



TD: ANALYSE STATIQUE DE CODE

Le code [StanleyDINNE/SAST_tp](#) est sur GitHub.

Sommaire

1 BANDIT	2
1.1 Installation	2
1.2 Analyse de cinq scripts Python	2
2 SEMGREP	4
2.1 Premier usage de SEMGREP sur des exemples	4
2.1.1 Installation et configuration de SEMGREP	4
2.1.2 Dossier 1/	5
2.1.2.1 Problèmes détectés	5
2.1.2.2 Corrections	6
2.1.3 Dossier 2/	7
2.1.3.1 Problèmes détectés	7
2.1.3.2 Corrections	7
2.1.3.3 Correction alternative pour sanitiser le direct-input	8
2.1.4 Dossier 3/	9
2.1.4.1 Problèmes détectés	9
2.1.4.2 Corrections	9
2.1.5 Dossier 4/	10
2.1.5.1 Problèmes détectés	10
2.1.5.2 Corrections	10
2.2 Audit d'une application complète avec SEMGREP	11
2.2.1 Réinstallation dans un environnement propre	11
2.2.2 Analyse et correction des problèmes trouvés	11
2.3 Création d'application avec vulnérabilité non détectée par SEMGREP	15
2.3.1 Premières idées	15
2.3.2 Application vulnérable	15
2.3.3 Conclusion	16

1 – BANDIT

1.1 – Installation

```
cd ex_1 && python3 -m venv .venv && source ./venv/bin/activate
python3 -m pip install bandit
python3 -m pip freeze > requirements.txt
bandit --version # bandit 1.7.7: python version = 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0]
bandit-config-generator -o bandit.cfg.yml
```

Code 1. – Installation de BANDIT

1.2 – Analyse de cinq scripts Python

```
mkdir -p reports
bandit --recursive . --exclude ./venv/ --format html --output ./reports/standard.html
bandit --recursive . --exclude ./venv/ --format txt --output ./reports/standard.txt
```

Code 2. – Analyse sans configuration spécifique des scripts Python dans [ex_1/](#)

Voici les résultats d'une analyse brute des cinq fichiers [1.py](#), [2.py](#), [3.py](#), [4.py](#) et [5.py](#), sans config.

Fichier	CWE	Sévérité	Occurrences	Signification de la CWE
1.py	CWE-89	MEDIUM	11 lignes	Improper Neutralization of Special Elements used in an SQL Command (< SQL Injection >)
2.py	CWE-78	MEDIUM	une fois	Improper Neutralization of Special Elements used in an OS Command (< OS Command Injection >)
3.py	CWE-259	LOW	14 lignes	Use of Hard-coded Password
4.py	CWE-22	MEDIUM	8 lignes	Improper Limitation of a Pathname to a Restricted Directory (< Path Traversal >)
5.py	CWE-89	MEDIUM	20 lignes	Improper Neutralization of Special Elements used in an SQL Command (< SQL Injection >)

En relançant l'analyse avec le fichier configuration par défaut (donc non configuré), la commande renvoie un code d'erreur (1), et on obtient le même rapport que sans configuration (surement un fallback sur une recherche de toutes les vulnérabilités). Si par contre on active un test pour une vulnérabilité qui n'apparaît pas dans le code (par exemple B103 : « *set bad file permissions* ») en le décommentant parmi les options listées et en l'ajoutant aux tests (sous tests), on obtient bien un rapport d'erreur vide

```
bandit -r . -x ./venv/ -f html -o ./reports/specific.html --configfile bandit.cfg.yml
bandit -r . -x ./venv/ -f txt -o ./reports/specific.txt --configfile bandit.cfg.yml
```

Code 3. – Analyse avec configuration des scripts Python dans [ex_1/](#)

Dans le fichier de configuration, voici la correspondance entre les CWE et les tests bandit

- B610, B608 (*django_extra_used*, *hardcoded_sql_expressions*) pour la CWE-89 (resp. [1.py](#), et [5.py](#))
- B102 (*exec_used*) pour la CWE-78
- B105, B106, B107 (*[hardcoded_password_]..string*, *..funcarg*, *..default*) pour la CWE-259
- B310 (*urllib_urlopen*) pour la CWE-22

On a donc décommenté tous les tests, et on a utilisé ceux-ci :

`tests:` [B608, B610, B102, B105, B106, B107, B310]

En relançant l'analyse, on obtient bien les mêmes résultats que lors des premières analyses sans configuration

2 — SEMGREP

2.1 — Premier usage de SEMGREP sur des exemples

2.1.1 — Installation et configuration de SEMGREP

Depuis l'exercice précédent (Chapitre 1), on change d'environnement :

```
deactivate # Depuis l'autre Virtual Environment pour l'exercice 1
cd ../ex_2 && python3 -m venv .venv && source ../venv/bin/activate
python3 -m pip install semgrep
python3 -m pip freeze > requirements.txt
semgrep --version # 1.60.1
```

Code 4. – Installation de SEMGREP

`semgrep scan --config auto` avait donné « *Missed out on 656 pro rules since you aren't logged in!* », donc nous nous sommes résolus à nous créer un compte et nous connecter avec `semgrep login`.

