LOG8100

DEVSECOPS

Laboratoire 2:

Tests de pénétration et Outils pour Intégration et Déploiement continus

Équipe 4

Raouf Tafat-Bouzid 2148502

Hamza Belahcen 2152372

Enrique Arsenio Rodriguez Rodriguez 2141087

22 octobre 2024

Introduction

Dans le cadre de ce travail pratique, il sera question de tester la sécurité de l'application suivante : Damn Vulnerable NodeJS Application (DVNA) qui, comme indiqué par son nom, est une application contenant diverses vulnérabilités, dont celles du top 10 de OWASP. Nous avons eu recours à des outils de tests de pénétrations afin d'analyser la sécurité de l'application dont OWASP ZAP et orchestron. Dans la deuxième partie de ce laboratoire, nous avons implémenté une interface CI/CD afin d'automatiser les tests de sécurité et la gestion de dépendances et de déployer l'application et la documentation.

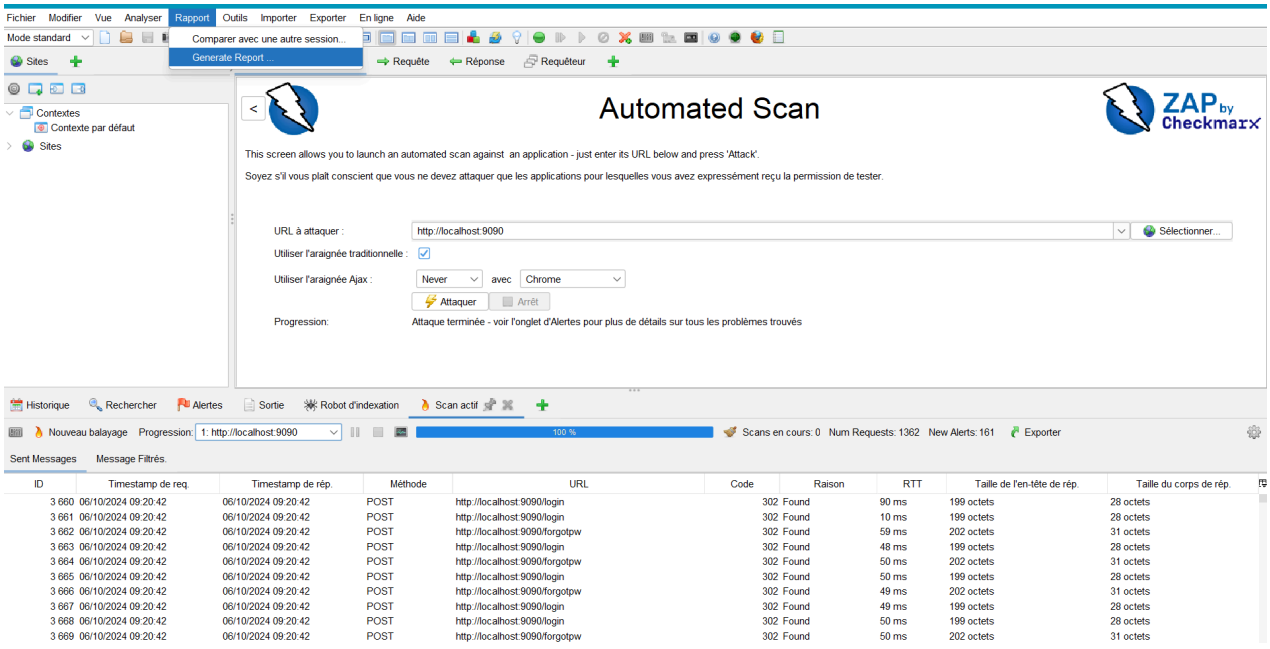
Ce rapport permettra de présenter les différentes étapes que nous avons effectuées afin de mener à bien le travail pratique ainsi que des explications quant à certains éléments.

Analyse de sécurité

Résultats des tests de pénétration:

Via application et outils OWASP ZAP:

Afin de faire usage de l'outil OWASP ZAP, la configuration de ce dernier était nécessaire. L'application devait être lancée sur un URL précis afin d'y faire référence dans l'outil et ainsi effectuer l'attaque en naviguant sur l'application.



