



**POLYTECHNIQUE
MONTRÉAL**

UNIVERSITÉ
D'INGÉNIERIE

LOG8100 – DevSecOps

Travail pratique No. 2

Tests de pénétration et outils pour intégration et déploiement continus

2067816 - Andy Chen

2270891 - Ali Hazime

2335476 - Louis Lalonde

Soumis à

Antoine Boucher

24 octobre 2024

Automne 2024

Introduction

Le but de ce travail pratique est d'analyser les vulnérabilités de *Damn Vulnerable NodeJS Application (dvna)*, une application web non sécurisée, à travers l'exécution de tests de pénétration, ainsi que d'implémenter une pipeline DevSecOps pour le projet.

Nous allons identifier et documenter les failles potentielles de l'application, puis proposer des stratégies de mitigation pour chaque vulnérabilité. Par ailleurs, nous implémenterons un pipeline CI/CD robuste permettant l'automatisation des tests et du déploiement sécurisé de l'application.

Pour ce faire, nous utilisons divers outils, notamment OWASP ZAP et Orchestron pour l'analyse de la sécurité de l'application, Github Actions pour configurer le pipeline CI, ainsi que Docker Hub et Azure pour le déploiement continu.

Analyse de sécurité

Avant l'exécution des tests de pénétration, il faut lancer l'application vulnérable dans un navigateur qui est surveillée par ZAP en cliquant sur 'Manual Explore'.

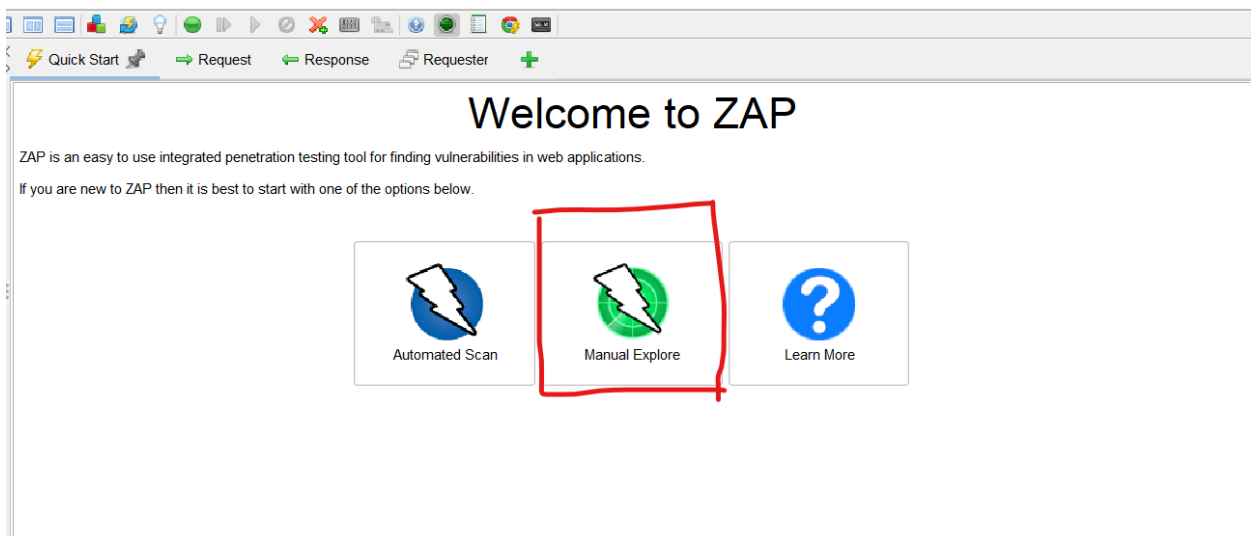


Fig. 1: Interface de démarrage de l'outil OWASP ZAP

Pour maximiser la couverture des tests de pénétration, nous naviguons manuellement dans les différents pages de l'application tout en soumettant des formulaires pendant que ZAP prend note des actions effectuées. Ensuite, nous attaquons l'application avec 'Spider' pour permettre de détecter toutes les pages.

Une fois que ces deux étapes sont complétées, nous effectuons les tests de pénétration avec l'option 'Active Scan'. Dans la prochaine section seront présentées trois vulnérabilités identifiées par les tests.

Vulnérabilités

A03:2021 - Injection

Page: /app/usersearch

User Search

Login

Submit

Fig. 2: Aperçu de la page /app/usersearch

Problème: Sur la page /app/usersearch, il y a un formulaire non sécurisé qui est vulnérable à des injections SQL. Ce type d'attaque est critique, car il permet à un attaquant d'injecter des commandes SQL malveillantes dans une application, ce qui peut compromettre l'intégrité et la confidentialité des données, entraîner des fuites d'informations sensibles ou permettre une prise de contrôle complète de la base de données. On peut voir ci-dessous un exemple d'injection SQL préparé par ZAP qui envoie le string 'test OR 1=1 --' dans le champ de saisie.


SQL Injection	
URL:	http://localhost:9090/app/usersearch
Risk:	 High
Confidence:	Medium
Parameter:	login
Attack:	test OR 1=1 --
Evidence:	
CWE ID:	89
WASC ID:	19
Source:	Active (40018 - SQL Injection)
Input Vector:	Form Query
Description:	
SQL injection may be possible.	
Other Info:	
<p>The page results were successfully manipulated using the boolean conditions [test AND 1=1 --] and [test OR 1=1 --]</p> <p>The parameter value being modified was NOT stripped from the HTML output for the purposes of the comparison.</p> <p>Data was NOT returned for the original parameter.</p>	
Solution:	
<p>Do not trust client side input, even if there is client side validation in place.</p> <p>In general, type check all data on the server side.</p> <p>If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?'</p>	

Fig. 3: Exemple d'injection SQL dans l'application *dvna*

Solution: Pour prévenir les injections SQL, la première étape consiste à valider et *sanitize* les inputs des utilisateurs avant de les utiliser dans une requête SQL. Ensuite, il est recommandé d'utiliser les requêtes préparées qui séparent les instructions SQL des données fournies par

l'utilisateur. En effet, au lieu de concaténer directement les inputs utilisateur dans une requête SQL, cette approche utilise des placeholders (:param ou ?) pour indiquer l'emplacement des valeurs dynamiques au moteur de la base de données. Ce dernier traitera les inputs utilisateur comme des données et non comme des commandes SQL exécutables. Voici un exemple d'utilisation des requêtes préparées pour prévenir l'injection SQL à la page /app/usersearch:

```
9 module.exports.userSearch = function (req, res) {
10   var query = "SELECT name,id FROM Users WHERE login='" + req.body.login + "'";
11   db.sequelize.query(query, {
12     model: db.User
13   }).then(user => {
```

Fig. 4: Exemple de concaténation des données dans la requête SQL (Mauvaise pratique)

```
10   var query = "SELECT name, id FROM Users WHERE login = :login";
11   db.sequelize.query(query, {
12     replacements: { login: req.body.login },
13     type: db.sequelize.QueryTypes.SELECT
14   }).then(user => {
```




Fig. 5: Exemple d'utilisation des requêtes préparées pour prévenir des injections SQL

A04:2021 - Insecure Design

Page: /app/admin/usersapi

```
Pretty-print ☒
{
  "success": true,
  "users": [
    {
      "id": 1,
      "name": "test",
      "login": "test",
      "email": "test@test.ca",
      "password": "$2a$10$OgaZRyHhjPbNtjGiU9Zciuo32/hQmCxf7KOmeVANgbaf1IDRbVAq0",
      "role": null,
      "createdAt": "2024-09-29T23:10:07.845Z",
      "updatedAt": "2024-09-30T20:31:26.878Z"
    },
    {
      "id": 2,
      "name": "ZAP",
      "login": "ZAP",
      "email": "zapproxy@example.com",
      "password": "$2a$10$ZpOpTGZAipz.Xh3CGWBSceC7MgEiFk3X.OsuGmb5bklmxJw7WZde2",
      "role": null,
      "createdAt": "2024-09-29T23:15:41.194Z",
      "updatedAt": "2024-09-29T23:15:41.194Z"
    }
  ],
}
```

Fig. 6: Aperçu de la page /app/admin/usersapi

Problème: Sur la page /app/admin/usersapi, il est possible de voir tous les utilisateurs avec leurs mots de passe hachés. Bien que les mots de passe soient chiffrés, le fait qu'ils soient accessibles via l'interface utilisateur implique une vulnérabilité critique et une mauvaise conception en termes de sécurité. Si un attaquant accède à ces données, il pourrait essayer de déchiffrer les mots de passe ou exploiter d'autres informations pertinentes en clair comme le courriel des utilisateurs. Un autre point important à souligner est le fait que la page est accessible à tous les utilisateurs, alors qu'elle devrait être exclusive à ceux qui ont le rôle admin.

