



WEBSCAN PRO

– AUTOMATED WEB APPLICATION
SECURITY TESTING TOOL

The Problem

- Web applications are highly vulnerable to attacks such as SQL Injection, XSS, and IDOR due to poor validation and access control.

Solution

WebScanPro is an automated security testing tool that scans a given URL, detects vulnerabilities, and generates a structured security report.

Objectives

- Detect common vulnerabilities (OWASP Top 10)
- Automate penetration testing
- Provide mitigation suggestions
- Generate visual security reports

Tools and test platforms

- **Languages:** Python
- **Libraries:** Requests, BeautifulSoup, Streamlit.
- **Testing Platforms:**
 - DVWA
 - OWASP Juice Shop
 - bWAPP
- **Output:** HTML/PDF Security Report

Week-1

Project Initialization & Setup

Problem

Manual security testing is time-consuming and inconsistent.

Approach

Defined project goals

Studied OWASP Top vulnerabilities

Set up vulnerable applications locally

Week-1

Solution Implemented

Installed DVWA / Juice Shop using XAMPP or Docker

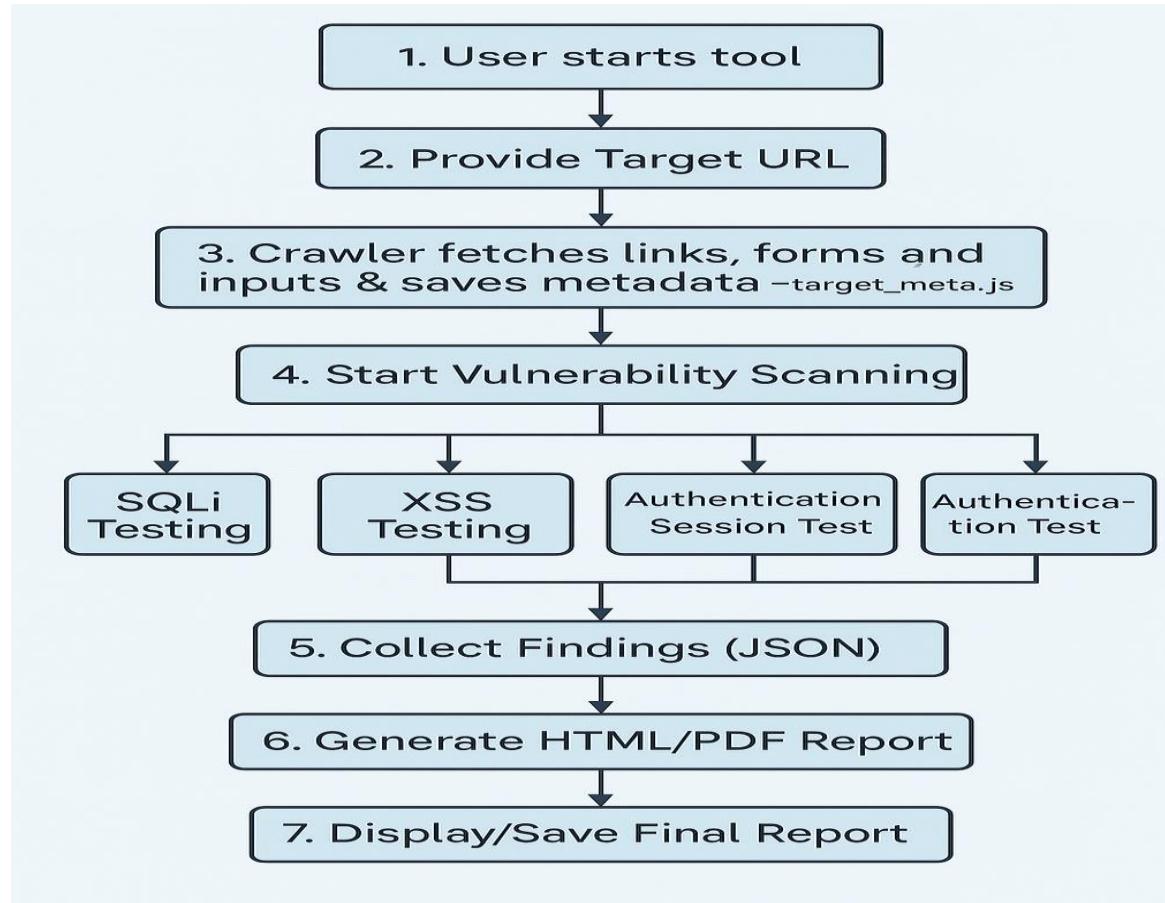
Explored pages, forms, URLs, parameters

Output

Working vulnerable web app environment

Identified attack surfaces

Workflow:



Week-2

Use tools like BeautifulSoup or Selenium to extract interactive elements.

Problem

- Security testing requires knowing all entry points (forms, links, inputs).

Approach

- Web crawling
- HTML parsing

Solution Implemented

- Developed crawler using **BeautifulSoup**

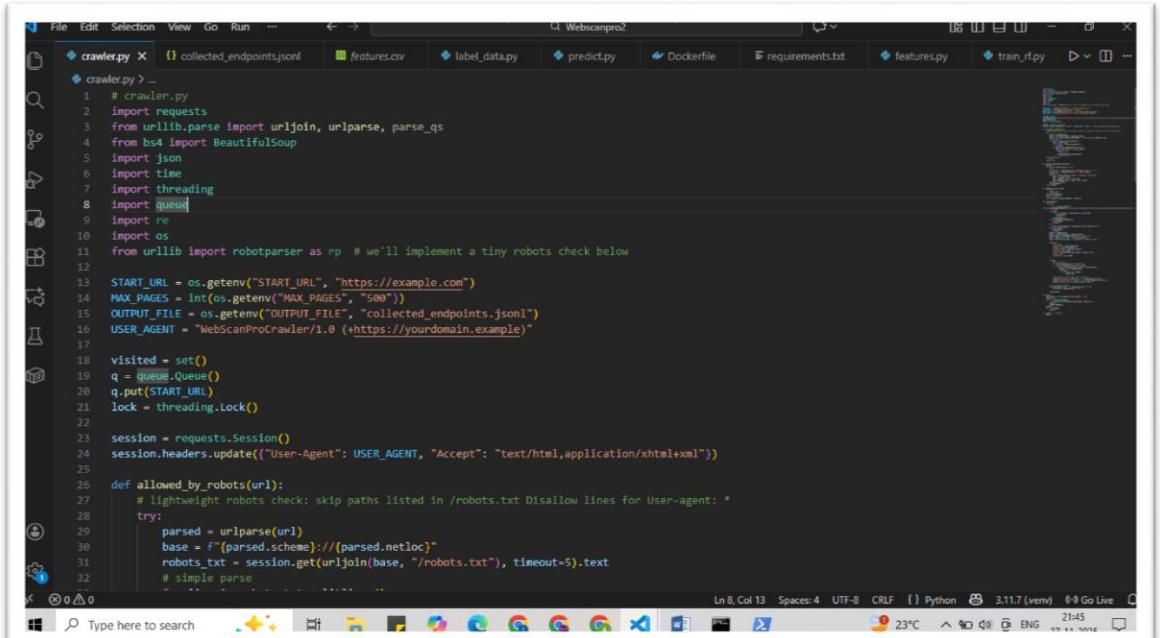
Week-2

Extracted:

- URLs
- Forms
- Input fields
- Parameters

Output

Structured metadata of target website



The screenshot shows a code editor window with a dark theme. The file 'crawler.py' is open, displaying the following code:

```
# crawler.py
# This script is a simple web crawler that collects endpoints from a target website.
# It uses the requests library to send HTTP requests and BeautifulSoup to parse HTML responses.
# It also uses a queue-based approach to handle multiple threads simultaneously.

# Import required libraries
import requests
from urllib.parse import urljoin, urlparse, parse_qs
from bs4 import BeautifulSoup
import json
import time
import threading
import queue
import re
import os
from urllib import_robotparser as rp # we'll implement a tiny robots check below

# Set environment variables
START_URL = os.getenv("START_URL", "https://example.com")
MAX_PAGES = int(os.getenv("MAX_PAGES", "500"))
OUTPUT_FILE = os.getenv("OUTPUT_FILE", "collected_endpoints.json")
USER_AGENT = "WebScanProCrawler/1.0 (<https://yourdomain.example>)"

# Initialize variables
visited = set()
q = queue.Queue()
q.put(START_URL)
lock = threading.Lock()

# Create a session
session = requests.Session()
session.headers.update({"User-Agent": USER_AGENT, "Accept": "text/html,application/xhtml+xml"})

# Define a function to check if a URL is allowed by robots.txt
def allowed_by_robots(url):
    # lightweight robots check: skip paths listed in /robots.txt Disallow lines for User-agent:
    try:
        parsed = urlparse(url)
        base = f"{parsed.scheme}://{parsed.netloc}"
        robots_txt = session.get(urljoin(base, "/robots.txt"), timeout=5).text
        # simple parse
        for line in robots_txt.split("\n"):
            if line.startswith("Disallow: ") and line[10:] == url.path:
                return False
    except Exception as e:
        print(f"Error checking robots.txt for {url}: {e}")
    return True
```

The code is a basic web crawler that starts at a specified URL and follows links to collect endpoints. It uses threads to handle multiple pages simultaneously and respects robots.txt files.

Week-3

SQL Injection Testing

Problem

Improper input validation leads to SQL Injection attacks.

Approach

- Payload-based testing
- Error response analysis

Solution Implemented

- Injected SQL payloads (' OR 1=1 --)
- Analyzed server responses
- Detected vulnerable endpoints

Output

- SQL Injection findings
- Severity classification
- Fix suggestions (Prepared Statements)