Suite au login depuis le terminal :

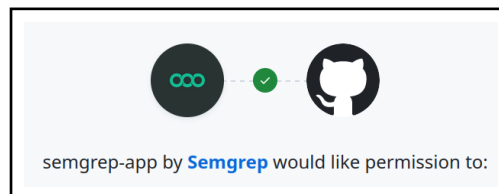


Fig. 1. – Liaison de Semgrep avec GitHub

1. Liaison avec *GitHub*
2. Création d'une organisation « `security_3.0_static_analysis` »
3. Token de connexion est renvoyé dans le terminal

Nous allons analyser le code localement, tout en étant connecté à Semgrep depuis le terminal.

Pour chacun des dossiers `1/`, `2/`, `3/` et `4/`, nous allons :

1. Réaliser un scan complet avec SEMGREP
2. Identifier les vulnérabilités de sévérité HIGH
3. Corriger ces vulnérabilités
4. Vérifier que ces vulnérabilités n'apparaissent plus en refaisant un scan

2.1.2 – Dossier 1/

cd 1 && semgrep ci

2.1.2.1 – Problèmes détectés

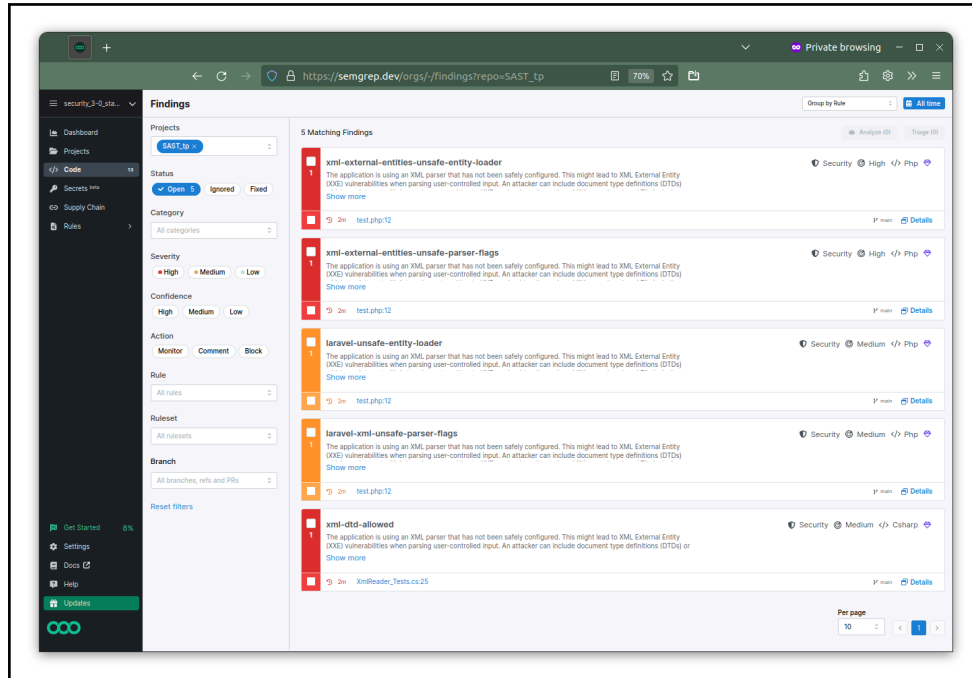


Fig. 2. – Problèmes trouvés par Semgrep dans le répertoire 1

- `XmlReader_Tests.cs`
 - 25| `XmlReader reader = XmlReader.Create(stream, settings);`
 - `csharp.dotnet-core.xxe.xml-dtd-allowed.xml-dtd-allowed`
- `test.php`
 - 12| `$document->loadXML($xml, LIBXML_NOENT | LIBXML_DTDLOAD);`
 - `php.lang.security.xml-external-entities-unsafe-entity-loader.xml-external-entities-unsafe-entity-loader`
 - `php.lang.security.xml-external-entities-unsafe-parser-flags.xml-external-entities-unsafe-parser-flags`
 - `php.laravel.security.laravel-unsafe-entity-loader.laravel-unsafe-entity-loader`
 - `php.laravel.security.laravel-xml-unsafe-parser-flags.laravel-xml-unsafe-parser-flags`

On va essayer de corriger toutes les vulnérabilités trouvées par ce scan par défaut, c'est-à-dire Medium et High.

Il se trouve que si on se rend sur les détails d'une vulnérabilité dans l'interface Semgrep, et qu'on clique sur le bouton ci-dessous, on peut trouver des exemples de solutions.