ID	Timestamp de req.	Timestamp de rép.	Méthode	URL	Code	Raison	RTT	Taille de l'en-tête de rép.	Taille du corps de rép.
3 660	06/10/2024 09:20:42	06/10/2024 09:20:42	POST	http://localhost:9090/login	302	Found	90 ms	199 octets	28 octets
3 661	06/10/2024 09:20:42	06/10/2024 09:20:42	POST	http://localhost:9090/login	302	Found	10 ms	199 octets	28 octets
3 662	06/10/2024 09:20:42	06/10/2024 09:20:42	POST	http://localhost:9090/forgotpw	302	Found	59 ms	202 octets	31 octets
3 663	06/10/2024 09:20:42	06/10/2024 09:20:42	POST	http://localhost:9090/login	302	Found	48 ms	199 octets	28 octets
3 664	06/10/2024 09:20:42	06/10/2024 09:20:42	POST	http://localhost:9090/forgotpw	302	Found	50 ms	202 octets	31 octets
3 665	06/10/2024 09:20:42	06/10/2024 09:20:42	POST	http://localhost:9090/login	302	Found	50 ms	199 octets	28 octets
3 666	06/10/2024 09:20:42	06/10/2024 09:20:42	POST	http://localhost:9090/forgotpw	302	Found	49 ms	202 octets	31 octets
3 667	06/10/2024 09:20:42	06/10/2024 09:20:42	POST	http://localhost:9090/login	302	Found	49 ms	199 octets	28 octets
3 668	06/10/2024 09:20:42	06/10/2024 09:20:42	POST	http://localhost:9090/login	302	Found	50 ms	199 octets	28 octets
3 669	06/10/2024 09:20:42	06/10/2024 09:20:42	POST	http://localhost:9090/forgotpw	302	Found	50 ms	202 octets	31 octets

Une fois l'analyse finie, l'application permet de générer un rapport présentant différentes informations. On retrouve notamment les différents types de vulnérabilités accompagnées de leur niveau de risque et de leur compte.

Alert counts by risk and confidence

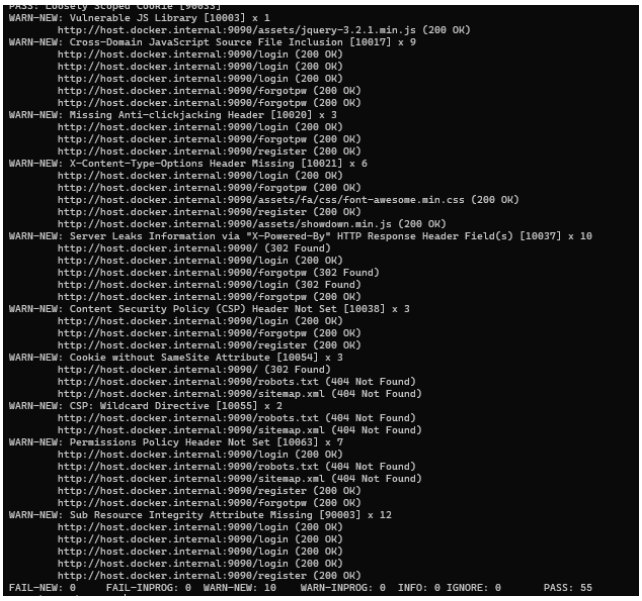
This table shows the number of alerts for each level of risk and confidence included in the report.

(The percentages in brackets represent the count as a percentage of the total number of alerts included in the report, rounded to one decimal place.)

