IT Management & Audits

Practical Lab Manual

Web Application Security Scanner

Practical P04

Learning Domain

Application Security & Vulnerability Assessment

Course Learning Outcomes

CLO04: Identify and assess common web application vulnerabilities

Unit

Unit III: Application Security Testing

Time Allocation: 3 hours

Learning Mode: Hands-on (80%) + Theory (20%)

Difficulty Level: Intermediate

Web Application Security Scanner

Practical P04

Quick Reference

Practical Code	P04
Practical Name	Web Application Security Scanner
Slot	T/P-4
Duration	3 hours
CLO Mapping	CLO04
Unit	Unit III: Application Security Testing
Delivery Mode	Hands-on Lab
Target Audience	Intermediate Level
India Integration	HIGH
Screenshot Count	5 Required

Prerequisites

- Basic understanding of web concepts (HTTP methods, HTML, URLs)
- Familiarity with Python programming (variables, functions, modules)
- Understanding of client-server architecture and request-response model
- Python 3.8+ installed on your system
- Completion of previous practicals recommended

Tools Required

Tool	**Version**	**Free**	**Notes**
Python	3.8+	✓	Required
Flask	Latest	✓	Vulnerable app framework
requests	Latest	✓	HTTP library for scanner
pip	Latest	✓	Included with Python
Web Browser	Latest	✓	Chrome or Firefox recommended

Learning Objectives

- ✓ Understand the OWASP Top 10 web application security risks
- ✓ Set up and run a deliberately vulnerable web application in a safe environment
- ✓ Manually identify SQL injection and Cross-Site Scripting (XSS) vulnerabilities
- ✓ Use an automated security scanner to detect common web vulnerabilities
- ✓ Analyze security headers and understand their protective role
- ✓ Generate and interpret vulnerability assessment reports with severity ratings
- ✓ Apply ethical hacking principles and understand Indian legal frameworks for security testing

What You Will Learn

By the end of this practical, you will:

1. Understand the OWASP Top 10 (2021) web application security risks
2. Set up a deliberately vulnerable Flask application on localhost
3. Manually exploit SQL injection on a login form
4. Manually exploit reflected Cross-Site Scripting on a search page
5. Run automated security header analysis to detect missing protections
6. Use automated scanners to detect SQL injection and XSS vulnerabilities
7. Perform a comprehensive full-scope security scan
8. Generate HTML vulnerability reports with CVSS-like severity ratings

Real-World Application

Web application security testing is critical for organizations operating digital services. In India, the Reserve Bank of India (RBI) mandates that banks and financial institutions conduct regular Vulnerability Assessment and Penetration Testing (VAPT) of their web applications. CERT-In (Indian Computer Emergency Response Team) coordinates vulnerability disclosure and incident response nationally. Companies such as **Paytm**, **PhonePe**, and **Razorpay** maintain dedicated application security teams that continuously scan their platforms for vulnerabilities before attackers can exploit them.

Hands-On Procedure

Ethical Use Disclaimer: The tools and techniques demonstrated in this lab are intended **strictly for educational purposes** within a controlled, local environment. The vulnerable web application must **ONLY** be run on `localhost` (127.0.0.1) and must **NEVER** be deployed on any public-facing server or network. Scanning or testing any system without explicit written authorization is **illegal** under the Information Technology Act, 2000 (Sections 43 and 66) and may result in criminal prosecution. Always obtain proper authorization before conducting security assessments. By proceeding with this lab, you agree to use these skills responsibly and ethically.

