Rapport de test de pénétration

Test de Sécurité

Table des matières

I. Analyse Applicatif 2

1. Analyse de la Base de Données MySQL 2

2. Analyse de la Faille Frontend 4

2.1 Création d'un Compte Utilisateur 4

2.2 Vulnérabilité des JSON Web Tokens (JWT) 4

2.3 Exploitation de la Vulnérabilité JWT 5

2.4 Attaque d'Usurpation de Privilèges 6

3. Obtention de la Base de Données 6

4. Obtention des données utilisateurs de la BDD 8

5. VM 8

6. Analyse Additionnelle 9

6.1 Utilisation de dirb 9

II. Analyse SAST 10

1. Frontend 11
2. Backend 13

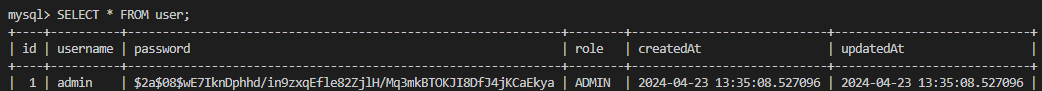
III. Mise en place d’une pipeline CI-CD 16

1. Configuration des repositories 17
2. Build des images Docker 18
3. Scan des images Docker 19
4. Déploiement des images Docker 20

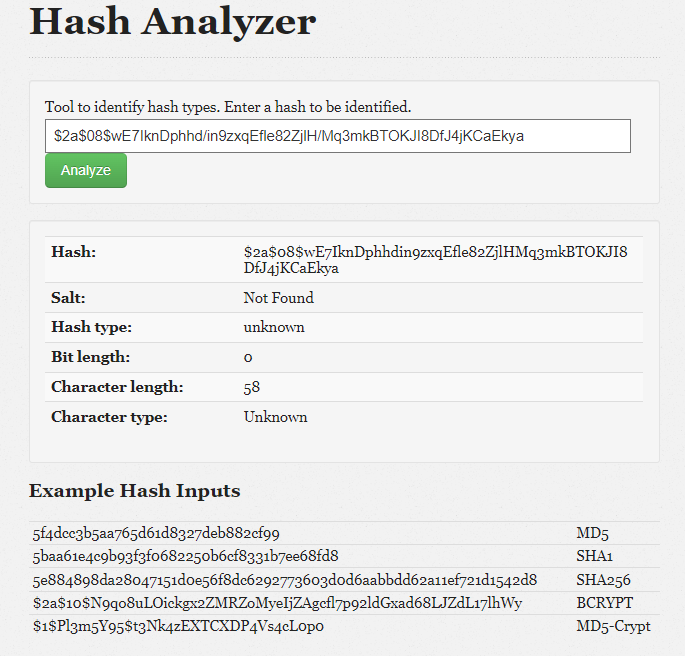
# I. Analyse Applicatif

## 1. Analyse de la Base de Données MySQL

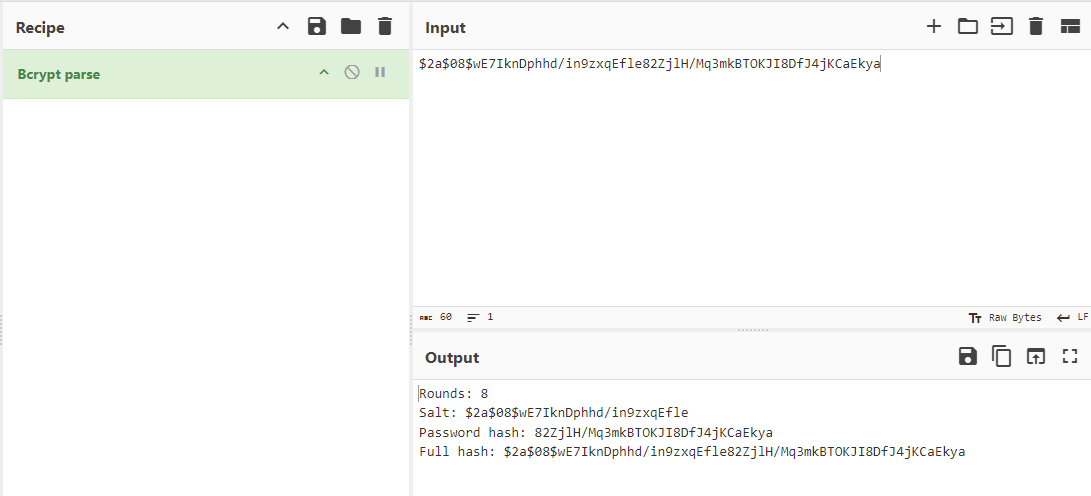
Nous avons lancé docker-compose pour déployer MySQL et accéder à la base de données fournie sur le dépôt Gandalf. En utilisant la commande **mysql -u root -p**, nous nous sommes connectés à la base **dev\_db**. En inspectant les données, nous avons remarqué que les mots de passe étaient hachés.



Pour identifier le type de hachage utilisé, nous avons constaté que le préfixe **$2a$** indique l'utilisation de l'algorithme bcrypt.



