

## Rendu de TP Analyse statique de code

11 février 2024

Florian Latapie – Élisabeth Roux

L'ensemble du code est disponible sur notre dépôt GitHub :

[github.com/FlorianLatapie/PNS-SI5-S9\\_Analyse\\_statique\\_code](https://github.com/FlorianLatapie/PNS-SI5-S9_Analyse_statique_code)

### Exercice 1

Nous avons scanné chacun des répertoires une première fois avec Bandit sans modifier le fichier de configuration. Le premier rapport que nous avons récupéré était le default-rapport.html. Ensuite, nous avons identifié les vulnérabilités de chaque fichier et avons modifié le fichier de configuration comme suit avant de régénérer un rapport ([ex 1/report.html](#) et [ex 1/report.txt](#) en sont la version définitive).

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

```
bandit -r . -f html -o default-report.html -c default-config.yml
bandit -r . -f txt -o default-report.txt -c default-config.yml
```


















```
bandit -r . -f html -o report.html -c my_config.yml
bandit -r . -f txt -o report.txt -c my_config.yml
```

Notre fichier de configuration

### Exercice 2

Par la suite, nous avons, à l'aide de l'outil Semgrep, audité l'ensemble des dossiers de l'exercice 2. Voici une analyse de notre démarche.

## ● Découverte des vulnérabilités

 2	<b>direct-response-write</b> Detected directly writing to a Response object from user-defined input. This bypasses any HTML escaping and may expose your application to a Cross-Site-scripting (XSS) vulnerability. Instead, use <code>resp.render()</code> . <a href="#">Show more</a>	🛡️ Security 🟡 Medium </> Javascript
 7m	2/y.js:9	🔍 HEAD +1 <a href="#">Details</a>
 7m	3/express.js:6	🔍 HEAD +1 <a href="#">Details</a>
 2	<b>taint-unsafe-echo-tag</b> Found direct access to a PHP variable without HTML escaping inside an inline PHP statement setting data from <code>\$_REQUEST[...]</code> . When untrusted input can be used to tamper with a web page rendering, it can lead <a href="#">Show more</a>	🛡️ Security 🟡 Medium </> Php 💡
 7m	3/dom.php:11	🔍 HEAD +1 <a href="#">Details</a>
 7m	3/example.php:7	🔍 HEAD +1 <a href="#">Details</a>
 2	<b>open-redirect-deepsemgrep</b> The application builds a URL using user-controlled input which can lead to an open redirect vulnerability. An attacker can manipulate the URL and redirect users to an arbitrary domain. Open redirect vulnerabilities can <a href="#">Show more</a>	🛡️ Security 🟠 High </> Javascript 💡
 7m	4/aa.js:17	🔍 HEAD +1 <a href="#">Details</a>
 7m	4/koa.js:8	🔍 HEAD +1 <a href="#">Details</a>
 1	<b>raw-html-format</b> User data flows into the host portion of this manually-constructed HTML. This can introduce a Cross-Site-Scripting (XSS) vulnerability if this comes from user-provided input. Consider using a sanitization library such <a href="#">Show more</a>	🛡️ Security 🟡 Medium </> Javascript
 7m	3/express.js:6	🔍 HEAD +1 <a href="#">Details</a>
 1	<b>xml-external-entities-unsafe-entity-loader</b> The application is using an XML parser that has not been safely configured. This might lead to XML External Entity (XXE) vulnerabilities when parsing user-controlled input. An attacker can include document type definitions (DTDs) which can interact with internal or external hosts. XXE can lead to other vulnerabilities, such as Local File Inclusion (LFI), Remote Code Execution (RCE), and Server-side request forgery (SSRF), depending on the application configuration. An attacker can also use DTDs to expand recursively, leading to a Denial-of-Service (DoS) attack, also known as a Billion Laughs Attack. The best defense against XXE is to have an XML parser that supports disabling DTDs. Limiting the use of external entities from the start can prevent the parser from being used to process untrusted XML files. Reducing dependencies on external resources is also a good practice for performance reasons. It is difficult to guarantee that even a trusted XML file on your server or during transmission has not been tampered with by a malicious third-party. <a href="#">Hide</a>	🛡️ Security 🟠 High </> Php 💡
 7m	1/test.php:12	🔍 HEAD +1 <a href="#">Details</a>
 1	<b>xml-external-entities-unsafe-parser-flags</b> The application is using an XML parser that has not been safely configured. This might lead to XML External Entity (XXE) vulnerabilities when parsing user-controlled input. An attacker can include document type definitions (DTDs) which can interact with internal or external hosts. XXE can lead to other vulnerabilities, such as Local File Inclusion (LFI), Remote Code Execution (RCE), and Server-side request forgery (SSRF), depending on the application configuration. An attacker can also use DTDs to expand recursively, leading to a Denial-of-Service (DoS) attack, also known as a Billion Laughs Attack. The best defense against XXE is to have an XML parser that supports disabling DTDs. Limiting the use of external entities from the start can prevent the parser from being used to process untrusted XML files. Reducing dependencies on external resources is also a good practice for performance reasons. It is difficult to guarantee that even a trusted XML file on your server or during transmission has not been tampered with by a malicious third-party. <a href="#">Hide</a>	🛡️ Security 🟠 High </> Php 💡
 4m	1/test.php:12	🔍 HEAD +1 <a href="#">Details</a>
 1	<b>laravel-unsafe-entity-loader</b> The application is using an XML parser that has not been safely configured. This might lead to XML External Entity (XXE) vulnerabilities when parsing user-controlled input. An attacker can include document type <a href="#">Show more</a>	🛡️ Security 🟡 Medium </> Php 💡
 7m	1/test.php:12	🔍 HEAD +1 <a href="#">Details</a>

