# TP SAST

## Exercice 1

Part_1: **django_extra_used**

Part_2: **exec_used**

Part_3: **hardcoded_password_string**

Changer la sévérité pour HIGH pas trouvé

Part_4: **blacklist**

B310

Part_5: **hardcoded_sql_expressions**

## Exercice 2

semgrep --config p/default doit être utilisé

```
 ┌──────────────────────────────────┐
| 16 Code Findings |
 └──────────────────────────────────┘

    SQLi.cs
   ❯❯❱ csharp.dotnet-core.sqli.systemdata-taint.systemdata-taint
          Untrusted input might be used to build a database query, which can lead
to a SQL injection
          vulnerability. An attacker can execute malicious SQL statements and gain
unauthorized access to
          sensitive data, modify, delete data, or execute arbitrary system
commands. To prevent this
          vulnerability, use prepared statements that do not concatenate user-
controllable strings and use
          parameterized queries where SQL commands and user data are strictly
separated. Also, consider using
          an object-relational (ORM) framework to operate with safer abstractions.
          Details: https://sg.run/4EpL

           25┆ using (SqlCommand cmd = new SqlCommand("SELECT * FROM
users WHERE userId = '" + id +
              "'"))

   ❯❯❱ csharp.lang.security.sqli.csharp-sqli.csharp-sqli
          Detected a formatted string in a SQL statement. This could lead to SQL
```