Nous avons ensuite utilisé CyberChef pour décoder le hachage et avons réussi à récupérer le mot de passe **admin** à l'aide de John the Ripper.



Une image contenant texte, capture d’écran, Police

Description générée automatiquement

## 2. Analyse de la Faille Frontend

### 2.1 Création d'un Compte Utilisateur

En examinant le frontend, nous avons identifié un formulaire d'inscription avec des champs pour le courriel et le mot de passe.

Une image contenant texte, capture d’écran, affichage, logiciel

Description générée automatiquement

Ainsi qu'un bouton pour créer un nouveau compte utilisateur. Nous avons créé un compte avec les identifiants **sauron:sauron**.

Une image contenant texte, capture d’écran, affichage, logiciel

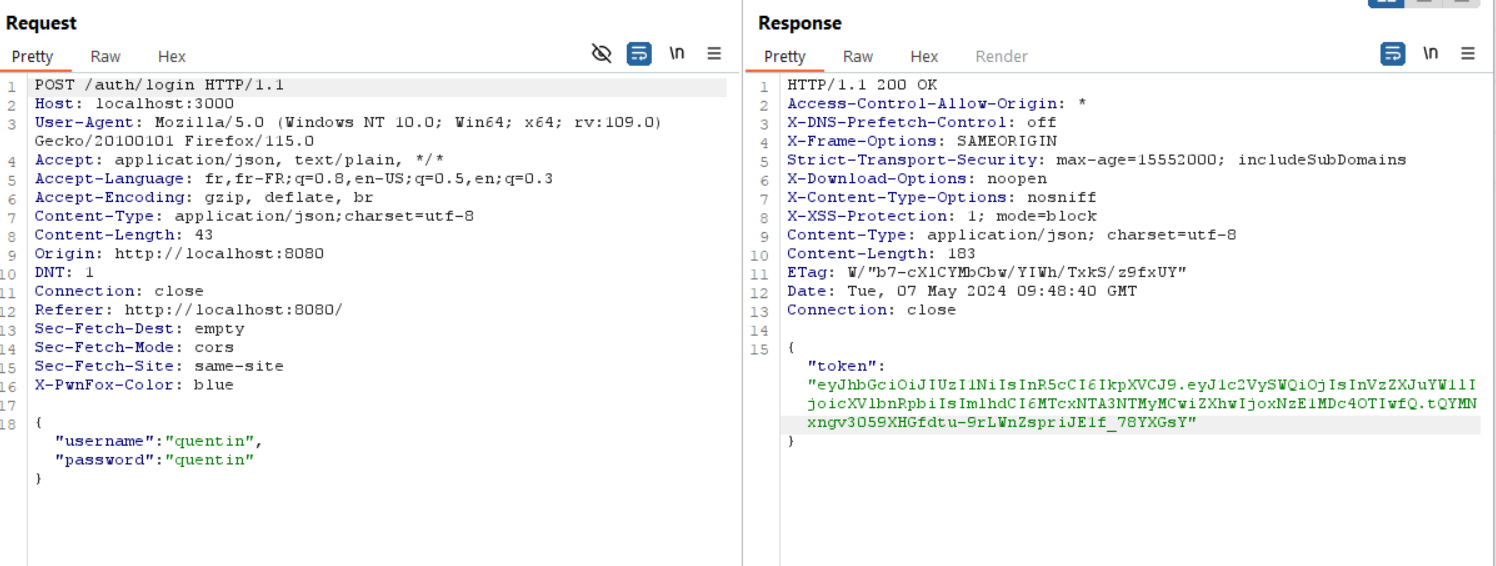
Description générée automatiquement

### 2.2 Vulnérabilité des JSON Web Tokens (JWT)

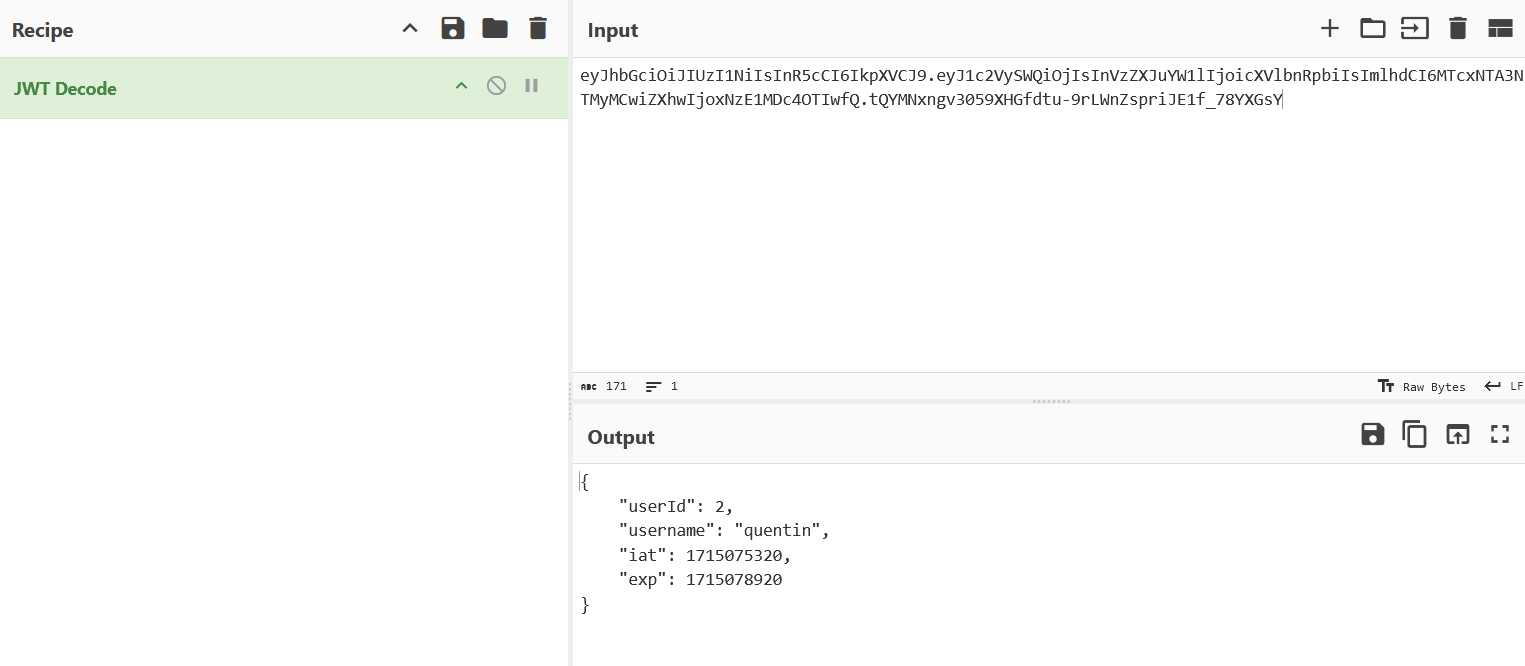
Il existe une faille de sécurité significative dans la gestion des JSON Web Tokens (JWT) due à l'utilisation d'une clé HMAC faible (**@QEGTUI**). Cette faiblesse expose les tokens JWT à des