1

laravel-xml-unsafe-parser-flags

Security

Medium

</> Php

1

The application is using an XML parser that has not been safely configured. This might lead to XML External Entity (XXE) vulnerabilities when parsing user-controlled input. An attacker can include document type

Show more

5m

1/test.php:12

HEAD +1

Details

1

xml-dtd-allowed

Security

Medium

</> Csharp

1

The application is using an XML parser that has not been safely configured. This might lead to XML External Entity (XXE) vulnerabilities when parsing user-controlled input. An attacker can include document type definitions (DTDs) or XIncludes which can interact with internal or external hosts. XXE can lead to other vulnerabilities, such as Local File Inclusion (LFI), Remote Code Execution (RCE), and Server-side request forgery (SSRF), depending on the application configuration. An attacker can also use DTDs to expand recursively, leading to a Denial-of-Service (DoS) attack, also known as a [Billion Laughs Attack](#). The best defense against XXE is to have an XML parser that supports disabling DTDs. Limiting the use of external entities from the start can prevent the parser from being used to process untrusted XML files. Reducing dependencies on external resources is also a good practice for performance reasons. It is difficult to guarantee that even a trusted XML file on your server or during transmission has not been tampered with by a malicious third-party.

Hide

5m

1/XmlReader\_Tests.cs:25

HEAD +1

Details

1

redos

Security

High

</> Javascript

1

The regular expression identified appears vulnerable to Regular Expression Denial of Service (ReDoS) through catastrophic backtracking. If the input is attacker controllable, this vulnerability can lead to systems being

Show more

8m

2/y.js:8

HEAD +1

Details

### • Vulnérabilités “high” à corriger :

1

xml-external-entities-unsafe-entity-loader

Security

High

</> Php

1

The application is using an XML parser that has not been safely configured. This might lead to XML External Entity (XXE) vulnerabilities when parsing user-controlled input. An attacker can include document type definitions (DTDs) which can interact with internal or external hosts. XXE can lead to other vulnerabilities, such as Local File Inclusion (LFI), Remote Code Execution (RCE), and Server-side request forgery (SSRF), depending on the application configuration. An attacker can also use DTDs to expand recursively, leading to a Denial-of-Service (DoS) attack, also known as a Billion Laughs Attack. The best defense against XXE is to have an XML parser that supports disabling DTDs. Limiting the use of external entities from the start can prevent the parser from being used to process untrusted XML files. Reducing dependencies on external resources is also a good practice for performance reasons. It is difficult to guarantee that even a trusted XML file on your server or during transmission has not been tampered with by a malicious third-party.

Hide

48m

1/test.php:12

main

Details

1

xml-external-entities-unsafe-parser-flags

Security

High

</> Php

1

The application is using an XML parser that has not been safely configured. This might lead to XML External Entity (XXE) vulnerabilities when parsing user-controlled input. An attacker can

Show more

48m

1/test.php:12

main

Details

1

xml-dtd-allowed

Security

Medium

</> Csharp

1

The application is using an XML parser that has not been safely configured. This might lead to XML External Entity (XXE) vulnerabilities when parsing user-controlled input. An attacker can

Show more

48m

1/XmlReader\_Tests.cs:25

main

Details

### • Correctif apporté

Analysons le fichier [ex 2/1/test.php](#) :

La ligne 12 : `$document→loadXML($xml, LIBXML_NOENT | LIBXML_DTDLOAD);` semble être la ligne problématique.