Output:

```
File Edit Selection View Go Run ... < > Q Webscanpro2 Dockerfile requirements.txt features.py train_rf.py .gitignore predict.py > ...
collected_endpoints.jsonl features.csv label_data.py predict.py Dockerfile requirements.txt features.py train_rf.py .gitignore predict.py > ...
1 import pandas as pd
2 import joblib
3 import os
4
5 # Load trained model
6 model_path = "models/rf_webscanpro.pkl"
7 features_path = "data/features.csv"
8 output_path = "data/predictions.csv"
9
10 # Check files exist
11 if not os.path.exists(model_path):
12     raise FileNotFoundError(f"Model not found at {model_path}. Train the model first.")
13
14 if not os.path.exists(features_path):
15     raise FileNotFoundError(f"Features file not found at {features_path}. Run features.py first.")
16
17 print("Loading model...")
18 model = joblib.load(model_path)
19
20 print("Loading features...")
21 df = pd.read_csv(features_path)
22
23 print("Preparing feature matrix...")
24 X = df.drop(columns=["url", "label"], errors="ignore")
25
26 print("Running predictions...")
27 preds = model.predict(X)
28 df["prediction"] = preds
```

```
File Edit Selection View Go Run ... < > Webscanpro2 0 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 crawler.py collected_endpoints.jsonl features.csv label_data.py predict.py Dockerfile requirements.txt features.py train_rf.py data > features.status.csv 1 url,status_code,content_length,num_links,num_forms,num_inputs,contains_js,num_query_params,avg_param_name_len,server_header_len,label 2 https://example.com,200,513,0,0,0,0,0,0,0,0,0 3 https://owasp.org/www-project-juice-shop/,200,92703,40,0,0,1,0,0,0,0,0 4 https://owasp.org,200,55315,28,1,1,0,0,0,0,0 5 https://owasp.org/store,200,439,1,0,0,1,0,0,0,0,0 6 https://owasp.org/,200,55315,28,1,1,1,0,0,0,0,0 7 https://owasp.org/cdn-cgi/l/email-protection,404,4713,0,0,0,1,0,0,0,0,0 8 https://owasp.org/www-project-top-ten,200,71194,161,0,0,1,0,0,0,0,0 9 https://owasp.org/projects/,200,139446,278,0,0,1,0,0,0,0,0 10 https://owasp.org/www-project-top-ten/,200,71194,161,0,0,1,0,0,0,0,0 11 https://owasp.org/www-project-application-security-verification-standard/,200,67870,39,0,0,1,0,0,0,0,0 12 https://owasp.org/donate?reponame=www-project-juice-shop&title=OWASP+Juice+Shop,200,42859,17,1,8,1,2,6,5,0,0 13 https://owasp.org/www-project-api-security/,200,55536,84,0,0,1,0,0,0,0,0 14 https://owasp.org/www-project-vulnerable-web-applications-directory,200,219787,37,0,0,1,0,0,0,0,0 15 https://owasp.org/slack_invite,200,26555,17,0,0,1,0,0,0,0,0 16 https://owasp.org/www-project-automated-threats-to-web-applications/,200,92978,80,0,0,1,0,0,0,0,0 17 https://owasp.org/supporters,200,33889,34,0,0,1,0,0,0,0,0 18 https://owasp.org/events/,200,36778,23,0,0,1,0,0,0,0,0 19 https://owasp.org/about/,200,34888,29,0,0,1,0,0,0,0,0 20 https://owasp.org/sitemap,200,30173,57,0,0,1,0,0,0,0,0 21 https://owasp.org/contact/,200,30850,23,0,0,1,0,0,0,0,0 22 https://owasp.org/store/,200,561,0,0,0,1,0,0,0,0,0 23 https://owasp.org/blog/2025/09/03/owasp-membership-drive.html,200,35109,40,0,0,1,0,0,0,0,0 24 https://owasp.org/donate?reponame=owasp.github.io,200,42859,17,1,8,1,1,0,0,0,0 25 https://owasp.org/www-policy/operational/privacy,200,72751,56,0,0,1,0,0,0,0,0 26 https://owasp.org/blog/2025/11/19/stacey-ebbs-marketing-hired.html,200,33891,38,0,0,1,0,0,0,0,0 27 https://owasp.org/blog/2025/09/26/Top10Survey.html,200,31821,37,0,0,1,0,0,0,0,0 28 https://owasp.org/blog/2025/10/01/GSoCRecap.html,200,37216,38,0,0,1,0,0,0,0,0 29 https://owasp.org/www-project-cheat-sheets/,200,35659,21,0,0,1,0,0,0,0,0 30 https://owasp.org/blog/2025/10/03/owasp-certified-secure-software-developer.html,200,30742,37,0,0,1,0,0,0,0,0 31 https://owasp.org/donate?reponame=www-project-top-ten&title=OWASP+Top+Ten,200,42859,17,1,8,1,2,6,5,0,0 32 https://owasp.org/top10/,200,29615,47,1,9,1,0,0,0,0,0
```

Week-4

Cross-Site Scripting (XSS) Testing

Problem

XSS allows attackers to execute malicious scripts in user browsers.

Approach

JavaScript payload injection

DOM & response inspection

Solution Implemented

Injected XSS payloads (<script>alert()</script>)

Detected reflected & stored XSS

Output

Vulnerable input fields identified

Mitigation tips (Input sanitization, CSP)

Output:

The screenshot shows a Microsoft Visual Studio Code (VS Code) interface with the following details:

- Title Bar:** Webscanpro2
- File Explorer:** Shows files: crawler.py, collected_endpoints.jsonl, features.csv, predictions.csv, xss_detector.py (current), requirements-ml.txt, requirements.txt, rf_webscanpro.p.
- Code Editor:** Displays the content of xss_detector.py:

```
backend > xss_detector.py > ...
1 import os
2 import re
3 import requests
4 from bs4 import BeautifulSoup
5 from urllib.parse import urlparse, urljoin, parse_qs, urlencode, urlunparse, ParseResult
6 import time
7 import random
8 import string
9
10 USER_AGENT = "WebScanPro-XSS-Detector/1.0 (+https://example.local)"
11 REQUEST_TIMEOUT = 12
12
13 def random_token(n=12):
14     return "__XSS_TOKEN__" + ''.join(random.choice(string.ascii_letters+string.digits) for _ in range(n)) + "__"
15
```
- Terminal:** Shows the command being run: `PS C:\Users\Anand sai\Desktop\WebScanpro2\backend> python xss_detector.py --url "https://owasp.org/www-project-juice-shop/"`. The output indicates it's running passive checks and provides a JSON response object.
- Status Bar:** Shows the following information: Ln 12, Col 1, Spaces: 4, UTF-8, CRLF, Python 3.11.7 (.venv), Go Live, Date: Fri, 05 Dec 2025 12:47:40 GMT, Temperature: 26°C Mostly cloudy, ENG, 18:17, Date: 05-12-2025.

