 07 janvier 2025

Lien testé

- **URL cible** : `http://localhost/login`

Résumé des Résultats

Les tests ont été réalisés avec un débit constant de **50 requêtes** par seconde sur une période de **30 secondes**. Voici les détails des performances enregistrées :

```
Requests      [total, rate, throughput]         1500, 50.03, 50.02
Duration      [total, attack, wait]             29.988s, 29.981s, 7.143ms
Latencies     [min, mean, 50, 90, 95, 99, max]  3.474ms, 7.289ms, 6.819ms, 8.415ms, 10.488ms, 19.92ms, 63.53ms
Bytes In      [total, mean]                     2919000, 1946.00
Bytes Out     [total, mean]                     0, 0.00
Success       [ratio]                           100.00%
Status Codes  [code:count]                      200:1500
Error Set:
```

1. **Requêtes :**
    - **Nombre total :** 1500 requêtes ont été envoyées.
    - **Débit constant :** 50.03 requêtes par seconde.
    - **Débit réel (throughput) :** 50.02 requêtes par seconde, ce qui indique que l'application a pu traiter presque toutes les requêtes sans retard notable.
2. **Durée du test :**
    - **Durée totale :** 29.988 secondes.
    - Cette durée inclut une durée d'attente moyenne (wait) de 7.143 ms pour les réponses.
3. **Latences (en millisecondes) :**
    - **Minimum :** 3.474 ms, représentant la meilleure performance en termes de réactivité.
    - **Moyenne :** 7.289 ms, qui est une performance globalement stable et rapide.
    - **Médiane (50%) :** 6.819 ms indiquant que la moitié des requêtes ont eu une latence inférieure à cette valeur.
    - **95e centile :** 10.488 ms, ce qui signifie que 95 % des requêtes ont été traitées en moins de 10.5 ms.
    - **Maximum :** 63.53 ms, un pic isolé qui peut indiquer un traitement légèrement plus long pour une requête spécifique.
4. **Volume de données :**

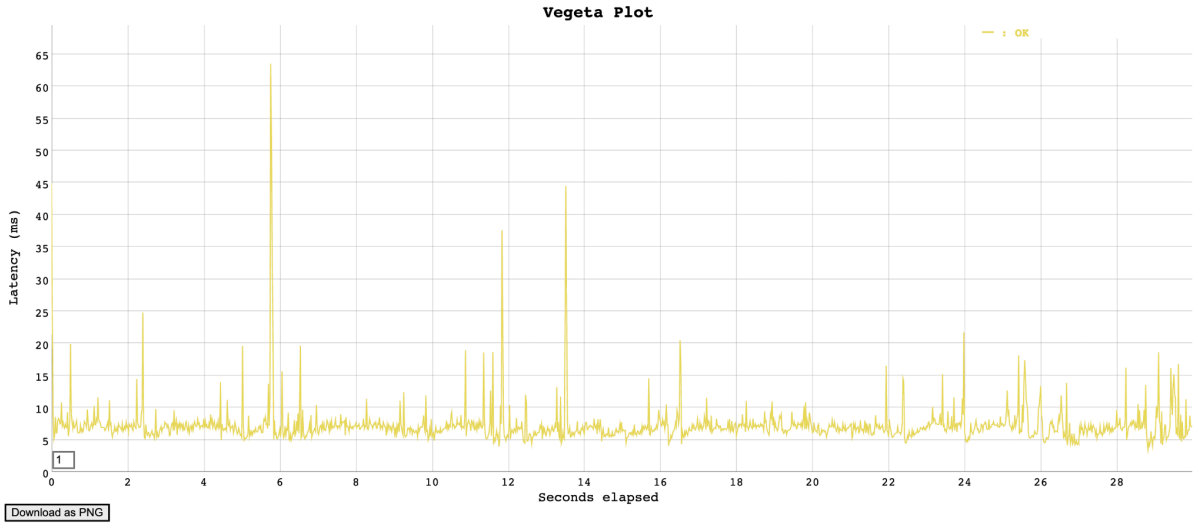
- **Bytes In :** 2 919 000 octets ont été reçus au total, soit une moyenne de 1946 octets par requête.
- **Bytes Out :** Aucune donnée n'a été envoyée, ce qui est typique pour des tests sur des endpoints de type "GET" sans charge utile (payload).

5. **Taux de Succès :**
   - **Ratio de succès :** 100 %.
   - **Réponses réussies :** Toutes les 1500 requêtes ont reçu un code HTTP 200, ce qui indique une stabilité complète de l'application pendant ce test.

Analyse Graphique

Un graphique a été généré pour visualiser l'évolution des latences pendant la durée du test. Ce graphique est disponible dans le fichier HTML interactif accessible dans le répertoire `tests-vegeta`.

