🔍 *Abstract – Advanced Web Application Security Scanner (CLI + Web UI)*

In the modern digital landscape, web applications are frequent targets of cyberattacks due to common vulnerabilities like Cross-Site Scripting (XSS), SQL Injection, and insecure configurations. To address this, we developed an **Advanced Web Application Security Scanner** — a Python-based tool that provides both a **Command-Line Interface (CLI)** and a **Flask-powered Web Dashboard** for scanning and reporting security issues in web applications.

The scanner is designed to be lightweight, educational, and user-friendly, enabling students, developers, and beginners in cybersecurity to perform basic vulnerability assessments with ease. It detects key issues such as XSS, SQL Injection, missing security headers (e.g., Content-Security-Policy), and cookie misconfigurations (e.g., missing HttpOnly or Secure flags). Scan results are logged in CSV format, visualized using interactive pie charts, and can be exported as Excel reports.

The tool emphasizes **modular design**, allowing reuse of scanning logic across CLI and web modes. It also features **log management**, **search filters**, and **severity categorization**, enhancing usability and reporting. While the scanner is not a substitute for enterprise-grade tools, it effectively demonstrates foundational concepts in web application security and provides a practical learning experience in both **secure coding** and **offensive testing**.

Table of Contents

📘 Chapter 1: Introduction

1.1 Overview

Web applications are the backbone of modern digital services. From online banking to e-commerce platforms, most user interactions today occur through the web. While this convenience benefits users and businesses alike, it also presents a wide attack surface for cybercriminals. Security vulnerabilities such as SQL injection, Cross-Site Scripting (XSS), cookie theft, and the absence of proper security headers can easily be exploited to compromise sensitive data or take over systems.

To combat these threats, it is essential to scan and analyze web applications regularly. This project aims to develop an **Advanced Web Application Security Scanner** capable of identifying several common vulnerabilities using both **a Command-Line Interface (CLI)** and **a Flask-based Web Dashboard**. This scanner is intended for educational and research purposes and helps developers, students, and cybersecurity learners understand the basics of secure coding and application hardening.

1.2 Problem Statement

Most traditional vulnerability scanners are either expensive, too complex for beginners, or limited in functionality. Open-source tools often lack user-friendly interfaces or integration between CLI and web components. There's also a gap in tools that combine scanning features like header checks, XSS detection, SQL injection attempts, and insecure content analysis in a beginner-friendly yet detailed way.

Hence, there is a need for a lightweight, open-source security scanner that can:

- Perform multiple types of basic vulnerability tests
- Be accessed both via terminal (CLI) and web UI
- Provide structured output and logs for future analysis
- Export findings to formats like Excel for reporting
- Visualize results (via pie charts) to understand severity levels

1.3 Objective

The main objective of this project is to:

> **Develop an advanced security scanner for web applications** that performs multiple types of security tests, logs results, displays them on a web dashboard, and allows export for analysis — combining both CLI and web-based modes.

1.4 Core Goals

1. Build a reusable scan engine (core module)
2. Create a CLI tool to run scans via terminal
3. Create a Flask-based web interface for scans
4. Perform at least **7 core vulnerability checks**
5. Store scan results in a structured format (CSV)
6. Display summary charts using Chart.js
7. Export scan history to Excel (.xlsx)
8. Provide a simple and clean UI (Bootstrap)

1.5 Scope of the Project

The project is designed primarily for educational and small enterprise use. It will not conduct deep penetration testing or automated crawling like professional tools (e.g., Burp Suite or OWASP ZAP), but it will focus on the following:

- **Input URL scanning only**
- **Single-page security testing**
- **Basic vulnerability detection logic**
- **Real-time results**
- **Logs and exports for audit and report writing**

The scanner **will not** include features like brute-force login attacks, full spidering, authentication handling, or DoS simulation, as those are out of scope and may violate responsible disclosure principles.

1.6 Key Deliverables

Deliverable	Description
✅ scanner_core.py	Core logic for security checks
✅ scanner.py	CLI interface for terminal-based scan
✅ app.py	Flask app with web UI
✅ index.html	Form + scan results + pie chart
✅ logs.html	Scan log viewer with filters + export
✅ scan_log.csv	Stores all scan entries
✅ exported_logs.xlsx	Excel export of logs

1.7 Technologies Used

Technology	Purpose
Python 3.x	Core language
Flask	Web app framework
BeautifulSoup	HTML parsing
Requests	HTTP request handling
Pandas	Data export to Excel
Chart.js	Chart rendering
Bootstrap	Front-end styling
HTML/CSS	UI design
CSV / Excel	Log storage & reporting

📘 Chapter 2: Literature Review

2.1 Introduction

With the growing digitalization of businesses and services, web applications have become essential for delivering value to users. However, the increasing reliance on web technologies has also led to a surge in cyberattacks targeting vulnerabilities in these applications. Understanding past research, tools, and frameworks that address web application security is essential to design an effective and practical solution. This chapter presents a review of the existing literature, frameworks like the **OWASP Top 10**, and comparisons with existing tools and technologies relevant to this project.

2.2 Evolution of Web Application Security

Web application security has evolved rapidly in response to emerging threats. Initially, static pages posed fewer risks, but with the rise of dynamic, database-driven, and JavaScript-heavy applications, the attack surface increased significantly. Threats like **Cross-Site Scripting (XSS)**, **SQL Injection**, and **session hijacking** have become common, and researchers have been developing automated detection techniques and tools to mitigate them.

Over the years, various open-source and commercial solutions have emerged, focusing on dynamic scanning, code auditing, penetration testing, and security policy enforcement. Many of these solutions are tailored for enterprise-grade security testing, often requiring deep technical expertise or high financial investment.

2.3 OWASP Top 10 Framework

The **OWASP Top 10** is a globally recognized standard for web application security. It identifies and ranks the most critical web application security risks. This project takes direct inspiration from this model, focusing on simplified checks for the following:

OWASP Risk	Included in Project
A01: Broken Access Control	❌ Out of scope
A02: Cryptographic Failures	❌ Not targeted
A03: Injection (SQL)	✅ Yes (SQL Injection Test)

A04: Insecure Design	❌ Architectural issue, not scanned
A05: Security Misconfiguration	✅ Partial (header, cookie checks)
A06: Vulnerable and Outdated Components	❌ Not targeted
A07: Identification and Authentication Failures	❌ Not targeted
A08: Software and Data Integrity Failures	❌ Not targeted
A09: Security Logging and Monitoring Failures	✅ Logging implemented
A10: Server-Side Request Forgery (SSRF)	❌ Not included

This project focuses mainly on **A03**, **A05**, and **A09**, aligning with its goal of lightweight, accessible, beginner-friendly scanning.

2.4 Existing Tools and Their Limitations

Several well-known tools exist for web application vulnerability scanning:

2.4.1 OWASP ZAP (Zed Attack Proxy)

- **Strengths:** Powerful, open-source, includes spidering, fuzzing, and automated testing
- **Limitations:** Complex UI, heavy for beginners, steep learning curve

2.4.2 Burp Suite

- **Strengths:** Industry-grade, intercepts traffic, advanced testing
- **Limitations:** Free version is limited, paid version is expensive, requires deep knowledge

2.4.3 Nikto

- **Strengths:** Fast, CLI-based, well-known tool for web server scanning
- **Limitations:** Outdated signatures, noisy output, no web interface

2.5 Why Build This Project?

Unlike ZAP or Burp, this project is designed for:

- 💡 **Simplicity** – Easy to understand, no installation headaches
- 🔄 **Flexibility** – Works in both CLI and browser
- 🎯 **Focused Scans** – Only the most common real-world issues
- 📊 **Reporting & Visualization** – Easy logs, Excel exports, and charts

This tool is ideal for:

- 🧑‍🎓 Students
- 🛠️ Developers
- 🧪 Security beginners
- 🧑‍🏫 Classroom demonstrations

2.6 Key Findings from Literature

Study / Tool	Key Takeaways
OWASP Reports	Developers often ignore headers, cookies, and basic validation
Academic Research	Lightweight scanners are effective for early-stage prevention
Real-world breaches	60%+ exploit known vulnerabilities (like XSS, SQLi)
User behavior studies	Tools must have simple UIs to promote frequent use

2.7 Conclusion

While many advanced tools exist in the cybersecurity world, there is a clear gap in beginner-friendly, accessible, and simple vulnerability scanners that balance functionality with usability. This project was designed with that exact objective in mind — **to provide a practical learning tool that covers real-world issues, without overwhelming the user.**

📘 Chapter 3: System Analysis

3.1 Introduction

System analysis is a critical stage in software development, focusing on understanding the objectives of the system, analyzing requirements, identifying stakeholders, and preparing for successful implementation. This chapter examines the **problems in existing security tools**, identifies **target users**, and defines both **functional** and **non-functional requirements** for the Advanced Web Application Security Scanner project.

3.2 Problem Analysis

In today's digital landscape, most web applications face a wide variety of security threats due to improperly implemented security mechanisms. While enterprise tools like Burp Suite and OWASP ZAP offer thorough scanning capabilities, they are often:

- Too complex for students or small developers
- Not user-friendly
- Require deep configuration
- Do not provide quick, visual summaries

Beginners or small teams lack access to simple tools that can help them learn, detect, and mitigate common vulnerabilities early in the development cycle. There is a strong need for a **lightweight, easy-to-use scanner** that can:

- Detect basic vulnerabilities
- Be run from both terminal and browser
- Offer exportable logs and quick visual reports

3.3 Existing System vs Proposed System

Criteria	Existing Tools (Burp, ZAP, Nikto)	Proposed Project
Usability	Complex, technical	Simple and beginner-friendly
Interface	CLI/GUI (complex)	CLI + Web UI (Bootstrap)
Risk Visualization	Limited	Pie chart (Chart.js)
Export Reports	Limited/complex	Excel export
Educational Value	Low	High (ideal for learning)
Installation	Heavy setup	Lightweight, Python-based

3.4 System Requirements

3.4.1 Functional Requirements (FR)

FR No.	Requirement
FR1	User should be able to enter a URL via CLI or web UI
FR2	System should run multiple scans (XSS, SQLi, etc.) on the given URL
FR3	Scan results should be logged in a CSV file
FR4	User should be able to view past scans from a web dashboard
FR5	Export of logs to Excel format should be possible
FR6	Web UI should display scan results in a clear, categorized table
FR7	Severity levels should be visualized in a pie chart

3.4.2 Non-Functional Requirements (NFR)

NFR No.	Requirement
NFR1	The scanner must execute and respond within 10 seconds for simple URLs

NFR2	The system must support up to 1000 log entries without performance issues
NFR3	The UI should be responsive and mobile-friendly
NFR4	The system should handle invalid input URLs gracefully
NFR5	Logs should be saved persistently in CSV format

3.5 Feasibility Study

a) Technical Feasibility

- Built entirely in Python using widely supported libraries (Flask, requests, pandas)
- Requires no third-party API
- Easily deployable on any machine with Python installed