Part A: Environment Setup

Step 1: Clone the Web Application Security Scanner Repository

Objective: Download and explore the security scanner project structure.
Instructions:

1. Open your terminal (bash on Linux/Mac, PowerShell on Windows)
2. Navigate to your working directory
3. Clone the repository from the provided URL
4. Navigate into the project directory
5. Explore the project structure and read the ethical disclaimer

Code/Command:

```
# Clone the repository
git clone <repository-url> webapp-security-scanner
cd webapp-security-scanner

# View the project structure
ls -la
# Directories: vulnerable_app/, scanner/, reports/, docs/

# Explore key directories
ls vulnerable_app/    # app.py, templates/, static/, database.db
ls scanner/           # cli.py, header_check.py, sqli_scanner.py,
                      # xss_scanner.py, port_scanner.py,
                        report_generator.py

# Read the ethical disclaimer
cat docs/ETHICAL_DISCLAIMER.md
```

Clone and Explore the Repository

Expected Output

Project directory contains:
`vulnerable_app/` – Deliberately vulnerable Flask application
`scanner/` – Security scanning modules (headers, SQLi, XSS, etc.)
`reports/` – Report generation templates
`docs/` – Documentation and ethical guidelines
`requirements.txt` – Python dependencies
`README.md` – Project overview and usage instructions

Read the ethical disclaimer in `docs/` carefully before proceeding. Never deploy the vulnerable application on any network-accessible server.

Step 2: Create Virtual Environment and Install Dependencies

Objective: Set up an isolated Python environment and install all required packages.
Instructions:

1. Create a Python virtual environment
2. Activate the virtual environment
3. Install project dependencies from `requirements.txt`
4. Verify the scanner CLI is accessible

Code/Command:

```
# Create virtual environment
python -m venv venv

# Activate (Linux/Mac)
source venv/bin/activate
# Activate (Windows PowerShell)
.\venv\Scripts\Activate.ps1

# Install dependencies (Flask, requests, beautifulsoup4, etc.)
pip install -r requirements.txt

# Verify scanner CLI
python scanner/cli.py --help
```

Python Environment Setup

Expected Output

CLI help output shows available commands:

```
Usage:  cli.py scan [OPTIONS]

Options:
-target URL       Target URL to scan
-check TYPE       Check type:  headers, sqli, xss, all
-output FILE      Output report file path
-help             Show this message and exit
```

If Flask installation fails: (1) Ensure Python 3.8+ is installed: `python -version`, (2) Try upgrading pip first: `python -m pip install -upgrade pip`, (3) On Windows, ensure Python is in your PATH, (4) Try: `python -m pip install -r requirements.txt`.

Part B: Running the Vulnerable Application

Step 3: Start the Deliberately Vulnerable Flask Application

Objective: Launch the vulnerable web application on localhost and explore its pages. Start the Flask app, open `http://localhost:5000` in your browser, and explore the login, search, and profile pages. Keep the Flask server running in this terminal and open a new terminal for subsequent steps.
Code/Command:

```
# Start the vulnerable Flask application
python vulnerable_app/app.py
# Output: * Running on http://127.0.0.1:5000
#         * WARNING: Deliberately vulnerable application.
#         *          DO NOT expose to any network.
```

Start the Vulnerable Application

Expected Output

Browser at `http://localhost:5000` shows: Home page with navigation, Login page (`/login`) with username/password form, Search page (`/search`) with search input, and Profile page (`/profile`). The application intentionally contains SQLi and XSS vulnerabilities with no security headers.

Open a **second terminal window** for running scanner commands while keeping the Flask server running in the first terminal. Remember to activate the virtual environment in the new terminal as well.

Step 4: Manually Test SQL Injection and XSS Vulnerabilities

Objective: Understand how SQL injection and XSS work by manually exploiting them. On the login page (`/login`), enter the SQL injection payload in the username field and observe the authentication bypass. On the search page (`/search`), enter the XSS payload and observe the JavaScript alert popup.
Code/Command:

```
# SQL Injection on Login Page (/login)
# Username field: admin' OR '1'='1
# Password field: anything
# Result: Authentication bypassed - logged in as admin

# XSS on Search Page (/search)
# Search field: <script>alert('XSS')</script>
# Result: JavaScript alert popup appears

# Alternative XSS payloads:
<img src=x onerror=alert('XSS')>
<svg onload=alert('XSS')>
```

Manual Vulnerability Testing

Expected Output

SQL Injection: Login page accepts the payload. User is authenticated as "admin" without valid credentials. Displays: "Welcome, admin!"
XSS: JavaScript alert dialog appears displaying "XSS". The search term is reflected directly into HTML without sanitization (reflected XSS).