attaques de compromission et d'escalade de privilèges. Pour remédier à cette vulnérabilité, il est impératif de mettre en place une clé HMAC forte d'au moins 32 octets pour sécuriser les tokens.

### 2.3 Exploitation de la Vulnérabilité JWT

En utilisant Burp Suite pour intercepter les requêtes, nous avons remarqué la présence d'un token JWT lors de la requête de connexion.



En utilisant CyberChef, nous avons décrypté le token JWT et constaté qu'il n'était pas suffisamment sécurisé.



En modifiant les valeurs **userId** et **username** pour **1** et **admin** respectivement, nous avons recodé le token pour obtenir des privilèges administratifs.

### 2.4 Attaque d'Usurpation de Privilèges

En substituant le JWT token du compte utilisateur normal par celui créé avec le rôle administratif, nous avons pu accéder à des fonctionnalités administratives sur le site. Cela incluait la possibilité de modifier les rôles des utilisateurs et d'accéder à des pages restreintes.

Avant :

Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement

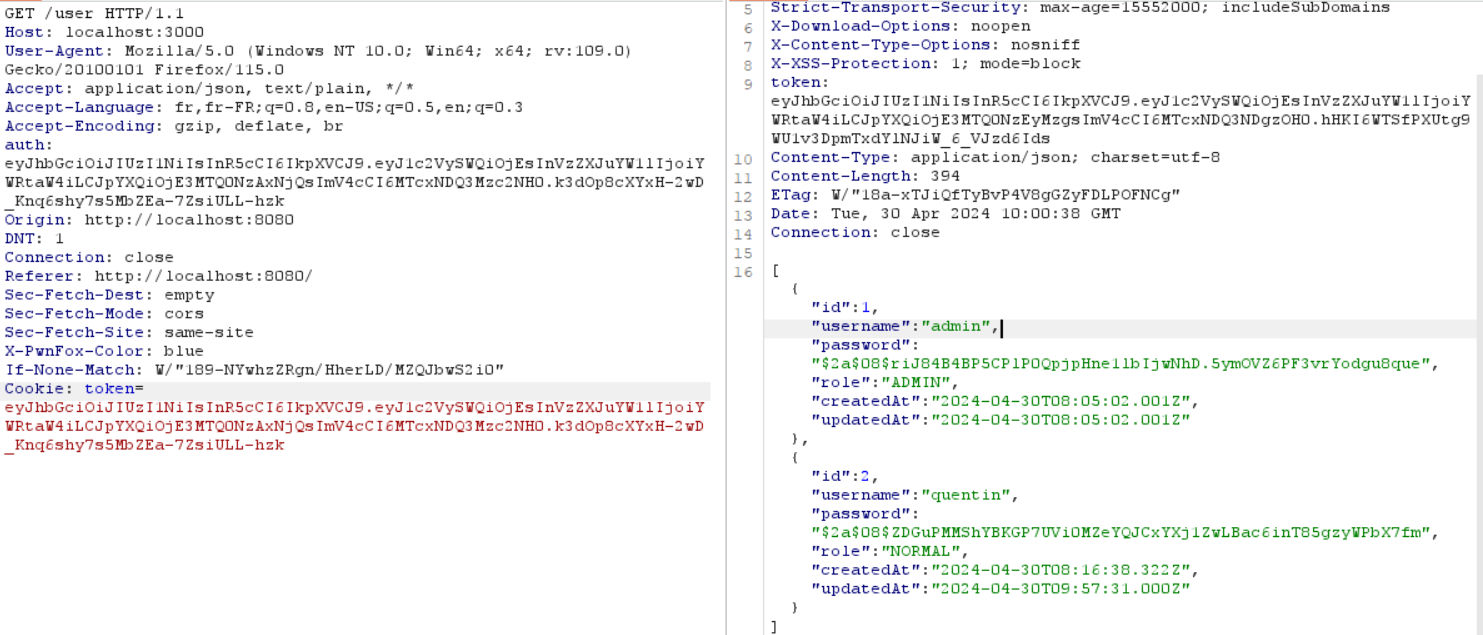
Après :