✅ **Feasible**

b) Operational Feasibility

- CLI tool is easy to use for terminal-based users
- Web UI is ideal for users who prefer visual interaction
- Export and chart features improve usability

✅ **Feasible**

c) Economic Feasibility

- No commercial dependencies
- Fully open-source libraries
- Can be deployed with zero cost

✅ **Feasible**

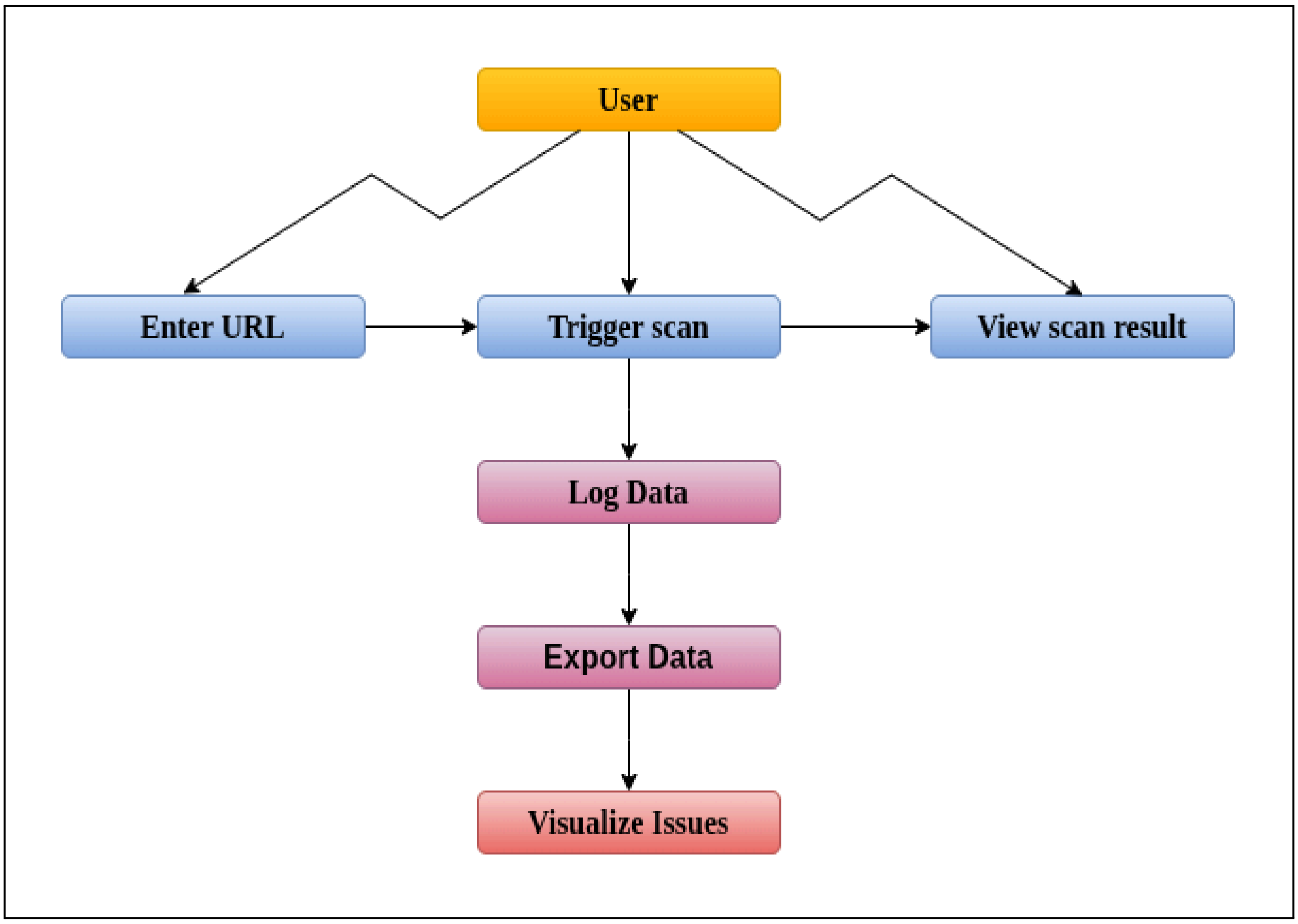
3.6 Use Case Diagram

📌 **Use Case**: "Scan a Web Application"

Actors:

- **User** (Developer / Tester)

Use Cases:



3.7 Conclusion

This chapter presented a thorough analysis of the existing problem, proposed solution, and technical requirements of the system. The feasibility study confirms that the scanner is realistic and achievable with limited resources. The system requirements will guide the subsequent stages of system design and development.

📘 Chapter 4: System Design

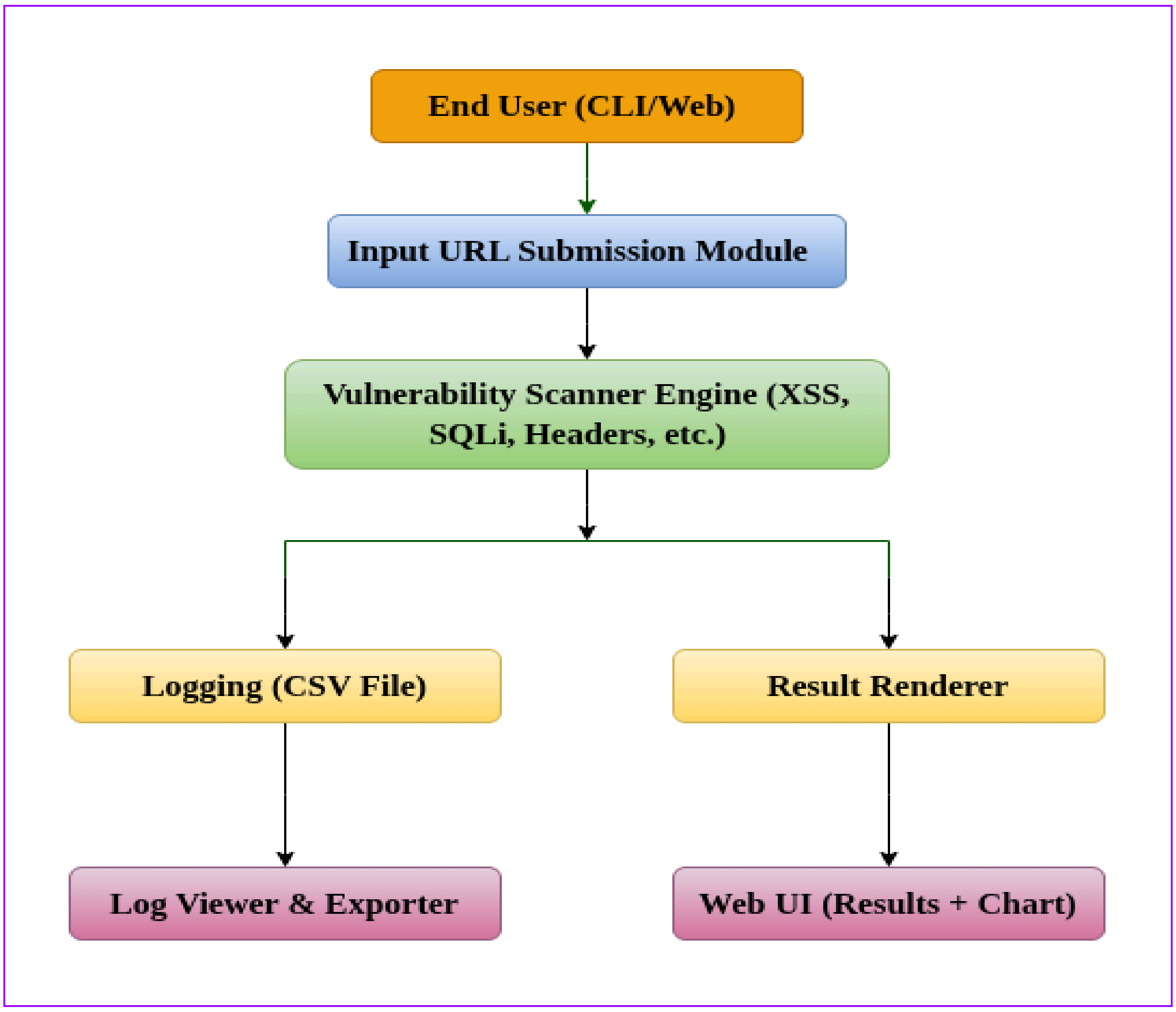
4.1 Introduction

System design is the blueprint for the system being developed. It converts the analysis phase's findings into a structured framework of modules, interfaces, workflows, and data flow. This chapter describes the **architecture**, **module structure**, **UI layout**, and **data flow** of the **Advanced Web Application Security Scanner** project.

4.2 System Architecture Diagram

The system is based on a **modular architecture** with clearly defined layers:

🔷 Architecture Components:



4.3 Modular Design

The system is divided into the following main modules:

Module	Description
scanner_core.py	Contains all scanning functions (shared by CLI & Flask)
scanner.py	CLI interface to run scans and show/save results
app.py	Flask web server handling form input, chart, and log routes
templates/index.html	Front-end page to submit URLs and see results
templates/logs.html	Log history page with export and filters
scan_log.csv	Persistent log storage for all scans
exported_logs.xlsx	Excel file generated from scan logs

4.4 User Interface Design

The system has a **responsive, clean Bootstrap UI**, divided into two key pages:

🔹 index.html

- Navbar with links to “New Scan” and “Logs”
- Input box for URL submission
- List of scan results
- Pie chart showing issue severity (High, Medium, Low)

🔹 logs.html

- Searchable and filterable table of all previous scans
- “Export to Excel” button
- Optional pie chart of all-time severity counts

Both pages use **Bootstrap 5** for mobile responsiveness and **Chart.js** for visualization.

4.5 File Structure

```
AdvancedScanner/
│
├── scanner_core.py       # Core scanning logic
├── scanner.py            # CLI script
├── app.py                # Flask server
├── scan_log.csv          # Log file
├── exported_logs.xlsx    # Exported Excel log
├── requirements.txt      # Python dependencies
│
├── templates/
│   ├── index.html        # Main scanner UI
│   └── logs.html         # Scan history UI
│
└── static/               # (Optional: CSS, JS)
```

4.6 Data Flow

1. **User Input**: URL is submitted via CLI or web form
2. **Scan Engine**: URL is passed to scanner_core.py which runs all selected tests
3. **Result Handling**:
   - ✅ Scan results are stored in a dictionary
   - ✅ Log entries are written to scan_log.csv
4. **Output**:
   - CLI: Results are printed in console
   - Web UI: Results shown in index.html, plus pie chart
   - Logs Page: Full log table shown in logs.html, with filters and export

4.7 Data Logging Format

Each row in scan_log.csv stores the following:

Column	Description
Timestamp	Date & time of scan
URL	Scanned site
Scan Type	What test was run (e.g., XSS)
Issue Type	Category of result (e.g., Vulnerability)
Details	Specific output (e.g., “Form vulnerable to XSS found”)

4.8 Design Considerations

- **Reusability**: The same scan logic is used by both CLI and Flask
- **Modularity**: Each component (UI, scanner, logger) is separated
- **Scalability**: More scan types can be added by modifying scanner_core.py
- **Responsiveness**: Bootstrap ensures it works on mobile/tablet screens
- **User-Friendliness**: Simple layout, pie charts, and Excel logs improve usability

---

4.9 Conclusion

The system is designed to be simple yet effective. By combining CLI and Web with a reusable scanning core, the design promotes modularity and extensibility. The chart-based visual UI and Excel exports also improve accessibility for less technical users.