These attacks succeed because the application concatenates user input directly into SQL queries and HTML output without sanitization. In production, this could lead to data theft and system compromise. **Never** test systems without written authorization.

Screenshot 1

What to paste: The vulnerable Flask application running in your web browser, showing the login page at `http://localhost:5000/login` with the username and password form fields visible.

Paste your screenshot here

Screenshot 2

What to paste: The result of the manual SQL injection attack on the login page, showing successful authentication bypass (e.g., “Welcome, admin!” message) after entering `admin' OR '1'='1` in the username field.

Paste your screenshot here

Part C: Automated Security Scanning

Step 5: Run Security Header Analysis

Objective: Check for missing HTTP security headers using the automated scanner.
Open a new terminal (activate venv), run the header check, and review which security headers are missing and what each protects against.
Code/Command:

```
# Run security header analysis
python scanner/cli.py scan \
    --target http://localhost:5000 \
    --check headers
# Checks: X-Frame-Options, CSP, HSTS,
#          X-Content-Type-Options, X-XSS-Protection, Referrer-
   Policy
```

Security Header Check

Expected Output

```
SECURITY HEADER CHECK - http://localhost:5000
X-Frame-Options: MISSING [HIGH]
Content-Security-Policy: MISSING [HIGH]
Strict-Transport-Security: MISSING [MEDIUM]
X-Content-Type-Options: MISSING [MEDIUM]
X-XSS-Protection: MISSING [LOW]
Referrer-Policy: MISSING [LOW]
Headers Present: 0/6 | Score: 0% | Severity: CRITICAL
```

A production application should have all security headers configured. CSP alone can prevent most XSS attacks.

Step 6: Run SQL Injection Scanner

Objective: Detect SQL injection points using the automated scanner.
Run the SQL injection scan, review detected injection points, note the triggering payloads, and understand the severity classification.
Code/Command:

```
# Run SQL injection scanner
python scanner/cli.py scan \
    --target http://localhost:5000 \
    --check sqli
```

SQL Injection Scan

Expected Output

```
SQL INJECTION SCAN - http://localhost:5000
[CRITICAL] /login - username - Payload:  '  OR '1'='1 - CVSS: 9.8
[CRITICAL] /login - username - Payload:  '  OR 1=1 -  - CVSS: 9.8
[HIGH]     /search - query - Payload:  '  UNION SELECT NULL - CVSS:
8.6
Total SQLi Findings:  3 (Critical:  2, High:  1)
```

Step 7: Run XSS Scanner

Objective: Detect Cross-Site Scripting vulnerabilities using the automated scanner. Run the XSS scan, review reflected XSS findings, and note which input fields are vulnerable.

Code/Command:

```
# Run XSS scanner
python scanner/cli.py scan \
    --target http://localhost:5000 \
    --check xss
```

XSS Scan

Expected Output

```
XSS SCAN - http://localhost:5000
[HIGH] /search - query - Reflected XSS - CVSS: 7.1
Payload:  <script>alert('XSS')</script>
[HIGH] /profile - name - Reflected XSS - CVSS: 7.1
Payload:  <img src=x onerror=alert(1)>
Total XSS Findings:  2 (High:  2)
```

Step 8: Run Full Comprehensive Scan

Objective: Perform a complete security scan combining all checks into a single report. Run the full scan with `-check all` to combine header analysis, SQLi, XSS, port scanning, and directory enumeration. Review the severity table and overall risk score.