Une image contenant texte, Police, nombre, capture d’écran

Description générée automatiquement

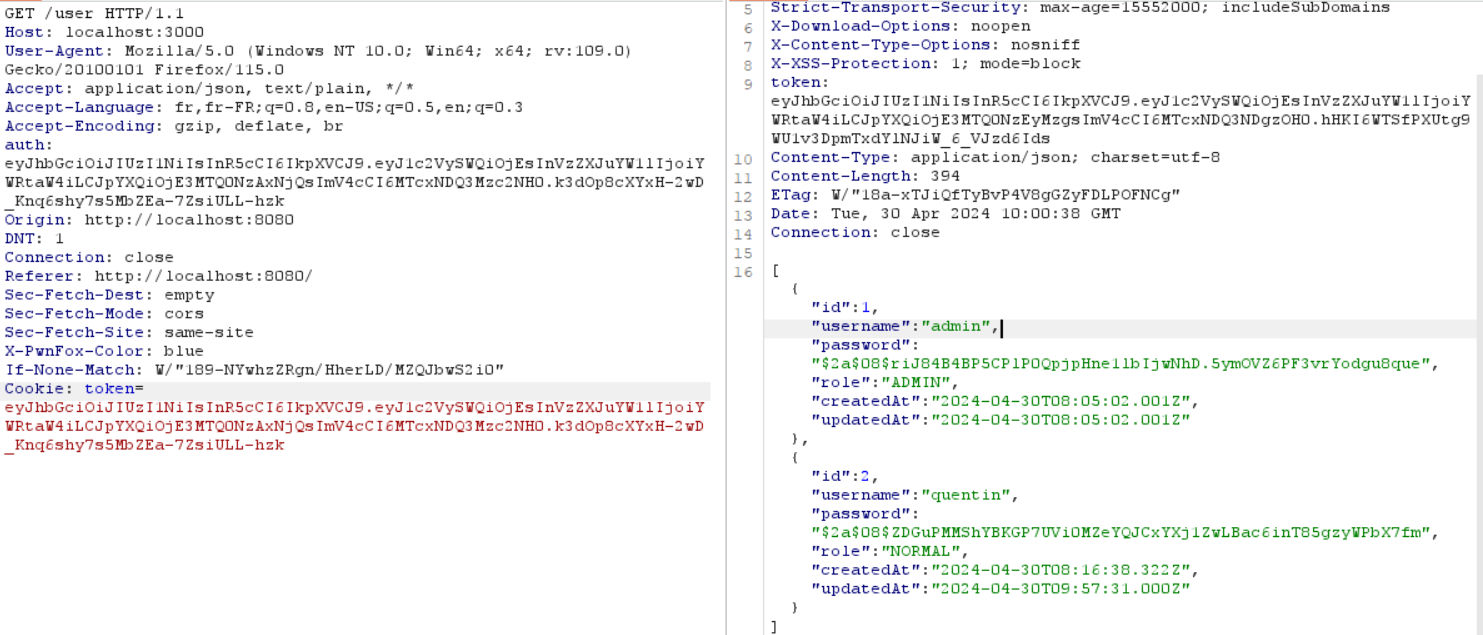
## 3. Obtention de la Base de Données

Grâce au token administratif obtenu, nous avons pu extraire l'intégralité de la base de données, ce qui soulève des préoccupations majeures en matière de sécurité.



## 4. Obtention des données utilisateurs de la BDD

Grâce au token administratif obtenu, nous avons pu extraire l'intégralité des données utilisateurs de la BDD ce qui soulève des préoccupations majeures en matière de sécurité.