Hash Disclosure - BCrypt	
URL:	http://localhost:9090/app/admin/usersapi
Risk:	High
Confidence:	High
Parameter:	
Attack:	
Evidence:	\$2a\$10\$.3LFrH7NUKLhBZEIBMtQeSrgjNks5ln1rmF/OkRVFrPtUZea6ZeS
CWE ID:	200
WASC ID:	13
Source:	Passive (10097 - Hash Disclosure)
Input Vector:	
Description:	A hash was disclosed by the web server. - BCrypt
Other Info:	
Solution:	Ensure that hashes that are used to protect credentials or other resources are not leaked by the web server or database. There is typically no requirement for password hashes to be accessible to the web browser.

Fig. 7: Exemple de *Insecure Design* dans l'application *dvna*

Solution: Deux actions doivent être effectuées pour fixer cette vulnérabilité. Tout d'abord, il faut absolument enlever les mots de passe de l'interface utilisateur, même s'ils sont chiffrés. Cela peut se faire en filtrant seulement les attributs d'utilisateur qu'on veut afficher à l'écran, comme on peut le voir dans l'image suivante. Cela empêche l'affichage des mots de passe chiffrés.

```
207 module.exports.listUsersAPI = function (req, res) {
208   db.User.findAll({attributes: [ 'id', 'name', 'email' ]}).then(users => {
209     res.status(200).json({
210       success: true,
211       users: users
212     })
213   })
214 }
```

Fig. 8: Spécification des attributs non secrets à afficher sur la page /app/admin/usersapi

Ensuite, on voudrait limiter l'accès à cette page confidentielle seulement aux utilisateurs ayant le rôle admin. Pour ce faire, il faut créer une fonction dont le but est de vérifier le rôle de l'utilisateur connecté, et qui sera appelée lors du routing vers la page /app/admin/usersapi. Le code devrait ressembler à ceci:

```

19 module.exports.isAdmin = function (req, res, next){
20   if(req.user.role=='admin')
21     next()
22   else
23     res.status(401).send('Unauthorized')
24 }

```

Fig. 9: Fonction vérifiant si l'utilisateur connecté a le rôle admin

```

36 router.get('/admin', authHandler.isAuthenticated, function (req, res) {
37   res.render('app/admin', {
38     admin: (req.user.role == 'admin')
39   })
40 })
41
42 router.get('/admin/usersapi', authHandler.isAuthenticated, authHandler.isAdmin, appHandler.listUsersAPI)

```

Fig. 10: Appel de la fonction *isAdmin* lors du routing vers la page /app/admin/usersapi

Ces deux suggestions devraient mitiger la vulnérabilité *Insecure Design* identifiée par ZAP à la page /app/admin/usersapi.

A07:2017 - Cross-Site Scripting (XSS)

Page: /app/products

✶ Damn Vulnerable NodeJS Application Logout				
☰ Available Products			Search Product	Add Product
#	Name	Code	Tags	Description
1	p	123	tag1	the letter p Edit
2	p	c:\Windows\system.ini	tag1	the letter p Edit
3	p	../../../../../../../../../../../../Windows/system.ini	tag1	the letter p Edit
4	p	c:\Windows\system.ini	tag1	the letter p Edit
5	p	../../../../../../../../../../../../Windows/system.ini	tag1	the letter p Edit
6	p	/etc/passwd	tag1	the letter p Edit
7	p	../../../../../../../../../../../../etc/passwd	tag1	the letter p Edit
8	p	/	tag1	the letter p Edit

Fig. 11: Aperçu de la page /app/products

Problème: Sur la page /app/products, il est possible de passer un script dans le navigateur dans le but de l'exécuter. Cette faille permet à un attaquant d'exécuter des scripts malveillants dans le contexte d'un utilisateur légitime, ouvrant ainsi la voie à diverses attaques, telles que le vol de cookies, la prise de contrôle de session, ou encore la redirection vers des sites

dangereux. On peut voir un exemple de ce type d'attaque critique, nommé *Cross Site Scripting*, dans la figure suivante où ZAP ajoute un script affichant une alerte dans le navigateur.

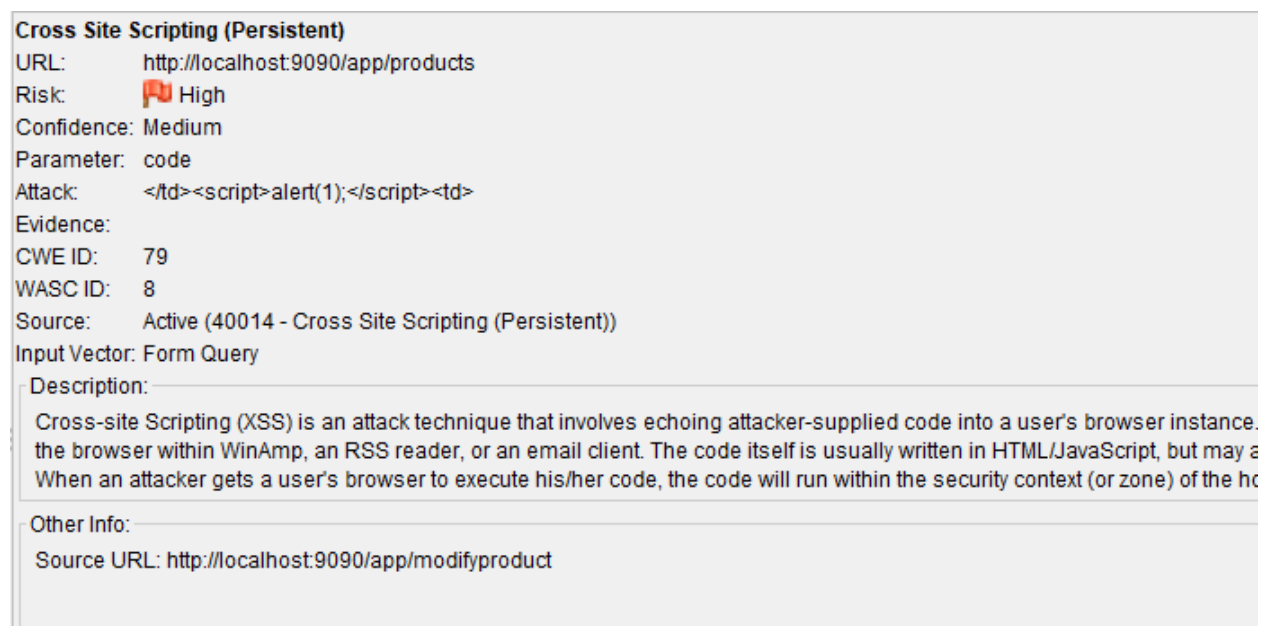


Fig. 12: Exemple de *Cross-Site Scripting* (XSS) dans l'application *dvna*

Solution: Pour prévenir les attaques XSS, nous proposons deux améliorations de code. Pour commencer, il est recommandé d'utiliser le *Content Security Policy* (CSP), qui est un header *HTTP* permettant de prévenir les attaques XSS en contrôlant les ressources qu'une page peut charger. Ce mécanisme applique des règles custom pour spécifier quels domaines sont autorisés à charger et exécuter du contenu, ce qui garantit que seules des sources fiables sont utilisées. Cela réduit le risque que des scripts dangereux soient injectés et exécutés sur la page, même si un attaquant parvient à y insérer du code.