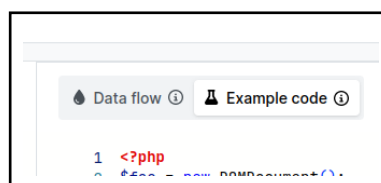


Fig. 3. – Bouton d'exemples de solutions à appliquer

2.1.2.2 – Corrections

Pour `XmlReader_Tests.cs`, la configuration de Semgrep est telle qu'elle reconnaît :

	source	sink
1	<code>\$X.DtdProcessing = DtdProcessing.Parse</code>	<code>XmlReader.Create(\$C, \$SETTINGS, ...)</code>
2	<code>\$X.XmlResolver = new XmlUrlResolver()</code>	<code>(XmlTextReader \$R).\$READ(...)</code>

L'idée serait de désactiver le traitement DTD et le XMLResolver ainsi :

```
settings.DtdProcessing = DtdProcessing.Prohibit; // Désactiver le traitement DTD
settings.XmlResolver = null; // Désactiver XmlResolver
...
xmlDocument.XmlResolver = null; // Ici aussi
```

Code 5. – Correction pour `XmlReader_Tests.cs`

Pour `test.php`, la vulnérabilité XXE vient de l'utilisation de `libxml_disable_entity_loader(false)`. Pour corriger le problème, on peut modifier `libxml_disable_entity_loader(true)`, et ajouter `libxml_set_external_entity_loader(static function () { return null; });`.

Bizarrement, `test2.php` n'a pas été détecté, mais il faut aussi updaté de la même façon que pour `test.php`.

En relançant le scan, on obtient un écran satisfaisant sans vulnérabilité High ni Medium trouvée :

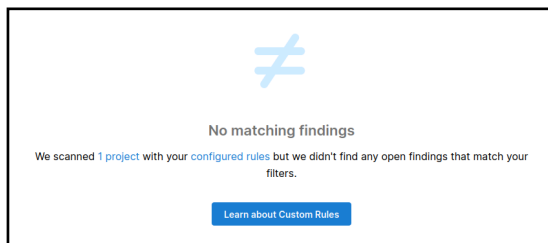


Fig. 4. – Toutes les vulnérabilités précédemment trouvées ont été corrigées

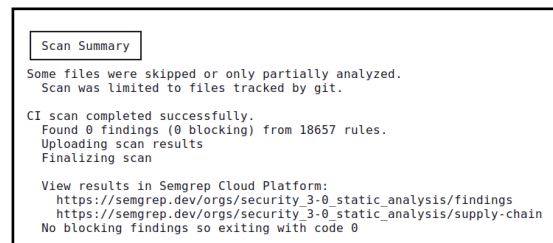


Fig. 5. – Récapitulatif de Semgrep en ligne de commande qui indique que tout a été corrigé

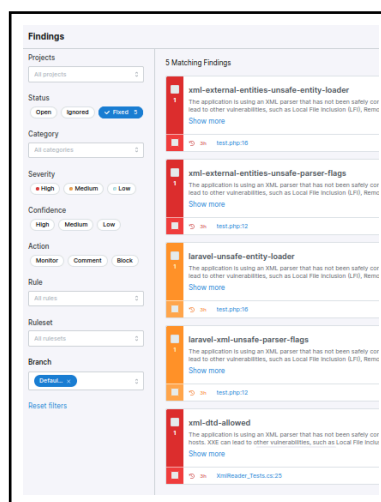


Fig. 6. – Interface Web de Semgrep avec le rappel des vulnérabilités, dorénavant corrigées

2.1.3 – Dossier 2/

```
cd ../2 && semgrep ci
```

2.1.3.1 – Problèmes détectés

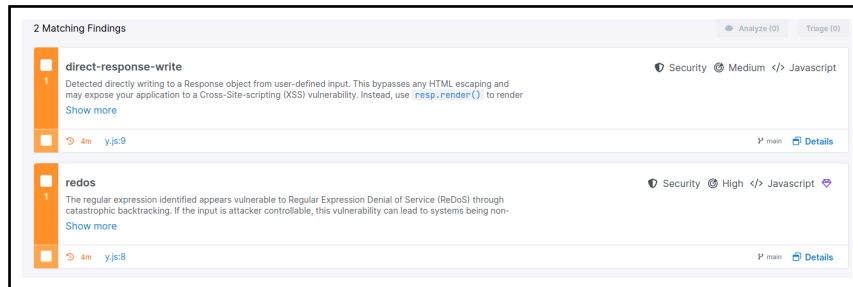


Fig. 7. – Toujours plus de vulnérabilités dans un si petit script

2.1.3.2 – Corrections

On peut sanitiser l'input avec les solutions données dans la page d'aide, à savoir importer le module xss :

```
var xss = require("xss");
...
res.send('<h1> Hello :' + xss(a) + "</h1>");
```

Code 6. – Sanitizer HTML proposé par Semgrep

Pour protéger contre ReDoS (Regular Expression Denial of Service), il faut simplifier le pattern de la regex, pour éviter que la recherche soit excessivement longue si l'input est conçu pour faire câbler une Regex. Dans notre cas, on peut utiliser `var r = /^[a-z]+$/;` à la place de `var r = /^[a-z]+$/;`.