Observons la documentation des paramètres de la fonction loadXML :

[php.net/manual/en/libxml.constants.php](http://php.net/manual/en/libxml.constants.php)

Florian Latapie, Élis Roux

3/6

**LIBXML\_NOENT (int)**  
Substitute entities

**Caution** Enabling entity substitution may facilitate XML External Entity (XXE) attacks.

Après avoir supprimé l'attrbyt problématique, il reste toujours une erreur :  
xml-external-entities-unsafe-entity-loader.

**xml-external-entities-unsafe-entity-loader** High severity High confidence Monitor

The application is using an XML parser that has not been safely configured. This might lead to XML External Entity (XXE) vulnerabilities when parsing user-controlled input. An attacker can include document type definitions (DTDs) which can interact with internal or external hosts. XXE can lead to other vulnerabilities, such as Local File Inclusion (LFI), Remote Code Execution (RCE), and Server-side request forgery (SSRF), depending on the application configuration. An attacker can also use DTDs to expand recursively, leading to a Denial-of-Service (DoS) attack, also known as a Billion Laughs Attack. The best defense against XXE is to have an XML parser that supports disabling DTDs. Limiting the use of external entities from the start can prevent the parser from being used to process untrusted XML files. Reducing dependencies on external resources is also a good practice for performance reasons. It is difficult to guarantee that even a trusted XML file on your server or during transmission has not been tampered with by a malicious third-party.

**REFERENCES**

- <https://websec.io/2012/08/27/Preventing-XXE-in-PHP.html>
- <https://www.php.net/manual/en/function.libxml-set-external-entity-...>
- [https://www.php.net/libxml\\_disable\\_entity\\_loader](https://www.php.net/libxml_disable_entity_loader)

**Activity**

- [New note](#)
- [Seen on](#)
- [main](#)
- Tue, 23 Jan 2024 15:35:11 GMT
- 48 minutes ago via GitHub

**Pattern** **Metadata** [Open in Editor](#) [Data flow](#) [Example code](#) [Run locally](#)

```

1 pattern-sources:
2   - patterns:
3     - pattern-either:
4       - pattern: |
5         $._GET
6       - pattern: |
7         $._POST
8       - pattern: |
9         $._COOKIE
10      - pattern: |
11        $._FILES
12      - pattern: |
13        $._REQUEST
14 pattern-sinks:
15   - patterns:
16     - pattern-inside: |
17       libxml_disable_entity_loader(false);
18     ...
19   - pattern-either:
20     - patterns:
21       - pattern: |
22         $X->loadXML(...)
23       - pattern-inside: |
24         $X = new DOMDocument;
25         ...

```

**1/test.php**

**Source**

Line:9

↓

**Traces**

Line:9

↓

**Sink**

Line:12

C'est ainsi que nous trouvons dans la partie patterns "libxml\_disable\_entity\_loader(false);" cela indique que la présence de ce morceau de code déclenche l'alerte dans semgrep. Ainsi passer du paramètre "false" au paramètre "true", l'appli est ainsi moins vulnérable d'après semgrep !

```
7 libxml_disable_entity_loader (true);
```

## Exercice 3

Nous avons ici également utilisé Semgrep afin d'auditer cette fois le dossier de l'exercice 3. Se trouvent ci-dessous les vulnérabilités "high" que nous avons détectées.

**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 safely, prepared statement should be used. SQLAlchemy provides

[Show more](#)

- [db.py:19](#) 35s [HEAD](#) [Details](#)
- [db\\_init.py:20](#) 35s [HEAD](#) [Details](#)
- [libuser.py:12](#) 35s [HEAD](#) [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](#) 2m [HEAD +1](#) [Details](#)
- [vulpy.py:16](#) 2m [HEAD +1](#) [Details](#)

Il y a ici deux types d'erreurs à modifier

- **Erreurs SQL**

La solution ici est d'utiliser une **requête paramétrée** plutôt qu'une concaténation de chaînes de caractères :

```

52
53 - c.execute("UPDATE users SET password = '{}' WHERE username = '{}'.format(password, username))
54     conn.commit()
55
56
57 + c.execute("UPDATE users SET password = ? WHERE username = ?", (password, username))
58     conn.commit()
59

```

Cela permet d'éviter une **SQL injection** si un hacker tente d'utiliser le caractère ";" ou un commentaire pour ajouter une autre ligne de SQL à notre requête préparée.