Code/Command:

```
# Run full scan (headers + SQLi + XSS + ports + directories)
python scanner/cli.py scan \
    --target http://localhost:5000 \
    --check all
```

Full Comprehensive Scan

Expected Output

```
FULL SECURITY SCAN - http://localhost:5000
Critical:  2 (SQLi) | High:  4 (XSS, Headers)
Medium:  3 (Headers, Port) | Low:  2 (Headers, Dir) | Info:  1
TOTAL: 12 findings | Overall Risk Score:  8.4/10 (Critical)
```

A risk score of 8.4/10 indicates critical vulnerabilities requiring immediate remediation. SQL injection findings alone would warrant emergency patching.

Screenshot 3

What to paste: Terminal output showing the automated scanner results, including the security header check (missing headers) and SQL injection detection results with severity ratings.

Paste your screenshot here

Screenshot 4

What to paste: Terminal output showing the full comprehensive scan results, including the severity summary table with Critical/High/Medium/Low counts and the overall risk score.

Paste your screenshot here

Part D: Vulnerability Reporting and Remediation

Step 9: Generate HTML Vulnerability Report

Objective: Generate a professional vulnerability assessment report with severity ratings. Run the full scan with `-output report.html`, open the report in your browser, and review the executive summary, CVSS scores, and remediation recommendations.
Code/Command:

```
# Generate HTML vulnerability report
python scanner/cli.py scan \
    --target http://localhost:5000 \
    --check all --output report.html

# Open report (Linux/Mac: open, Windows: start)
start report.html
```

Generate HTML Vulnerability Report

Expected Output

HTML report generated: `report.html`

Report sections: Executive Summary (risk score, target URL), Findings by Severity (Critical/High/Medium/Low with color coding), each finding with CVSS score, affected URL, evidence, and remediation recommendation.

This report format follows industry conventions used by OWASP ZAP, Burp Suite, and Nessus.

Step 10: Analyze Report and Prioritize Remediation

Objective: Interpret the vulnerability report and create a remediation priority plan. Review findings by severity (Critical → High → Medium → Low), apply the CVSS scoring system, and document which fixes require immediate vs. planned action.
Remediation Priority Framework:

Severity	CVSS	Timeline	Action
Critical	9.0–10.0	Immediate (24 hrs)	Emergency patch required
High	7.0–8.9	Within 1 week	High priority fix
Medium	4.0–6.9	Within 30 days	Scheduled remediation
Low	0.1–3.9	Within 90 days	Best-effort improvement

```
# Remediation by severity:
```

```
# Critical (SQLi): Use parameterized queries immediately
# High (XSS): Encode output, add CSP header within 1 week
# Medium (Headers): Add X-Frame-Options, HSTS within 30 days
# Low (Info): Remove server banners within 90 days
```

Analyze Findings Summary

Expected Output

```
PRIORITY 1 (Immediate):  [CRITICAL] SQLi on /login, /search
PRIORITY 2 (1 week):  [HIGH] XSS on /search, /profile; missing CSP,
X-Frame-Options
PRIORITY 3 (30 days):  [MEDIUM] Missing HSTS, X-Content-Type-Options
```

Screenshot 5

What to paste: The HTML vulnerability report (`report.html`) opened in your web browser, showing the executive summary with overall risk score, severity distribution, and at least one detailed vulnerability finding with its CVSS score and remediation recommendation.

Paste your screenshot here

Conceptual Background

OWASP Top 10 (2021)

The Open Web Application Security Project (OWASP) publishes the Top 10 most critical web application security risks. The 2021 edition includes:

A1: Broken Access Control – Unauthorized access to data or functions due to missing restrictions

A2: Cryptographic Failures – Weak or missing encryption exposing sensitive data

A3: Injection – Untrusted data sent to interpreters (SQL, OS, LDAP) as commands

A4: Insecure Design – Fundamental design flaws that implementation cannot fix

A5: Security Misconfiguration – Default configs, missing headers, verbose errors

A6: Vulnerable Components – Using libraries with known unpatched vulnerabilities

A7: Identification Failures – Weak authentication and session management

A8: Data Integrity Failures – Missing integrity verification in updates and pipelines

A9: Logging Failures – Insufficient logging delaying breach detection

A10: Server-Side Request Forgery – Fetching remote resources without URL validation

SQL Injection In-Depth

SQL Injection (SQLi) occurs when an attacker inserts malicious SQL code into application queries through user input fields. A vulnerable login query concatenates user input directly: `SELECT * FROM users WHERE username='` + input + `'`. When the attacker enters `admin' OR '1'='1`, the condition becomes always true, bypassing authentication.