- **Fichier graphique :** `plot-20250107.html`



Interprétation du graphique :

- La majorité des latences sont restées stables et proches de la moyenne (7.289 ms).
- Quelques pics isolés sont visibles (notamment autour de 60 ms), mais ils n'ont pas affecté le taux de succès global.
- Ces pics pourraient être dus à des facteurs tels que la gestion des ressources serveur ou des tâches simultanées plus exigeantes.

Conclusion et Perspectives

Ces tests montrent que l’application est capable de gérer un flux de **50 requêtes par seconde sur une période de 30 secondes**, avec des latences globalement faibles et aucune erreur.

2. Tests de sécurité et rapport avec SQLMap

SQLMap est un outil automatisé d'audit et d'exploitation des vulnérabilités liées aux injections SQL. Il permet de tester des paramètres d'URL ou des données POST pour identifier des failles exploitables. Cet audit utilise SQLMap pour évaluer la sécurité d'une application web, avec comme cible une URL de connexion.

Procédure Utilisée

1. Téléchargement de SQLMap

SQLMap est exécuté à l'aide d'une image Docker. Voici la commande utilisée pour télécharger l'image officielle :

```
docker pull googlesky/sqlmap
```

2. Exécution de SQLMap

SQLMap a été lancé pour tester une URL de connexion :

```
docker run --rm -it googlesky/sqlmap -u
http://host.docker.internal:80/login
```

- Le paramètre `-u` indique l'URL cible pour les tests

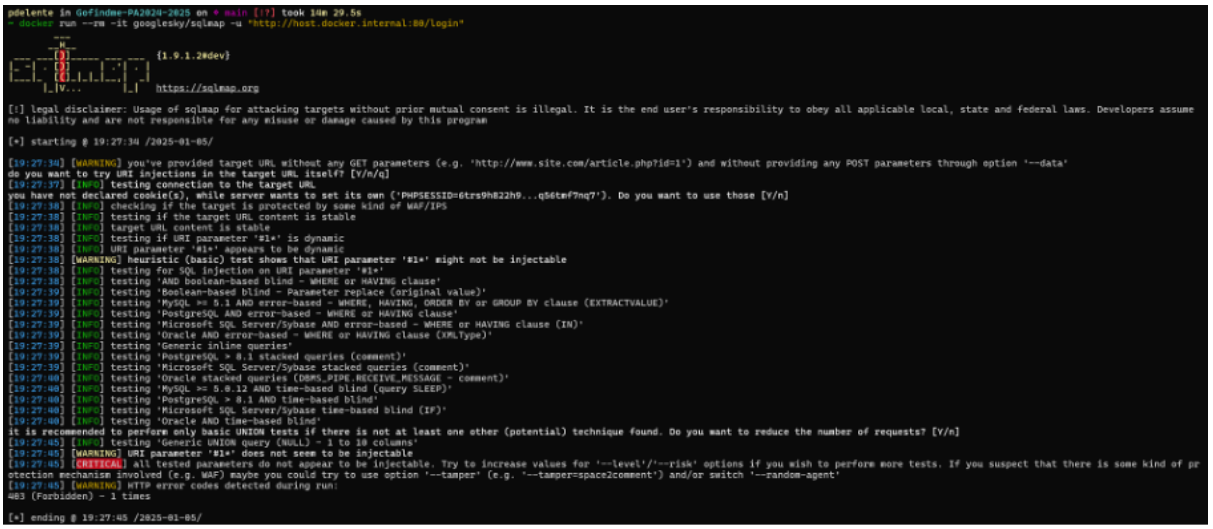
3. Tests effectués

SQLMap exécute plusieurs types de tests sur les paramètres dynamiques pour détecter des vulnérabilités, notamment :

- **Boolean-based Blind :** Cette méthode exploite des requêtes SQL qui retournent des réponses différentes (vrai ou faux) selon la validité d'une condition logique, permettant de déduire des informations sans retour explicite.
- **Error-based :** Cette technique force la base de données à produire des messages d'erreur contenant des informations sensibles sur la structure ou les données du système.
- **Time-based Blind :** Utilise des commandes SQL qui induisent des délais d'exécution (par exemple, `SLEEP()`) pour déterminer indirectement si une condition est vraie ou fausse.

- **UNION-based SQL Injection :** Combine des résultats de plusieurs requêtes SQL via l'opérateur UNION pour extraire des données sensibles, souvent visibles dans la réponse HTTP de l'application.

Ces tests visent à déterminer si les entrées utilisateur peuvent manipuler des requêtes SQL de manière non intentionnelle.



Résultats

Avertissements rencontrés :

- Le paramètre `#!%` a été identifié comme dynamique, mais ne semble pas exploitable directement.
- Une erreur `HTTP 403 (Forbidden)` a été détectée, ce qui pourrait indiquer la présence d'un pare-feu applicatif web (WAF) ou de restrictions sur le serveur cible.