## 5. VM

Pour trouver le mot de passe des machines virtuelles, nous avons entrepris une attaque par brute force manuelle en testant des combinaisons de mots de passe classiques. Nous avons commencé par essayer tout simplement :

admin:admin

root:root

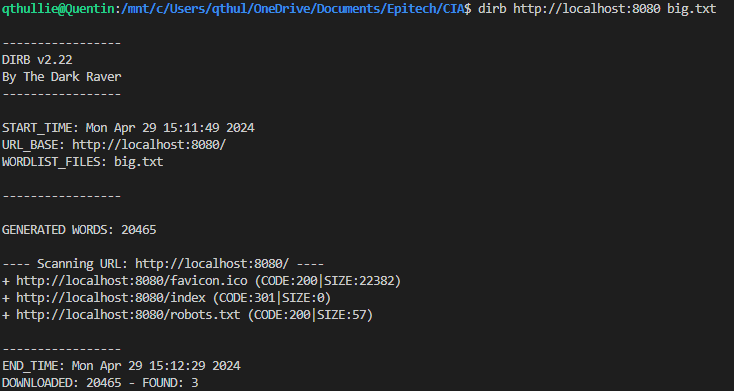
Finalement, nous avons réussi à nous authentifier en utilisant admin:root

Étant donné que nous avons trouvé le mot de passe correct à la troisième tentative, il n'a pas été nécessaire d'utiliser des outils de brute forces plus sophistiquées ou dirtycow. Cette méthode simple et rapide nous a permis de confirmer la vulnérabilité des mots de passe par défaut et de souligner l'importance de les changer pour des options plus sécurisées.

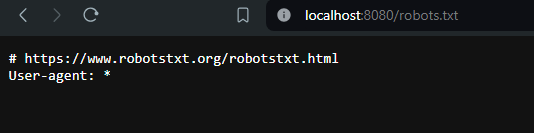
## 6. Analyse Additionnelle

### 6.1 Utilisation de dirb

Dans une tentative pour découvrir d'éventuels fichiers ou URLs cachés, nous avons utilisé **dirb** avec le dictionnaire **big.txt** sur le site web **http://localhost:8000**.



Malheureusement, cette méthode n'a pas révélé de fichiers intéressants, même le fichier **robots.txt** était vide.



# II. Analyse SAST

Semgrep et ZAP pour une Utilisation de SAST

Semgrep (Semantic Grep) et ZAP (OWASP Zed Attack Proxy) sont deux outils importants dans le domaine de la sécurité des applications, utilisés pour l'analyse statique de la sécurité des applications (SAST).

Semgrep est un outil de détection de vulnérabilités basé sur des règles, capable de scanner le code source pour identifier des motifs dangereux ou des failles de sécurité. Il est léger, rapide et supporte de nombreux langages de programmation. Semgrep permet de créer des règles personnalisées pour s'adapter aux besoins spécifiques d'une application.

ZAP, bien qu'initialement un outil de test de sécurité dynamique (DAST), inclut des fonctionnalités d'analyse statique. Il peut être utilisé pour scanner le code et les configurations de l'application à la recherche de vulnérabilités connues, aidant ainsi à renforcer la sécurité dès les premières phases du développement.

## Frontend

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Issue** | **Version** | **Patch** | **CVE** |
| Critical | Code Injection in fsevents | 1.2.9 | 1.2.11 | <https://www.cve.org/CVERecord?id=CVE-2023-45311> |

**Description :** Affected versions of fsevents are vulnerable to Improper Control Of Generation Of Code ('Code Injection'). fsevents depends on the <https://fsevents-binaries.s3-us-west-2.amazonaws.com/> URL. If an attacker at any point gains control of this URL, they can execute arbitrary code on any JavaScript project that depends on fsevents or uses code distributed from this URL.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Issue** | **Version** | **Patch** | **CVE** |
| High | Uncontrolled Resource Consumption in tmpl | 1.0.4 | 1.0.5 | <https://www.cve.org/CVERecord?id=CVE-2021-3777> |

**Description :** tmpl versions before 1.0.5 are vulnerable to Uncontrolled Resource Consumption when formatting a string.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Issue** | **Version** | **Patch** | **CVE** |
| High | Inefficient Regular Expression Complexity in terser | 4.3.4 | 4.8.1 | <https://www.cve.org/CVERecord?id=CVE-2022-25858> |

**Description :** terser versions before 4.8.1, >= 5.0.0 before 5.14.2 are vulnerable to Inefficient Regular Expression Complexity.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Issue** | **Version** | **Patch** | **CVE** |
| High | Improper Verification of Cryptographic Signature in browserify-sign | 4.0.4 | 4.2.2 | https://www.cve.org/CVERecord?id=CVE-2023-46234 |

**Description :** Affected versions of browserify-sign are vulnerable to Improper Verification Of Cryptographic Signature. The vulnerability lies in the checkValue function incorrectly verifying the upper bounds of the r and s components in a signature, enabling attackers to manipulate the s component by setting it to the prime number q, thereby simulating a zero value for s and potentially resulting in the unauthorized acceptance of maliciously signed messages during signature verification.