Week-5

Authentication & Session Testing

Problem

Weak authentication leads to account compromise.

Approach

Credential testing

Session analysis

Solution Implemented

Tested weak/default credentials

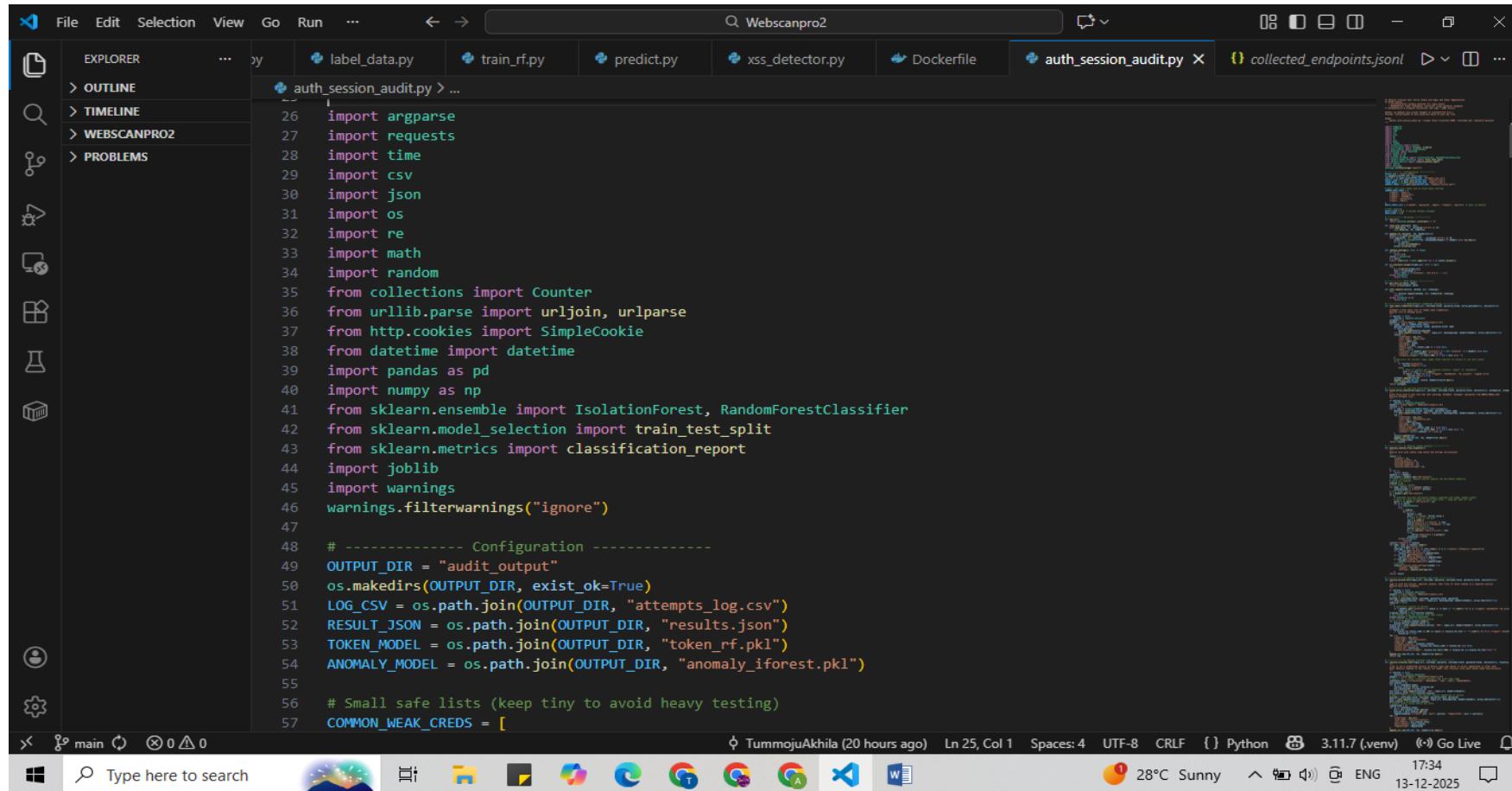
Checked cookies & session fixation

Simulated brute-force attempts

Output:

Authentication flaws

Secure session recommendations



The screenshot shows a code editor interface with the title bar "Webscanpro2". The left sidebar includes icons for Explorer, Outline, Timeline, and Problems. The main area displays a Python script named "auth_session_audit.py". The script imports various modules like argparse, requests, time, csv, json, os, re, math, random, Counter, urljoin, urlparse, SimpleCookie, datetime, pandas, np, IsolationForest, RandomForestClassifier, train_test_split, classification_report, joblib, and warnings. It also uses os.makedirs to create a directory named "audit_output". The script then defines several file paths: LOG_CSV, RESULT_JSON, TOKEN_MODEL, and ANOMALY_MODEL, all relative to the OUTPUT_DIR. A comment indicates small safe lists for testing. Finally, it defines a list of common weak credentials. The status bar at the bottom shows the file name "main", line count "0", and other system information.

```
File Edit Selection View Go Run ... ← → Q Webscanpro2
EXPLORER auth_session_audit.py label_data.py train_rf.py predict.py xss_detector.py Dockerfile auth_session_audit.py collected_endpoints.jsonl ...
OUTLINE
TIMELINE
WEBSCANPRO2
PROBLEMS

26 import argparse
27 import requests
28 import time
29 import csv
30 import json
31 import os
32 import re
33 import math
34 import random
35 from collections import Counter
36 from urllib.parse import urljoin, urlparse
37 from http.cookies import SimpleCookie
38 from datetime import datetime
39 import pandas as pd