📘 Chapter 5: Implementation

5.1 Introduction

This chapter explains the **actual development and implementation** of each module in the project. It provides a breakdown of the source code structure, key functions, user interface, log generation, export capabilities, and other features that make up the Advanced Web Application Security Scanner.

The implementation was done using Python 3, Flask for the web interface, and supporting libraries such as requests, beautifulsoup4, pandas, and Chart.js.

5.2 Technology Stack

Layer	Technology Used	Purpose
Language	Python 3.x	Core programming
UI	HTML, Bootstrap 5	Front-end
Backend	Flask	Web server
Libraries	BeautifulSoup, Requests	HTML parsing, HTTP scanning
Data	CSV, Pandas, OpenPyXL	Logging and export
Charts	Chart.js	Pie chart visualization

5.3 Code File Breakdown

📄 1. scanner_core.py – Main Scan Engine

This module contains all the reusable functions for different types of scans. Both the CLI and web interface use this core file.

```
def scan_xss(url):
    # Looks for forms and checks if input is reflected
    ...

def scan_sql_injection(url):
    # Tries SQL payloads and checks for error patterns
    ...

def check_security_headers(response):
    # Checks for common security headers
    ...

def scan_cookie_flags(response):
    # Checks for HttpOnly and Secure flags in cookies
    ...

# All functions return a list of dicts like:
# [{ "ScanType": "XSS", "IssueType": "Vulnerability", "Details": "Found XSS on input form" }]
```

These functions return structured data, so the CLI and web components can format and display results independently.

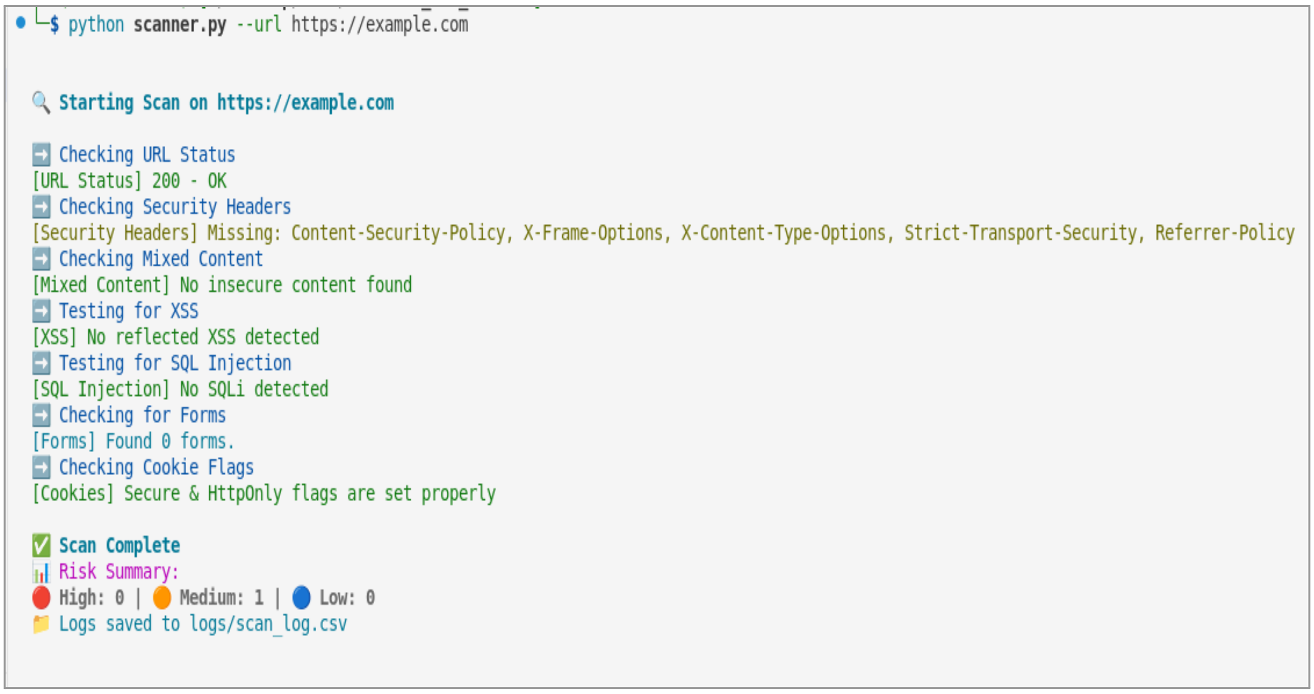
📄 2. scanner.py – CLI Interface

This is the **command-line tool** to run scans without a web browser.

```
$ python scanner.py https://example.com
```

- Accepts a URL
- Calls all scan functions
- Displays results in terminal
- Writes logs to `scan_log.csv`

Sample CLI Output:



```
● └$ python scanner.py --url https://example.com


🔍 Starting Scan on https://example.com

➡️ Checking URL Status
[URL Status] 200 - OK
➡️ Checking Security Headers
[Security Headers] Missing: Content-Security-Policy, X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security, Referrer-Policy
➡️ Checking Mixed Content
[Mixed Content] No insecure content found
➡️ Testing for XSS
[XSS] No reflected XSS detected
➡️ Testing for SQL Injection
[SQL Injection] No SQLi detected
➡️ Checking for Forms
[Forms] Found 0 forms.
➡️ Checking Cookie Flags
[Cookies] Secure & HttpOnly flags are set properly

✅ Scan Complete
📊 Risk Summary:
🔴 High: 0 | 🟠 Medium: 1 | 🔵 Low: 0
📁 Logs saved to logs/scan_log.csv
```

📄 3. app.py – Flask Web Application

This is the entry point for the web interface. It defines all Flask routes:

Route	Function
/	Displays input form and results
/logs	Shows all past scans
/export	Downloads log as Excel file

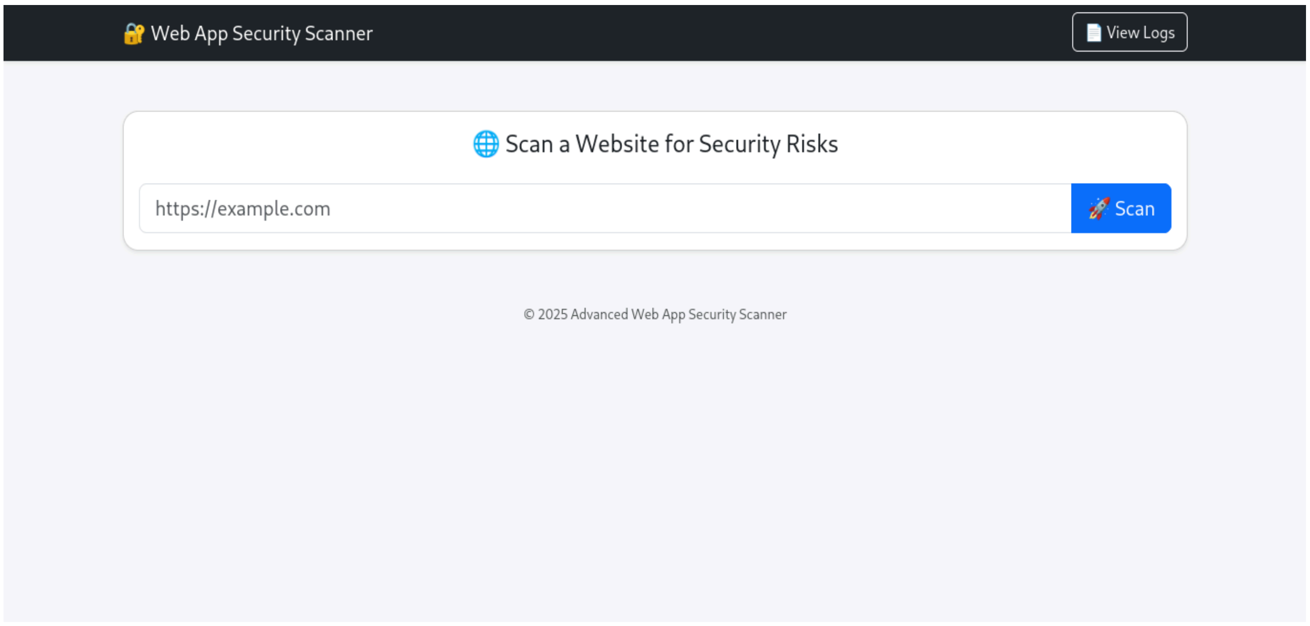
Key Logic:

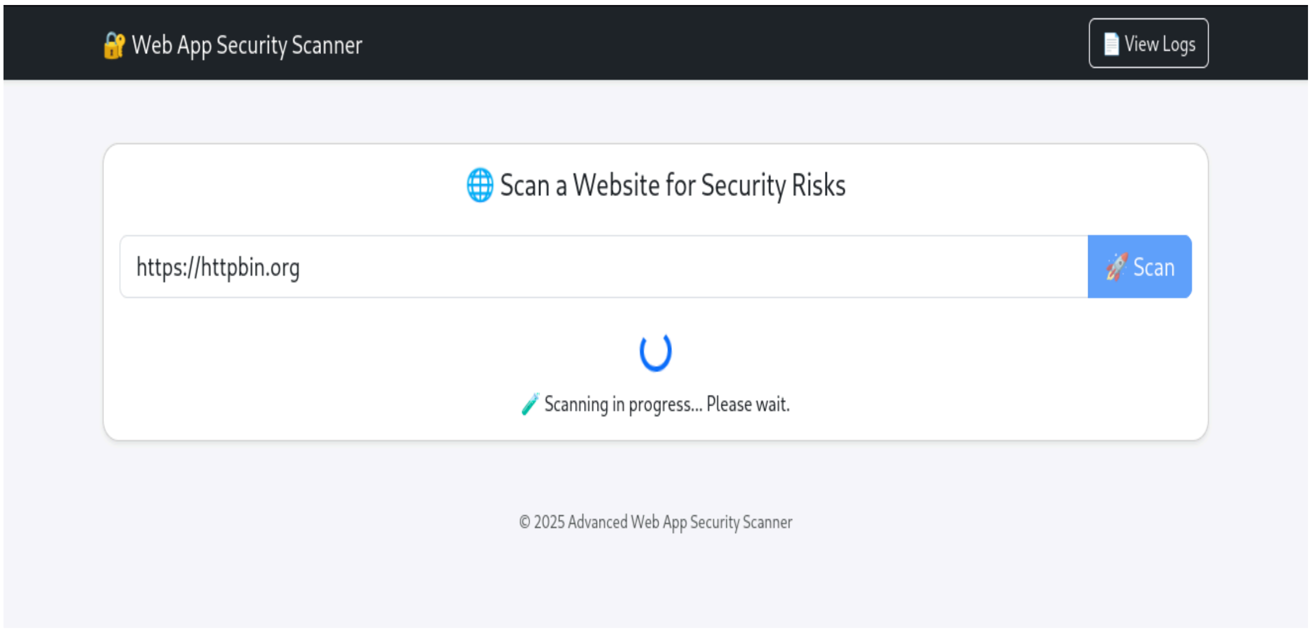
- When user submits a URL, it is passed to scanner_core.py
- Results are shown on the same page
- Log is updated
- Pie chart summarizes result severity