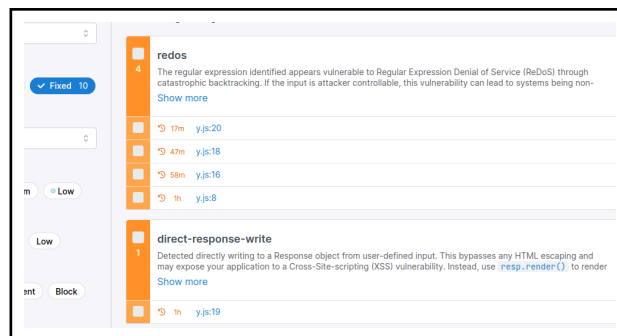


Fig. 8. – Les résultats qui font plaisir

2.1.3.3 – Correction alternative pour sanitiser le direct-input

Ou alors on peut créer notre propre sanitizer, en utilisant une regex qui va attraper tous les caractères suspects et renvoyer leur équivalent sanitisé depuis un mapping issu d'un dictionnaire :

```
const escapeHTML = str => str.replace(/&<>'"/g, tag => ({  
  '&': '&amp;',  
  '<': '&lt;',  
  '>': '&gt;',  
  '"': '&#39;',  
  "'": '&quot;',  
}[tag]));
```

Code 7. – Sanitizer HTML fait-maison

Mais si on fait ça, il faut l'ajouter dans les règles de Semgrep, pour qu'il le reconnaisse comme un sanitizer avec quelque chose comme

```
pattern-sanitizers:  
- patterns:  
  - pattern-either:  
    - pattern-inside: |  
      const escapeHTML = $P => $P.replace(/&<>'"/g, $I => ({  
        '&': '&amp;',  
        '<': '&lt;',  
        '>': '&gt;',  
        '"': '&#39;',  
        "'": '&quot;',  
      }[$I]));  
      ...  
    - pattern: escapeHTML(...)
```

Code 8. – Règles à ajouter pour Semgrep

Mais c'était un peu trop long à setup dans l'interface en ligne, alors on a eu recours aux propositions, telles que `require('xss')`.

2.1.4 – Dossier 3/

```
cd ../3 && semgrep ci
```

2.1.4.1 – Problèmes détectés

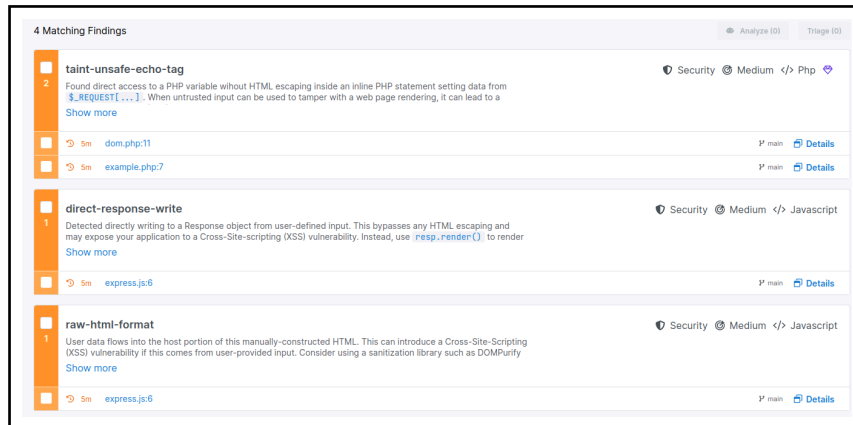


Fig. 9. – Vulnérabilités trouvées dans le dossier 3

2.1.4.2 – Corrections

Pour raw-html-format, il ne faut pas renvoyer de html directement : `res.send('Hello : ' + name);` ; plutôt que `res.send('<h1> Hello : ' + name + '</h1>')`, ou alors utiliser DOMPurify.

Pour direct-response-write, on peut sanitiser avec `const xss = require("xss");` puis `xss(name)` comme l'exercice précédent (Chapitre 2.1.3).

Pour taint-unsafe-echo-tag, on peut utiliser `htmlspecialchars` ou `htmlspecialchars` de PHP pour sanitiser. Le seul endroit où ça ne fonctionnerait pas, ce serait si le php était invoqué dans un contexte `<script> </script>` dans lequel on pourrait appeler des fonctions et ainsi invoquer du code sans avoir besoin de caractères tels que `"`, `'`, `<`, `>`, etc. Mais là ce n'est que du php donc ça va, `htmlspecialchars` fera l'affaire.

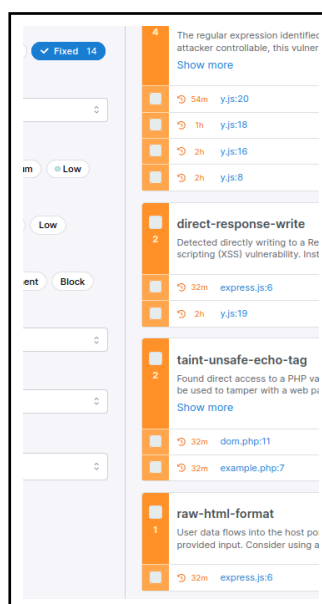


Fig. 10. – Alerts gone..