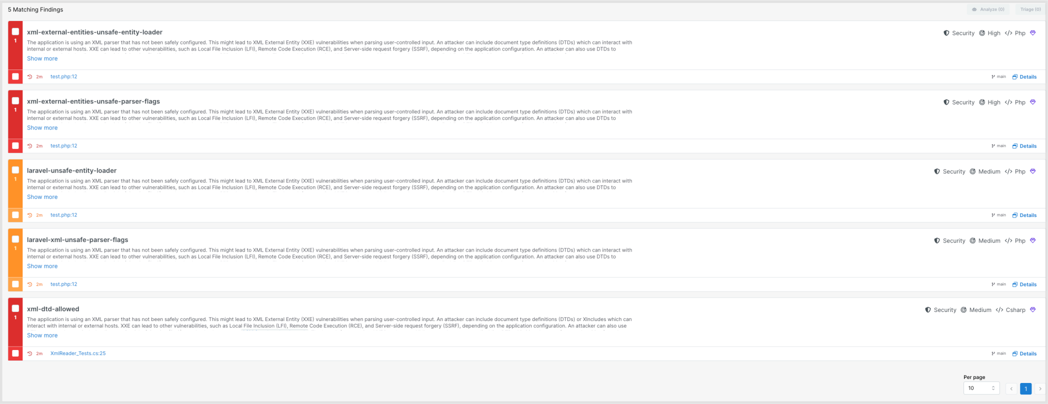
```
injection if variables in the
         SQL statement are not properly sanitized. Use a prepared statements
instead. You can obtain a
         PreparedStatement using 'SqlCommand' and 'SqlParameter'.
         Details: https://sg.run/d2Xd

          25┆ using (SqlCommand cmd = new SqlCommand("SELECT * FROM
users WHERE userId = '" + id +
            "'"))

    blindsqli.php
   ❯❯❱ php.lang.security.injection.tainted-sql-string.tainted-sql-string
         User data flows into this manually-constructed SQL string. User data can
be safely inserted into SQL
         strings using prepared statements or an object-relational mapper (ORM).
Manually-constructed SQL
         strings is a possible indicator of SQL injection, which could let an attacker
steal or manipulate
         data from the database. Instead, use prepared statements (`$mysqli-
>prepare("INSERT INTO test(id,
         label) VALUES (?, ?)");`) or a safe library.
         Details: https://sg.run/lZYG

          17┆ $count = $db->querySingle('select count(*) from secrets where
id = ' . $_GET['id']);

    example1.rb
   ❯❯❱ ruby.rails.security.injection.tainted-sql-string.tainted-sql-string
         Detected user input used to manually construct a SQL string. This is
usually bad practice because
         manual construction could accidentally result in a SQL injection. An
attacker could use a SQL
         injection to steal or modify contents of the database. Instead, use a
parameterized query which is
         available by default in most database engines. Alternatively, consider
using an object-relational
         mapper (ORM) such as ActiveRecord which will protect your queries.
         Details: https://sg.run/Y85o

           7┆ con.query 'UPDATE users set name = ' + params[:name] +

    example2.js
    ❯❱ problem-based-packs.insecure-transport.js-node.using-http-
server.using-http-server
         Checks for any usage of http servers instead of https servers.
Encourages the usage of https
         protocol instead of http, which does not have TLS and is therefore
```

```
unencrypted. Using http can lead
          to man-in-the-middle attacks in which the attacker is able to read
sensitive information.
          Details: https://sg.run/x1zL

            5┆ var req = http.request(options, function(res)

    mysql.js
   ❯❯❱ javascript.express.security.injection.tainted-sql-string.tainted-sql-
string
          Detected user input used to manually construct a SQL string. This is
usually bad practice because
          manual construction could accidentally result in a SQL injection. An
attacker could use a SQL
          injection to steal or modify contents of the database. Instead, use a
parameterized query which is
          available by default in most database engines. Alternatively, consider
using an object-relational
          mapper (ORM) such as Sequelize which will protect your queries.
          Details: https://sg.run/66ZL

           19┆ sql : "SELECT * FROM users WHERE id=" + userId

    ❯❱ javascript.express.mysql.express-mysql-sqli.express-mysql-sqli
          Untrusted input might be used to build a database query, which can lead
to a SQL injection
          vulnerability. An attacker can execute malicious SQL statements and gain
unauthorized access to
          sensitive data, modify, delete data, or execute arbitrary system
commands. To prevent this
          vulnerability, use prepared statements that do not concatenate user-
controllable strings and use
          parameterized queries where SQL commands and user data are strictly
separated. Also, consider using
          an object-relational (ORM) framework to operate with safer abstractions.
          Details: https://sg.run/306W

           21┆ connection.query(query,(err, result) => {
            ⋮┆----------------------------------------
           28┆ connection.query("SELECT * FROM users WHERE id=" + userId,
(err, result) => {

   ❯❯❱ javascript.express.security.injection.tainted-sql-string.tainted-sql-
string
          Detected user input used to manually construct a SQL string. This is
usually bad practice because
          manual construction could accidentally result in a SQL injection. An
```

```
attacker could use a SQL
        injection to steal or modify contents of the database. Instead, use a
parameterized query which is
        available by default in most database engines. Alternatively, consider
using an object-relational
        mapper (ORM) such as Sequelize which will protect your queries.
        Details: https://sg.run/66ZL