Types: (1) **Classic (In-Band)** – same channel for injection and retrieval; (2) **Blind (Boolean)** – infer data from true/false response differences; (3) **Blind (Time-Based)** – infer data from response delays using `SLEEP()`; (4) **Out-of-Band** – exfiltrate data via DNS or HTTP to attacker server.

Prevention: Use parameterized queries (prepared statements), ORM frameworks (SQLAlchemy, Django ORM), input validation with whitelists, least-privilege database accounts, and Web Application Firewalls (WAF).

Cross-Site Scripting (XSS)

XSS occurs when an application includes untrusted data in web pages without proper escaping, allowing attackers to execute scripts in victims' browsers.

Types: (1) **Reflected XSS** – script is reflected off the server in search results or error messages (demonstrated in this lab); (2) **Stored XSS** – script is permanently stored on the server (database, comments) and executes for every visitor; (3) **DOM-Based XSS** – client-side JavaScript writes untrusted data to the DOM without sanitization.

Prevention: Encode all output (HTML entity encoding), use Content-Security-Policy headers, enable template auto-escaping (Jinja2, React), validate input server-side, and set HTTPOnly/Secure flags on cookies.

Security Headers

HTTP security headers instruct browsers to enable security features that protect against common attacks:

Header	**Protects Against**	**Recommended Value**
Content-Security-Policy	XSS, injection	`default-src 'self'`
Strict-Transport-Security	Protocol downgrade	`max-age=31536000`
X-Frame-Options	Clickjacking	`DENY`
X-Content-Type-Options	MIME sniffing	`nosniff`
Referrer-Policy	Info leakage	`strict-origin-when-cross-origin`

CVSS Scoring Basics

The Common Vulnerability Scoring System (CVSS) provides a standardized method for rating the severity of security vulnerabilities on a scale of 0.0 to 10.0:

Score	**Rating**	**Description**
9.0–10.0	Critical	Easily exploitable, full system compromise, no interaction required
7.0–8.9	High	Significant impact, relatively easy to exploit
4.0–6.9	Medium	Moderate impact, requires some conditions to exploit
0.1–3.9	Low	Limited impact, difficult to exploit
0.0	None	Informational only

CVSS uses three metric groups: **Base** (inherent), **Temporal** (evolves over time), and **Environmental** (organization-specific).

Ethical Hacking Principles

Security testing must follow ethical guidelines: (1) **Authorization** – obtain written permission before any testing; (2) **Scope** – stay within agreed boundaries; (3) **Confidentiality** – protect findings and discovered data; (4) **Non-Disruption** – avoid damaging systems; (5) **Responsible Disclosure** – report to appropriate parties, not publicly; (6) **Documentation** – maintain detailed records; (7) **Legal Compliance** – follow all applicable laws.

India-Specific Legal and Regulatory Context

Information Technology Act, 2000

Key sections: **Section 43** – penalty for unauthorized access (up to INR 1 crore compensation); **Section 66** – criminal punishment for hacking (up to 3 years imprisonment); **Section 43A** – liability for failure to protect personal data; **Section 72A** – punishment for breach of lawful contract.

CERT-In Vulnerability Disclosure

The Indian Computer Emergency Response Team (CERT-In) under MeitY coordinates responsible vulnerability disclosure, mandates incident reporting within 6 hours (2022 directive), maintains the National Vulnerability Database, and conducts cybersecurity audits of critical information infrastructure.