2.1.5 – Dossier 4/

```
cd ../4 && semgrep ci
```

2.1.5.1 – Problèmes détectés

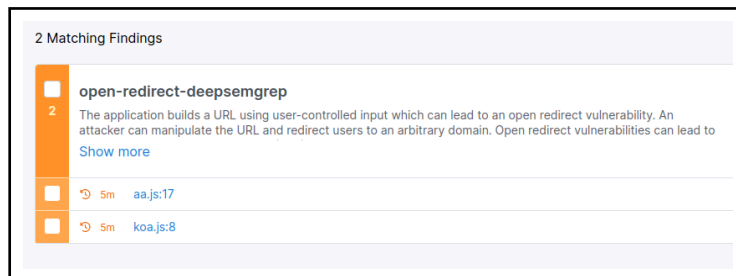


Fig. 11. – Vulnérabilités sur le répertoire 4

2.1.5.2 – Corrections

Il faut éviter de rediriger vers un domaine géré par l'input, donc préfixer par un domaine de confiance. Dans notre cas, nous ne savons pas quel domaine mettre pour cette appli', donc nous allons mettre example.com, dans `koa.js`. Pour la redirection dans `aa.js`, on peut renvoyer une page pour demander à l'utilisateur s'il veut bien être redirigé vers une autre page (Cf `aa.js`).

Ça a introduit une vulnérabilité medium (javascript.lang.security.audit.unknown-value-with-script-tag.unknown-value-with-script-tag) mais avec une faible confiance donc on va dire qu'on va l'ignorer car on a déjà sanitisé l'input.

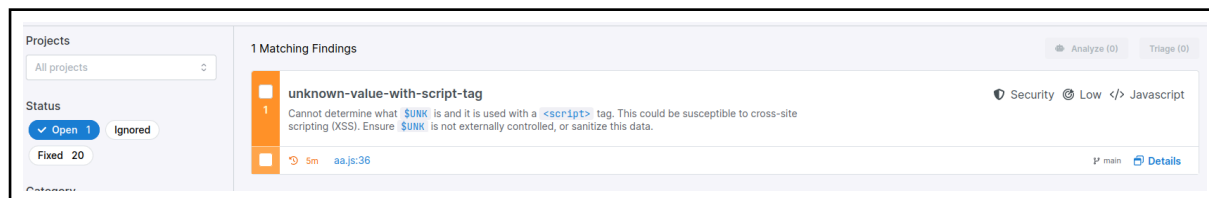


Fig. 12. – Après quelques essais-erreurs, on arrive à une correction satisfaisante

Rules summary			
<div> <div>Most fired</div> <div>Most ignored</div> <div>Most fixed</div> </div>			
Rule	Ignored	Fix rate	Total
redos	0%	100%	4
direct-response-write	0%	100%	3
open-redirect-deepsemgrep	0%	100%	3
unknown-value-with-script-tag	0%	50%	2
raw-html-format	0%	100%	2
taint-unsafe-echo-tag	0%	100%	2
xml-external-entities-unsafe-entity-loader	0%	100%	1
xml-external-entities-unsafe-parser-flags	0%	100%	1
laravel-unsafe-entity-loader	0%	100%	1
laravel-xml-unsafe-parser-flags	0%	100%	1

Fig. 13. – Récapitulatif des corrections

2.2 – Audit d'une application complète avec SEMGREP

2.2.1 – Réinstallation dans un environnement propre

On réinstalle juste Semgrep dans un autre Virtual Environment pour avoir une installation propre (même si on aurait pu réutiliser celle de `ex_2/`).

```
deactivate && cd ../../ex_3 && python3 -m venv .venv && source ./venv/bin/activate
python3 -m pip install pyotp flask qrcode # Utilisés à plusieurs endroits, mais tous les trois en même temps dans
"mod_mfa.py"
python3 -m pip install semgrep
python3 -m pip freeze > requirements.txt
```

Code 9. – Réinstallation de semgrep dans un environnement clean pour l'exercice 3

\$ `semgrep ci`

27 Non-blocking Code Findings

Fig. 14. – Uh oh...

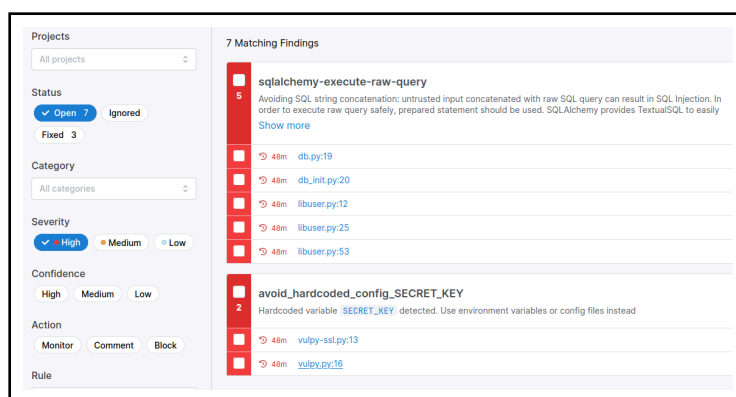


Fig. 15. – Extrait des vulnérabilités (High) trouvées dans `ex_3`

Dans un à plusieurs fichiers pour chacune, on a

Sévérité	Nombre de vulnérabilités	Vulnérabilités
HIGH	2	1. sqlalchemy-execute-raw-query 2. avoid_hardcoded_config_SECRET_KEY
MEDIUM	5	1. django-no-csrf-token 2. formatted-sql-query 3. avoid_using_app_run_directly 4. debug-enabled 5. missing-integrity
LOW	1	1. request-with-http