Ensuite, une deuxième amélioration possible est de toujours *escape* les inputs utilisateurs pour empêcher l'exécution de code HTML non voulu. Cela implique de transformer les caractères spéciaux, tels que `<`, `>` et `&`, à leurs entités HTML pour éviter qu'ils soient interprétés comme du code. Avec cette stratégie, le script injecté par ZAP dans l'image précédente deviendrait:

`<script>alert(1)</script>`

plutôt que

`<script>alert(1)</script>`

et il sera affiché comme texte plutôt qu'être exécuté par le navigateur. On peut soit créer notre propre logique pour *escape* des inputs utilisateurs ou utiliser des bibliothèques existantes comme *lodash* avec la méthode `escape(string)`.¹

¹ <https://lodash.com/docs#escape>

Rapport de test avec Orchestron

Pour générer un rapport de test à l'aide de Orchestron et OWASP ZAP, nous avons d'abord créé un projet sur Orchestron en suivant les instructions dans le fichier README sur le répertoire github suivant: <https://github.com/we45/orchestron-community>.

Une fois que l'application roule localement sur notre machine, il suffit de créer une application dans un projet afin de générer une commande pour effectuer un webhook. Nous pouvons voir la commande *curl* à utiliser dans la figure 13.

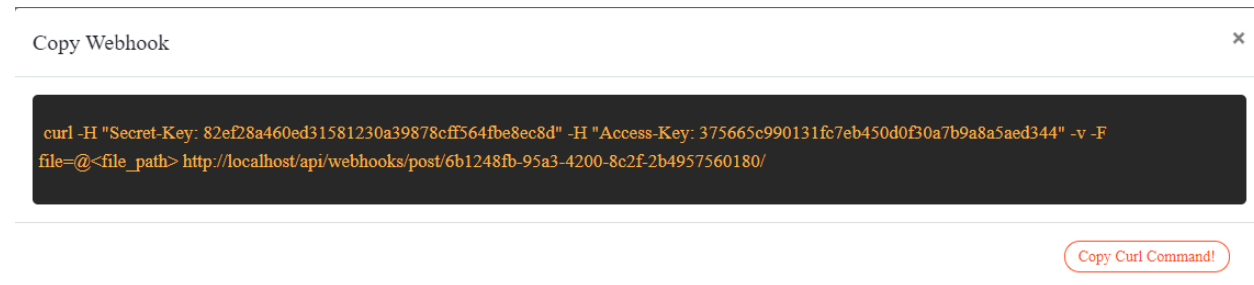


Fig. 13: Commande permettant de transférer un fichier sur Orchestron

En remplaçant le `<file_path>` par le chemin où on sauvegarde le rapport de OWASP ZAP, cette commande permet de transférer le fichier sur le serveur de Orchestron. Ainsi, nous obtenons un aperçu des vulnérabilités, comme illustré à la figure 14. Noter qu'il est aussi possible d'utiliser l'interface de Orchestron pour soumettre un fichier, l'utilisation d'un webhook permet d'automatiser le processus dans le cadre d'un projet en développement continu.

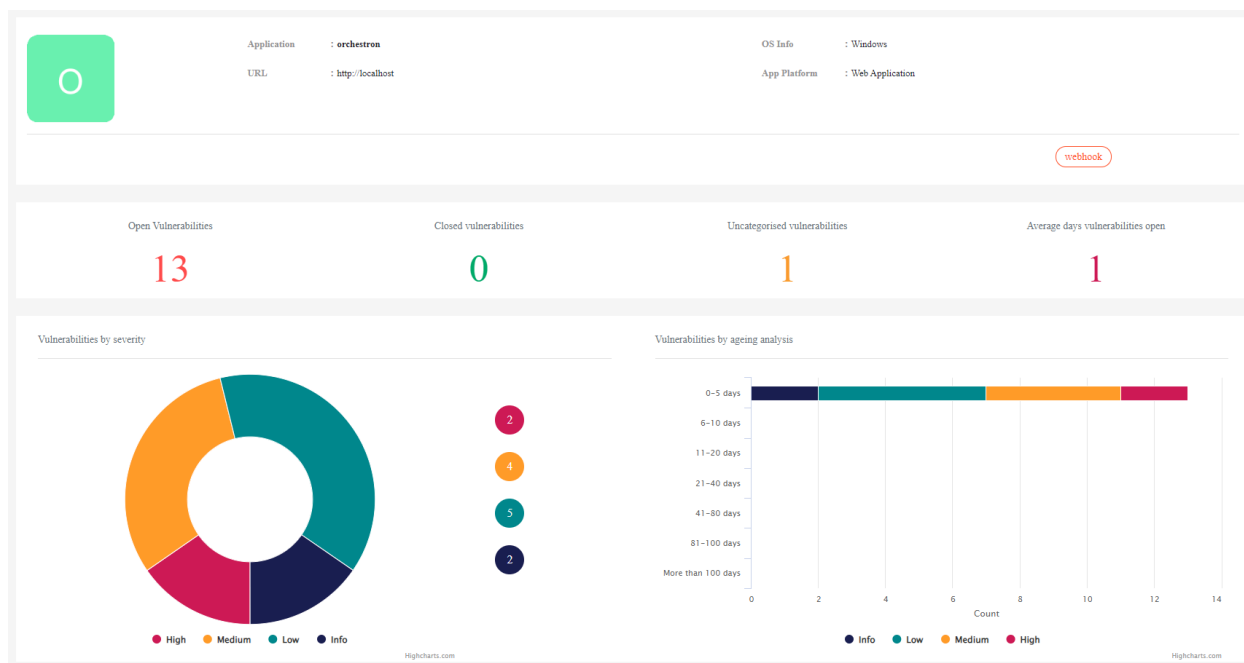


Fig. 14: Aperçu des vulnérabilités du projet sur Orchestron Community

Nous pouvons voir qu'il y a 13 vulnérabilités détectées à des niveaux différents, dont 2 sont *High*, 4 *Medium*, 5 *Low* et 2 seulement informationnelles dont les détails sont montrés dans les figures suivantes.