## Backend

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Issue** | **Version** | **Patch** | **CVE** |
| Critical | Cross-site Scripting in class-validator | 0.10.2 | 0.14.0 | <https://www.cve.org/CVERecord?id=CVE-2019-18413> |

**Description :** Affected version of class-validator is vulnerable to Improper Neutralization Of Input During Web Page Generation ('Cross-Site Scripting') / Improper Neutralization Of Special Elements Used In An Sql Command ('Sql Injection'). validate() input validation can be bypassed because certain internal attributes can be overwritten via a conflicting name. The optional forbidUnknownValues parameter can be used to reduce the risk of this validation bypass.

**Lines :**

Reachable via 4 usages (Collapse)

back\_student/src/controller/UserController.ts:90

back\_student/src/controller/UserController.ts:47

back\_student/src/controller/AuthController.ts:115

back\_student/src/controller/AuthController.ts:18

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Issue** | **Version** | **Patch** | **CVE** |
| Critical | Uncontrolled Resource Consumption in tmpl | 1.0.4 | 1.0.5 | <https://www.cve.org/CVERecord?id=CVE-2021-3777> |

**Description** : tmpl versions before 1.0.5 are vulnerable to Uncontrolled Resource Consumption when formatting a string.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Issue** | **Version** | **Patch** | **CVE** |
| Critical | Prototype Pollution in typeorm | 0.2.20 | 0.2.25 | <https://www.cve.org/CVERecord?id=CVE-2020-8158> |

**Description** : typeorm versions before 0.2.25 are vulnerable to Modification Of Assumed-Immutable Data (Maid) / Improperly Controlled Modification Of Object Prototype Attributes ('Prototype Pollution'). The vulnerability allows an attacker to save a specially crafted object to pollute the Object prototype and cause side effects on the library/application logic, such as denials of service attacks and/or SQL injections, by adding arbitrary properties to any object in the runtime.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Issue** | **Version** | **Patch** | **CVE** |
| Medium | Improper Authentication in jsonwebtoken | 8.5.1 | 9.0.0 | <https://www.cve.org/CVERecord?id=CVE-2022-23540> |

**Description** : Affected versions of jsonwebtoken are vulnerable to Improper Authentication / Use Of A Broken Or Risky Cryptographic Algorithm / Improper Verification Of Cryptographic Signature. The lack of algorithm definition and a falsy secret or key in the jwt.verify() function can lead to signature validation bypass due to defaulting to the none algorithm for signature verification. Users are affected only if ALL of the following conditions are true for the jwt.verify() function: 1. A token with no signature is received. 2. No algorithms are specified. 3. A falsy (e.g., null, false, undefined) secret or key is passed.

Lines: back\_student/src/middlewares/checkJwt.ts:17

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Issue** | **Version** | **Patch** | **CVE** |
| Medium | Use of a Broken or Risky Cryptographic Algorithm in jsonwebtoken | 8.5.1 | 9.0.0 | <https://www.cve.org/CVERecord?id=CVE-2022-23539> |

**Description** : Affected versions of jsonwebtoken are vulnerable to Use Of A Broken Or Risky Cryptographic Algorithm. The library could be misconfigured so that legacy, insecure key types are used for signature verification. For example, DSA keys could be used with the RS256 algorithm.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Issue** | **Version** | **Patch** | **CVE** |
| Medium | Improper Authentication in jsonwebtoken | 8.5.1 | 9.0.0 | <https://www.cve.org/CVERecord?id=CVE-2022-23541> |

**Description** : Affected versions of jsonwebtoken are vulnerable to Improper Authentication. Insecure implementation of key retrieval function could lead to Forgeable Public/Private Tokens from RSA to HMAC. Reachable if you are using a poorly implemented key retrieval function and your application is supporting usage of both symmetric key and asymmetric key in jwt.verify() implementation with the same key retrieval function.