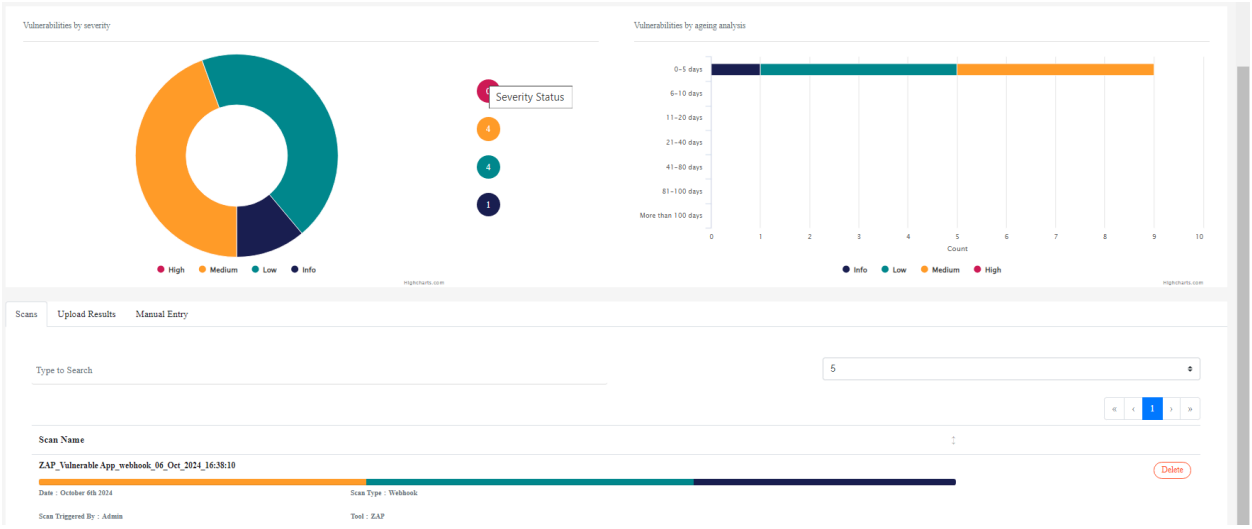
		Confidence				
		User Confirmed	Haut	Moyen	Faible	Total
Risk	Haut	0 (0,0 %)	0 (0,0 %)	0 (0,0 %)	0 (0,0 %)	0 (0,0 %)
	Moyen	0 (0,0 %)	2 (15,4 %)	2 (15,4 %)	0 (0,0 %)	4 (30,8 %)
	Faible	0 (0,0 %)	0 (0,0 %)	4 (30,8 %)	0 (0,0 %)	4 (30,8 %)
	Pour information	0 (0,0 %)	1 (7,7 %)	2 (15,4 %)	2 (15,4 %)	5 (38,5 %)
	Total	0 (0,0 %)	3 (23,1 %)	8 (61,5 %)	2 (15,4 %)	13 (100%)

Alert type	Risk	Count
CSP: Wildcard Directive	Moyen	9 (69,2 %)
Content Security Policy (CSP) Header Not Set	Moyen	13 (100,0 %)
Missing Anti-clickjacking Header	Moyen	13 (100,0 %)
Vulnerable JS Library	Moyen	1 (7,7 %)
Cookie without SameSite Attribute	Faible	4 (30,8 %)
Cross-Domain JavaScript Source File Inclusion	Faible	39 (300,0 %)
Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)	Faible	33 (253,8 %)
X-Content-Type-Options Header Missing	Faible	16 (123,1 %)
Authentication Request Identified	Pour information	1 (7,7 %)
Cookie faiblement couplé	Pour information	7 (53,8 %)
Information Disclosure - Suspicious Comments	Pour information	3 (23,1 %)
Session Management Response Identified	Pour information	11 (84,6 %)
User Agent Fuzzer	Pour information	161 (1 238,5 %)
Total		13

Nous avons aussi eu recours à l'image docker zaproxy/zap-stable afin de trouver les vulnérabilités présentes. (commande : /zap/zap-baseline.py -t -g gen.conf -x testreport.xml)



À partir des rapports générés dans les étapes précédentes, nous étions en mesure d'utiliser un autre outil, soit orchestron, permettant un suivi des vulnérabilités à partir d'une interface visuelle.



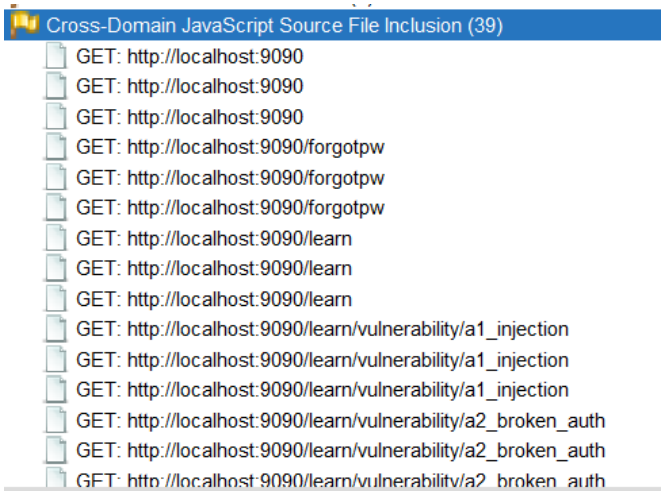
Présentation de quelques vulnérabilités:

Vulnérabilité 1 : Cross-Domain JavaScript Source File Inclusion

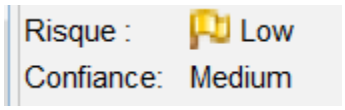
i. Nature des vulnérabilités (ex : injection SQL, XSS, etc.)