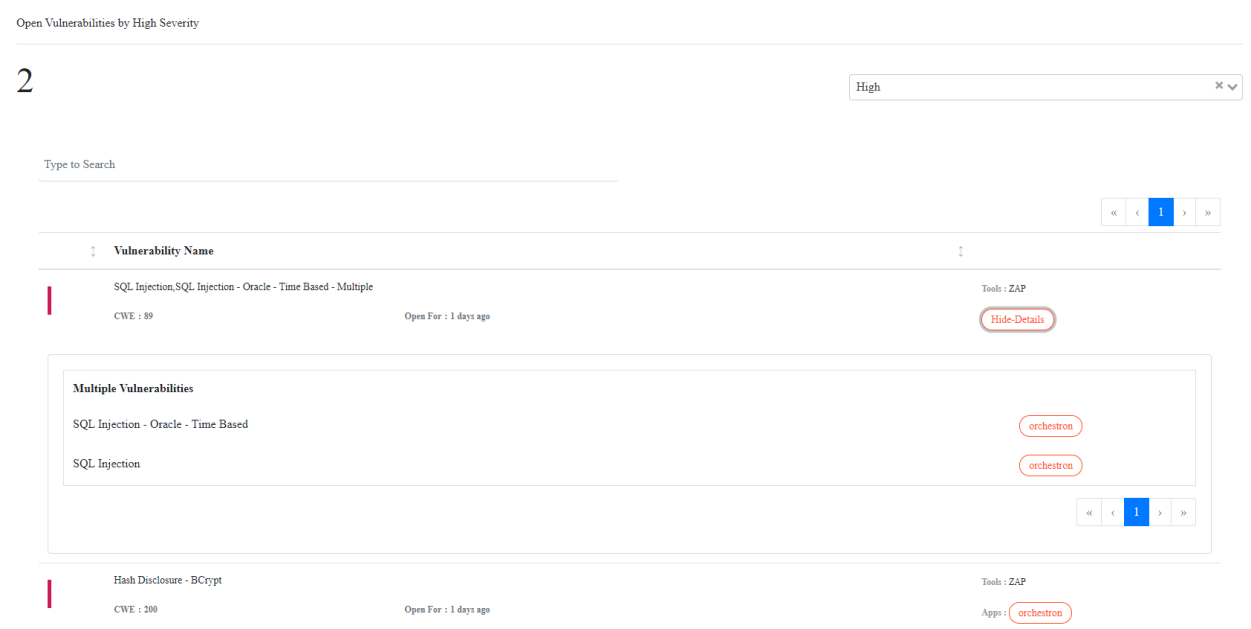


Fig. 15: Vulnérabilités *High* sur Orchestron

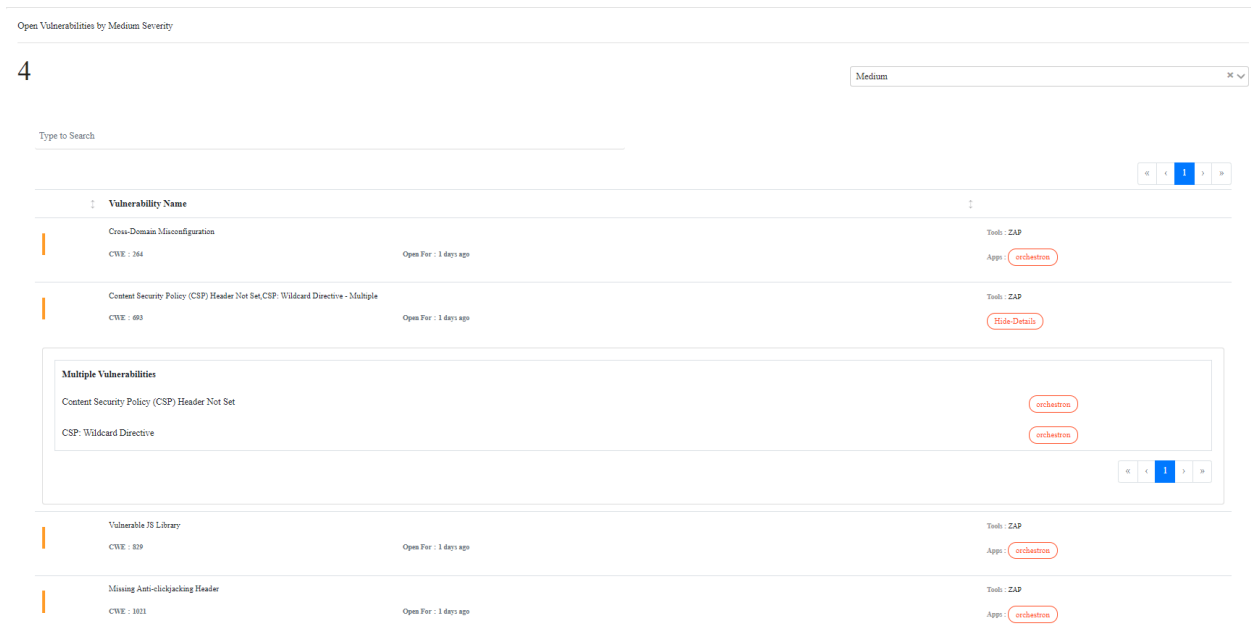


Fig. 16: Vulnérabilités *Medium* sur Orchestron

Open Vulnerabilities by Low Severity

5

Low

Type to Search

Vulnerability Name			
Application Error Disclosure, Private IP Disclosure, Server Leaks Information via "X-P...	CWE : 200	Open For : 1 days ago	Tools : ZAP App: orchestron
<div>Multiple Vulnerabilities</div> <div>Application Error Disclosure</div> <div>Private IP Disclosure</div> <div>Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)</div>			
Strict-Transport-Security Header Not Set	CWE : 339	Open For : 1 days ago	Tools : ZAP App: orchestron
X-Content-Type-Options Header Missing	CWE : 693	Open For : 1 days ago	Tools : ZAP App: orchestron
Onion Domain JavaScript Source File Inclusion	CWE : 829	Open For : 1 days ago	Tools : ZAP App: orchestron
Cookies without SameSite Attribute	CWE : 1275	Open For : 1 days ago	Tools : ZAP App: orchestron