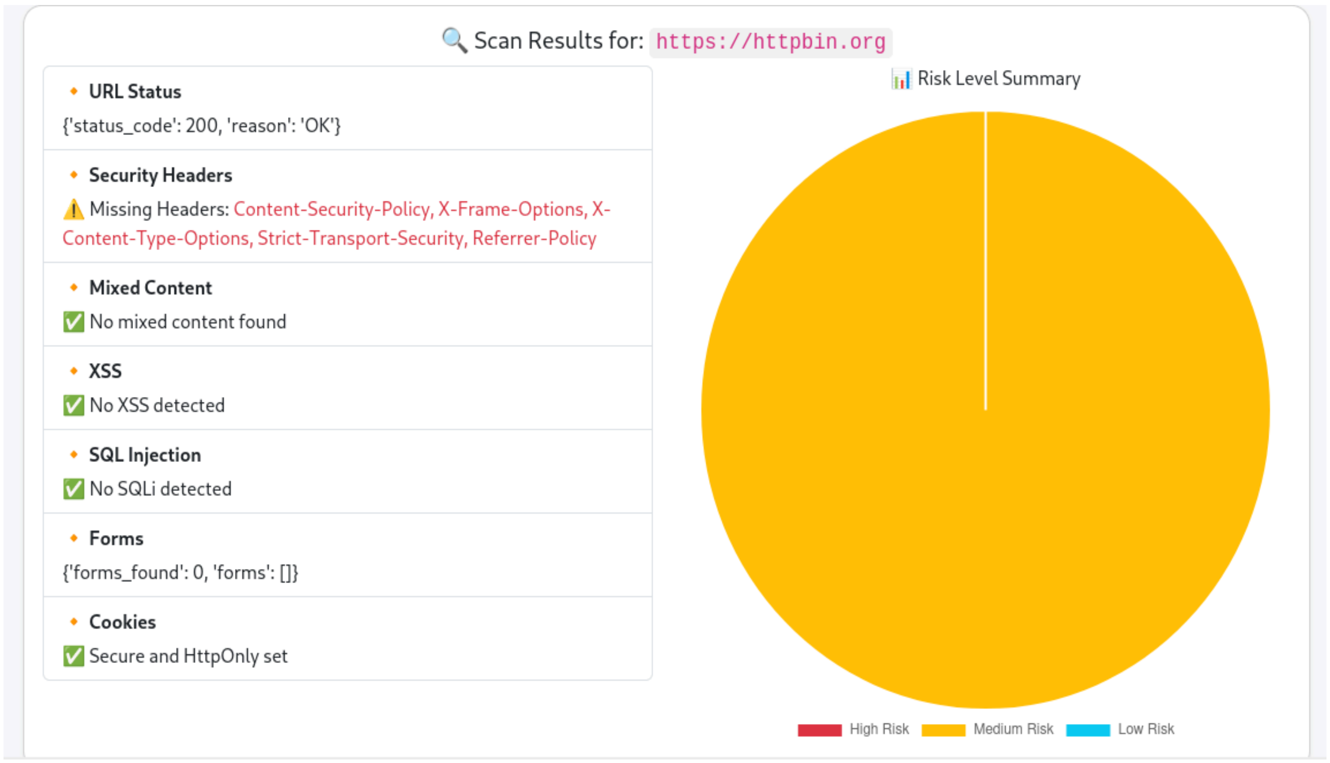
📄 4. index.html – Web Scanner UI

Features:

- Input field for URL
- Display of all scan results
- Pie chart for severity levels (High, Medium, Low)
- Bootstrap layout for responsiveness


🔐 Web App Security Scanner
📄 View Logs
🌐 Scan a Website for Security Risks
https://example.com
🚀 Scan
© 2025 Advanced Web App Security Scanner


🔐 Web App Security Scanner
📄 View Logs
🌐 Scan a Website for Security Risks
https://httpbin.org
🚀 Scan
🧪 Scanning in progress... Please wait.
© 2025 Advanced Web App Security Scanner



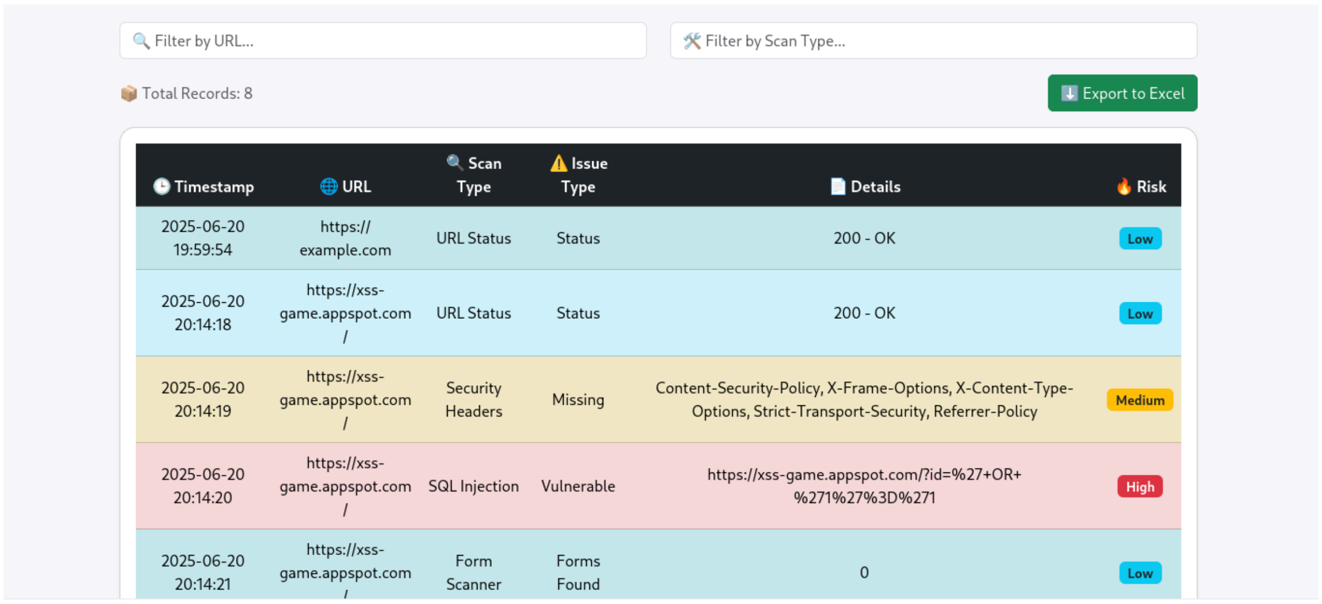
🔍 Scan Results for: `https://httpbin.org`

- **URL Status**
  {'status_code': 200, 'reason': 'OK'}
- **Security Headers**
  ⚠️ Missing Headers: Content-Security-Policy, X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security, Referrer-Policy
- **Mixed Content**
  ✅ No mixed content found
- **XSS**
  ✅ No XSS detected
- **SQL Injection**
  ✅ No SQLi detected
- **Forms**
  {'forms_found': 0, 'forms': []}
- **Cookies**
  ✅ Secure and HttpOnly set

📄 5. logs.html – Log Viewer Page

Features:

- Table displaying all logs
- Search filters (by URL, scan type)
- Export to Excel button
- Optional pie chart of total severity distribution



📦 Total Records: 8

🕒 Timestamp	🌐 URL	🔍 Scan Type	⚠️ Issue Type	📄 Details	🔥 Risk
2025-06-20 19:59:54	https:// example.com	URL Status	Status	200 - OK	Low
2025-06-20 20:14:18	https://xss-game.appspot.com /	URL Status	Status	200 - OK	Low
2025-06-20 20:14:19	https://xss-game.appspot.com /	Security Headers	Missing	Content-Security-Policy, X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security, Referrer-Policy	Medium
2025-06-20 20:14:20	https://xss-game.appspot.com /	SQL Injection	Vulnerable	https://xss-game.appspot.com/?id=%27+OR+%271%27%3D%271	High
2025-06-20 20:14:21	https://xss-game.appspot.com	Form Scanner	Forms Found	0	Low

5.4 Logging: scan_log.csv

Each scan result is stored in a CSV file using this format:

Timestamp	URL	Scan Type	Issue Type	Details

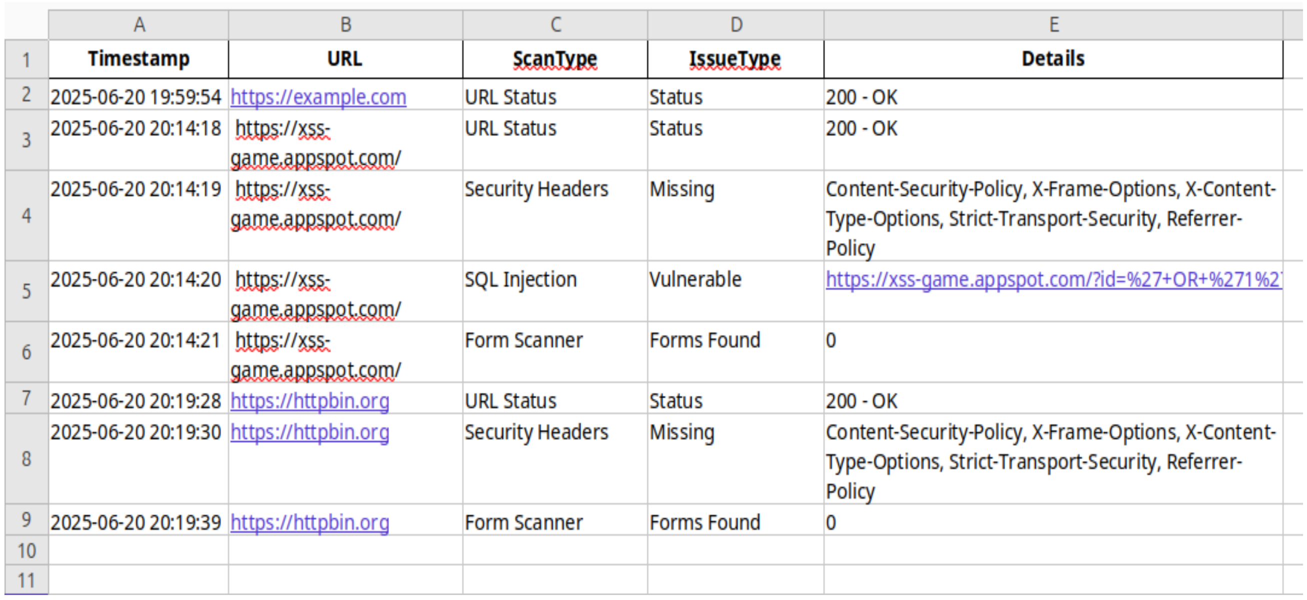
Example Row:

```
Timestamp	URL	ScanType	IssueType	Details
2025-06-20 19:59:54	https://example.com	URL Status	Status	200 - OK
2025-06-20 20:14:19	https://xss-game.appspot.com/	Security Headers	Missing	Content-Security-Policy, X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security, Referrer-Policy
2025-06-20 20:14:20	https://xss-game.appspot.com/	SQL Injection	Vulnerable	https://xss-game.appspot.com/?id=%27+OR+%271%27%3D%271
```