XSS / man in the middle / DoS / Détournement d'api / CSP

ii. Localisation dans le code (fichiers ou modules affectés)



iii. Gravité des vulnérabilités (ex : critique, élevée, moyenne, faible)



iv. Risques associés à chaque vulnérabilité (impact potentiel sur la sécurité).

Cette vulnérabilité permettrait à un attaquant d'exécuter du code sur le navigateur de la victime, menant à l'exposition de divers risques de sécurité.

Stratégies de mitigation pour vulnérabilité.

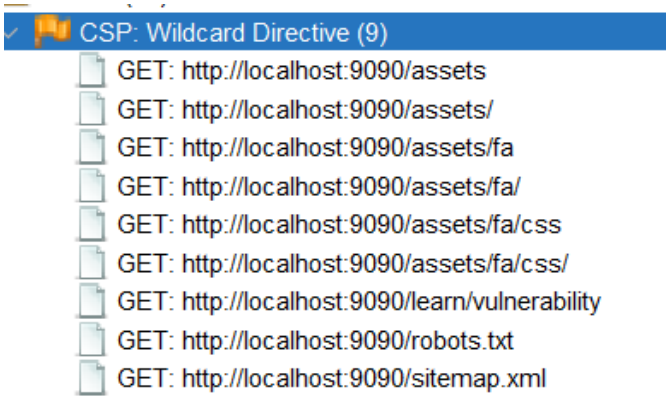
Afin de faire face à cette vulnérabilité il faudrait s'assurer que les fichiers sources proviennent de ressources fiables et sécuritaires. De plus, il ne faudrait pas permettre le contrôle de fichiers par les utilisateurs.

Vulnérabilité 2 : CSP: Wildcard Directive

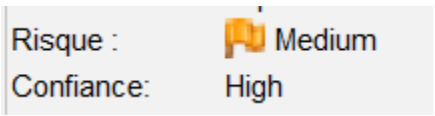
i. Nature des vulnérabilités (ex : injection SQL, XSS, etc.)

XSS / Remote Code Execution / Exfiltration de données / Violation de politique de confiance

ii. Localisation dans le code (fichiers ou modules affectés)



iii. Gravité des vulnérabilités (ex : critique, élevée, moyenne, faible)



iv. Risques associés à chaque vulnérabilité (impact potentiel sur la sécurité).

Une règle CSP permet de charger sur une application n'importe quelle source de contenu. Cela rend le site vulnérable à diverses attaques.

Stratégies de mitigation pour vulnérabilité.

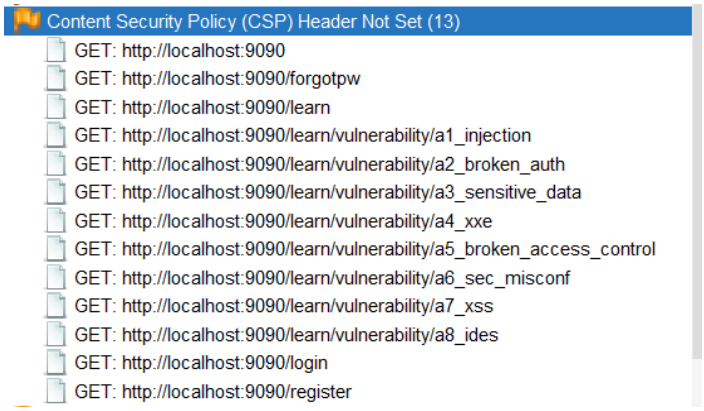
Afin de faire face à cette vulnérabilité il faudrait s'assurer que le serveur web, le serveur d'application et l'équilibreur de charges soient configurés adéquatement afin de mettre en place des politiques de sécurité.

Vulnérabilité 3 : Content Security Policy (CSP) Header Not Set

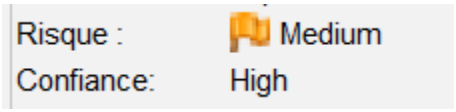
i. Nature des vulnérabilités (ex : injection SQL, XSS, etc.)