RBI Cyber Security Framework for Banks

The RBI requires banks to conduct VAPT at least annually and after every major release, engage CERT-In empanelled auditors, maintain a Security Operations Centre (SOC), report cyber incidents within specified timelines, and implement Board-level cybersecurity oversight.

Real-World Example: VAPT in Indian Banking

Indian banks follow a structured VAPT process: (1) **Scoping** target applications (internet banking, UPI); (2) **Automated scanning** with OWASP ZAP, Burp Suite, Nessus; (3) **Manual testing** for business logic flaws; (4) **Reporting** with CVSS scores and evidence; (5) **Remediation** within mandated timelines; (6) **Re-testing** to verify fixes; (7) **Compliance** reports submitted to RBI as part of annual IS audit.

Assessment & Deliverables

Assessment Questions

Answer the following questions in your submission:

Q1. List any five of the OWASP Top 10 (2021) risks and explain how each one could impact a banking web application in India.

Q2. Explain how SQL injection works. Why does the payload `admin' OR '1'='1` bypass authentication? What is the correct prevention method?

Q3. What is the difference between reflected, stored, and DOM-based XSS? Which type did you observe in this lab, and how could it be exploited in a real attack scenario?

Q4. Explain the purpose of three different HTTP security headers. Why were all headers missing in the vulnerable application?

Q5. What are the key principles of ethical hacking? Under which sections of the IT Act 2000 could unauthorized security testing be prosecuted?

Q6. Describe the CVSS scoring system. How would you prioritize remediation of vulnerabilities with scores of 9.8, 7.1, and 4.5?

Q7. What is CERT-In and what role does it play in India's cybersecurity ecosystem? What are the mandatory incident reporting timelines?

Q8. Explain the RBI's VAPT requirements for banks. How does the scanning approach used in this lab relate to real-world VAPT engagements?

Deliverables Checklist

Item	**Description**	**Type**	**Status**
Screenshot 1	Vulnerable app login page	Paste	☐
Screenshot 2	SQL injection bypass success	Paste	☐
Screenshot 3	Header check + SQLi scanner	Paste	☐
Screenshot 4	Full scan severity table	Paste	☐
Screenshot 5	HTML vulnerability report	Paste	☐
Answers	Q1–Q8 written responses	Text	☐
Report File	Generated `report.html`	File	☐
Remediation	Priority list with timelines	Text	☐

Verification Checklist

- ☐ Repository cloned, virtual environment set up, CLI runs with `-help`
- ☐ Vulnerable Flask application running on `http://localhost:5000`
- ☐ SQL injection manually tested (authentication bypass confirmed)
- ☐ XSS manually tested on search page (alert popup confirmed)
- ☐ Automated scans completed: headers, SQLi, XSS, full scan
- ☐ HTML vulnerability report generated and opened in browser
- ☐ Remediation priority list created with severity-based timelines
- ☐ All 5 screenshots captured and all 8 questions answered

Grading Rubric

Criteria	**Description**	**Points**	**Score**
Setup	Repo cloned, environment ready, app running	10	___/10
Manual Testing	SQLi and XSS manually demonstrated	15	___/15
Header Scan	Security header analysis completed	10	___/10
SQLi Scan	Automated SQL injection scan run	10	___/10
XSS Scan	Automated XSS scan run	10	___/10
Full Scan	Comprehensive scan with severity table	10	___/10
Report	HTML report generated and analyzed	15	___/15
Remediation	Priority plan with timelines	10	___/10
Assessment	Q1–Q8 answered correctly	10	___/10
	TOTAL	**100**	___/100

Appendix A: OWASP Top 10 Quick Reference

ID	Risk	Impact	Prevention
A01	Broken Access Control	Unauthorized access	RBAC, deny by default
A02	Cryptographic Failures	Data exposure	TLS, strong algorithms
A03	Injection	System compromise	Parameterized queries
A04	Insecure Design	Security gaps	Threat modeling
A05	Security Misconfig	Data leak	Hardening, audits
A06	Vulnerable Components	Known exploits	Patch management
A07	Identification Failures	Account takeover	MFA, strong passwords
A08	Data Integrity Failures	Supply chain attacks	Code signing
A09	Logging Failures	Delayed detection	Centralized logging
A10	SSRF	Internal access	URL allowlists

