

TD – Analyse statique du code

Khadidja MEHDI, Valentin NASONE

Lien du repo GitHub : <https://github.com/khad-mdi/semgrep-tp>.

Exercice 1 : BANDIT

BANDIT : SAST pour analyser les programmes Python.

Il fonctionne de 2 manières : sans fichier de config (dans ce cas, il effectue tous les tests), ou avec un fichier de config.

Pour générer des rapports, on utilise les commandes suivantes :

```
bandit "*.py" -f html -o report_without_config.html
```

```
bandit "*.py" -f txt -o report_without_config.txt
```

Ces rapports sont composés d'informations sur la vulnérabilité et sur le test utilisé pour la découvrir, de la ou les ligne(s) de code problématique(s), mais également de 2 paramètres importants :

- **SEVERITY** : la gravité de la vulnérabilité identifiée,
- **CONFIDENCE** : la confiance de Bandit dans la détection de la vulnérabilité.

□ 1.py

À première vue, on peut rapidement déduire que le code est vulnérable aux injections SQL. En effet, des requêtes sont créées à partir d'input utilisateur et aucune mesure n'est mise en place pour vérifier cet input. Cette hypothèse est confirmée par le rapport généré par Bandit, qui ne renvoie que des vulnérabilités SQLi, voici un exemple :

```
django_extra_used: Use of extra potential SQL attack vector.  
Test ID: B610  
Severity: MEDIUM  
Confidence: MEDIUM  
CWE: CWE-89  
File: 1.py  
Line number: 12  
More info: https://bandit.readthedocs.io/en/1.7.6/plugins/b610\_django\_extra\_used.html  
  
11  
12     User.objects.filter(username='admin').extra(dict(could_be='insecure'))  
13     User.objects.filter(username='admin').extra(select=dict(could_be='insecure'))
```

□ 2.py

À première vue, ce fichier d'une ligne contient du code vulnérable à l'injection de code. La fonction **exec** en Python n'effectue aucune vérification particulière sur le code qu'elle exécute, donc si la chaîne de caractères qu'elle prend en paramètre contient du code malveillant, il sera exécuté sans aucune restriction. Cette hypothèse est confirmée par le rapport généré par Bandit :

```
exec_used: Use of exec detected.  
Test ID: B102  
Severity: MEDIUM  
Confidence: HIGH  
CWE: CWE-78  
File: 2.py  
Line number: 1  
More info: https://bandit.readthedocs.io/en/1.7.6/plugins/b102\_exec\_used.html  
  
1      exec("pas bien")
```

□ 3.py

À première vue, on remarque que la plupart des mots de passe de ce fichier sont « harcodés ». Le rapport généré par Bandit confirme cette hypothèse, mais considère que c'est une vulnérabilité peu grave (LOW SEVERITY) En effet, elle requiert d'avoir accès au code source et ne peut pas être facilement exploitable depuis l'extérieur :

```
hardcoded_password_string: Possible hardcoded password: 'class_password'  
Test ID: B105  
Severity: LOW  
Confidence: MEDIUM  
CWE: CWE-259  
File: 3.py  
Line number: 4  
More info: https://bandit.readthedocs.io/en/1.7.6/plugins/b105\_hardcoded\_password\_string.html  
  
3      class SomeClass:  
4          password = "class_password"  
5
```

□ 4.py

À première vue, on remarque que ce fichier permet d'ouvrir des fichiers en local à l'aide de `file://`, ce qui représente une vulnérabilité de sécurité puisqu'on peut potentiellement avoir accès à des fichiers sensibles. Cette hypothèse est vérifiée par le rapport généré par Bandit :

```
blacklist: Audit url open for permitted schemes. Allowing use of file:/ or custom schemes is often unexpected.  
Test ID: B310  
Severity: MEDIUM  
Confidence: HIGH  
CWE: CWE-22  
File: 4.py  
Line number: 9  
More info: https://bandit.readthedocs.io/en/1.7.6/blacklists/blacklist\_calls.html#b310-urllib-urlopen  
  
8      # Python 3  
9      urllib.request.urlopen('file:///bin/ls')  
10     urllib.request.urlretrieve('file:///bin/ls', '/bin/ls2')
```