Résumé des résultats :

SQLMap n'a pas détecté de vulnérabilités exploitables dans les paramètres testés. Le système semble protégé contre les attaques SQL Injection.

Conclusion

L'outil SQLMap n'a pas révélé de vulnérabilités exploitables dans le cadre de cet audit initial. Cela suggère que des mécanismes de protection sont en place, mais il est recommandé d'exécuter des tests plus approfondis et de renforcer les contrôles de sécurité existants.

3. Tests Web avec OWASP ZAP

Dans le cadre de la sécurisation de notre application web, nous avons intégré **OWASP ZAP** (Zed Attack Proxy) dans nos services Docker pour effectuer des tests de sécurité automatisés. Cet outil permet d'identifier diverses vulnérabilités potentielles au sein de notre application web.

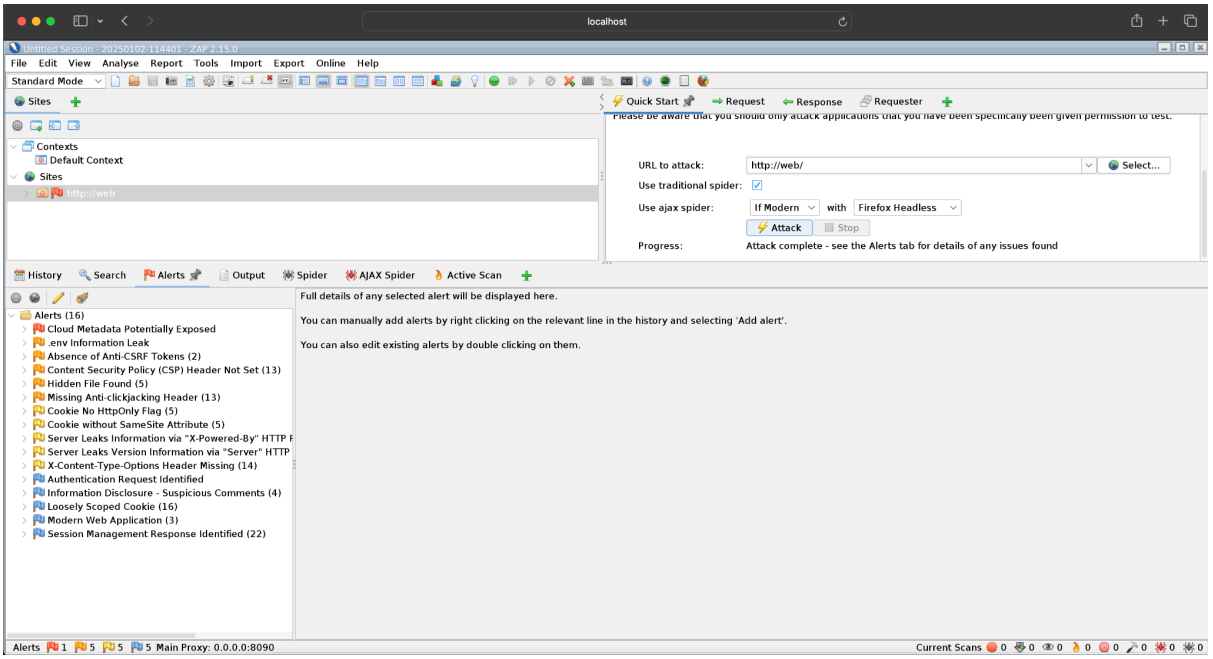
Mise en Place de OWASP ZAP

Accès à ZAP

- Ouvrir le navigateur et accéder à l'interface de ZAP via : http://localhost:8081/zap/

Scan Automatisé

1. Choisir **"Automated Scan"** dans l'interface.
2. Entrer l'URL à tester (par exemple : `http://web/login` au lieu de localhost en raison de la configuration Docker).
3. Cliquer sur **"Attack"** et patienter jusqu'à la fin du scan.



Résultats de l'Analyse

Les résultats de l'analyse sont affichés dans l'onglet **Alerts**, qui classe les vulnérabilités par niveau de gravité :

- **High (Rouge) :** Vulnérabilités critiques à corriger immédiatement.
- **Medium (Jaune/Orange) :** Vulnérabilités importantes à traiter rapidement.
- **Low (Bleu) :** Vulnérabilités mineures ou suggestions d'amélioration.