Fig. 17: Vulnérabilités *Low* sur Orchestron

Open Vulnerabilities by Info Severity

2

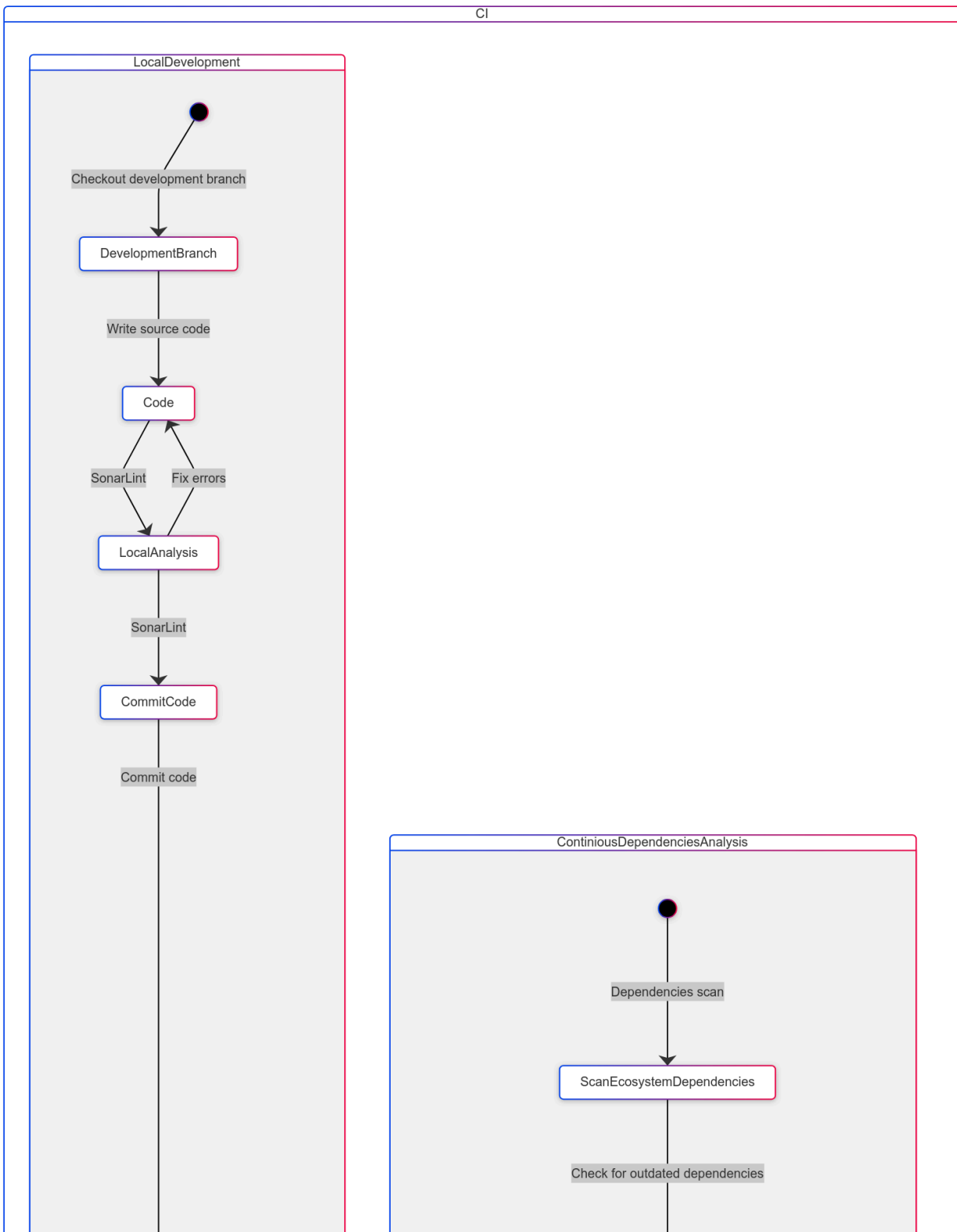
Info

Type to Search

Vulnerability Name			
User Agent Fuzzer	CWE : 0	Open For : 1 days ago	Tools : ZAP App: orchestron
GET for POST	CWE : 16	Open For : 1 days ago	Tools : ZAP App: orchestron

Fig. 18: Vulnérabilités *Info* sur Orchestron

Pipeline d'intégration continue (CI)



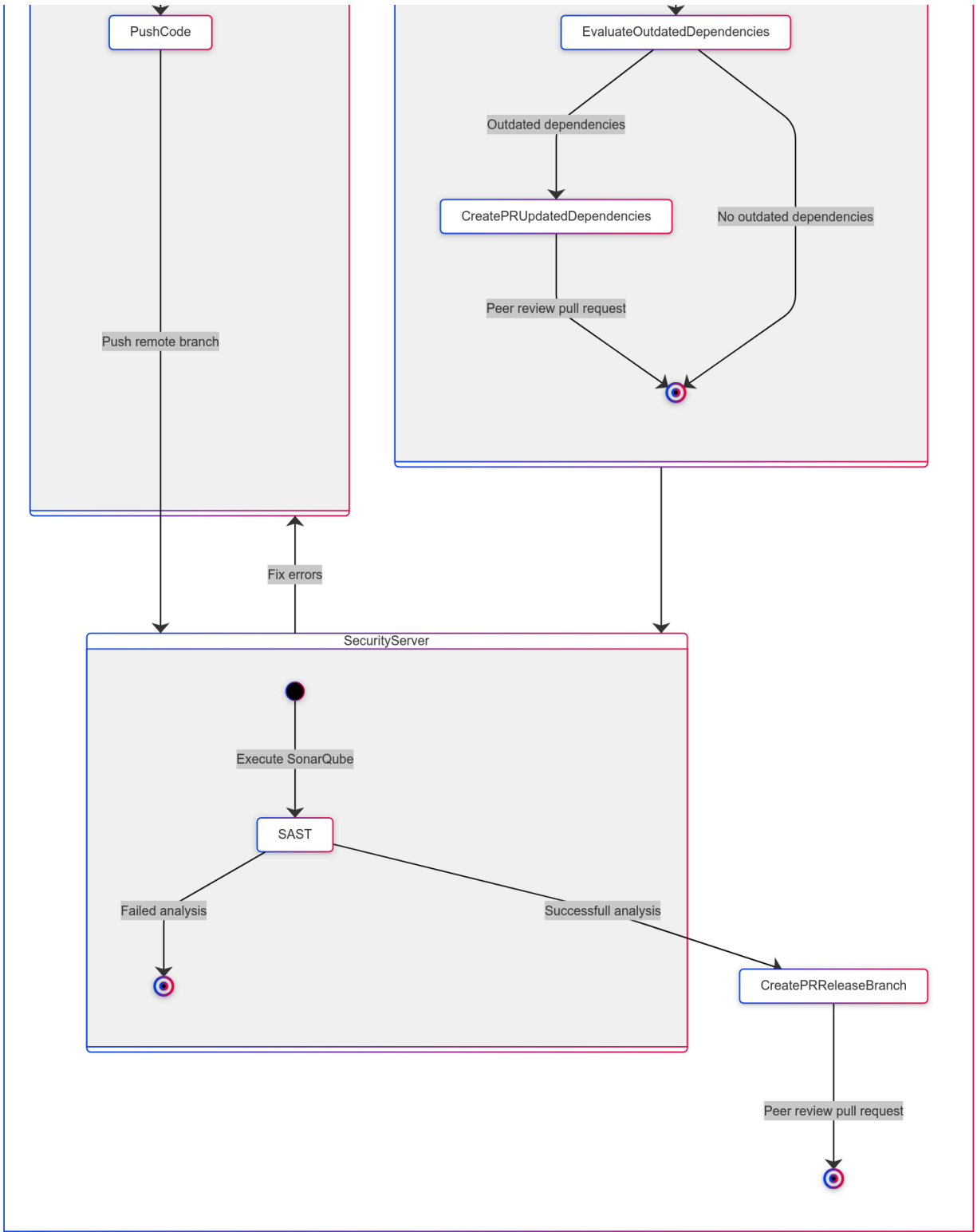


Fig 19. Diagramme d'état de la pipeline d'intégration continue du projet.

Outils utilisés

- [GitHub](#): Fournisseur de système de contrôle de version cloud et On-Premises.
- [SonarLint](#): Librairie d'analyse locale du code pour améliorer la qualité et la sécurité avant le commit.
- [Dependabot](#): Module intégré sous forme de GitHub action qui permet l'automatisation de la gestion des dépendances, vérifiant régulièrement les mises à jour et créant des pull requests pour celles-ci.
- [SonarQube](#): Utilisé pour l'analyse statique du code afin de détecter les vulnérabilités potentielles.

Description

1. Développement local

- [Checkout Development Branch](#): Les développeurs commencent par récupérer la branche de développement pour leur fonctionnalité en fonction de la tâche créée à partir du tableau Kanban sur GitHub.
- [Write Source Code](#): Les développeurs écrivent la logique de métier de la fonctionnalité.
- [SonarLint](#): Utilisé de façon continue sur l'environnement de développement local des développeurs pour analyser le code et identifier les problèmes de qualité ou de sécurité avant le commit.
- [Push code](#): Après avoir corrigé les erreurs détectées par SonarLint, le code est commit et publié sur la branche distante.

2. Analyse continue des dépendances

- [Scan Ecosystem Dependencies](#): Analyse de façon continue les dépendances du repository distant pour vérifier s'il existe des mises à jour disponibles ou des vulnérabilités connues nécessitant une mise à jour.
- [Create PR for Updated Dependencies](#): Si des dépendances sont obsolètes ou vulnérables, Dependabot crée une pull request de mise à jour pour chacune des dépendances devant être mise à jour. Les vulnérabilités sont rapportées dans la section Dependabot de la ressource Security de GitHub.
- [Peer Review Pull Request](#): Les pull requests créés par Dependabot sont examinées par les pairs et acceptées s'il y a un accord commun.

3. Serveur de sécurité

- [Execute SonarQube \(SAST\)](#): Analyse statique du code pour détecter les vulnérabilités de sécurité avec SonarQube hébergé sur un serveur de sécurité.
- [Successful Analysis](#): Si l'analyse réussit, le développeur est autorisé à créer une pull request pour intégrer sa fonctionnalité à la branche release.

- Failed Analysis : Si l'analyse échoue, le développeur doit corriger les problèmes de sécurité rapportés par SonarQube dans son environnement local ce qui redémarre le cycle d'intégration continu.

Pipeline de déploiement continu (CD)

Voir l'[annexe A](#) pour un supplément de diagramme d'état du pipeline de déploiement continu.