Appendix B: Common SQL Injection Payloads

Payload	Type	Purpose
`’ OR ’1’=’1`	Auth bypass	Always-true condition
`’ OR 1=1 -`	Auth bypass	Comment out remainder
`’ UNION SELECT NULL -`	Extraction	Column count detection
`’; DROP TABLE users -`	Destructive	Table deletion
`’ AND 1=1 - / ’ AND 1=2 -`	Boolean blind	True/false inference
`’ AND SLEEP(5) -`	Time blind	Delay-based inference

These payloads are for educational reference only. Using them without explicit written authorization is illegal under the IT Act 2000.

Appendix C: Security Headers Checklist

Header	OK	Expected Value
Content-Security-Policy	☐	Restrictive policy, no `unsafe-inline`
Strict-Transport-Security	☐	`max-age` $\geq$ 31536000
X-Frame-Options	☐	`DENY` or `SAMEORIGIN`
X-Content-Type-Options	☐	`nosniff`
Referrer-Policy	☐	`strict-origin-when-cross-origin`
Permissions-Policy	☐	Restrict unnecessary features

Appendix D: Ethical Guidelines for Security Testing

▷ **Before:** Obtain written authorization, define scope (URLs, IP ranges, exclusions), agree on testing window, establish emergency contacts, confirm rules of engagement

▷ **During:** Stay within scope, document all actions with timestamps, stop on unintended disruption, do not access real user data, report critical findings immediately

▷ **After:** Securely delete test artifacts, prepare comprehensive report, present through secure channels, verify remediation via re-testing, maintain confidentiality indefinitely

Appendix E: Troubleshooting Guide

Problem: `ModuleNotFoundError: No module named 'flask'` or port already in use
Solutions:

1. Ensure virtual environment is activated: check for `(venv)` prefix in terminal
2. Install Flask: `pip install flask`
3. Port 5000 in use: kill existing process or change port in `app.py`
4. On macOS, AirPlay may use port 5000 — disable it or use port 5001

Problem: `ConnectionRefusedError` or `requests.exceptions.ConnectionError`
Solutions:

1. Verify the Flask application is running: check the first terminal window
2. Confirm the URL is correct: `http://localhost:5000` (not `https`)
3. Check firewall settings are not blocking localhost connections
4. Try using `http://127.0.0.1:5000` instead of `localhost`

Problem: Report file is created but shows blank or broken layout in browser
Solutions:

1. Ensure the scan completed successfully before report generation
2. Check that the `reports/templates/` directory exists and contains template files
3. Try generating with a different output path: `-output /tmp/report.html`
4. Open the file in a different browser (Chrome recommended)
5. Check terminal for Jinja2 template errors

Appendix F: Additional Resources

References

→ OWASP Top 10: `https://owasp.org/www-project-top-ten/`

→ OWASP Testing Guide: `https://owasp.org/www-project-web-security-testing-guide/`

→ CERT-In: `https://www.cert-in.org.in/`

→ CVSS Calculator: `https://www.first.org/cvss/calculator/3.1`

→ PortSwigger Web Security Academy (free online labs)

→ “The Web Application Hacker’s Handbook” – Stuttard and Pinto

Tools Used in This Practical

Tool	Purpose	Cost
Python 3.8+	Programming language runtime	Free
Flask	Vulnerable web application framework	Free
requests	HTTP library for scanner modules	Free
BeautifulSoup4	HTML parsing for response analysis	Free
Jinja2	HTML report template engine	Free
colorama	Terminal color output	Free

—END OF LAB MANUAL—

Document Version: 1.0

IT Management & Audits – Practical Lab Series