- **Secrets "hard codés" dans le code**

Et pour résoudre le problème du secret "hard codé", la vulnérabilité "avoid hardcoded config SECRET\_KEY", nous l'avons déplacé dans un fichier.

```

app = Flask('vulpy')
#app.config['SECRET_KEY'] = 'aaaaaaa'

with open('secret.txt', 'r') as f:
    app.config['SECRET_KEY'] = f.read().strip()

```

Cette correction est donnée à titre d'exemple, il serait nécessaire d'utiliser un gestionnaire de secrets comme [Infisical](#) dans une application utilisée en production. Il est évidemment nécessaire de supprimer le code précédent commenté.

## Exercice 4

Nous avons ici choisi d'utiliser un code vulnérable à une XSS.

```

from flask import Flask, request, render_template_string
app = Flask(__name__)

@app.route('/')
def index():
    name = request.args.get('name')
    template = f"<h1>Welcome {name}!</h1>"
    return template

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

```

Seule la vulnérabilité debug enabled est repérée par notre SAST.

3

debug-enabled

Security

High

</> Python

Detected Flask app with debug=True. Do not deploy to production with this flag enabled as it will leak sensitive information. Instead, consider using Flask configuration variables or setting `debug` using system

Show more

26m	ex_4/app.py:11	main	Details
19d	ex_3/vulpy-ssl.py:32	main	Details
19d	ex_3/vulpy.py:58	main	Details

La vulnérabilité XSS vulnérabilité n'est pas détectée par le SAST puisque ce n'est pas une faille de code, mais c'est une faille d'implémentation.

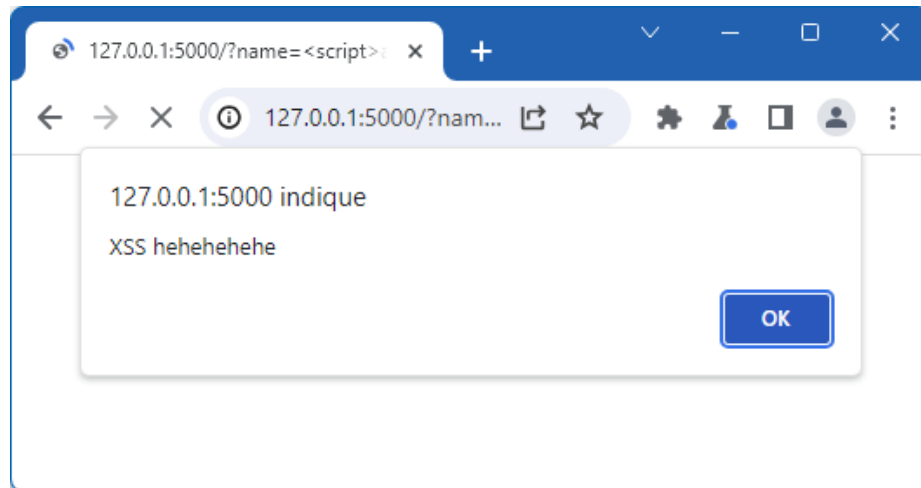
En exécutant une requête spécifique avec Burp, on remarque que le code est bien vulnérable :

Request	Response
Pretty Raw Hex	Pretty Raw Hex Render
<pre> 1 GET /?name=%3Cscript%3Ealert(1)%3C/script%3E HTTP/1.1 2 Host: localhost:5000 3 sec-ch-ua: "Chromium";v="121", "Not A(Brand";v="99" 4 sec-ch-ua-mobile: ?0 5 sec-ch-ua-platform: "Windows" 6 Upgrade-Insecure-Requests: 1 7 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/121.0.6167.160 Safari/537.36 8 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7 9 Sec-Fetch-Site: none 10 Sec-Fetch-Mode: navigate 11 Sec-Fetch-User: ?1 12 Sec-Fetch-Dest: document 13 Accept-Encoding: gzip, deflate, br 14 Accept-Language: fr-FR,fr;q=0.9,en-US;q=0.8,en;q=0.7 15 Connection: close </pre>	<pre> 1 HTTP/1.1 200 OK 2 Server: Werkzeug/3.0.1 Python/3.11.0rc1 3 Date: Sun, 11 Feb 2024 15:53:15 GMT 4 Content-Type: text/html; charset=utf-8 5 Content-Length: 43 6 Connection: close 7 8 &lt;h1&gt;   Welcome &lt;script&gt;     alert(1)   &lt;/script&gt;   ! &lt;/h1&gt; </pre>

En effet, si on met une balise script dans notre attribut name, le code JavaScript est bien récupéré.

Florian Latapie, Élis Roux

6/6



L'ensemble des analyses semgrep sont disponibles ici :  
[semgrep.dev/orgs/florian-latapie-etu-unice-fr/findings](https://semgrep.dev/orgs/florian-latapie-etu-unice-fr/findings)

**Fin du document**