Outils utilisés

- [GitHub](#)
- [Docker](#): Plateforme de conteneurisation pour le packaging et le déploiement d'applications.
- [Docker Hub](#): Registre pour stocker et distribuer les images Docker.
- [ZAP \(OWASP Zed Attack Proxy\)](#): Outil pour les tests de sécurité dynamiques (DAST).
- [Watchtower](#): Service pour automatiser la mise à jour des conteneurs Docker.

Description

1. Déclenchement du CD

- [Merge Pull Request](#): Le processus CD est déclenché lorsqu'une pull request est fusionnée dans la branche release.

2. Staging

- [Checkout Repo](#): Récupération du code source de la branche release.
- [Docker Login Staging](#): Authentification sur Docker Hub pour accéder au registre d'images.
- [Build Staging Docker Image](#): Tagging & construction de l'image Docker pour l'environnement de staging.
- [Push Staging Docker Image](#): Publication de l'image Docker de staging sur Docker Hub.
- [Generate Attestation Staging](#): Génération d'une attestation pour la provenance de l'image de staging.

3. Staging Server

- [Watchtower Service](#): Surveillance continue des nouvelles images staging Docker disponibles à une fréquence prédéterminée. Ce service est déployé sous forme de Docker container sur l'hôte.
- [Staging Server Deploy Image](#): Déploiement automatique de la nouvelle image de staging.
- [DAST \(ZAP Scan\)](#): Exécution de tests de sécurité dynamiques sur le serveur de staging lorsque la nouvelle version de l'application est déployée. Un rapport d'état de l'analyse est publiée dans la section issues du repository GitHub.

- i. Successful Analysis: Si les tests DAST réussissent, le processus continue vers la production.
- ii. Detected Vulnerabilities: Selon la sévérité des vulnérabilités détectées, le gestionnaire des releases de l'application décide si les corrections de vulnérabilités à apporter peuvent être réalisées à même la release branch ou si une fix branch doit être ouverte.

4. Production

- Checkout Code Prod: Récupération du code source pour la production.
- Docker Login Prod: Authentification sur Docker Hub pour l'environnement de production.
- Create Tag: Création d'un tag Git, en fonction de la version utilisée pour annoter la branch release d'origine, p. ex., release/**1.0.2**, pour marquer la version de production.
- Create Release: Création d'une release GitHub avec les artefacts nécessaires.
- Publish GitHub Pages: Mise à jour du site web du repository à partir de la documentation.
- Build Docker Image Prod: Tagging & construction de l'image Docker pour la production.
- Push Docker Image Prod: Publication de l'image Docker de production sur Docker Hub.
- Generate Attestation Prod: Génération d'une attestation pour la provenance de l'image de production.

5. Production Server

- Watchtower Service Prod: Surveillance continue des nouvelles images Docker de production.
- Prod Server Deploy Image: Déploiement automatique de la nouvelle image de production.
- Monitoring: Surveillance continue de l'application en production. P. ex., Prometheus, Grafana, Elastic Stack, etc.

Diagramme de déploiement

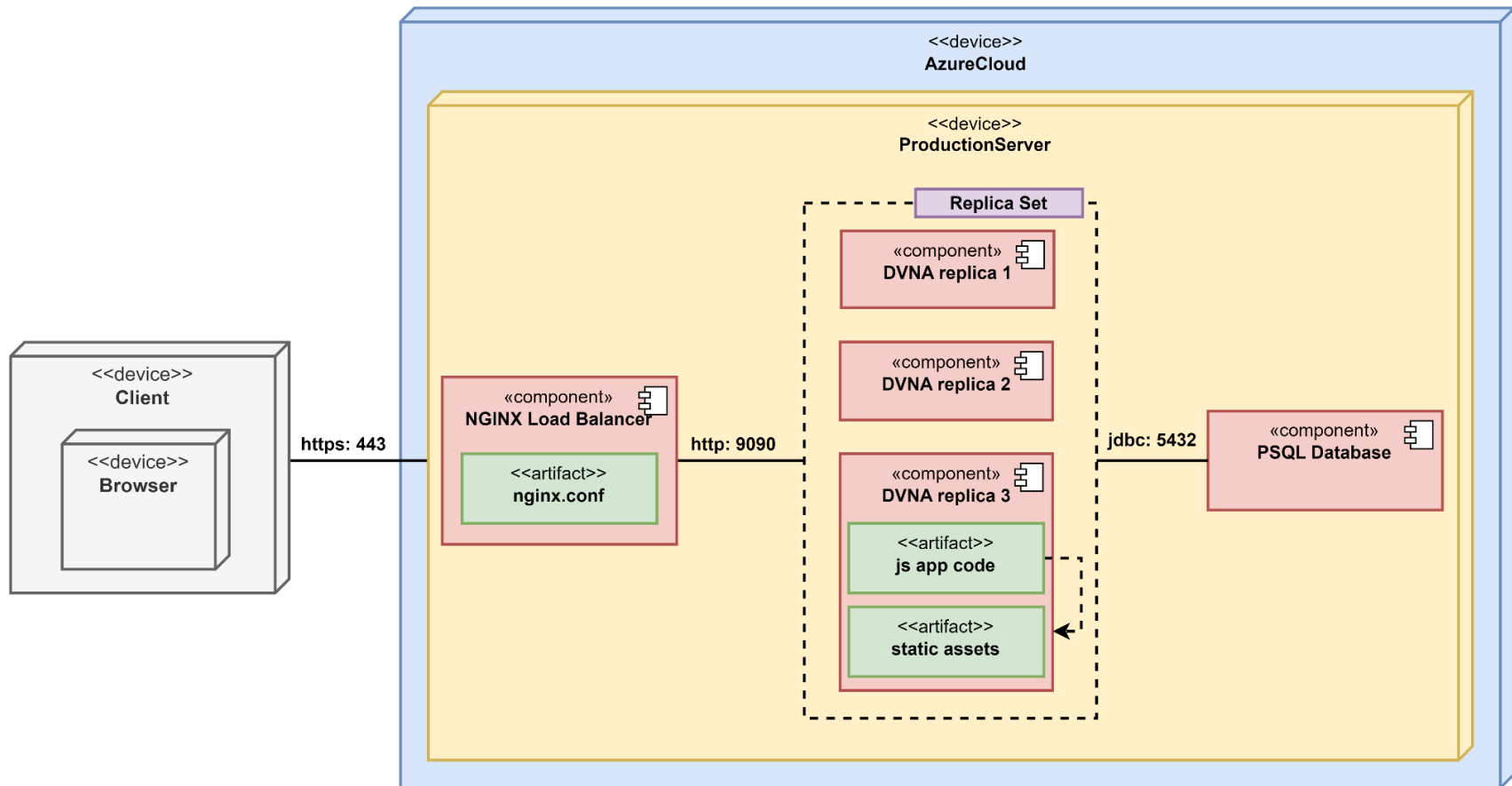


Fig 20. Diagramme de déploiement du projet.