5.5 Excel Export: /export Route

Using the pandas and openpyxl libraries, logs from the CSV are converted into an Excel file:

```
@app.route('/export')
def export_logs():
    df = pd.read_csv("scan_log.csv")
    df.to_excel("exported_logs.xlsx", index=False)
    return send_file("exported_logs.xlsx", as_attachment=True)
```



	A	B	C	D	E
1	Timestamp	URL	ScanType	IssueType	Details
2	2025-06-20 19:59:54	https://example.com	URL Status	Status	200 - OK
3	2025-06-20 20:14:18	https://xss-game.appspot.com/	URL Status	Status	200 - OK
4	2025-06-20 20:14:19	https://xss-game.appspot.com/	Security Headers	Missing	Content-Security-Policy, X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security, Referrer-Policy
5	2025-06-20 20:14:20	https://xss-game.appspot.com/	SQL Injection	Vulnerable	https://xss-game.appspot.com/?id=%27+OR+%271%2
6	2025-06-20 20:14:21	https://xss-game.appspot.com/	Form Scanner	Forms Found	0
7	2025-06-20 20:19:28	https://httpbin.org	URL Status	Status	200 - OK
8	2025-06-20 20:19:30	https://httpbin.org	Security Headers	Missing	Content-Security-Policy, X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security, Referrer-Policy
9	2025-06-20 20:19:39	https://httpbin.org	Form Scanner	Forms Found	0
10					
11					

5.6 Pie Chart Visualization

Using **Chart.js**, a pie chart displays:

- 🔴 High Risk (XSS, SQLi)
- 🟡 Medium Risk (Security headers)
- 🔵 Low Risk (Cookies, Mixed Content)

```
data: [{{ chart_data.High }}, {{ chart_data.Medium }}, {{ chart_data.Low }}]
```

5.7 Error Handling

The system handles errors gracefully:

- Invalid URLs show "Invalid input" message
- Logs are skipped if files don't exist
- Try-except blocks prevent scan failures from crashing the app

5.8 Optional Features (Added Later)

Feature	Description
🔎 Filter logs	JS-based table filtering by URL/Scan Type
📤 Export logs to Excel	Added in `/export` route
📊 Pie chart in logs.html	Shows overall severity trend
📋 Unified CSV logging	Same format used by CLI and Flask

5.9 Conclusion

The implementation involved integrating multiple modules in a clean and maintainable architecture. Reusability and user experience were prioritized. The scanner delivers real-time, easy-to-understand results to both technical and non-technical users. Export and chart features enhance usability and value for educational/demo purposes.

📘 Chapter 6: Testing and Result Analysis

6.1 Introduction

Once the development phase is completed, rigorous testing is essential to verify that the scanner works as expected across different conditions and inputs. This chapter outlines the **testing strategy**, documents **test cases**, presents **sample results**, and provides an analysis of the project's performance, reliability, and limitations.

6.2 Testing Methodology

We used **manual testing** with black-box techniques, targeting:

- ✅ Functionality testing (each scan feature)
- ✅ Input validation (invalid/malformed URLs)
- ✅ UI responsiveness (mobile & desktop views)
- ✅ Log generation and export accuracy
- ✅ Web server behavior on edge cases

The system was tested using various real and test websites, including safe demo URLs such as:

- http://testphp.vulnweb.com/
- https://example.com
- http://httpbin.org
- Intentionally malformed URLs

6.3 Test Environment

Component	Specification
OS	Windows 11 & 12 / Linux Ubuntu & kali

Python	Python 3.10 / 3.11
Browser	Chrome, Firefox
IDE	VS Code / PyCharm
Terminal	CMD / Bash
Libraries	Flask, requests, BeautifulSoup, pandas

6.4 Sample Test Cases

🔎 Test Case 1: XSS Scan

Parameter	Value
Input URL	http://testphp.vulnweb.com/
Expected Result	Detect input forms reflecting JS
Actual Result	✅ XSS vulnerability detected

🔎 Test Case 2: SQL Injection Detection

Parameter	Value
Input URL	http://testphp.vulnweb.com/listproducts.php?cat=1
Expected Result	SQL error message on injecting payload
Actual Result	✅ SQL error found in response

🔎 Test Case 3: Security Headers Check

Parameter	Value
Input URL	https://example.com
Expected Result	Missing headers flagged
Actual Result	✅ Content-Security-Policy missing
Notes	Safe site, but headers are minimal

🔎 Test Case 4: Cookie Flag Test

Parameter	Value
Input URL	http://httpbin.org/cookies/set?mycookie=value
Expected Result	Check for HttpOnly & Secure flags
Actual Result	✅ HttpOnly flag missing (as expected)

🔎 Test Case 5: Export Logs

Parameter	Value
Action	Visit /logs → Click “Export”
Expected	Excel file is downloaded with full logs
Actual	✅ File downloaded, values match CSV
Format	.xlsx, with all scan columns

❌ Test Case 6: Invalid URL Handling

Input	http:/bad_url
Expected	Error message
Actual	✅ Error handled, user shown “Invalid URL”

6.5 UI Testing

- ✅ Works on mobile (Bootstrap)
- ✅ Filters work instantly using JavaScript
- ✅ No crashes during scan reruns
- ✅ Clear color-coded chart summaries

6.6 Performance Observations

URL Type	Avg. Scan Time

Normal HTML page	3–5 seconds
JavaScript-heavy page	7–10 seconds
Broken URL	< 1 second (fails early)
CLI execution	Fast, low memory usage
Flask server	Light, even on low RAM systems

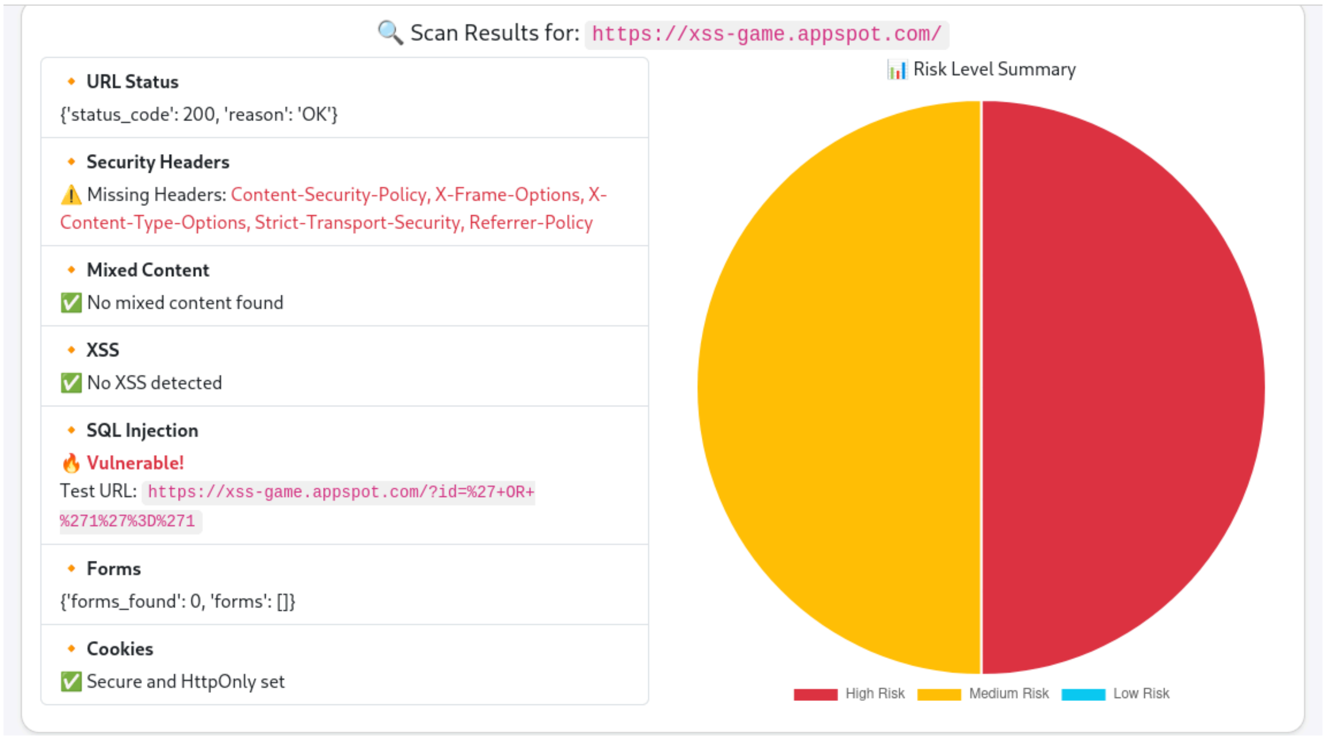
6.7 Result Analysis

The scanner produces reliable results on:

- XSS: form input reflection detection
- SQLi: error message detection from payloads
- Headers: missing standard headers
- Cookies: flag check

📊 Visualization Output

The pie chart component displayed accurate summaries of High, Medium, and Low risks.