2.2.2 – Analyse et correction des problèmes trouvés

1. `avoid_hardcoded_config_SECRET_KEY` nécessite que l'on n'hardcode pas les secret dans l'application, ni même dans un fichier d'environnement visible dans le dépôt GitHub. Les secrets doivent être dans des variables d'environnement, seulement en local. Ainsi, on peut faire les modifications nécessaires dans les fichiers `vulpy*.py` :

```
import os
...
app.config['SECRET_KEY'] = os.environ['VULPY_SECRET_KEY']
```

Si on utilise bash par exemple, il faudrait définir `export VULPY_SECRET_KEY='aaaaaaa'` dans `~/.bashrc`.

2. sqlalchemy-execute-raw-query nécessite que l'on sanitize les input utilisateur qui pourraient entrer dans une query SQL, pour éviter les risques de SQL injections, comme les bases de données sont des éléments critiques. Selon [ce commentaire sur le sujet de SQL injections sur Stack Overflow](#). Ainsi, on modifie les fichiers `db.py`, `db_init.py` en utilisant des requêtes préparées / paramétrées, notamment en remplaçant `for ... : Connection.execute` par `Connection.executemany` pour les deux premiers fichiers, et par exemple `c.execute("UPDATE users SET password = ? WHERE username = ?", (password, username))` pour une des lignes du troisième.
3. django-no-csrf-token indique que la méthode de django pour éviter les CSRF n'a pas été utilisée. On peut s'aider de [la documentation sur leur site à ce sujet](#), pour ainsi corriger les problèmes dans les fichiers dans `templates/` que sont `mfa.enable.html`, `posts.view.html`, `user.chpasswd.html`, `user.create.html`, `user.login.html`, `user.login.mfa.html` et `welcome.html`. En somme, tous les fichiers utilisant un `<form method="POST">...</form>` à remplacer avec `<form method="POST">{% csrf_token %} ...</form>`.