40 import numpy as np
41 from sklearn.ensemble import IsolationForest, RandomForestClassifier
42 from sklearn.model_selection import train_test_split
43 from sklearn.metrics import classification_report
44 import joblib
45 import warnings
46 warnings.filterwarnings("ignore")
47
48 # ----- Configuration -----
49 OUTPUT_DIR = "audit_output"
50 os.makedirs(OUTPUT_DIR, exist_ok=True)
51 LOG_CSV = os.path.join(OUTPUT_DIR, "attempts_log.csv")
52 RESULT_JSON = os.path.join(OUTPUT_DIR, "results.json")
53 TOKEN_MODEL = os.path.join(OUTPUT_DIR, "token_rf.pkl")
54 ANOMALY_MODEL = os.path.join(OUTPUT_DIR, "anomaly_iforest.pkl")
55
56 # Small safe lists (keep tiny to avoid heavy testing)
57 COMMON_WEAK_CREDS = [
```

Week-6

Access Control & IDOR Testing

Problem

- Unauthorized users accessing restricted resources.

Approach

- Parameter manipulation
- Role-based testing

Solution Implemented

- Modified user IDs in URLs
- Tested horizontal & vertical privilege escalation

Output

- IDOR vulnerabilities detected
- Access control fixes (RBAC, ABAC)

Output:

The screenshot shows a Microsoft Visual Studio Code (VS Code) window titled "Webscanpro2". The left sidebar displays a file tree with several Python files and configuration files under a folder named "WEBSCANPRO2". The main editor area contains the code for "access_control_analyzer.py". The terminal at the bottom shows the execution of the script and its output, which includes a list of access control issues found during the audit.

```
# access_control_analyzer.py (AUTO LOGIN VERSION)
import requests
import json
import pandas as pd
from difflib import SequenceMatcher
from datetime import datetime
import os

BASE_URL = "http://localhost:3000"
LOGIN_API = "/rest/user/login"

OUTPUT_DIR = "access_control_logs"
os.makedirs(OUTPUT_DIR, exist_ok=True)

# Test credentials (LAB ONLY)
```