Xss / Clickjacking / Data injection / CSRF

ii. Localisation dans le code (fichiers ou modules affectés)



iii. Gravité des vulnérabilités (ex : critique, élevée, moyenne, faible)



iv. Risques associés à chaque vulnérabilité (impact potentiel sur la sécurité).

Ce genre d'attaque rend l'application vulnérable face au vol de données, à la dégradation de site ou à la distribution de logiciels malveillants.

Stratégies de mitigation pour vulnérabilité.

Afin de faire face à cette vulnérabilité il faudrait s'assurer

Tout comme la vulnérabilité précédente, il faudrait s'assurer que le serveur web, le serveur d'application et l'équilibreur de charges soient configurés adéquatement afin de mettre en place des politiques de sécurité et ainsi faire face à la vulnérabilité.

**Afin de réaliser les parties suivantes CI/CD certaines modifications ont dû être appliqué dans le code

Création d'un DockerFile

Dockerfile
KiKero99, last week | 2 authors (You and one other)

```
FROM node:18

# Set the working directory
WORKDIR /dvna-master

# Copy the package.json and package-lock.json files
COPY dvna-master/package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application, including the .env file
COPY dvna-master .

# Expose the port for your app
EXPOSE 9090

# Set the default command to start the app
CMD ["npm", "start"]

```

Ajout de code

```
module.exports = {
    listen: process.env.APP_LISTEN || '0.0.0.0',
    port: process.env.APP_PORT || process.env.PORT || 9090
}       KiKero99, last week • merde
```

Changement de la base de données (Nous avons dû utiliser une base de données postgreSQL)

```
const URL = process.env.POSTGRES_URL;


// Initialize Sequelize connection
const sequelize = new Sequelize( URL, {
  dialect: 'postgres',
  logging: false,        You, last week • test action
  dialectOptions: {
    ssl: true, // Use SSL if your PostgreSQL instance requires it (like on Azure)
  },
});
```

Mise à jour de la version du bcrypt dans le package.json pour la rendre compatible avec le reste du projet:

```
"bcrypt": "^5.0.0",
```

Utilisation de node 18

Description du pipeline d'intégration continue (CI)

Description du pipeline et des outils utilisés:

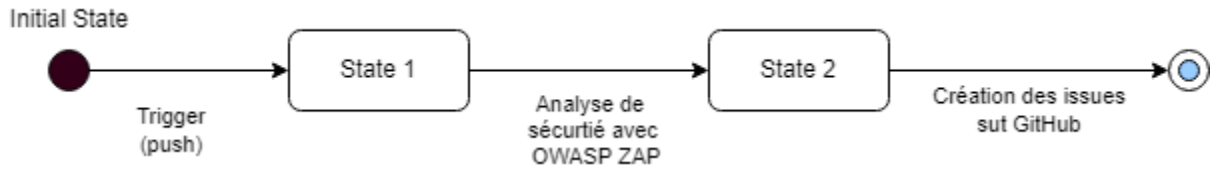
Les pipelines d'intégration continue (CI) sont utilisées pour automatiser diverses tâches lorsqu'une action est effectuée sur les branches d'un projet, telles que l'analyse de sécurité, l'exécution des tests et la gestion des dépendances. Dans notre projet, trois pipelines sont déployées automatiquement : **CodeQL**, **OWASP ZAP** et **Dependabot**.

Les technologies utilisées pour la mise en place de ces pipelines sont les suivantes :

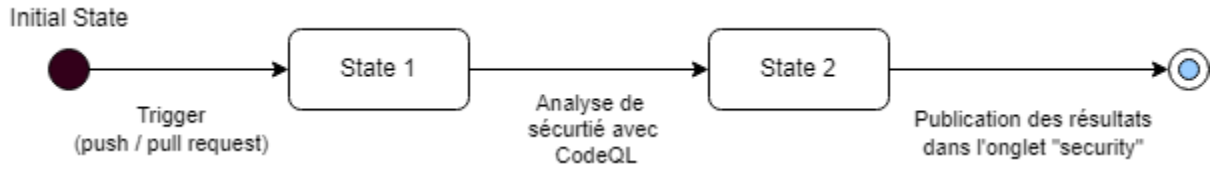
- CodeQL: Outil d'analyse statique utilisé pour examiner le code source des applications et détecter les failles de sécurité dans les langages supportés tels que JavaScript, TypeScript, Swift, entre autres.
- OWASP ZAP: Outil de test de sécurité permettant l'identification de vulnérabilités dans une application web et ainsi découvrir différentes failles présentes.
- Dependabot: Outil intégré à Github et permettant la gestion des dépendances en effectuant des "pull requests".
- Github Actions: Plateforme d'automatisation qui execute des workflows CI/CD, déclencheurs basés sur des événements comme les pushs et les pull requests
- Github Secrets: Fonctionnalité permettant de stocker des données dites "secrètes" afin de ne pas les exposer dans le code.