# III. Mise en place d’une pipeline CI-CD

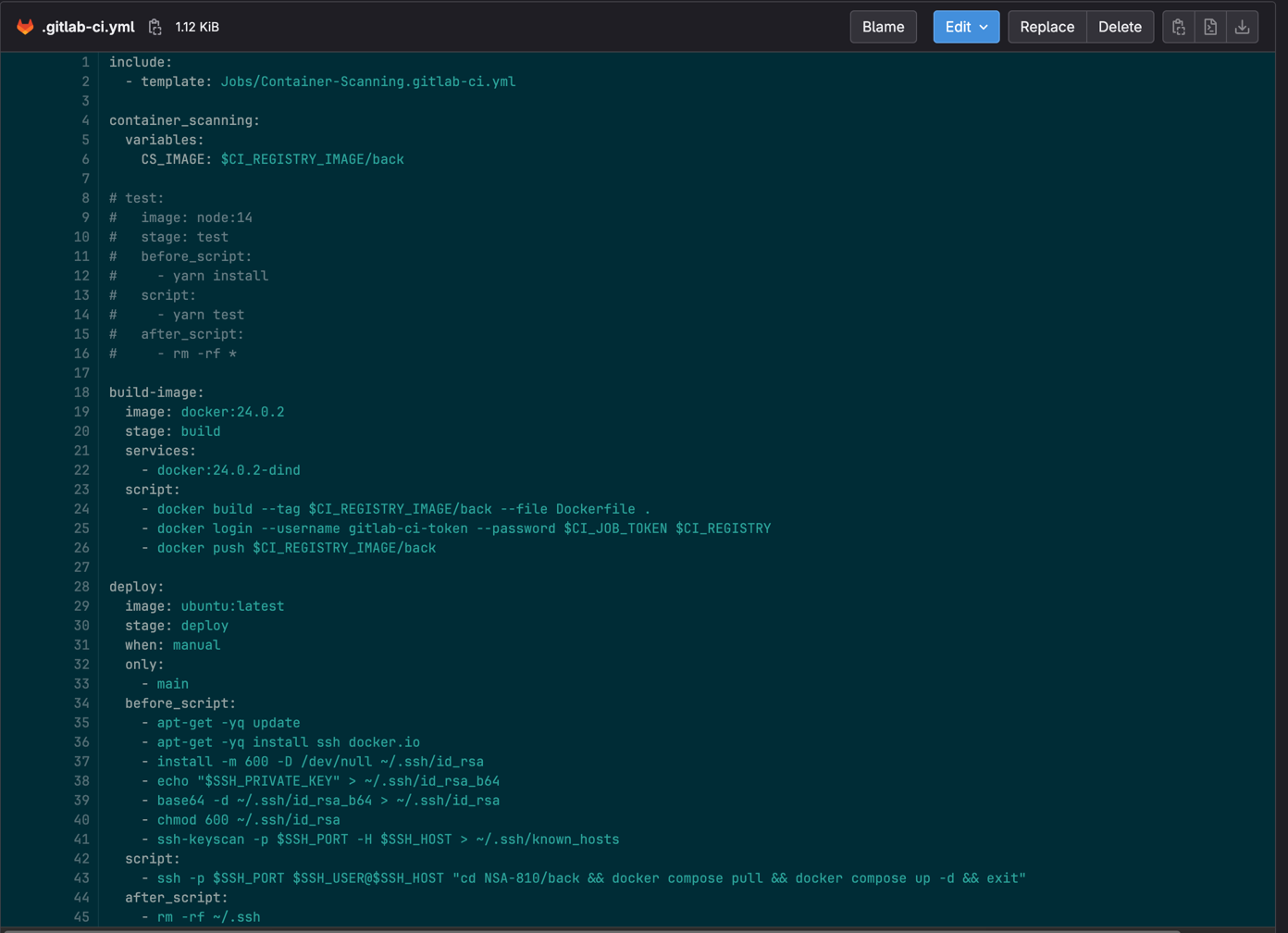
Nous avons décidé d’utiliser [Gitlab](https://gitlab.com/nsa-8101) pour la partie intégration et déploiement continue. L’objectif est de créer des images Docker pour le backend ainsi que pour le frontend qui sont par la suite analysé par l’outil Scanning-Container fournit sur Gitlab pour ensuite être déployer sur un serveur. Ainsi, un groupe nommé nsa-810 a été créé ainsi que deux repository nommé Front et Back.