         28┆ connection.query("SELECT * FROM users WHERE id=" + userId,
(err, result) => {

   ❯❱ javascript.express.mysql.express-mysql-sqli.express-mysql-sqli
        Untrusted input might be used to build a database query, which can lead
to a SQL injection
        vulnerability. An attacker can execute malicious SQL statements and gain
unauthorized access to
        sensitive data, modify, delete data, or execute arbitrary system
commands. To prevent this
        vulnerability, use prepared statements that do not concatenate user-
controllable strings and use
        parameterized queries where SQL commands and user data are strictly
separated. Also, consider using
        an object-relational (ORM) framework to operate with safer abstractions.
        Details: https://sg.run/306W

         35┆ connection.query({
         36┆    sql : "SELECT * FROM users WHERE id=" +userId
         37┆ },(err, result) => {

  ❯❯❱ javascript.express.security.injection.tainted-sql-string.tainted-sql-
string
        Detected user input used to manually construct a SQL string. This is
usually bad practice because
        manual construction could accidentally result in a SQL injection. An
attacker could use a SQL
        injection to steal or modify contents of the database. Instead, use a
parameterized query which is
        available by default in most database engines. Alternatively, consider
using an object-relational
        mapper (ORM) such as Sequelize which will protect your queries.
        Details: https://sg.run/66ZL

         36┆ sql : "SELECT * FROM users WHERE id=" +userId

    sql.js
    ❱ javascript.express.security.audit.express-check-csurf-middleware-
usage.express-check-csurf-middleware-usage
```

```
          A CSRF middleware was not detected in your express application. Ensure you are either using one such
          as `csurf` or `csrf` (see rule references) and/or you are properly doing CSRF validation in your
          routes with a token or cookies.
          Details: https://sg.run/BxzR

            3┆ var app = express()

    ❯❱ javascript.sequelize.node-sequelize-hardcoded-secret-argument.node-sequelize-hardcoded-secret-argument
          A secret is hard-coded in the application. Secrets stored in source code, such as credentials,
          identifiers, and other types of sensitive data, can be leaked and used by internal or external
          malicious actors. Use environment variables to securely provide credentials and other secrets or
          retrieve them from a secure vault or Hardware Security Module (HSM).
          Details: https://sg.run/E7ZB

            5┆ const sequelize = new Sequelize('database', 'username', 'password', {

   ❯❯❱ javascript.sequelize.security.audit.sequelize-injection-express.express-sequelize-injection
          Detected a sequelize statement that is tainted by user-input. This could lead to SQL injection if
          the variable is user-controlled and is not properly sanitized. In order to prevent SQL injection, it
          is recommended to use parameterized queries or prepared statements.
          Details: https://sg.run/gjoe

           11┆ sequelize.query('SELECT * FROM Products WHERE name LIKE ' + req.body.username);

   ❯❯❱ javascript.express.security.injection.tainted-sql-string.tainted-sql-string
          Detected user input used to manually construct a SQL string. This is usually bad practice because
          manual construction could accidentally result in a SQL injection. An attacker could use a SQL
          injection to steal or modify contents of the database. Instead, use a parameterized query which is
          available by default in most database engines. Alternatively, consider using an object-relational
          mapper (ORM) such as Sequelize which will protect your queries.
          Details: https://sg.run/66ZL
          11┆ sequelize.query('SELECT * FROM Products WHERE name LIKE ' +
req.body.username);

    sqli.php
   ❯❯❱ php.lang.security.injection.tainted-sql-string.tainted-sql-string
          User data flows into this manually-constructed SQL string. User data can
be safely inserted into SQL
          strings using prepared statements or an object-relational mapper (ORM).
Manually-constructed SQL
          strings is a possible indicator of SQL injection, which could let an attacker
steal or manipulate
          data from the database. Instead, use prepared statements (`$mysqli-
>prepare("INSERT INTO test(id,
          label) VALUES (?, ?)");`) or a safe library.
          Details: https://sg.run/lZYG

          17┆ $count = $db->querySingle('select count(*) from secrets where
id = ' . $_GET['id']);
```