🔍 Scan Results for: `https://xss-game.appspot.com/`

🔸 **URL Status**
{'status_code': 200, 'reason': 'OK'}

🔸 **Security Headers**
⚠️ Missing Headers: Content-Security-Policy, X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security, Referrer-Policy

🔸 **Mixed Content**
✅ No mixed content found

🔸 **XSS**
✅ No XSS detected

🔸 **SQL Injection**
🔥 **Vulnerable!**
Test URL: `https://xss-game.appspot.com/?id=%27+OR+%271%27%3D%271`

🔸 **Forms**
{'forms_found': 0, 'forms': []}

🔸 **Cookies**
✅ Secure and HttpOnly set

6.8 Strengths

- ✅ Dual-mode scanning: CLI + Web
- ✅ Reusable scan engine
- ✅ Chart-based visualization
- ✅ Filterable and exportable logs
- ✅ Beginner-friendly and lightweight

6.9 Limitations

Limitation	Reason
No deep crawler	To keep the tool lightweight
Limited to public URLs	Cannot scan behind authentication
No advanced payload fuzzing	Basic XSS/SQLi logic only
No SSL/TLS inspection	Out of project scope
One-page at a time	No site-wide scan or spidering

6.10 Conclusion

Testing confirmed that the scanner meets its intended goals. It detects basic web vulnerabilities, provides clear output and visualization, and serves as a valuable tool for students and developers learning web security. Though not a replacement for enterprise scanners, its simplicity and modular design make it a highly effective learning tool.

📘 Chapter 7: Conclusion and Future Work

7.1 Conclusion

The **Advanced Web Application Security Scanner** project was developed with the primary objective of providing a beginner-friendly, lightweight, and practical tool to identify common vulnerabilities in web applications. The tool supports both **CLI** and **Flask-based web UI**, enabling users to interact with it according to their preferences or environment.

Through this project, we achieved the following:

- Developed a **modular scan engine (**scanner_core.py**)** capable of detecting:
    - Cross-Site Scripting (XSS)
    - SQL Injection
    - Missing security headers
    - Cookie flag issues
    - Mixed content risks
- Built a **command-line interface (**scanner.py**)** for quick scans directly from the terminal
- Designed a clean **Bootstrap-based web dashboard (**app.py **+ HTML templates)** for visual scanning, chart-based summaries, and scan history management
- Implemented **logging**, **filtering**, and **Excel export** for audit trails and documentation
- Made the tool educational and useful for students, developers, and early-stage security testers

This scanner is a balance between **functionality, clarity, and simplicity**, focusing on real-world vulnerabilities and visual reporting to improve security awareness and learning.

7.2 Achievements

Feature	Completed?

CLI Scan Engine	✅ Yes
Flask Web Dashboard	✅ Yes
Log to CSV	✅ Yes
Excel Export	✅ Yes
Severity Pie Chart	✅ Yes
Form & Input Validation	✅ Yes
Modular Code Structure	✅ Yes
Reusable Core Logic	✅ Yes

The final system was tested thoroughly and responded accurately across a wide range of websites and inputs.

7.3 Key Learnings

This project helped reinforce practical knowledge of:

- Web application vulnerabilities
- Python modules for requests, parsing, and data handling
- Web development using Flask and Bootstrap
- Working with real-time data logging
- Creating interactive UI components using Chart.js
- Structuring a project for **both CLI and GUI** environments

You also gained exposure to **software architecture**, modularity, reusable code practices, and **clear UI/UX design** — all essential for a full-stack security tool.

7.4 Future Enhancements

While the project is feature-complete as per the goals, several improvements can be considered in future versions:

Future Feature	Description
🔐 Authentication support	Scan behind login pages using session handling
🕷️ Site spidering	Crawl all pages of a site before scanning
🔄 Scheduling	Run automatic scans at intervals
📊 Dashboard filters	Add filters by date, severity, or scan type
🌐 API-based scanning	Offer a REST API for integration with other systems
🛡️ Machine Learning	Analyze historical scan data to predict risks
📱 Mobile App	Create a mobile-friendly version using Flutter or React Native

7.5 Final Thoughts

This project fulfills its purpose of providing a **practical, understandable, and effective** security scanner. It enables users to:

- Learn about web vulnerabilities
- Test their own applications
- Document results professionally
- Visualize risk patterns

It stands as a **foundation for larger, more sophisticated tools** and serves as an ideal project for academic demonstration, training, or workshops.

📘 Chapter 8: References and Appendix

8.1 References

Below is a list of all sources, libraries, tools, and standards referred to during the design and development of this project.

📚 Web Security Standards & Documentation

1. OWASP Foundation. "OWASP Top 10 Web Application Security Risks."
   🔗 https://owasp.org/www-project-top-ten/

2. Mozilla Web Security Guidelines.
   🔗 https://infosec.mozilla.org/guidelines/web_security

3. RFC 6265 – HTTP State Management Mechanism (Cookies).
   🔗 https://tools.ietf.org/html/rfc6265

💻 Python Libraries and Tools

4. Requests: HTTP for Humans –
   🔗 https://docs.python-requests.org/

5. BeautifulSoup4: HTML/XML parser –
   🔗 https://www.crummy.com/software/BeautifulSoup/

6. Flask: Web framework for Python –
   🔗 https://flask.palletsprojects.com/

7. Pandas: Data analysis and Excel export –
   🔗 https://pandas.pydata.org/

8. OpenPyXL: Excel file handling –
   🔗 https://openpyxl.readthedocs.io/

📊 Front-End Tools

9. Bootstrap 5 –
   🔗 https://getbootstrap.com/

10. Chart.js –
🔗 https://www.chartjs.org/

11. DataTables (optional for filtering) –
🔗 https://datatables.net/

🔍 Learning Resources

12. "SQL Injection Cheat Sheet" – PortSwigger
🔗 https://portswigger.net/web-security/sql-injection/cheat-sheet

13. "XSS Cheat Sheet" – OWASP
🔗 https://owasp.org/www-community/xss-filter-evasion-cheatsheet

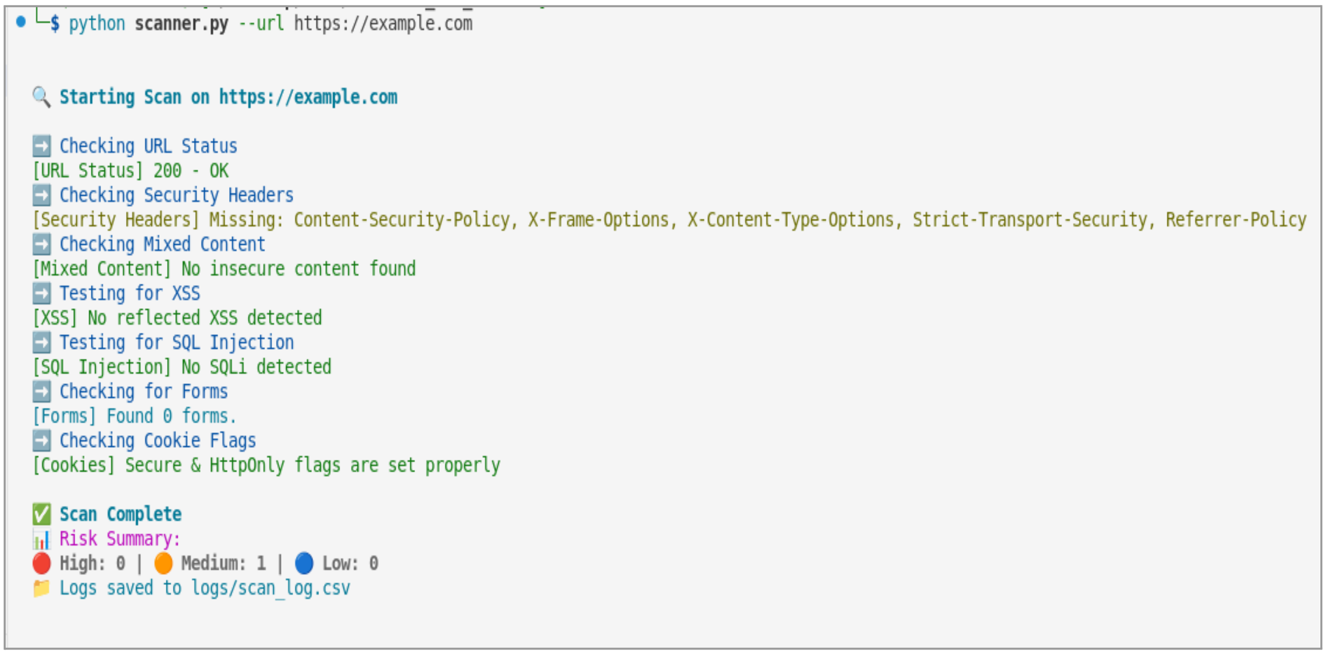
14. Sample test sites used:
    - http://testphp.vulnweb.com/
    - https://example.com
    - http://httpbin.org

8.2 Appendix

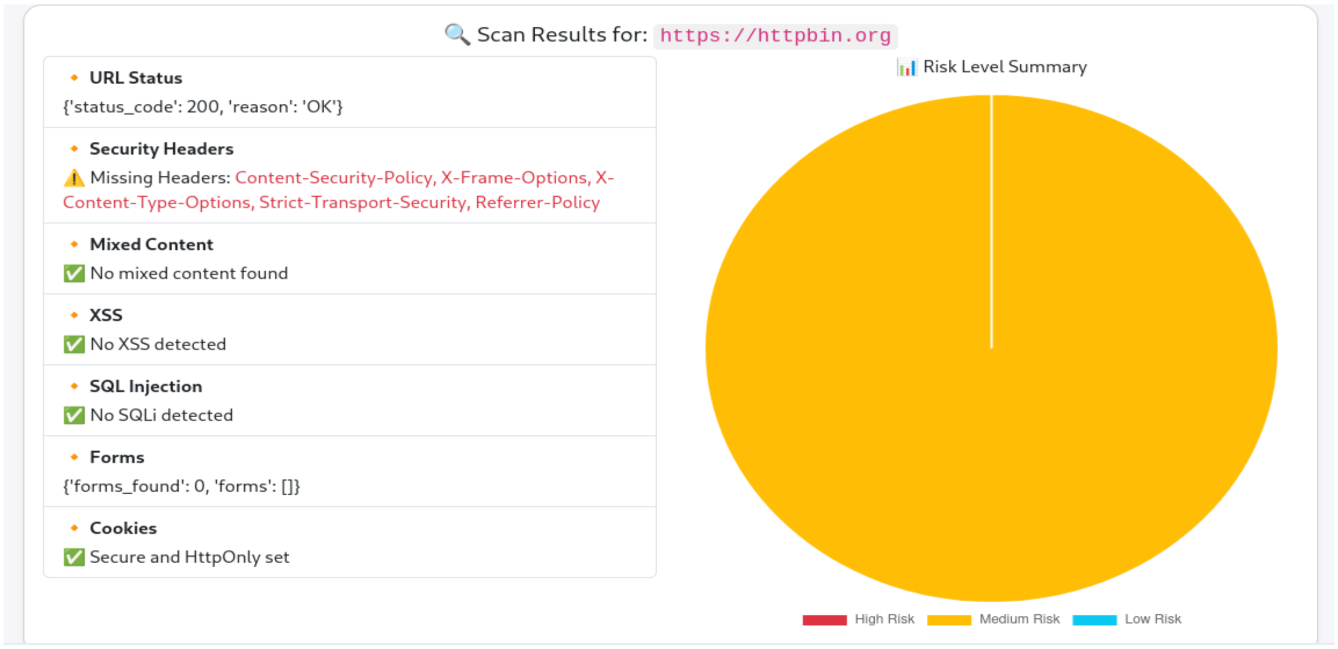
This section will include **screenshots and additional information** to support your report and presentation. Insert them manually into your Word document as shown below.