Une image contenant texte, Logiciel multimédia, logiciel, capture d’écran

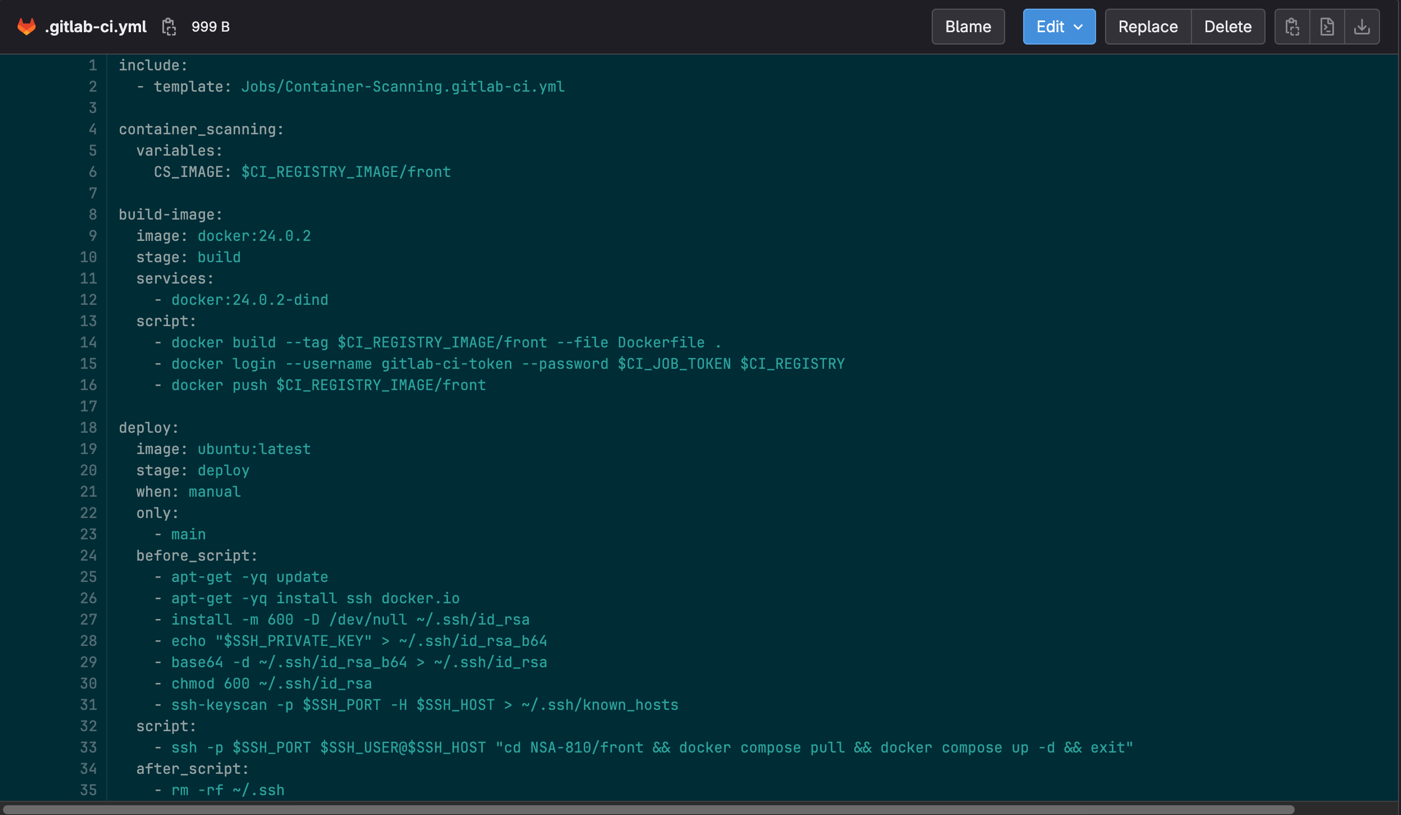
Description générée automatiquement

## Configuration des repositories

Dans chacun des repository, nous avons créé une fichier .gitlab-ci.yml, afin de lancer les pipelines pour build les images Docker afin de scanner ces dernières puis de lancer le déploiement sur le serveur.

Fichier .gitlab-ci.yml du back : 

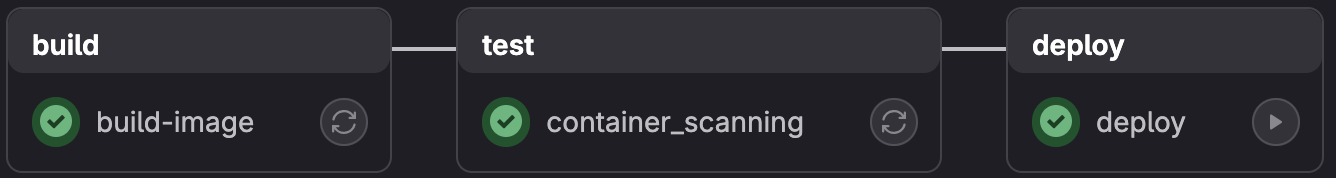
Fichier .gitlab-ci.yml du front :



## Build des images Docker

Lorsqu’un push est effectué, une build est déclenchée pour chaque repository. Voici les étapes détaillées du processus de build :

* **Construction de l'Image Docker** : Utilisation des Dockerfiles présents dans chaque repository pour créer les images Docker. Par exemple :
  + Pour le frontend : docker build -t registry.gitlab.com/nsa-810/front:latest .
  + Pour le backend : docker build -t registry.gitlab.com/nsa-810/back:latest .



## Scan des images Docker

Si on entre dans le détail du scan, on peut voir les différentes failles qui ont été trouvé :

* Exemple pour le front :

Une image contenant texte, capture d’écran, menu

Description générée automatiquement

* Exemple pour le back :

Une image contenant texte, capture d’écran

Description générée automatiquement

Ces résultats sont également disponibles en fichiers JSON dans les artifacts :

Une image contenant capture d’écran, texte, noir

Description générée automatiquement

## Déploiement des images Docker

Pour la suite de ce pipeline, nous avons décidé de réaliser le déploiement via GitLab de manière manuelle. Cela signifie que pour que la tâche de déploiement soit effectuée, il faut la valider une fois que la tâche de scan est terminée.

* **Validation Manuelle** : Une fois le scan terminé et les résultats analysés, nous procédons à la validation manuelle du déploiement. Cette étape permet de s'assurer que seules les images saines et sécurisées sont mises en production.

Une image contenant capture d’écran, texte, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, logo, conception

Description générée automatiquement

* **Déploiement sur le Serveur** : Une fois validée, l’image Docker est déployée sur le serveur cible.

Une image contenant texte, capture d’écran

Description générée automatiquement

Une fois le déploiement réaliser, le site est accessible depuis le serveur sur lequel l’image docker a été déployer.