Documentation

<https://alisandro104.github.io/LOG8100/>

Environnement de déploiement

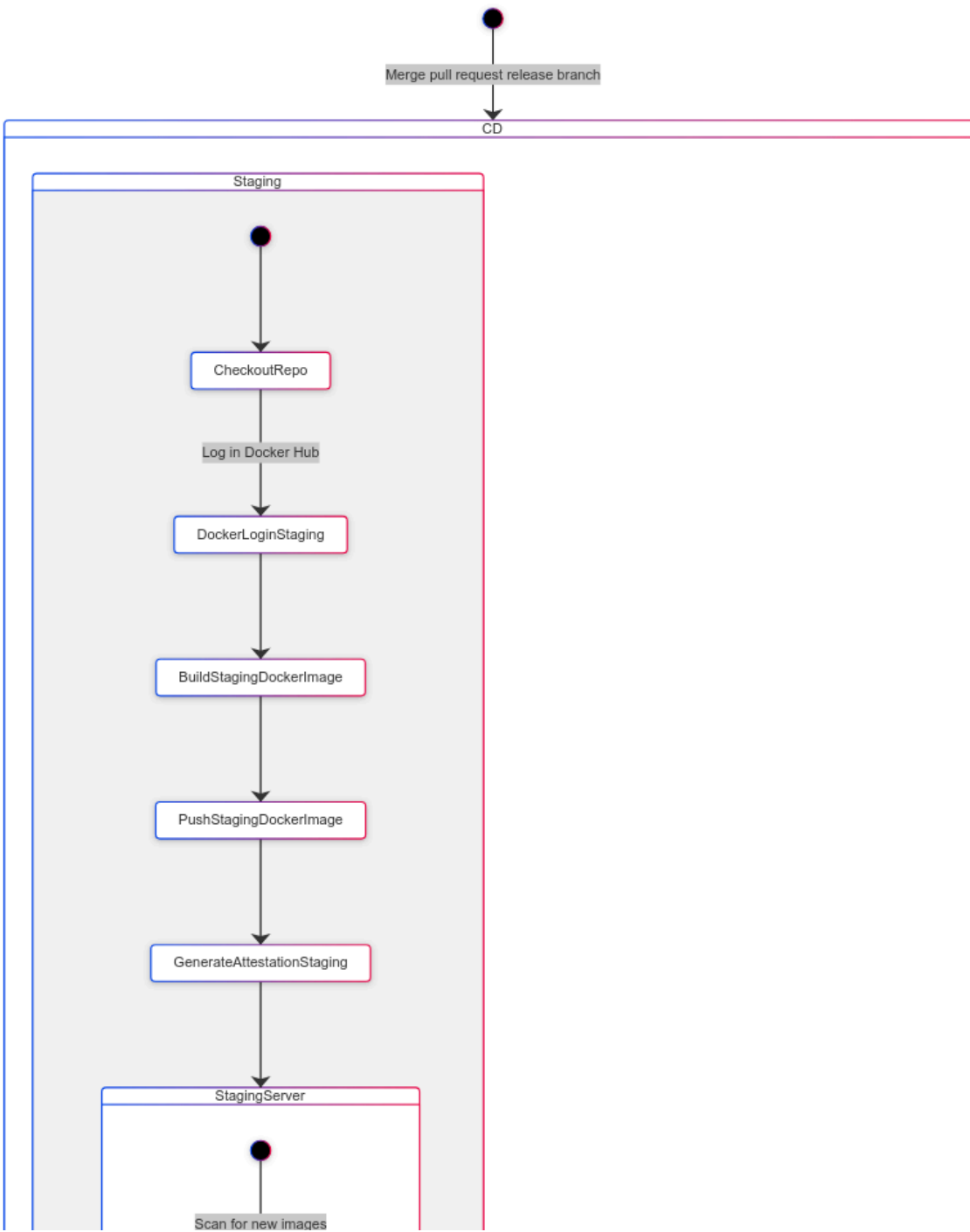
<https://dvna-team-1.canadacentral.cloudapp.azure.com>