□ 5.py

À première vue, on remarque également dans ce fichier des lignes de code vulnérables aux SQLi puisque les inputs de l'utilisateur ne sont pas vérifiés mais sont utilisées dans des requêtes SQL. Cette hypothèse est vérifiée par le rapport généré par Bandit :

hardcoded_sql_expressions: Possible SQL injection vector through string-based query construction.
Test ID: B608
Severity: MEDIUM
Confidence: LOW
CWE: [CWE-89](#)
File: [5.py](#)
Line number: 4
More info: https://bandit.readthedocs.io/en/1.7.6/plugins/b608_hardcoded_sql_expressions.html

```
3      # bad
4      query = "SELECT * FROM foo WHERE id = '%s'" % identifier
5      query = "INSERT INTO foo VALUES ('a', 'b', '%s')" % value
```

Maintenant, nous allons modifier le fichier de configuration pour effectuer seulement les tests que nous jugeons pertinents. Après lecture du rapport, on ajoute la ligne suivante au fichier de configuration :

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

Ensuite, on génère les rapports avec les commandes suivantes :

```
bandit "*.py" -c config.yml -f html -o report_with_config.html
```

```
bandit "*.py" -c config.yml -f txt -o report_with_config.txt
```

Exercice 2 : SEMGREP

Semgrep : SAST qui permet de rechercher des motifs dans le code source pour identifier des vulnérabilités de sécurité, des anti-patterns, des bugs, des erreurs de configuration, des problèmes de performance, etc. Contrairement à Bandit qui ne concerne que Python, on peut utiliser Semgrep pour plusieurs langages de programmation.

Dans le dossier `ex_2`, on lance la commande suivante: **semgrep ci**. Puis, on accède aux rapports. On trouve 13 vulnérabilités au total, et 3 vulnérabilités HIGH que nous allons corriger.

1

xml-external-entities-unsafe-entity-loader

Security High </> Php

The application is using an XML parser that has not been safely configured. This might lead to XML External Entity (XXE) vulnerabilities

[Show more](#)

6m 1/test.php:12

main Details

Vulnérabilité

Cette vulnérabilité se produit lorsqu'une application utilise un analyseur XML configuré de manière non sécurisée. Elle concerne les lignes suivantes du fichier **1/test.php** :

- ❑ La ligne `libxml_disable_entity_loader (false)` ; désactive la désactivation par défaut du chargeur d'entités XML, et donc l'application est configurée pour autoriser le chargement d'entités XML.
- ❑ La ligne `$document->loadXML($xml, LIBXML_NOENT | LIBXML_DTDLOAD)` ; charge le XML à partir de la valeur de `$_GET['xml']` dans un objet DOMDocument. Les options `LIBXML_NOENT` et `LIBXML_DTDLOAD` indiquent que le chargement de l'entité XML et du Document Type Definition (DTD) est activé.

Exploitation

Un attaquant peut exploiter cette vulnérabilité en fournissant un XML contenant des entités externes malveillantes, ce qui peut entraîner plusieurs types d'attaques comme de l'exécution de code à distance (RCE) ou du déni de service (DoS).

Contremesure

Pour corriger cette vulnérabilité, nous allons activer la désactivation par défaut du chargeur d'entités XML :

```
libxml_disable_entity_loader (true);
```

1

xml-external-entities-unsafe-parser-flags

Security High </> Php

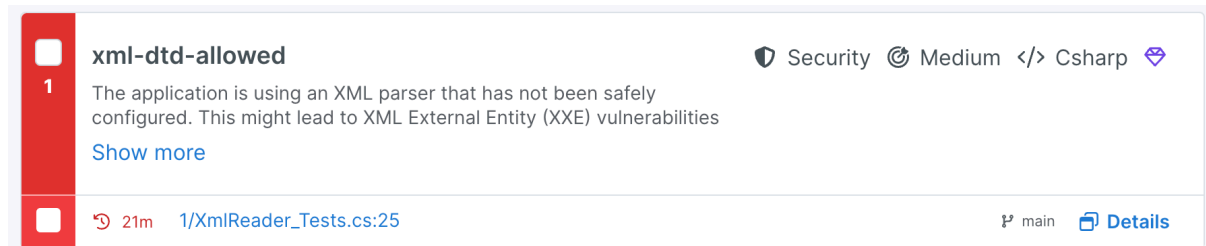
The application is using an XML parser that has not been safely configured. This might lead to XML External Entity (XXE) vulnerabilities

[Show more](#)

17m 1/test.php:12

main Details

Cette vulnérabilité est liée à la précédente car les entités qui sont chargés vont être parsés par la suite : “This might lead to XML External Entity (XXE)”. Cependant, comme nous avons corrigé le fait de charger des entités externes, cette vulnérabilité n’a pas lieu d’être et constitue un faux positif.