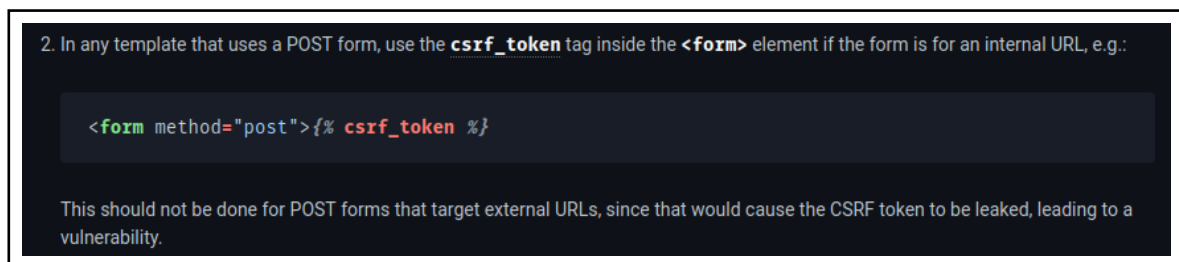


Fig. 16. – Conseil de la doc de django sur les CSRF

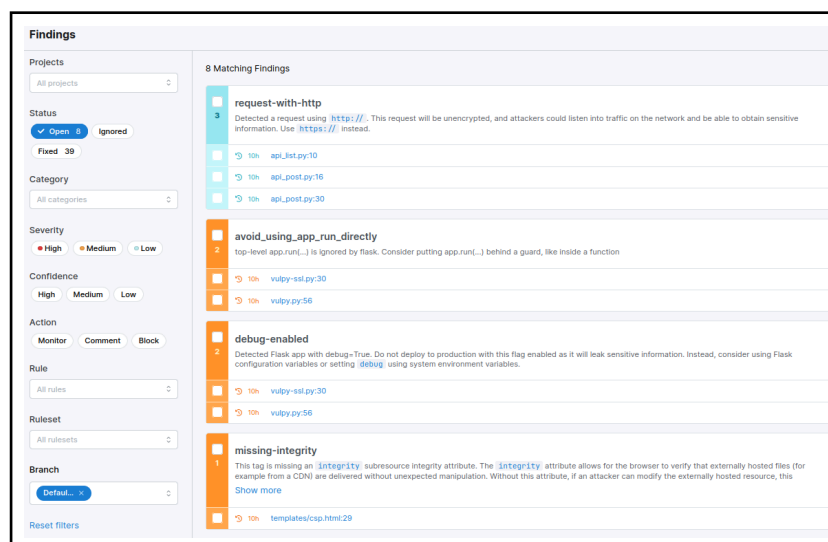


Fig. 17. – Point sur la progression sur la correction des problèmes détectés

4. missing-integrity dans le fichier `templates/csp.html` est soulevée par Simgrep car il faudrait ajouter un paramètre integrity avec le hash du script à importer dans la balise `<script/>`. Pour cela, je commence par télécharger le script en local en allant à l'URL, (ici <https://apis.google.com/js/platform.js>), je le hash avec `sha256sum` en local et je modifie la balise ainsi :

```
<script ...
integrity="sha256-0bcb6531cb0967359e17b655d4142b55d1eac2aed3fe5340f8ce930a7000e5d3">
</script>
```

(on aurait aussi pu utiliser `openssl sha256 platform.js` pour donner un équivalent à `platform.js`).

5. `avoid_using_app_run_directly` (pour les fichiers `vulpy-ssl.py` et `vulpy.py`), problème de Broken Access Control, pour lequel il faut mettre les appels à `app.run()` derrière « une garde » (comme une fonction, ou un `if __name__ == '__main__': ...`). Nous avons décidé d'utiliser `if __name__ == '__main__': ...`, comme c'est une best-practice en Python, qui permet notamment d'indiquer que le fichier est un script exécutable et non juste une librairie.
6. `debug-enabled`, toujours avec les deux même fichiers `vulpy-ssl.py` et `vulpy.py`, indique que `debug=True` en tant que paramètre de `app.run` est problématique : si l'app est lancée en production avec cette configuration, des info sensibles peuvent potentiellement leak dans les logs. Il vaut mieux préférer set cette valeur avec des variables d'environnement, en la définissant par défaut à `False` si non définie. Par exemple avec

```
DEBUG = (_lower() == 'true') if (_ := os.environ.get("VULPY_DEBUG", None)) is not
None else False
app.run(..., debug = DEBUG, ...)
```

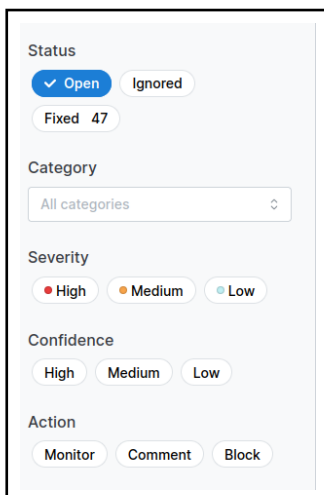
En fait, `os.environ` va load les variables d'environnement en tant que string par défaut, et toute string non vide est considérée comme `True`. On va donc utiliser cette expression ternaire créée de toute pièce va vérifier que la variable d'environnement est bien définie à `'true'`, ou `'True'`, etc.

- Si oui, on vérifie que sa version minuscule correspond bien à `'true'`, auquel cas on renvoie `True`
- Si non, on renvoie simplement `False`, la valeur par défaut.

Nous avons déjà importé `os` lors du patch sur les secrets précédemment, donc pas besoin de le réimporter. Évidemment, il faut aussi définir la variable d'environnement (comme pour les secrets avant) en local.

7. La dernière `request-with-http` trouvée dans les fichiers `api_list.py` et `api_post.py` est explicite d'elle-même. Le problème pour la corriger est que si on fait un appel à un site qui ne supporte pas HTTPS mais juste HTTP, on ne peut pas faire grand chose de plus (à notre connaissance). Là c'est une requête vers la loopback (127.0.0.1) donc on peut mettre `https` si on setup correctement en local avec les certificats, et tout.

On va modifier en supposant que les admin' de la machine du serveur auraient fait les bonnes config' et générations de certificats.



The image shows a sidebar filter panel for a vulnerability management system. It contains the following sections:

- Status:** Includes buttons for 'Open' (with a checkmark icon), 'Ignored', and 'Fixed 47'.
- Category:** A dropdown menu currently showing 'All categories'.
- Severity:** Includes buttons for 'High' (with a red dot), 'Medium' (with an orange dot), and 'Low' (with a blue dot).
- Confidence:** Includes buttons for 'High', 'Medium', and 'Low'.
- Action:** Includes buttons for 'Monitor', 'Comment', and 'Block'.

Fig. 18. – Toutes les vulnérabilités trouvées par Semgrep ont été corrigées

2.3 – Création d'application avec vulnérabilité non détectée par SEMGREP

cd ../ex_4/

2.3.1 – Premières idées

La première idée que nous avons eu était de berner Semgrep. C'est-à-dire écrire du code vulnérable, mais en le camouflant, afin qu'une analyse statique avec des patterns ne puisse pas la détecter.