Diagramme d'état/d'activité pour la séquence des étapes de la pipeline:

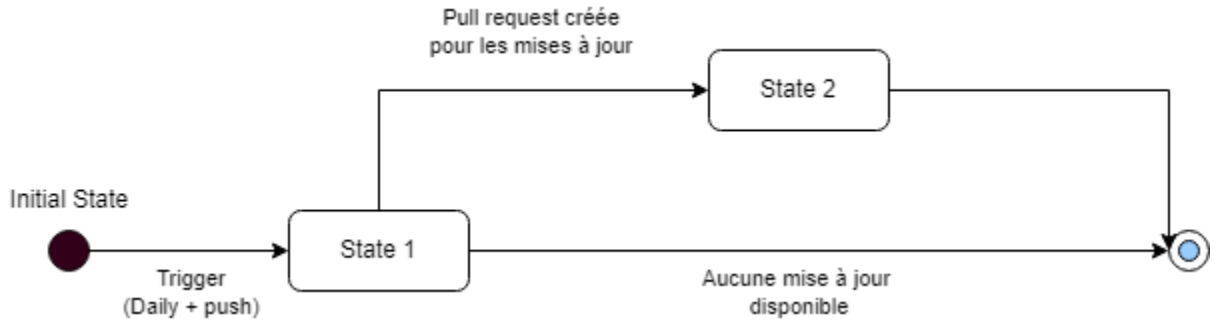
- Diagramme d'état pour OWASP ZAP



- Diagramme d'état pour CodeQL



- Diagramme d’état pour Dependabot



Description du pipeline de déploiement continu (CD)

Étapes du processus de déploiement:

Dans le cadre du processus de déploiement, la pipeline de déploiement continu (CD) garantit la mise à jour automatique des applications en intégrant progressivement les nouvelles versions dans les environnements de production, assurant ainsi un déploiement en continu. Dans notre projet pratique, nous avons automatisé la construction des images Docker à l'aide d'un Dockerfile. Ces images sont ensuite envoyées sur Docker Hub, où elles sont accessibles à ceux disposant des autorisations appropriées. Enfin, les images sont déployées sur un environnement cloud Azure, dans un serveur dédié aux applications contenues dans des containers Docker : **"Container Apps"**.