1



5 Matching Findings

xml-external-entities-unsafe-entity-loader

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

Security High Php

2m test.php:12 main

xml-external-entities-unsafe-parser-flags

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

Security High Php

2m test.php:12 main

laravel-unsafe-entity-loader

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

Security Medium Php

2m test.php:12 main

laravel-xml-unsafe-parser-flags

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

Security Medium Php

2m test.php:12 main

xml-dtd-allowed

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

Security Medium Csharp

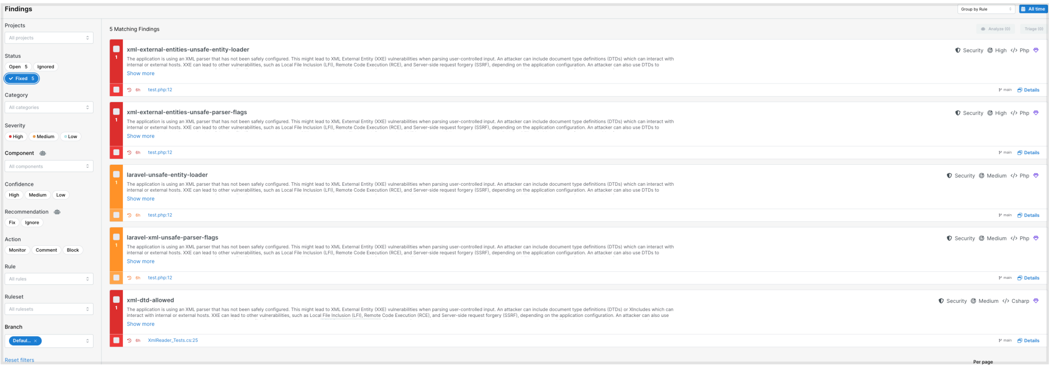
2m XmlReader_Tests.cs:25 main

Per page: 10

```
1 % semgrep --config p/default
/Users/olivier/Library/Python/3.9/lib/python/site-packages/urllib3/
__init__.py:35: NotOpenSSLWarning: urllib3 v2 only supports OpenSSL 1.1.1+,
currently the 'ssl' module is compiled with 'LibreSSL 2.8.3'. See: https://
github.com/urllib3/urllib3/issues/3020
  warnings.warn(


 ┌────────────────────────┐
| Scan Status |
 └────────────────────────┘
  Scanning 5 files (only git-tracked) with 1751 Code rules:

  CODE RULES

  Language      Rules   Files         Origin      Rules
  ─────────────────────────────────────────────────────
────────────────────────────────────

  <multilang>      55      10         Community    1095
  csharp           67       2        Pro rules      656
  php              61       2
  js              241       1


  SUPPLY CHAIN RULES

  💎 Run `semgrep ci` to find dependency
    vulnerabilities and advanced cross-file findings.


  PROGRESS

────────────────────────────────────────────────────────
──── 100% 0:00:00


 ┌──────────────────────────────┐
| 5 Code Findings |
 └──────────────────────────────┘


    XmlReader_Tests.cs
   ❯❯❱ csharp.dotnet-core.xxe.xml-dtd-allowed.xml-dtd-allowed
           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.
          Details: https://sg.run/0zlv

           25┆ XmlReader reader = XmlReader.Create(stream, settings);

    test.php
   ❯❯❱ php.lang.security.xml-external-entities-unsafe-entity-loader.xml-
external-entities-unsafe-entity-loader
          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.
          Details: https://sg.run/5qPA

           12┆ $document->loadXML($xml, LIBXML_NOENT |
LIBXML_DTDLOAD);

   ❯❯❱ php.lang.security.xml-external-entities-unsafe-parser-flags.xml-
external-entities-unsafe-parser-flags
          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.
          Details: https://sg.run/GJxp

           12┆ $document->loadXML($xml, LIBXML_NOENT | LIBXML_DTDLOAD);

    ❯❱ php.laravel.security.laravel-unsafe-entity-loader.laravel-unsafe-entity-loader
          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.
          Details: https://sg.run/lePG
          12┆ $document->loadXML($xml, LIBXML_NOENT |
LIBXML_DTDLOAD);

    ❯❱ php.laravel.security.laravel-xml-unsafe-parser-flags.laravel-xml-
unsafe-parser-flags
          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.
          Details: https://sg.run/YoAo

          12┆ $document->loadXML($xml, LIBXML_NOENT |
LIBXML_DTDLOAD);



┌──────────────┐
│ Scan Summary │
└──────────────┘
Some files were skipped or only partially analyzed.
  Scan was limited to files tracked by git.

Ran 422 rules on 5 files: 5 findings.
```



5 Matching Findings

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 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

Show more

6h test.php:12 — main — Details

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 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