Vulnérabilités Détectées et Actions Correctives

Cloud Metadata Potentially Exposed

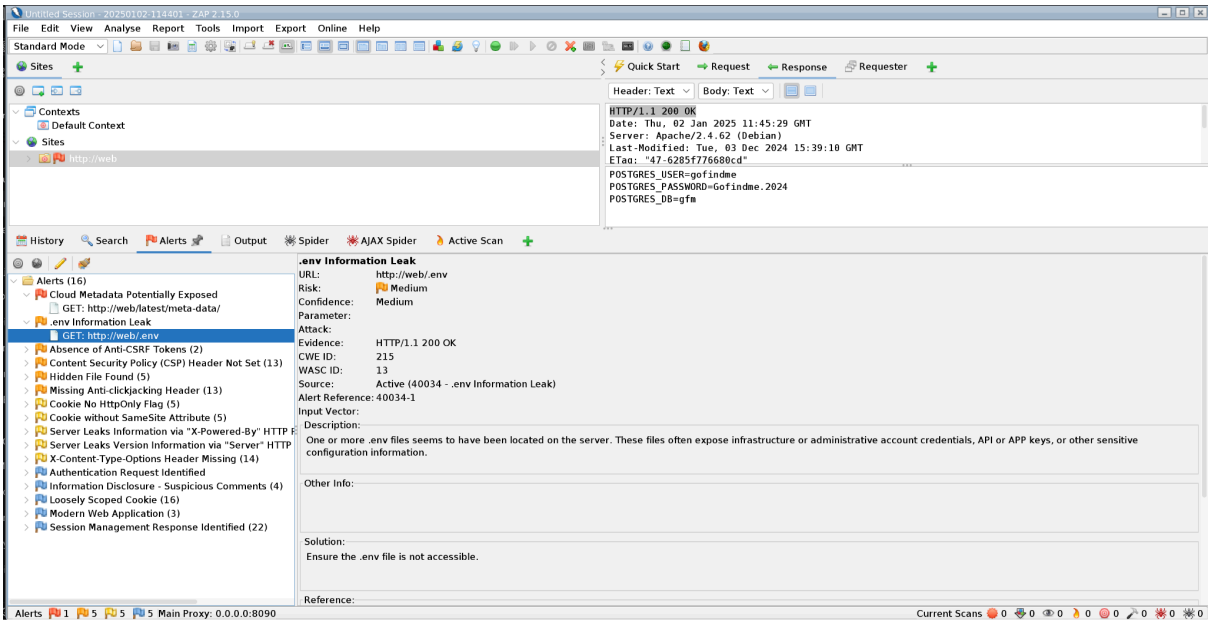
- **Problème :** Des métadonnées de l'environnement cloud sont accessibles publiquement.
- **Solution :**
  - Restreindre l'accès aux métadonnées via des règles de pare-feu.
  - Désactiver les endpoints exposés non nécessaires.

.env Information Leak

- **Problème :** Le fichier `.env` contenant des variables sensibles (mots de passe, clés API) est accessible publiquement.
- **Solution :**

1. Bloquer l'accès au fichier `.env` dans la configuration du serveur (Apache, Nginx).
2. Vérifier que le fichier `.env` n'est pas exposé dans le déploiement.

Exemple de fuite de fichier `.env` détectée :



Absence de CSRF Tokens

- **Problème :** Les formulaires ne sont pas protégés contre les attaques Cross-Site Request Forgery (CSRF).
- **Solution :**
  - Implémenter des tokens CSRF dans les formulaires pour empêcher les soumissions malveillantes.

Content Security Policy (CSP) Header Not Set

- **Problème :** L'en-tête CSP est absent, augmentant le risque d'exécution de scripts malveillants.
- **Solution :**
  - Définir une Content Security Policy stricte pour contrôler les sources de contenu.

Hidden File Found

- **Problème :** Des fichiers ou répertoires cachés ont été détectés.

- **Solution :**
    1. Analyser les fichiers identifiés et supprimer ceux non nécessaires.
    2. Restreindre l'accès via la configuration serveur.

Cookie without HttpOnly Flag

- **Problème :** Les cookies critiques n'ont pas le flag HttpOnly, les rendant vulnérables aux attaques JavaScript.
- **Solution :**
    - Configurer le serveur pour ajouter le flag HttpOnly aux cookies sensibles.

Recommandations

- **Priorité haute :** Corriger immédiatement les vulnérabilités critiques **(High)** et importantes **(Medium).**
- **Sécurisation des données sensibles :** Protéger les fichiers de configuration comme `.env.`
- **Mise en place de politiques de sécurité :** Ajouter des en-têtes HTTP de sécurité (CSP, X-Content-Type-Options).

Conclusion

L'intégration d'OWASP ZAP a permis de détecter plusieurs vulnérabilités critiques et importantes dans notre application web. Les corrections proposées doivent être appliquées rapidement pour renforcer la sécurité globale du système. Une nouvelle analyse devra être effectuée après correction pour valider la résolution des problèmes identifiés.