Vulnérabilité

Cette vulnérabilité se produit lorsqu'une application utilise un analyseur XML qui permet le traitement de définitions de DTD, qui sont des documents utilisés pour définir la structure d'un document XML. Elle concerne les lignes suivantes du fichier **1/XmlReader_Tests.cs** :

- La ligne `settings.DtdProcessing = DtdProcessing.Parse;` indique que les DTDs sont autorisées à être analysées.

Exploitation

Un attaquant peut exploiter cette vulnérabilité en fournissant un DTD mal configurées qui entraîneront des XXE (XML External Entity). Ce qui pourra être utilisé pour plusieurs types d'attaques comme de l'exécution de code à distance (RCE) ou du déni de service (DoS). Comme les DTD sont autorisées à être analysées, un attaquant peut injecter une DTD malveillante qui sera analysée, et pouvant engendrer des attaques XXE.

Contremesure

Pour corriger cette vulnérabilité, on ignore l'analyse de DTD, comme ceci :

```
settings.DtdProcessing = DtdProcessing.Ignore;
```

Toutes les vulnérabilités critiques (HIGH) ont été corrigées.



Exercise 3

Nous allons auditer les fichiers de l'exercice 3 en utilisant Semgrep. Nous utiliserons également Bandit en parallèle pour les fichiers Python afin de comparer les deux SAST.

Analyse fichiers Python : BANDIT vs SEMGREP

L'analyse Bandit met en avant 2 vulnérabilités HIGH dans les fichiers Python.

```
Total issues (by severity):
  Undefined: 0
  Low: 5
  Medium: 14
  High: 2
```

L'analyse SEMGREP met en avant 7 vulnérabilités HIGH dans les fichiers Python.

7 Matching Findings

Analyze (0) Triage (0)

5 **sqlalchemy-execute-raw-query** Security Low Python
Avoiding SQL string concatenation: untrusted input concatenated with raw SQL query can result in SQL Injection. In order to execute raw query
[Show more](#)

- db.py:19 main Details
- db_init.py:20 main Details
- libuser.py:12 main Details
- libuser.py:25 main Details
- libuser.py:53 main Details

2 **avoid_hardcoded_config_SECRET_KEY** Security Low Python
Hardcoded variable `SECRET_KEY` detected. Use environment variables or config files instead

- vulpy-ssl.py:13 main Details
- vulpy.py:16 main Details

Cette comparaison illustre le fait que les SAST fonctionnent tous différemment. Semgrep considère qu'un secret « harcodé » est critique, alors que Bandit classe cette vulnérabilité avec **SEVERITY=LOW**. À l'inverse, Semgrep considère qu'activer le mode DEBUG n'est que **SEVERITY=MEDIUM**, alors que Bandit le classe en **SEVERITY=HIGH**. Enfin, les lignes de codes sensibles aux SQLI que nous allons corriger dans cette exercice car jugées critiques par Semgrep ont une sévérité **MEDIUM** pour Bandit. Ce sont seulement quelques exemples de résultats différents que l'on obtient en utilisant différents SAST. C'est donc le travail de l'ingénieur sécurité d'étudier toutes les vulnérabilités qui ont été remontées par les différents outils et de juger de la criticité de chacune, selon les spécificités de son projet.

Correction des vulnérabilités avec SEMGREP

Dans ce TP, nous utiliserons Semgrep qui prend en charge plusieurs langages de programmation. Étant donné que notre application ne contient pas que du Python, nous serons limités avec Bandit.