Un exemple tout bête : si une règle détecte l'usage de la fonction d'exécution de code sous forme de string dans Python `exec`, il suffit de l'invoquer sans utiliser son nom. On peut la récupérer du dictionnaire des variables globales et l'appeler :

```
getattr(
    globals()[['_builtins_']],
    'e' + 'xQc'.replace('Q', 'e')
)(''
arbitrary = lambda: print("Exécution de code issu d'une string")
arbitrary()
'')
```

Code 10. – Exemple simple d'un appel à `exec(...)` non détecté par un SAST

On a passé le modules des builtins présent dans l'environnement global dans la fonction `getattr` qui va récupérer la fonction associée 'exec' (string obfusquée aux regard des SAST) puis passer le paramètre qui est le code à exécuter, sous forme de string (qui peut venir un input non sanitisé).

2.3.2 – Application vulnérable

Nous avons utilisé Zap comme DAST et nous nous sommes inspirés du code d'Achraf & Sarah : les fichiers de l'application *Flask* sont dans `ex_4/`, que l'on lance simplement avec `python3 vulnerable.py`.

Même en sanitisant l'input, Semgrep trouve toujours à tort qu'il y a une vulnérabilité `directly-returned-format-string`, de sévérité medium. Par contre, ce qu'il ne voit pas et que Zap voit (Cf Fig. 19), c'est l'absence de token pour prévenir les CSRF.

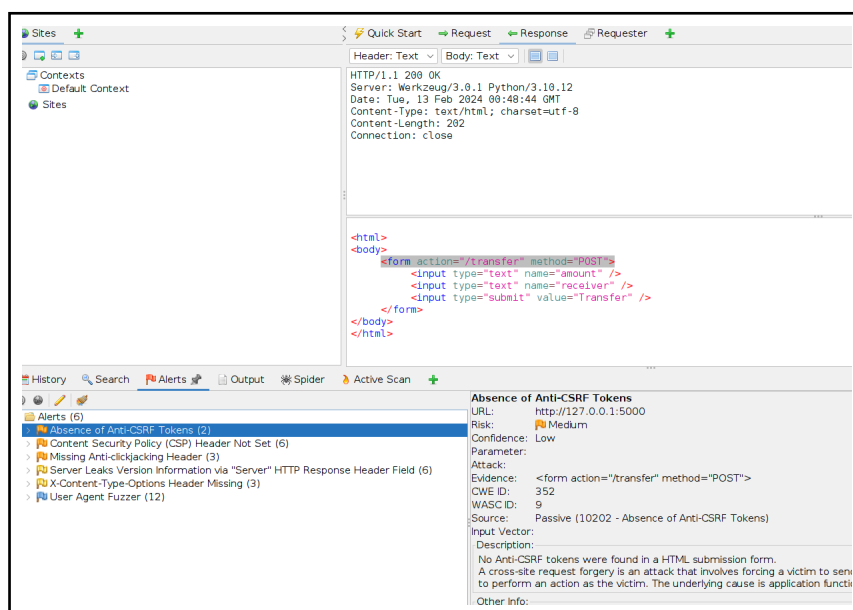


Fig. 19. – Zap trouve l'absence de token CSRF

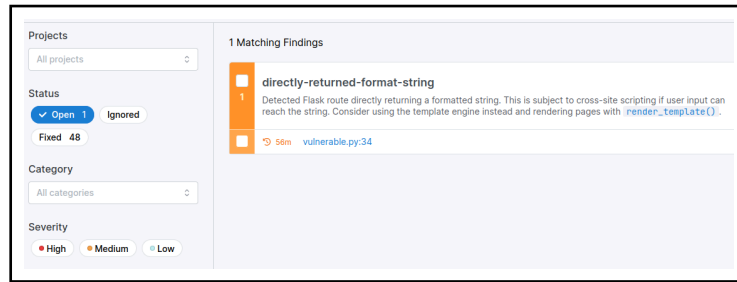


Fig. 20. – Semgrep donne des faux positifs, et met en faux négatif le CSRF

Le problème ici n'est pas que Semgrep n'est pas capable de détecter l'absence de ce token, puisqu'il l'a fait pour `django-no-csrf-token` dans le Chapitre 2.2.2, mais plutôt comment Semgrep fonctionne. De ce que nous avons compris, il utilise un ensemble de fichiers de règles pour trouver des patterns, et à l'aide notamment de logique de source et sink, il va pouvoir déterminer si tel morceau de code correspond à un pattern (ils essayent d'être exhaustifs) de vulnérabilité connu.

Sauf que là, la vulnérabilité est dans une string renvoyée, qui n'est pas vérifiée par Semgrep. Et on voit même que, pour une vulnérabilité différente (`directly-returned-format-string`), Semgrep va toujours s'alarmer alors que l'input est sanitisé.

Pour les deux cas, il faudrait lui rajouter des règles, d'une part pour vérifier plus de choses, d'autres part, pour ajouter des sanitizers.

2.3.3 – Conclusion

Cela n'était qu'un exemple, mais on peut aussi avoir des config' qui ne sont pas vérifiées par un SAST, ou alors des obfuscations qui sortent des patterns comme l'exemple théorique dans le Chapitre 2.3.1.