Terminal Output:

```
PS C:\Users\Anand sai\Desktop\Webscanpro2> python access_control_analyzer.py
[*] Logging in as Admin...
[*] Logging in as User...
[✓] Tokens obtained successfully
[!] Access Control Issues Found:
```

time	endpoint	issue	severity
0	/rest/user/whoami	Vertical Privilege Escalation	High
1	/rest/user/whoami	Horizontal Privilege Escalation	Medium
2	/rest/admin/application-version	Vertical Privilege Escalation	High
3	/rest/admin/application-version	Horizontal Privilege Escalation	Medium

Week-7

Security Report Generation

Problem

- Raw vulnerability data is hard to understand.

Approach

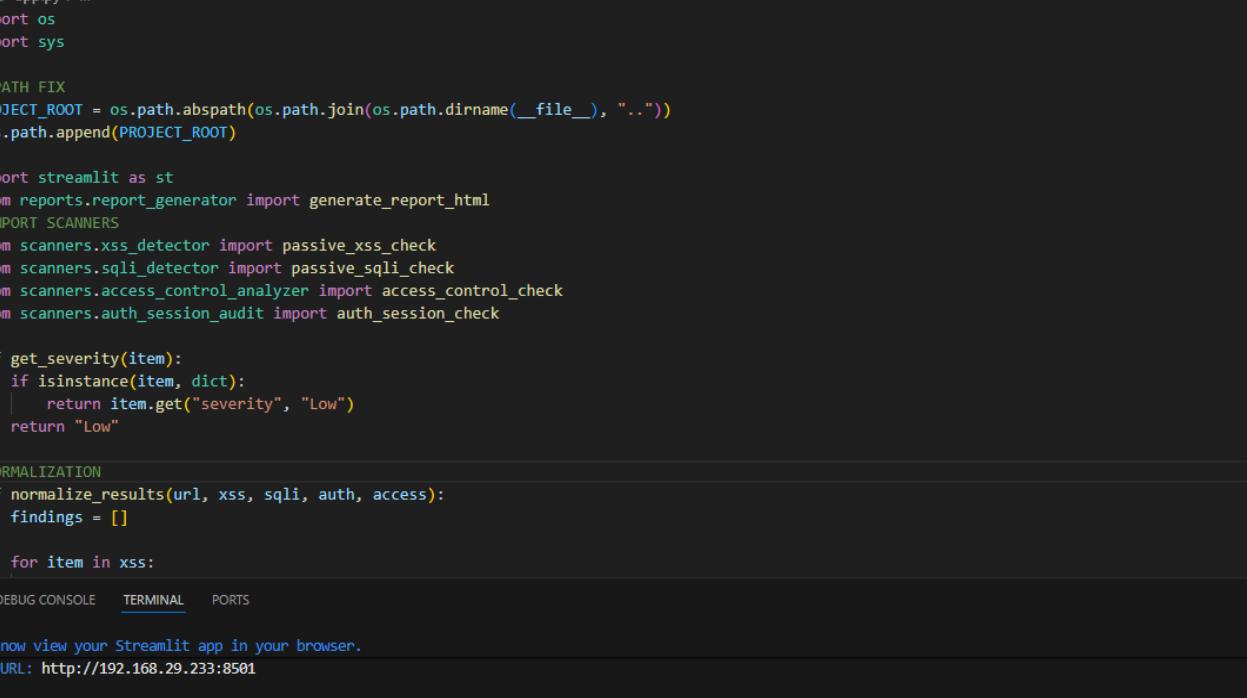
- Structured reporting
- Severity classification

Solution Implemented

- Generated report with:
- Vulnerability Type
- Affected Endpoint
- Severity (Low / Medium / High)
- Mitigation