Figure 1 section CD: Notre Serveur Azure

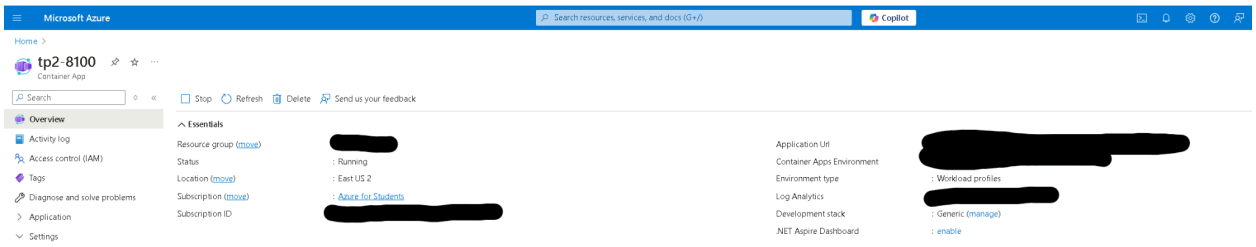
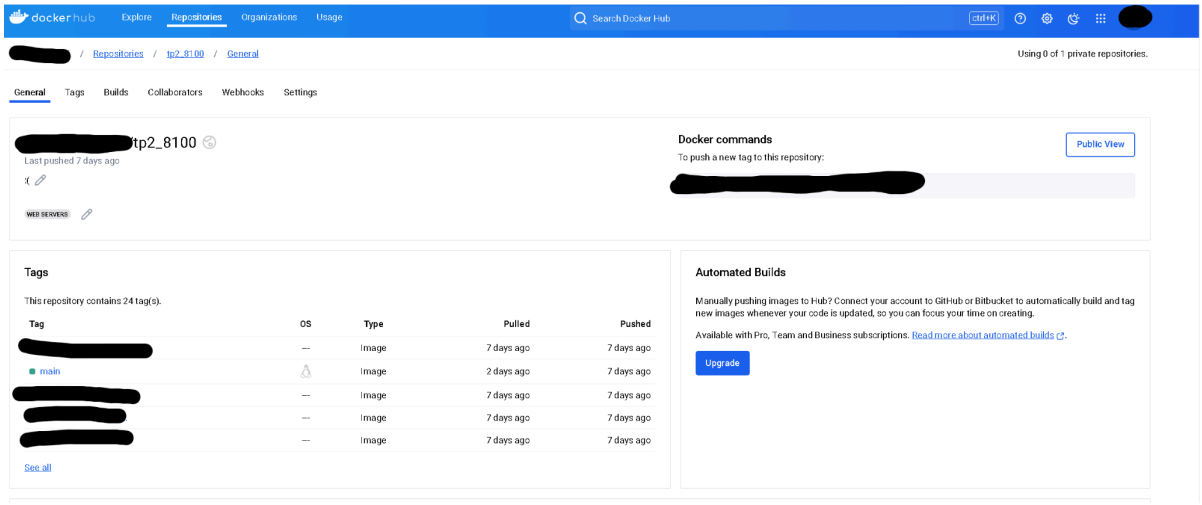


Figure 2 section CD: Notre instance DockerHub



Étapes du processus de déploiement:

Les étapes suivantes décrivent le véritable processus des pipelines de déploiement continu (CD) pour notre projet pratique. En cas de doute, vous pouvez consulter les différents fichiers YAML du projet. Le fichier YAML pour le déploiement sur Azure a été généré automatiquement par la plateforme, avec un accès configuré à notre dépôt Git.

1- **Élément déclencheur**: Les pipelines de déploiement continu (CD) sont déclenchés automatiquement par un push sur la branche principale. Dans le cas du déploiement sur Azure, il est également possible de les déclencher manuellement via la commande workflow_dispatch.

2- **Authentification sur DockerHub**: Le pipeline se connecte à DockerHub avec les jetons secrets du github pour garantir la sécurité des éléments de production.