Show more

6h test.php:12 — main — Details

laravel-unsafe-entity-loader — Security · Medium · Php

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

Show more

6h test.php:12 — main — Details

laravel-xml-unsafe-parser-flags — Security · Medium · Php

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

Show more

6h test.php:12 — main — Details

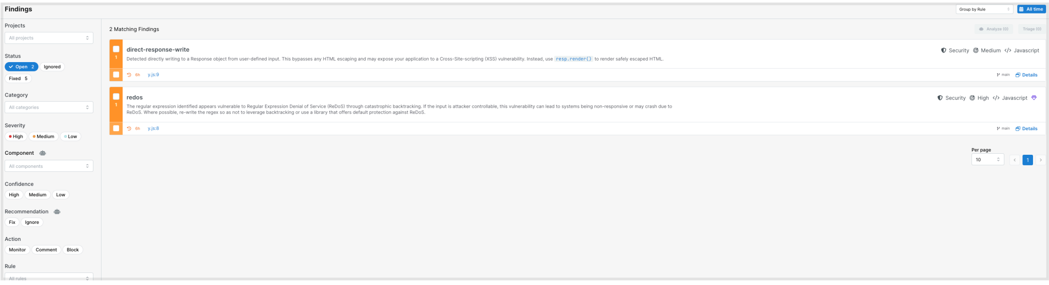
xml-dtd-allowed — Security · Medium · Csharp

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

Show more

6h XmlReader_Tests.cs:25 — main — Details

2



2 Matching Findings

direct-response-write — Security · Medium · Javascript

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 `resp.render()` to render safely escaped HTML.

6h y.js:9 — main — Details

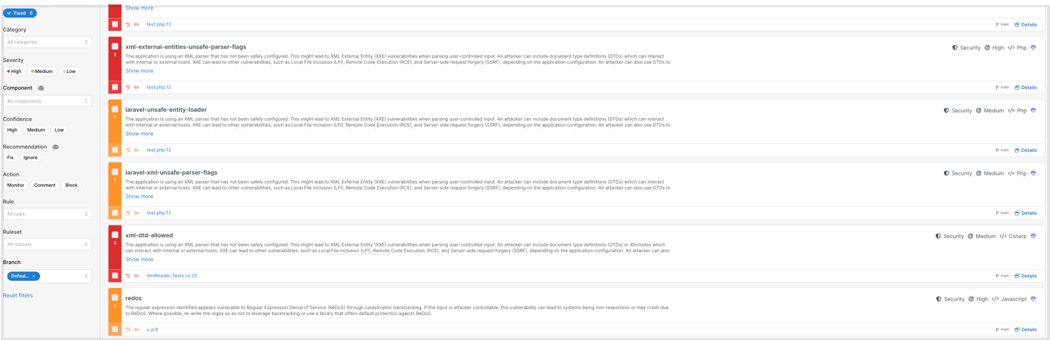
redos — Security · High · Javascript

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 non-responsive or may crash due to ReDoS. Where possible, re-write the regex so as not to leverage backtracking or use a library that offers default protection against ReDoS.

6h y.js:8 — main — Details

```
2 % semgrep --config p/default
/Users/olivier/Library/Python/3.9/lib/python/site-packages/urllib3/
__init__.py:35: NotOpenSSLWarning: urllib3 v2 only supports OpenSSL 1.1.1+,
currently the 'ssl' module is compiled with 'LibreSSL 2.8.3'. See: https://
github.com/urllib3/urllib3/issues/3020
  warnings.warn(


 ┌─────────────┐
| Scan Status |
 └─────────────┘
  Scanning 1 file (only git-tracked) with 1751 Code rules:

  CODE RULES

  Language      Rules   Files          Origin      Rules
 ─────────────────────────────────────────────────────
─────────────────────────────────

  <multilang>     55       2          Community    1095
  js             241       1        Pro rules      656



  SUPPLY CHAIN RULES
  💎 Run `semgrep ci` to find dependency
    vulnerabilities and advanced cross-file findings.


  PROGRESS


━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
━━━━━ 100% 0:00:00


  ┌──────────────────┐
│ 2 Code Findings │
  └──────────────────┘


    y.js
   ❯❱ javascript.express.redos.express-redos.redos
          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 non-responsive or may crash due to ReDoS. Where
possible, re-write the regex
          so as not to leverage backtracking or use a library that offers default
protection against ReDoS.
          Details: https://sg.run/2ZLz5

            8┆ let match = r.test(req.params.id);

   ❯❱ javascript.express.security.audit.xss.direct-response-write.direct-
response-write
          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
          'resp.render()' to render safely escaped HTML.
          Details: https://sg.run/vzGl

            9┆ res.send(match)



  ┌───────────────┐
│ Scan Summary │
  └───────────────┘
Some files were skipped or only partially analyzed.
  Scan was limited to files tracked by git.
```

```
Ran 296 rules on 1 file: 2 findings.
olivier@MacBook-Air-de-Olivier 2 % ls
y.js
```