Output

- HTML / PDF security report
 - Visual summaries



The screenshot shows a Streamlit application running in a browser window. The Streamlit interface includes a sidebar with icons for file operations, search, and help. The main area displays Python code for a Streamlit application, which imports necessary modules and defines functions for severity calculation and result normalization. Below the code, a message indicates the app is running and provides a network URL. The bottom of the screen shows a taskbar with various icons and a system tray.

```
backend > app.py > ...
1 import os
2 import sys
3
4 # PATH FIX
5 PROJECT_ROOT = os.path.abspath(os.path.join(os.path.dirname(__file__), ".."))
6 sys.path.append(PROJECT_ROOT)
7
8 import streamlit as st
9 from reports.report_generator import generate_report_html
10 #IMPORT SCANNERS
11 from scanners.xss_detector import passive_xss_check
12 from scanners.sql_injection import passive_sql_injection_check
13 from scanners.access_control_analyzer import access_control_check
14 from scanners.auth_session_audit import auth_session_check
15
16 def get_severity(item):
17     if isinstance(item, dict):
18         return item.get("severity", "Low")
19     return "Low"
20
21 #NORMALIZATION
22 def normalize_results(url, xss, sql_injection, auth, access):
23     findings = []
24
25     for item in xss:
26
27         if item["severity"] == "High" and item["type"] == "XSS":
```

Final Output:

The screenshot shows a Microsoft Edge browser window with the title bar "WebScanPro" and the address bar "localhost:8501". The page content is a dark-themed interface for a web vulnerability scanner. At the top, there are three buttons: "File change.", "Rerun", and "Always rerun". Below this is the main heading "WebScanPro - Web Vulnerability Scanner" with a padlock icon. A form field labeled "Enter Target URL" contains the value "https://example.com". A large "Start Scan" button is positioned below the URL field. The bottom of the screen shows the Windows taskbar with various pinned icons and the system tray.

Generated Report

The screenshot shows the WebScanPro application running in a browser window. The main content area displays a JSON report with two main sections: 'XSS' and 'SQL Injection'. The 'XSS' section contains one item with details about inline JavaScript detection. The 'SQL Injection' section contains one item with details about session cookie analysis for GitHub's DVWA application. The browser's address bar shows 'localhost:3501'. The taskbar at the bottom includes icons for various Windows applications like File Explorer, Microsoft Word, and Google Chrome.

Vulnerabilités detected

The screenshot shows a 'WebScanPro Security Report' window. At the top, it displays the 'Scan Date: 2025-12-26 15:24:42'. Below this is a 'Summary' section with a bulleted list of findings: Total vulnerabilities: 9, High: 1, Medium: 1, Low: 7. Underneath is a 'Detailed Findings' section containing a table with columns for 'Vulnerability', 'Affected Endpoint', 'Severity', and 'Suggested Mitigation'. The table lists nine entries corresponding to the findings in the JSON report, including XSS, SQL Injection, and various session management issues. The browser's address bar shows 'C:/Users/Anand%20sai/Downloads/WebScanPro_Report.html'. The taskbar at the bottom shows a similar set of application icons as the first screenshot.

Vulnerability	Affected Endpoint	Severity	Suggested Mitigation
Cross-Site Scripting (XSS)	https://github.com/digininja/DVWA	Medium	Apply output encoding, input validation, and Content Security Policy (CSP)
SQL Injection	https://github.com/digininja/DVWA	Low	Use parameterized queries and avoid dynamic SQL
SQL Injection	https://github.com/digininja/DVWA	Low	Use parameterized queries and avoid dynamic SQL
SQL Injection	https://github.com/digininja/DVWA	Low	Use parameterized queries and avoid dynamic SQL
SQL Injection	https://github.com/digininja/DVWA	Low	Use parameterized queries and avoid dynamic SQL
Authentication / Session Management	https://github.com/digininja/DVWA	Low	Use Secure and HttpOnly cookies, proper session expiry, and strong authentication
Authentication / Session Management	https://github.com/digininja/DVWA	High	Use Secure and HttpOnly cookies, proper session expiry, and strong authentication
Authentication / Session Management	https://github.com/digininja/DVWA	Low	Use Secure and HttpOnly cookies, proper session expiry, and strong authentication
Access Control	https://github.com/digininja/DVWA	Low	Implement role-based access control and server-side authorization checks

THANK YOU