Web Security

This lab deals with understanding how scanners enumerate attack surfaces and spot common vulnerabilities. We will be using Wapiti (web scanning tool) to identify weak points in a web application's input handling and response behaviour.

Main Vulnerabilities

Main vulnerabilities that are of high impact and easy to miss manuall, wapiti will identify are as follows:

- Cross Site Scripting (XSS)
- SQL Injection
- Command Injection
- File Inclusion
- Insecure File Uploads
- Server-Side Request Forgery (SSRF)

Ethical Consideration

This lab explicitly states that we must only scan the provided sites, and scanning anything else without explicit permission is illegal.

Platform	URL	Purpose
OWASP Juice Shop	https://juice-shop.herokuapp.com	Modern vulnerable app (XSS, SQLi, etc.)
Google Gruyere	https://google-gruyere.appspot.com/123456/ *(replace 123456 with your instance ID)*	Classic Google training app
PentesterLab	https://54.80.249.209/	Simple vulnerable endpoints
Hack The Box Academy	https://academy.hackthebox.com/ *(requires free account)*	Guided vulnerable scenarios

Setting Up Wapiti - The Web Scanning Tool

```
# !pip install wapiti3
# !wapiti --version
```

Preparing The Target

Google Gruyere

It is a purposely vulnerable web applicaion that is used for safe, controlled security testing. It contains flaws such as XSS, injection issues, and insecure file handling, so testers like us can practice and find real vulnerabilities without affecting the live systems. Each user gets their own isolated instance to ensure ethical and contained testing.

Gruyere Instance

Google Gruyere requires us to create a personal instance:

- https://google-gruyere.appspot.com → Create Instance

After visiting the website and creating the Gruyere instance id, the final target looks like this:

- https://google-gruyere.appspot.com/438049961685840348819228436588982022134/

Scanning The Online Google's Learning App

In this step we carried out the following activities:

- Installed and configured the Wapiti (wapiti3) tool to perform automated black-box vulnerability scanning on a controlled target.
- Used a dedicated Google Gruyere instance to ensure the assessment remained isolated and ethically compliant.
- Ran a complete scan, allowing Wapiti to crawl the application, enumerate inputs, and inject payloads.
- Tested for common weaknesses including XSS, SQL injection, command injection, file inclusion, insecure file uploads, and SSRF.

```
!wapiti -u "https://google-gruyere.appspot.com/438049961685840348819228436588982022134/" -f html -o "./gruyere-report.html"
```

Show hidden output

Script For Displaying The Generated Report From The Above Step

This part concerns with reporting the detected vulnerabilities:

- Used the provided Python script to automatically locate the generated Wapiti HTML report.
- Rendered the report inside the environment to confirm the scan executed correctly.
- Reviewed the findings to validate vulnerabilities detected during the scan.

```
import os
from IPython.display import display, HTML
import glob
import pathlib
# Base 'out' name you passed to Wapiti (-o). Update if you used a different name.
out_name = "./gruyere-report.html"

def find_html_report(out):
    # If out is a file and ends with .html, use it
    if os.path.isfile(out) and out.lower().endswith('.html'):
        return os.path.abspath(out)
    # If out is a directory, search for the newest .html inside it
    if os.path.isdir(out):
        matches = glob.glob(os.path.join(out, "*.html"))
        if matches:
            # choose the most recent file
            matches.sort(key=os.path.getmtime, reverse=True)
            return os.path.abspath(matches[0])
    # If out doesn't exist as provided, search for any html file pattern that looks like wapiti output
    candidates = glob.glob(out + "*/*.html") + glob.glob(out + "*.html")
    if candidates:
        candidates.sort(key=os.path.getmtime, reverse=True)
        return os.path.abspath(candidates[0])
    return None

report_path = find_html_report(out_name)

if report_path:
    print("Report file found:", report_path)
    try:
        with open(report_path, 'r', encoding='utf-8') as f:
            html = f.read()
        display(HTML(html))
    except Exception as e:
        print("Could not render HTML inline (error):", e)
        print("Open the file in your browser instead:", report_path)
else:
    print("No HTML report found. Check that the scan produced the report and that 'out_name' matches the -o argument passed
```

Show hidden output

Vulnerability Report

- This scan report shows that the app is missing several key security headers, including CSP, X-Frame-Options, HSTS, and X-Content-Type Options. Without all these, the browsers cannot protect the user from clickjacking, unsafe file handling and insecure connections.
- Cookies were also not protected with HttpOnly or secure flags. Thus, this means they can be accessed by scripts or sent over an encrypted connection.
- There is no input sanitization as Wapiti found one reflected XSS issue and multiple stored XSS vulnerabilities, Thus, this means the app intakes whatever the user enters and displays back on the page without checking. And due to this issue, injected JavaScript gets executed every time an affected page loads, it is a serious security flaw.

Reflection

In this lab, I have learnt the following:

- Black-Box web scanning, like how it enumerates the attack surface without having access to the source code of the web application
- Used Wapiti tool to identify the common web application's vulnerabilities and reviewed the resulting report.
- Ethical boundaries were reinforced as scanning is only valid and allowed in controlled, premitted environments.

- And I used the provided script to load and review the HTML report inside the notebook environment.