Vulnérabilité 1 : sqlalchemy-execute-raw-query	
<p>Cette vulnérabilité concerne le fait d'exécuter des requêtes SQL brutes sans précautions, concaténées avec des inputs utilisateurs qui n'ont pas été vérifiées.</p> <p>Pour la corriger, nous utilisons des instructions préparées avec des marqueurs ?, au lieu de concaténer directement les valeurs de u, p, et 0 dans la requête SQL, Cela signifie que les valeurs sont passées séparément et ne sont pas interprétées comme du code SQL potentiellement malveillant.</p>	
Fichier	db.py:19
Code vulnérable	<code>c.execute("INSERT INTO users (user, password, failures) VALUES ('%s', '%s', '%d')" % (u, p, 0))</code>
Code corrigé	<code>c.execute("INSERT INTO users (user, password, failures) VALUES (?, ?, ?)", (u, p, 0))</code>
Fichier	db_init.py:20
Code vulnérable	<code>c.execute("INSERT INTO users (username, password, failures, mfa_enabled, mfa_secret) VALUES ('%s', '%s', '%d', '%d', '%s')" % (u, p, 0, 0, ''))</code>
Code corrigé	<code>c.execute("INSERT INTO users (username, password, failures, mfa_enabled, mfa_secret) VALUES (?, ?, ?, ?, ?)", (u, p, 0, 0, ''))</code>
Fichier	libuser.py:12
Code vulnérable	<code>user = c.execute("SELECT * FROM users WHERE username = '{}' and password = '{}'".format(username, password)).fetchone()</code>
Code corrigé	<code>user = c.execute("SELECT * FROM users WHERE username = ? AND password = ?", (username, password)).fetchone()</code>
Fichier	libuser.py:25
Code vulnérable	<code>c.execute("INSERT INTO users (username, password, failures, mfa_enabled, mfa_secret) VALUES ('%s', '%s', '%d', '%d', '%s')" % (username, password, 0, 0, ''))</code>
Code corrigé	<code>c.execute("INSERT INTO users (username, password, failures, mfa_enabled, mfa_secret) VALUES (?, ?, ?, ?, ?)", (username, password, 0, 0, ''))</code>
Fichier	libuser.py:53
Code vulnérable	<code>c.execute("UPDATE users SET password = '{}' WHERE username = '{}'".format(password, username))</code>
Code corrigé	<code>c.execute("UPDATE users SET password = ? WHERE username = ?", (password, username))</code>
Vulnérabilité 2: avoid_hardcoded_config_SECRET_KEY	
<p>Cette vulnérabilité concerne une variable appelée SECRET_KEY qui est harcodée. Pour la corriger, il faut passer par l'utilisation de variable d'environnement. On ajoute une condition qui vérifie que la variable est bien définie. Cette vulnérabilité apparaît 2 fois dans le rapport de Semgrep.</p>	
Fichier	vulpy-ssl.py:13 et vulp.py :16 (les lignes de code vulnérables sont identiques)
Code vulnérable	<code>app.config['SECRET_KEY'] = 'aaaaaaa'</code>
Code corrigé	<pre>SECRET_KEY = os.environ.get('SECRET_KEY') if SECRET_KEY is None: raise ValueError("SECRET_KEY n'est pas défini dans les variables d'environnement.") app.config['SECRET_KEY'] = SECRET_KEY</pre>

Suite à ces corrections, nous n'avons plus aucune vulnérabilité HIGH sur notre application.

Fixed 7

Exercice 4

La plus importante limite d'un SAST est qu'il ne peut pas détecter les vulnérabilités pendant le runtime. Pour illustrer cet exemple, on va simuler et prouver que Semgrep (SAST) le traite comme un faux négatif. Dans un second temps, on utilisera un DAST pour détecter la vulnérabilité non relevée par le SAST. Pour cela, on va coder une application web simple en Python Flask, où un utilisateur peut soumettre un nom de fichier pour vérifier s'il existe sur le serveur. Code exécute une commande système pour vérifier la présence d'un fichier, ce qui constitue une vulnérabilité **d'injection de commande**.

```
from flask import Flask, request, jsonify
import os

app = Flask(__name__)

def execute(command):
    result = os.popen(command).read()
    return result

@app.route('/checkfile', methods=['GET'])
def check_file():
    filename = request.args.get('filename')
    command_result = execute(f"ls -l {filename} 2>/dev/null")
    return jsonify({'command_result': command_result})

if __name__ == '__main__':
    app.run(debug=False)
```

D'abord, on utilise Semgrep pour chercher la vulnérabilité dans le code, mais on ne trouve aucun problème signalé. Semgrep peut rater une vulnérabilité d'injection de commande parce qu'il regarde seulement le code sans le faire tourner. Il a besoin de règles très précises pour repérer ce problème, et si ces règles ne sont pas là ou si le problème ne se montre qu'en faisant fonctionner le code avec certaines données, Semgrep ne le détectera pas.



No matching findings

Nous allons utiliser **Burp**, un outil d'analyse dynamique (DAST), pour tester notre application Flask. Après avoir lancé l'application avec la commande **python3 vulnerable_app.py**, elle est accessible à l'URL **http://127.0.0.1:5000/checkfile**. Pour configurer Burp, on active l'interception des requêtes dans l'application en allant dans **Proxy > Intercept is On**, puis on utilise le navigateur intégré à Burp pour naviguer vers l'URL (Chromium).