Show more

6h test.php:12 main Details

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 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

Show more

6h test.php:12 main Details

laravel-unsafe-entity-loader — Security Medium Php

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

Show more

6h test.php:12 main Details

laravel-xml-unsafe-parser-flags — Security Medium Php

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

Show more

6h test.php:12 main Details

xml-dtd-allowed — Security Medium Csharp

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

Show more

6h XmlReader_Tests.cs:25 main Details

redos — Security High Javascript

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 non-responsive or may crash due to ReDoS. Where possible, re-write the regex so as not to leverage backtracking or use a library that offers default protection against ReDoS.

6h y.js:8 main Details

Redos est fix, la sévérité est en soit médium mais la confidence étant high c’était le plus proche de la consigne

3

```
3 % semgrep --config p/default
/Users/olivier/Library/Python/3.9/lib/python/site-packages/urllib3/__init__.py:35: NotOpenSSLWarning: urllib3 v2 only supports OpenSSL 1.1.1+, currently the 'ssl' module is compiled with 'LibreSSL 2.8.3'. See: https://github.com/urllib3/urllib3/issues/3020
  warnings.warn(


 ┌──────────────────────────┐
│ Scan Status │
 └──────────────────────────┘
  Scanning 6 files (only git-tracked) with 1751 Code rules:

  CODE RULES

  Language      Rules   Files          Origin      Rules
 ─────────────────────────────────────────────────────────
───────────────────────────────────────

  <multilang>      75      12          Community    1095
  php             61       2          Pro rules      656
  js             241       1
  java            231       1
  csharp            67       1
  html              1       1
 SUPPLY CHAIN RULES

 💎 Run `semgrep ci` to find dependency
   vulnerabilities and advanced cross-file findings.


 PROGRESS

━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
━━━━━━━ 100% 0:00:00


 ┌─────────────────┐
│ 4 Code Findings │
 └─────────────────┘

   dom.php
   ❯❱ php.lang.security.taint-unsafe-echo-tag.taint-unsafe-echo-tag
          Found direct access to a PHP variable wihout HTML escaping inside an inline PHP statement setting
          data from `$_REQUEST[...]`. When untrusted input can be used to tamper with a web page rendering, it
          can lead to a Cross-site scripting (XSS) vulnerability. XSS vulnerabilities occur when untrusted
          input executes malicious JavaScript code, leading to issues such as account compromise and sensitive
          information leakage. To prevent this vulnerability, validate the user input, perform contextual
          output encoding or sanitize the input. In PHP you can encode or sanitize user input with
          `htmlspecialchars` or use automatic context-aware escaping with a template engine such as Latte.
          Details: https://sg.run/RlGe

           11┆ Hi, <?= $_GET['name']; ?>

   example.php
   ❯❱ php.lang.security.taint-unsafe-echo-tag.taint-unsafe-echo-tag
          Found direct access to a PHP variable wihout HTML escaping inside an inline PHP statement setting
          data from `$_REQUEST[...]`. When untrusted input can be used to tamper with a web page rendering, it
          can lead to a Cross-site scripting (XSS) vulnerability. XSS vulnerabilities occur when untrusted
          input executes malicious JavaScript code, leading to issues such as account compromise and sensitive
          information leakage. To prevent this vulnerability, validate the user input, perform contextual
          output encoding or sanitize the input. In PHP you can encode or sanitize user input with
          `htmlspecialchars` or use automatic context-aware escaping with a template engine such as Latte.
          Details: https://sg.run/RlGe

            7┆ echo 'Hello, ' . $_GET['name']

    express.js
    ❯❱ javascript.express.security.injection.raw-html-format.raw-html-format
          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 as DOMPurify to sanitize the HTML within.
          Details: https://sg.run/5DO3

            6┆ res.send('<h1> Hello :'+ name +"</h1>")

    ❯❱ javascript.express.security.audit.xss.direct-response-write.direct-response-write
          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
          'resp.render()' to render safely escaped HTML.
          Details: https://sg.run/vzGl

            6┆ res.send('<h1> Hello :'+ name +"</h1>")



┌──────────────┐
│ Scan Summary │
└──────────────┘
Some files were skipped or only partially analyzed.
  Scan was limited to files tracked by git.

Ran 673 rules on 6 files: 4 findings.
```