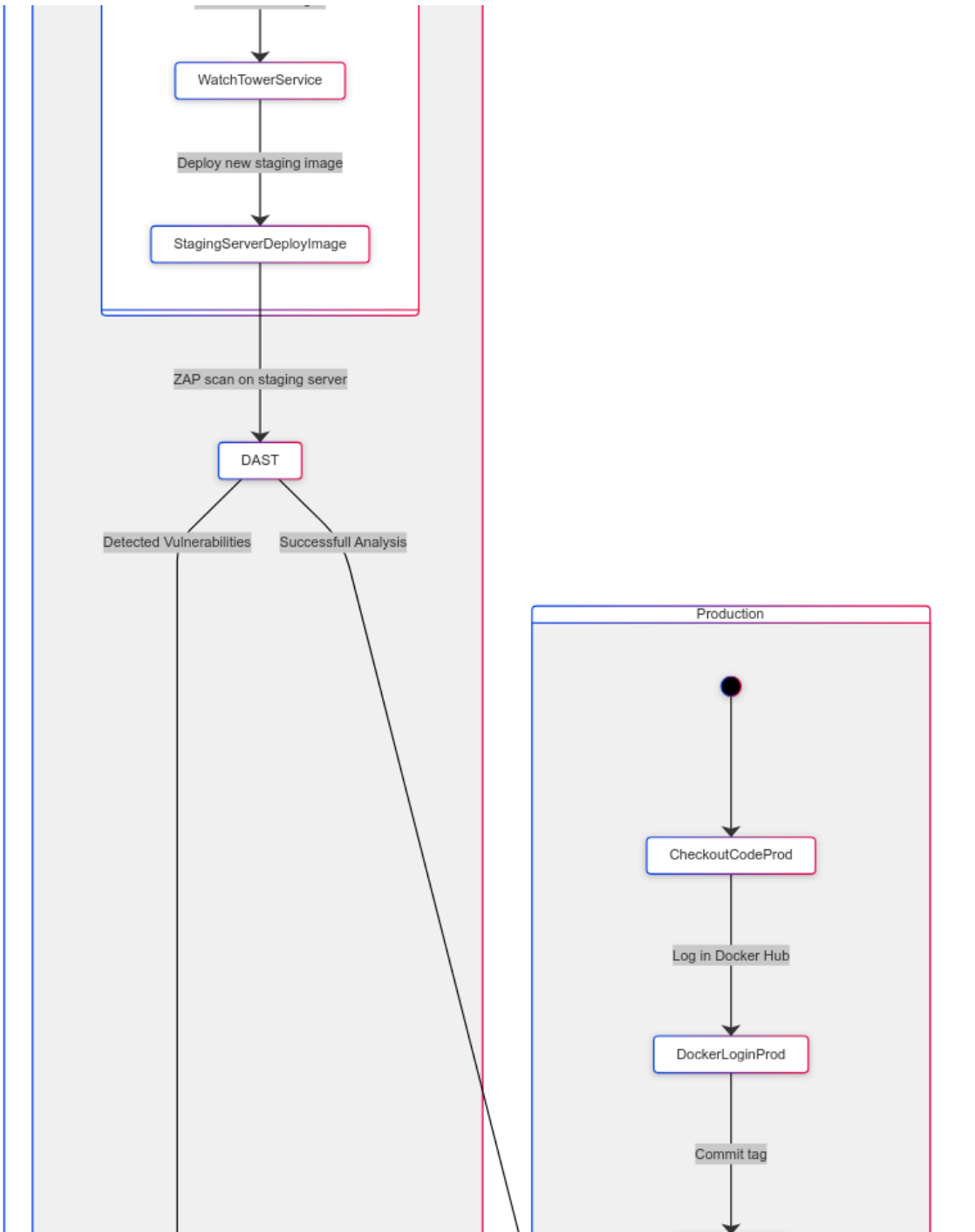
Conclusion

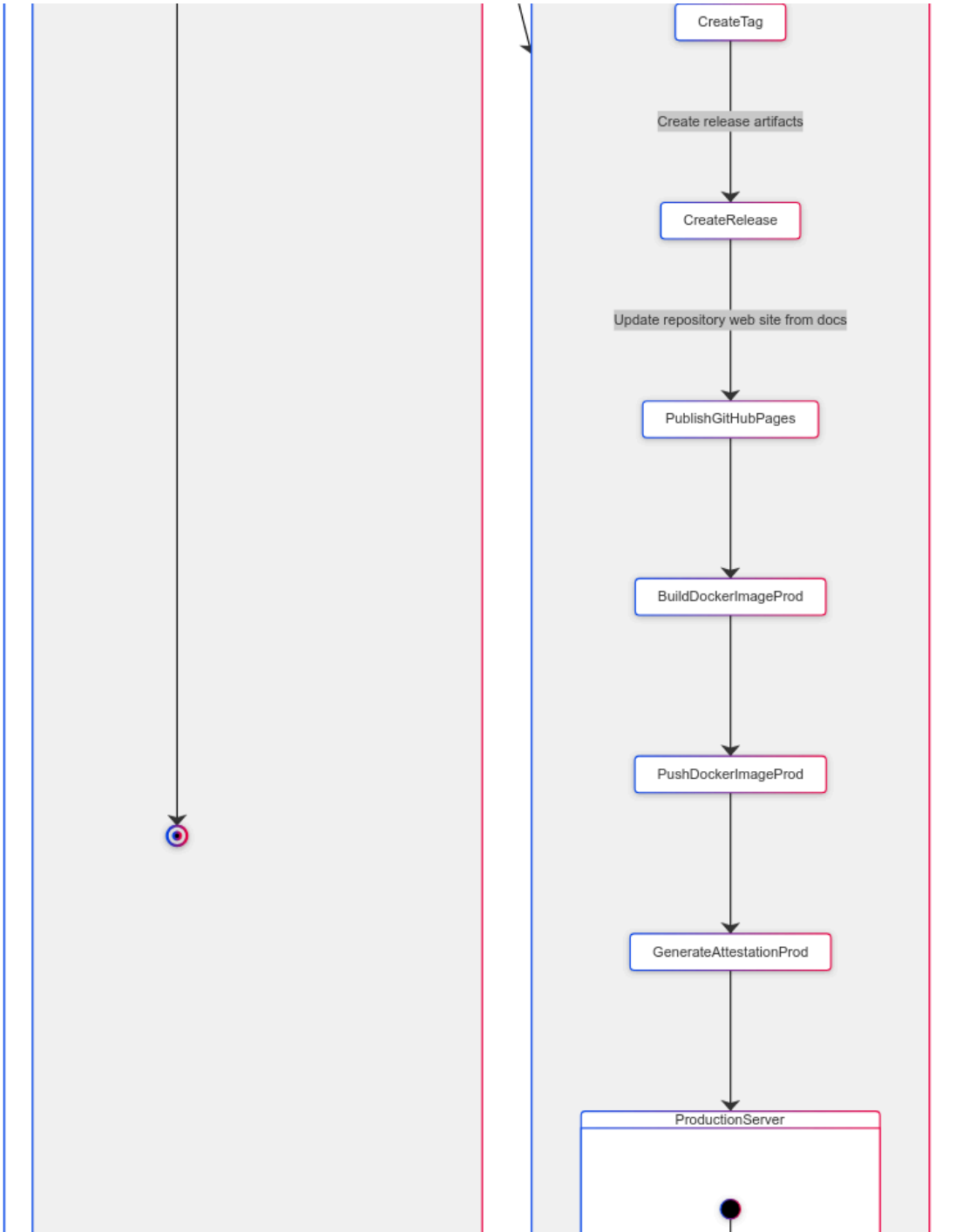
En résumé, nous avons apporté quelques modifications au code de l'application Damn Vulnerable NodeJS Application (dvna) afin de pouvoir l'exécuter dans un Docker container avant de le déployer dans le *Cloud* avec Azure. Nous avons ensuite identifié et corrigé plusieurs vulnérabilités à l'aide de tests de pénétration avec OWASP ZAP et Orchestron, puis intégré des outils d'automatisation tels que SonarLint, Dependabot et SonarQube dans la pipeline pour l'intégration et le déploiement continu.

Parmi les vulnérabilités ciblées dans notre rapport se trouvent des injections SQL, un design non sécurisé permettant d'accéder à des données d'authentification et des vulnérabilités XSS. Pour ces trois vulnérabilités, nous avons proposé des solutions à chacune afin d'augmenter la sécurité de l'application. Les solutions proposées se généralisent par l'ajout d'une étape de validation pour les entrées utilisateurs, la restriction de l'affichage de données sensibles et l'ajout de logique pour avoir du contrôle sur les sources d'exécution de code.

Annexe A







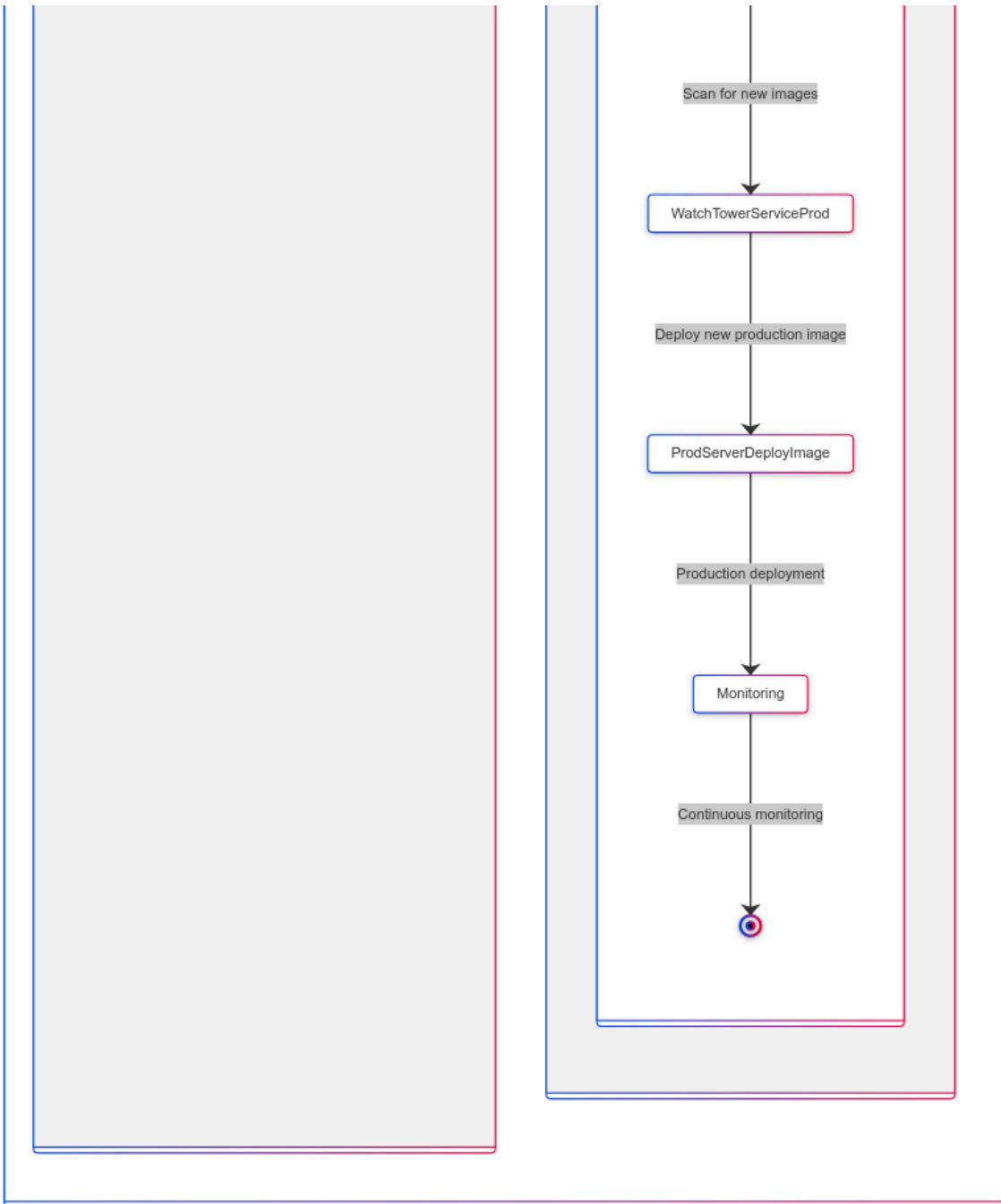


Fig 21. Diagramme d'état de la pipeline de déploiement contenu du projet.