D'abord, on effectue un test sans tenter d'injection, en visitant **`http://127.0.0.1:5000/checkfile?filename=password.txt`**. La réponse indique l'existence du fichier **`password.txt`** sans en montrer le contenu. En tant que cyberattaquant, on va tenter de récupérer le contenu de ce fichier.

```
{"command_result":"-rw-r--r--  1 khad  staff  45 Feb 10 12:05 password.txt\n"}
```

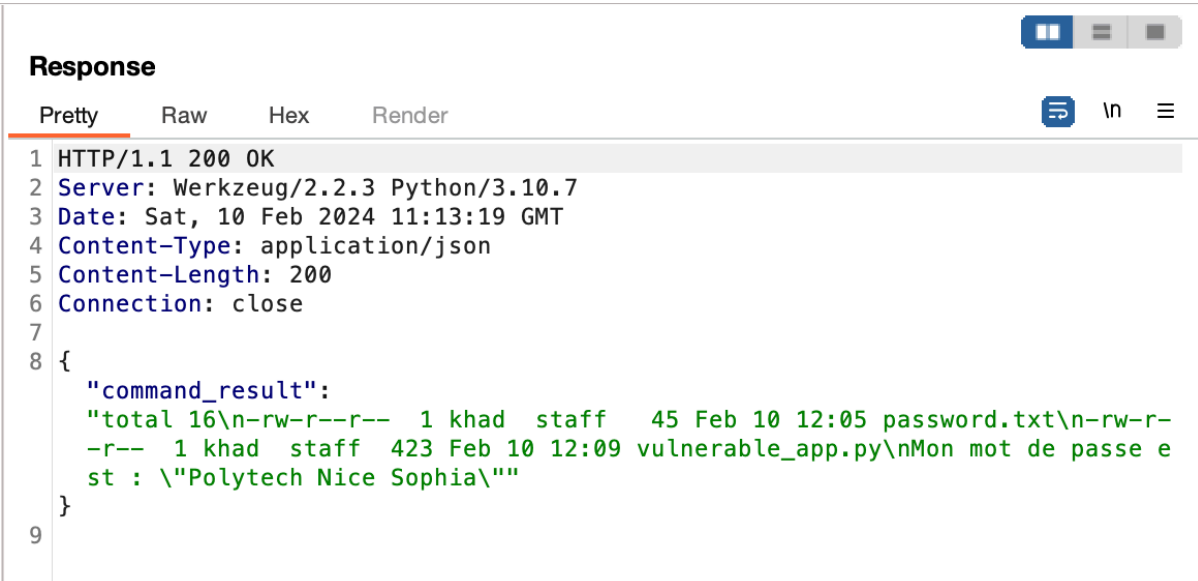
Pour cela, après avoir intercepté la requête initiale dans Burp, on désactive l'interception et on utilise cette requête comme base pour une injection. On se rend dans l'onglet **HTTP history**, sélectionne la requête GET, puis utilise l'option **Send to repeater** pour la modifier dans l'onglet **Repeater**. La première ligne de la requête est :

`GET /checkfile?filename=password.txt HTTP/1.1`

On la modifie pour tenter une injection de commande qui va afficher le contenu du fichier `password.txt` en utilisant :

`GET /checkfile?filename=;cat%20password.txt HTTP/1.1`

⚠ Il faut encoder l'espace en %20 pour éviter les erreurs de syntaxe HTTP.



```
Response
Pretty Raw Hex Render
1 HTTP/1.1 200 OK
2 Server: Werkzeug/2.2.3 Python/3.10.7
3 Date: Sat, 10 Feb 2024 11:13:19 GMT
4 Content-Type: application/json
5 Content-Length: 200
6 Connection: close
7
8 {
  "command_result":
    "total 16\n-rw-r--r--  1 khad  staff   45 Feb 10 12:05 password.txt\n-rw-r--r--  1 khad  staff  423 Feb 10 12:09 vulnerable_app.py\nMon mot de passe est : \"Polytech Nice Sophia\""
}
9
```

On parvient donc à afficher le contenu du fichier **`password.txt`**, ce qui prouve l'existence d'une vulnérabilité d'injection de commande. Cet exemple démontre que les DAST peuvent identifier des vulnérabilités qui peuvent échapper aux SAST.