4

```
4 % semgrep --config p/default /Users/olivier/Library/Python/3.9/lib/python/site-packages/urllib3/
```

```
__init__.py:35: NotOpenSSLWarning: urllib3 v2 only supports OpenSSL 1.1.1+,
currently the 'ssl' module is compiled with 'LibreSSL 2.8.3'. See: https://
github.com/urllib3/urllib3/issues/3020
  warnings.warn(


 ┌──────────────────────┐
│ Scan Status │
 └──────────────────────┘
  Scanning 3 files (only git-tracked) with 1751 Code rules:

  CODE RULES

  Language      Rules   Files          Origin      Rules
 ──────────────────────────────────────────────────────────
─────────────────────────────────────

  <multilang>      55       6          Community    1095
  js             241       2         Pro rules     656
  php              61       1


  SUPPLY CHAIN RULES

  💎 Run `semgrep ci` to find dependency
     vulnerabilities and advanced cross-file findings.


  PROGRESS


──────────────────────────────────────────────────────────────────────────
────── 100% 0:00:00


 ┌──────────────────────────────┐
│ 2 Code Findings │
 └──────────────────────────────┘

    aa.js
    ❯❱ javascript.express.open-redirect-deepsemgrep.open-redirect-
deepsemgrep
          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 lead to issues such as Cross-site scripting
(XSS) or redirecting to a
          malicious domain for activities such as phishing to capture users'
```

```
credentials. To prevent this
          vulnerability perform strict input validation of the domain against an allowlist of approved
          domains. Notify a user in your application that they are leaving the website. Display a domain where
          they are redirected to the user. A user can then either accept or deny the redirect to an untrusted
          site.
          Details: https://sg.run/BDbW

           17┆ res.redirect(url);

    koa.js
    ❯❱ javascript.express.open-redirect-deepsemgrep.open-redirect-deepsemgrep
          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 lead to issues such as Cross-site scripting (XSS) or redirecting to a
          malicious domain for activities such as phishing to capture users' credentials. To prevent this
          vulnerability perform strict input validation of the domain against an allowlist of approved
          domains. Notify a user in your application that they are leaving the website. Display a domain where
          they are redirected to the user. A user can then either accept or deny the redirect to an untrusted
          site.
          Details: https://sg.run/BDbW

            8┆ ctx.redirect(url);



┌──────────────┐
│ Scan Summary │
└──────────────┘
Some files were skipped or only partially analyzed.
  Scan was limited to files tracked by git.
  Partially scanned: 1 files only partially analyzed due to parsing or internal Semgrep errors

Ran 356 rules on 3 files: 2 findings.

Github Personal Access token:
```

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