PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
Ministry of Higher Education and Scientific Research
University of Skikda — Department of Computer Science
Master's in Information Security — Semester 3 (M2) — UEF322
Web and Mobile Application Security
Updated on 2026-09-22 at 23:28:28

Lab 5 — Deploying and Testing a CSP Policy

Theoretical reminder

Chapter 5 presents CSP as a "defense in depth": it does not prevent the injection itself, it prevents the injected script from **running**. This lab demonstrates it strikingly: you will deliberately reproduce the same stored XSS as in Lab 3, then observe that a single HTTP header line is enough to neutralize it without touching the underlying vulnerable code – and, conversely, you will see that *report-only* mode protects against nothing until you switch to blocking mode.

Objective

Configure a CSP on a local page, observe an injected script being blocked, then measure the effect of *report-only* mode (see chapter 5).

Prerequisites

Having studied chapter 5 (CSP, directives, nonces) and completed Lab 3 (stored XSS).

Tools and installation

- Python 3 with Flask: `pip install flask`.
- A browser with DevTools (*Console* tab).

Procedure

Step 1 — Create a deliberately vulnerable page

Create an `app.py` file:

```
from flask import Flask, request

app = Flask(__name__)
```

```
comment = ""

@app.route('/', methods=['GET', 'POST'])
def index():
    global comment
    if request.method == 'POST':
        comment = request.form['text']
    return f"""
    <form method="POST">
      <input name="text">
      <button>Send</button>
    </form>
    <div>Comment: {comment}</div>
    """

app.run(port=5000, debug=True)
```

Run it with `python app.py` and open `http://localhost:5000`. This page reinjects the comment **without any encoding**: it is the same flaw as the one exploited in Lab 3.

Step 2 — Confirm the XSS without CSP

In the field, enter `<script>alert('XSS')</script>` and submit. An alert box should appear: the script runs, no CSP is active.

Step 3 — Enable a blocking CSP

Modify the route to add the header:

```
from flask import Flask, request, make_response

@app.route('/', methods=['GET', 'POST'])
def index():
    global comment
    if request.method == 'POST':
        comment = request.form['text']
    resp = make_response(f"""..."""")  # same HTML content as in step 1
    resp.headers['Content-Security-Policy'] = "default-src 'self'"
    return resp
```

Restart the server, reload the page, and submit `<script>alert('XSS')</script>` again. The script no longer runs. Open the console: note the CSP error message displayed by the browser.

Step 4 — Switch to *report-only* mode

Replace the header with `Content-Security-Policy-Report-Only` (same value). Reload and submit the same payload again: the alert reappears, but a violation is logged in the console. Note the difference in behavior.

Step 5 — Block embedding in an iframe

Add the `frame-ancestors 'none'` directive to the CSP header (blocking mode). Create a second static HTML page containing `<iframe src="http://localhost:5000"></iframe>` and open it: the iframe should stay empty, with an error in the console.

Comprehension questions

1. The Flask route code remained vulnerable (no encoding added) from the beginning to the end of the lab. Why does the XSS no longer run from step 3 onwards?
2. What is the concrete difference in risk between leaving a CSP in *report-only* mode indefinitely and switching it to blocking mode?
3. How does the `frame-ancestors 'none'` directive in step 5 constitute a defense against the clickjacking seen in chapter 3?

Deliverable

Three screenshots (without CSP – with a blocking CSP – with a report-only CSP) and one sentence explaining why the CSP alone would not have prevented the XSS injection itself (see chapter 3, protection techniques).