A. Screenshots to Include (with headings)

Screenshot Name	Description
1. CLI Scan Output	Terminal result from scanner.py
2. Web Scanner UI	Form + results view from index.html
3. Scan Summary Pie Chart	Chart showing risk categories
4. Logs Table	All scan logs listed
5. Filter and Search	Demonstration of filtering scan logs
6. Export to Excel	Screenshot of exported_logs.xlsx
7. File Structure	Folder view showing project layout

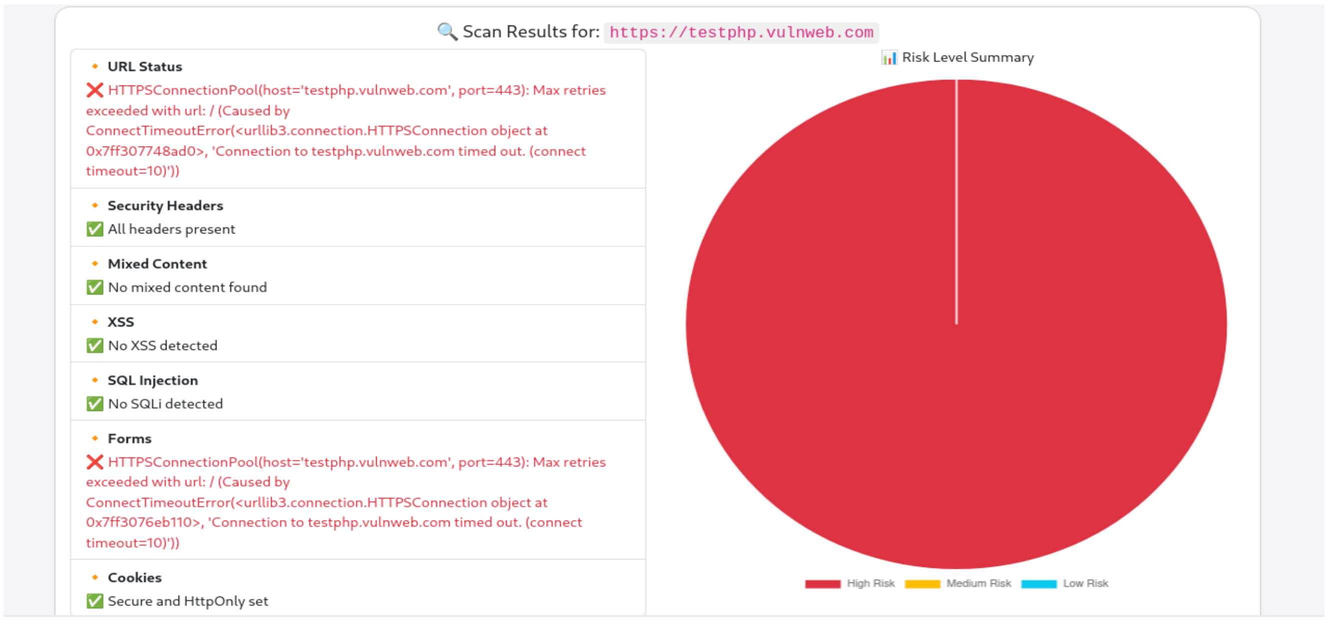


```
└─$ python scanner.py --url https://example.com


🔍 Starting Scan on https://example.com

➡️ Checking URL Status
[URL Status] 200 - OK
➡️ Checking Security Headers
[Security Headers] Missing: Content-Security-Policy, X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security, Referrer-Policy
➡️ Checking Mixed Content
[Mixed Content] No insecure content found
➡️ Testing for XSS
[XSS] No reflected XSS detected
➡️ Testing for SQL Injection
[SQL Injection] No SQLi detected
➡️ Checking for Forms
[Forms] Found 0 forms.
➡️ Checking Cookie Flags
[Cookies] Secure & HttpOnly flags are set properly

✅ Scan Complete
📊 Risk Summary:
🔴 High: 0 | 🟠 Medium: 1 | 🔵 Low: 0
📁 Logs saved to logs/scan_log.csv
```



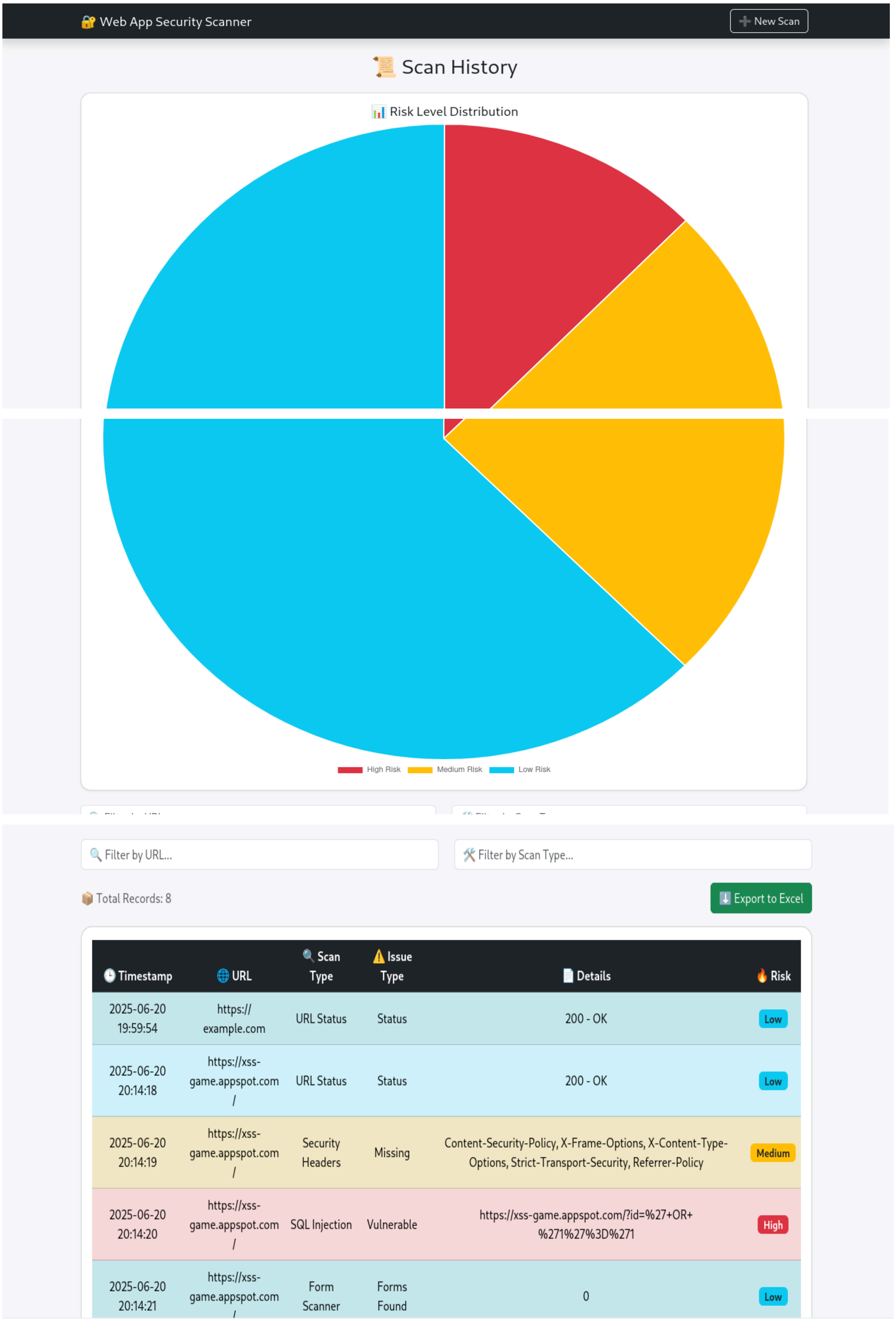
🔍 Scan Results for: `https://httpbin.org`

- **URL Status**
  {'status_code': 200, 'reason': 'OK'}
- **Security Headers**
  ⚠️ Missing Headers: Content-Security-Policy, X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security, Referrer-Policy
- **Mixed Content**
  ✅ No mixed content found
- **XSS**
  ✅ No XSS detected
- **SQL Injection**
  ✅ No SQLi detected
- **Forms**
  {'forms_found': 0, 'forms': []}
- **Cookies**
  ✅ Secure and HttpOnly set



🔍 Scan Results for: `https://testphp.vulnweb.com`

- **URL Status**
  ❌ HTTPSConnectionPool(host='testphp.vulnweb.com', port=443): Max retries exceeded with url: / (Caused by ConnectTimeoutError(<urllib3.connection.HTTPSConnection object at 0x7ff307748ad0>, 'Connection to testphp.vulnweb.com timed out. (connect timeout=10)'))
- **Security Headers**
  ✅ All headers present
- **Mixed Content**
  ✅ No mixed content found
- **XSS**
  ✅ No XSS detected
- **SQL Injection**
  ✅ No SQLi detected
- **Forms**
  ❌ HTTPSConnectionPool(host='testphp.vulnweb.com', port=443): Max retries exceeded with url: / (Caused by ConnectTimeoutError(<urllib3.connection.HTTPSConnection object at 0x7ff3076eb110>, 'Connection to testphp.vulnweb.com timed out. (connect timeout=10)'))
- **Cookies**
  ✅ Secure and HttpOnly set