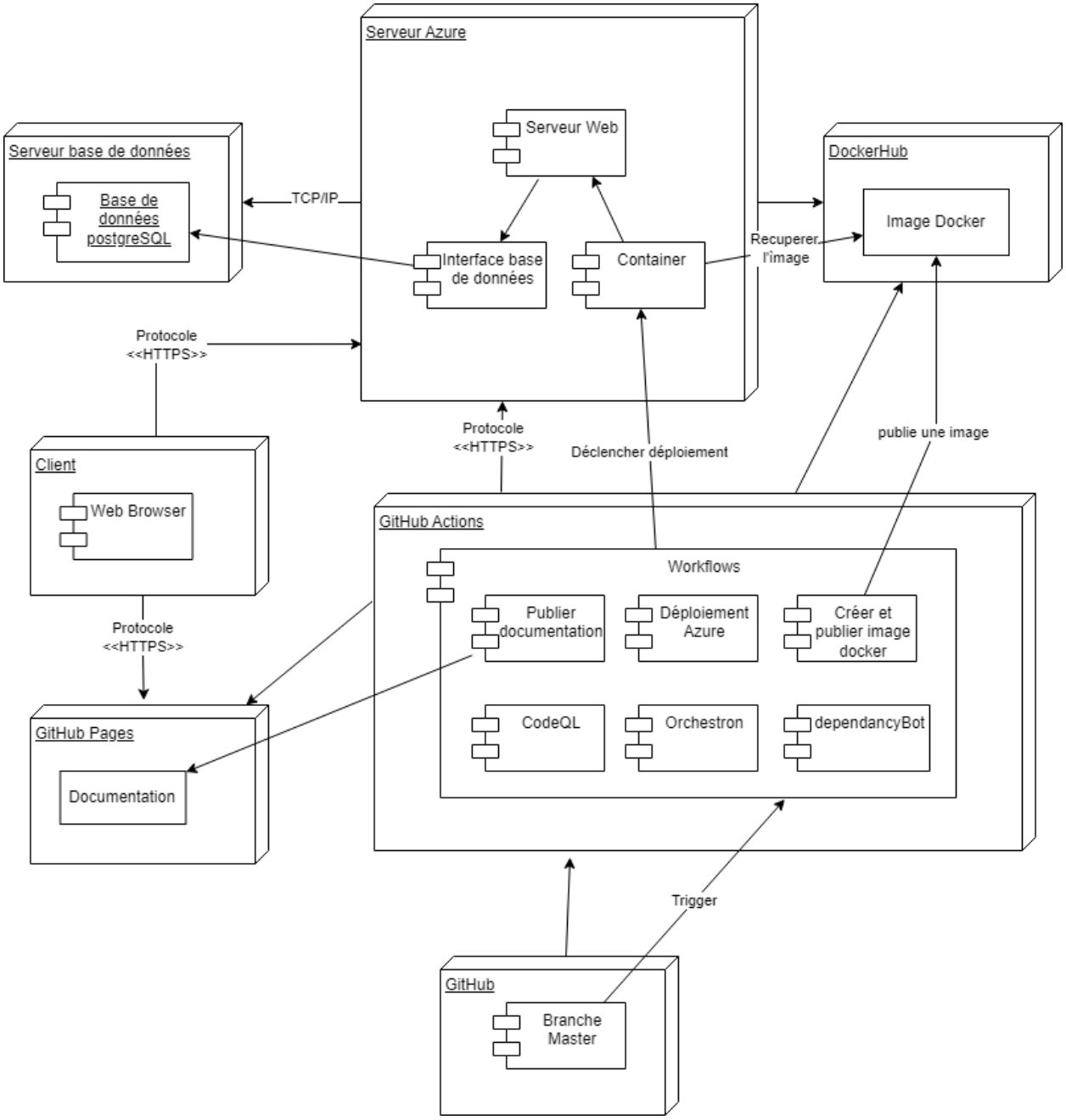
3- **Extraction des métadonnées**: les tags et les labels de l'image Docker sont générées pour sa construction

4- **Construction et publication de l'image**:Une nouvelle image Docker est construite et envoyée sur DockerHub, en y incluant les métadonnées générées lors de l'étape précédente.

5- **Authentification sur Azure**: Le pipeline se connecte à Azure, encore à l'aide des secrets

6- **Déploiement de l'application**: L'image Docker est déployée automatiquement dans notre environnement Azure pour l'application DVNA, où elle est accessible à tous en production.

Diagramme de déploiement de l'application:



Lien vers la documentation et l'environnement de déploiement:

Lien vers l'application deployée: https://tp2-8100.wittyisland-f2df98dd.eastus2.azurecontainerapps.io/login

Lien vers la documentation: https://kikero99.github.io/TP2_LOG8100/LOG8100_TP2.pdf

Conclusion

À travers ce travail pratique, nous avons été en mesure d'effectuer une analyse de sécurité nous permettant de déterminer les différentes vulnérabilités de l'application DVNA à l'aide d'outils comme OWASP ZAP et Orchestron. Parmi ces dernières, on en retrouve un grand nombre faisant partie du top 10 de L'OWASP et qui, en cas de non-correction, pourraient compromettre la sécurité d'un utilisateur visitant l'application. En ce qui concerne l'utilisation du pipeline d'intégration continue, elle s'est avérée utile pour l'automatisation des tests de sécurité et l'automatisation de la gestion des dépendances. Le pipeline de déploiement, quant à lui, nous a permis de garantir un déploiement continu sans interruption.

Afin d'améliorer une application de ce genre, il serait intéressant de faire recours à un pipeline d'intégration continue. Ce pipeline permettrait d'automatiser la sécurité de l'application en fournissant des analyses de vulnérabilités à chaque fois que le code est modifié à l'aide d'outils comme ZAP ou CodeQL. On serait donc en mesure d'identifier les vulnérabilités en temps réel, poussant ainsi les développeurs à corriger de telles erreurs. L'utilisation de git permettrait aussi l'usage de secrets menant à une non-exposition d'informations sensibles dans le code. L'utilisation d'un pipeline de déploiement continu serait aussi intéressante, car le code serait analysé avant chaque déploiement, diminuant ainsi le risque de vulnérabilité et les possibles erreurs humaines liées au déploiement.