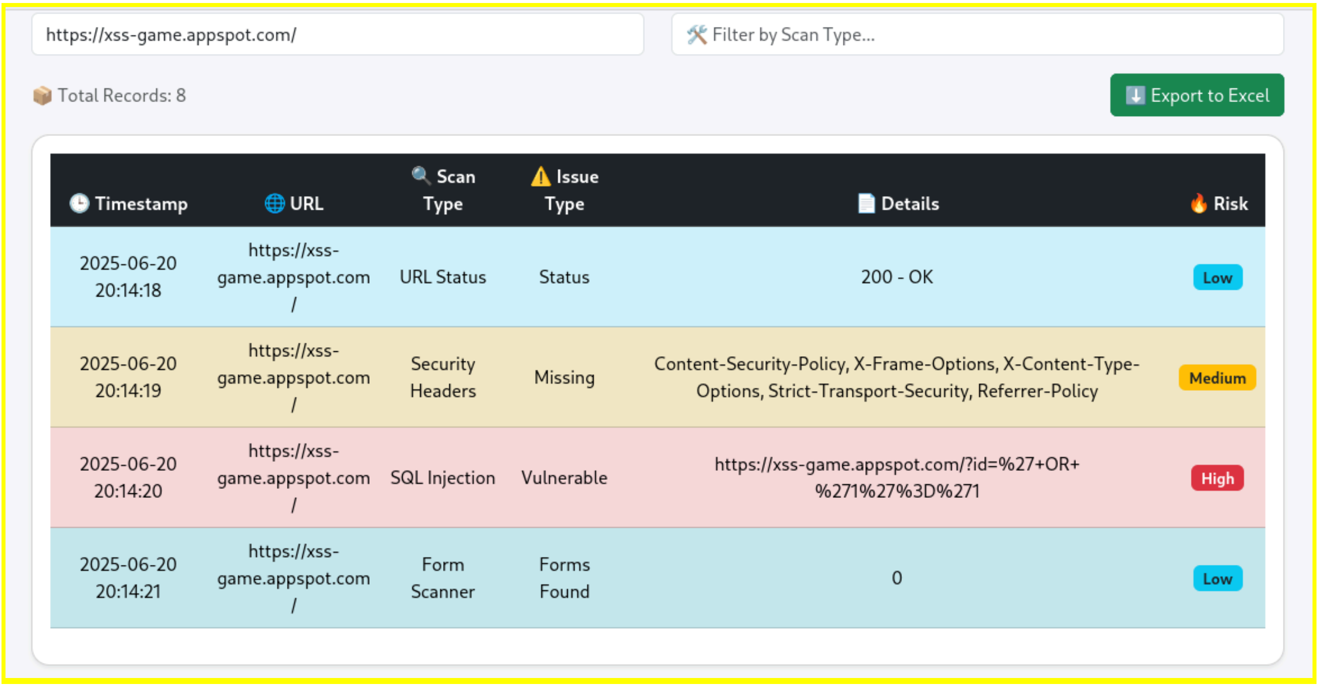


📜 Scan History

📊 Risk Level Distribution

📦 Total Records: 8

🕒 Timestamp	🌐 URL	🔍 Scan Type	⚠️ Issue Type	📄 Details	🔥 Risk
2025-06-20 19:59:54	https://example.com	URL Status	Status	200 - OK	Low
2025-06-20 20:14:18	https://xss-game.appspot.com/	URL Status	Status	200 - OK	Low
2025-06-20 20:14:19	https://xss-game.appspot.com/	Security Headers	Missing	Content-Security-Policy, X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security, Referrer-Policy	Medium
2025-06-20 20:14:20	https://xss-game.appspot.com/	SQL Injection	Vulnerable	https://xss-game.appspot.com/?id=%27+OR+%271%27%3D%271	High
2025-06-20 20:14:21	https://xss-game.appspot.com/	Form Scanner	Forms Found	0	Low

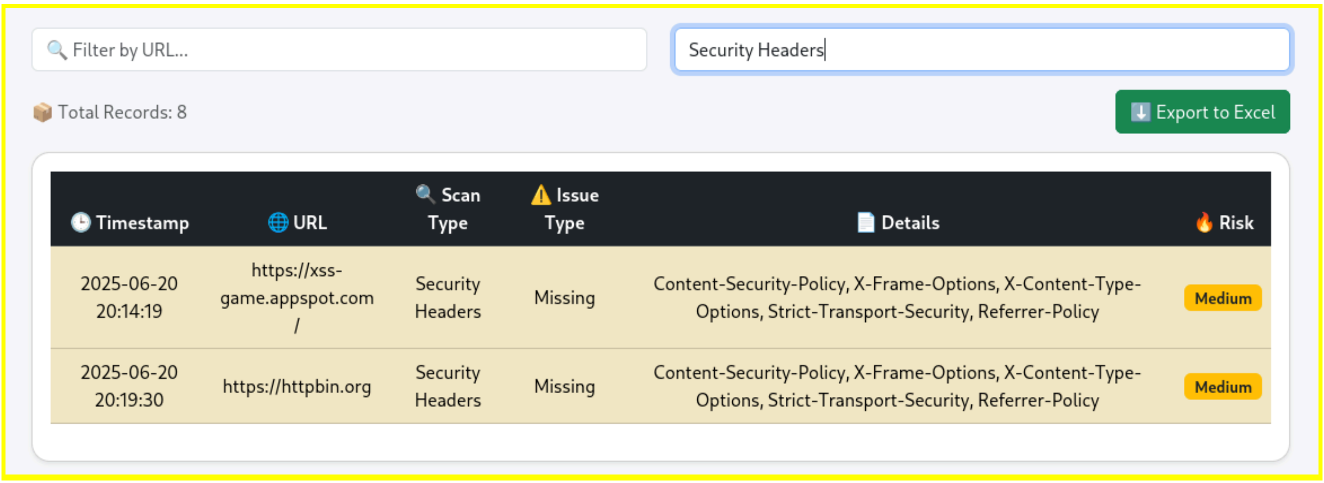

https://xss-game.appspot.com/

Filter by Scan Type...

Total Records: 8

Export to Excel

Timestamp	URL	Scan Type	Issue Type	Details	Risk
2025-06-20 20:14:18	https://xss-game.appspot.com/	URL Status	Status	200 - OK	Low
2025-06-20 20:14:19	https://xss-game.appspot.com/	Security Headers	Missing	Content-Security-Policy, X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security, Referrer-Policy	Medium
2025-06-20 20:14:20	https://xss-game.appspot.com/	SQL Injection	Vulnerable	https://xss-game.appspot.com/?id=%27+OR+%271%27%3D%271	High
2025-06-20 20:14:21	https://xss-game.appspot.com/	Form Scanner	Forms Found	0	Low

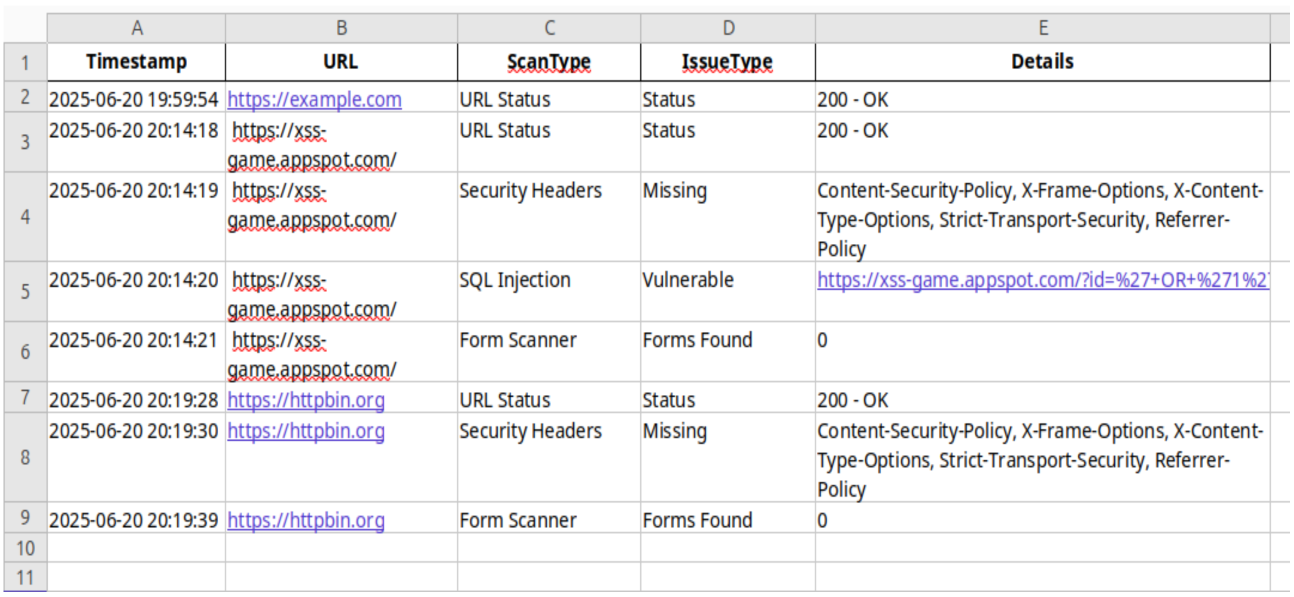

Filter by URL...

Security Headers

Total Records: 8

Export to Excel

Timestamp	URL	Scan Type	Issue Type	Details	Risk
2025-06-20 20:14:19	https://xss-game.appspot.com/	Security Headers	Missing	Content-Security-Policy, X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security, Referrer-Policy	Medium
2025-06-20 20:19:30	https://httpbin.org	Security Headers	Missing	Content-Security-Policy, X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security, Referrer-Policy	Medium


	A	B	C	D	E
1	Timestamp	URL	ScanType	IssueType	Details
2	2025-06-20 19:59:54	https://example.com	URL Status	Status	200 - OK
3	2025-06-20 20:14:18	https://xss-game.appspot.com/	URL Status	Status	200 - OK
4	2025-06-20 20:14:19	https://xss-game.appspot.com/	Security Headers	Missing	Content-Security-Policy, X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security, Referrer-Policy
5	2025-06-20 20:14:20	https://xss-game.appspot.com/	SQL Injection	Vulnerable	https://xss-game.appspot.com/?id=%27+OR+%271%2
6	2025-06-20 20:14:21	https://xss-game.appspot.com/	Form Scanner	Forms Found	0
7	2025-06-20 20:19:28	https://httpbin.org	URL Status	Status	200 - OK
8	2025-06-20 20:19:30	https://httpbin.org	Security Headers	Missing	Content-Security-Policy, X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security, Referrer-Policy
9	2025-06-20 20:19:39	https://httpbin.org	Form Scanner	Forms Found	0
10					
11					

```
advanced_web_scanner/
│
├── scanner_core.py        # Core scanning logic
├── scanner.py             # CLI script
├── app.py                 # Flask server
├── scan_log.csv           # Log file
├── exported_logs.xlsx     # Exported Excel log
├── requirements.txt       # Python dependencies
│
├── templates/
│   ├── index.html         # Main scanner UI
│   └── logs.html          # Scan history UI
│
└── static/                # (Optional: CSS, JS)
```

B. Sample Log File Format (CSV)

```
Timestamp      URL    ScanType       IssueType       Details
2025-06-20 19:59:54  https://example.com   URL Status     Status  200 - OK
2025-06-20 20:14:19   https://xss-game.appspot.com/         Security Headers       Missing
Content-Security-Policy, X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security,
Referrer-Policy
```

C. Commands to Run the Tool

✅ **CLI Scan:**

```
python scanner.py https://example.com
```

✅ **Start Flask App:**

```
python app.py
```

8.3 Final Note

This appendix serves as an additional reference for viva examiners and evaluators, offering a visual and technical snapshot of the system functionality, implementation, and outcomes.