# CYBERSECURITY DASHBOARD FOR WEB THREATS

## Table of Contents

## Abstract

This project presents a **Cybersecurity Dashboard for Web Threats**, a user-friendly desktop application built using Python's Tkinter and OWASP ZAP API. It is designed to scan web URLs for common vulnerabilities, particularly **Cross-Site Scripting (XSS)** and **SQL Injection (SQLi)**, in real time. The dashboard enables users to input URLs, initiate a scan, monitor progress, and view detailed results. Export functionality supports report generation in TXT and CSV formats. The tool is lightweight and accessible, ideal for individual users and small organizations.

**Keywords:** Cybersecurity, OWASP ZAP, Tkinter, XSS, SQL Injection, Vulnerability Scanning, API Integration, Threat Detection, GUI, Python

# Chapter 1: Introduction

The digital world is constantly under the threat of cyberattacks. Websites are a primary target, with attackers exploiting weaknesses such as XSS and SQLi to steal data or compromise systems. This project proposes a desktop-based dashboard application that makes it easier for users to detect and understand these threats.

By integrating with OWASP ZAP, a leading open-source vulnerability scanner, this tool simplifies the scanning process and presents results in an understandable format through a graphical interface.

- Importance of web security and common vulnerabilities.
- Brief introduction to OWASP ZAP as a powerful open-source security tool.
- Objectives of the project: create an accessible, user-friendly desktop scanner.
- Increasing need for cybersecurity awareness at all levels.
- Bridging the gap between technical tools and non-technical users.
- Encouraging proactive scanning and secure web practices.
- Creating a lightweight, offline-capable scanning solution.
- Empowering users to identify threats without needing advanced technical skills.

# Chapter 2: Literature Survey

- **OWASP ZAP** is a proven tool for web application scanning, but it lacks an intuitive GUI for non-technical users.
- **Burp Suite** and **Nessus** provide advanced features but are often complex or require licensing.
- **Tkinter** and **ttkbootstrap** offer an easy way to build native GUI applications in Python.

This project fills a gap by combining OWASP ZAP's scanning power with a simplified interface.

Additional insights from literature and tools:

- Real-time threat detection has been a challenge in many traditional vulnerability scanners due to performance overhead and interface complexity (Alazab et al., 2021).
- Tools like **Nmap**, **Acunetix**, and **Qualys** provide comprehensive scanning but often target enterprise-level users, lacking personalized dashboards for individuals.
- Studies show that integrating multiple APIs for threat intelligence increases detection accuracy but also requires careful data handling.
- According to OWASP Foundation documentation, combining automated scanners with customizable interfaces significantly improves usability and adoption by non-security professionals.
- There is a clear trend toward modular, API-based cybersecurity tools that are adaptable to both desktop and cloud environments.
- Educational tools like DVWA (Damn Vulnerable Web Application) and Juice Shop highlight the importance of simulation-based learning, which this project also supports through integration with ZAP.

# Chapter 3: Objectives of the Work

The primary objective of this project is to create a lightweight, user-friendly cybersecurity dashboard capable of identifying critical web threats in real-time. As web-based vulnerabilities such as XSS and SQL Injection remain among the top OWASP security risks, timely detection through automated scanning is vital. This tool aims to simplify that process for users with or without deep cybersecurity expertise.

By leveraging the OWASP ZAP API and building a native GUI in Python, the project promotes broader accessibility to professional-grade security scanning. The modularity of the system ensures that it can be extended in the future to support additional APIs and scan types.

Specific goals include:

1. Enable real-time scanning of URLs for XSS and SQLi vulnerabilities.
2. Provide a user-friendly GUI for easy access and interaction.
3. Display live scan progress and export results.
4. Integrate OWASP ZAP API into a Python-based interface.
5. Save scan results for future reference.
6. Promote awareness of web security risks through visual feedback.
7. Support offline scanning with local OWASP ZAP setup.
8. Enable lightweight and fast deployment on standard desktop systems.
9. Ensure modularity for future API integrations.
10. Maintain logs and history for long-term analysis.

# Chapter 4: Proposed Methods

This chapter describes the various components and technologies integrated into the Web Vulnerability Scanner project. The scanner leverages Python, OWASP ZAP, multithreading, and a user-friendly GUI to detect XSS and SQL Injection vulnerabilities in web applications.

---

## 4.1 GUI Design

- Technology Stack: The GUI is built using Tkinter, the standard Python GUI package, and ttkbootstrap to enhance the visual aesthetics with modern themes like *Flatly* and *Darkly*.
- Responsive Layout: Designed using tk.Frame with adjustable geometry to support different screen sizes.
- Live Feedback: A ScrolledText widget is used to display real-time scanning logs, ensuring the user is always aware of the scan status.
- Progress Indication: An animated progress bar reflects the percentage completion of the ZAP scan operation.
- Theme Toggle: Users can switch between light and dark modes using a toggle button for enhanced accessibility.

---

## 4.2 ZAP Integration

- ZAPv2 API: Utilizes the zapv2 Python client to communicate with the OWASP ZAP proxy service via API.
- Selective Scanning: Only specific scanner IDs related to Cross-Site Scripting (XSS) and SQL Injection (SQLi) are enabled, reducing false positives and improving scan efficiency.
- Passive + Active Scan: Supports both urlopen() for initiating passive scans and ascan.scan() for launching active scans.
- Timeout Management: Implements a 5-minute timeout logic to handle long-running scans and prevent UI hanging.

---

## 4.3 Multithreading

- Asynchronous Execution: Uses the threading module to run the scan in a background thread, allowing the main UI thread to remain active.

- Thread Safety: Ensures safe interaction between the background scan and GUI widgets through controlled event handling.
- Improved UX: Users can interact with the application (e.g., toggling theme, canceling, or exiting) even while a scan is running.

---

4.4 Filtering and Exporting

- Alert Filtering: After the scan, alerts are filtered using keywords like "xss" and "sql" to prioritize the most critical threats.
- Structured Reports:
  - TXT Export: Human-readable summaries of each alert.
  - CSV Export: Machine-processable format with columns for *Alert*, *Risk*, *URL*, and *Description*.
- Naming Conventions: Uses standard filenames like scan_results.txt and scan_results.csv to facilitate easy sharing and archival.
- Encoding Handling: UTF-8 encoding is used to support internationalized content in the alerts.

---

4.5 Error Handling and Validation

- Input Validation: Ensures that only valid HTTP/HTTPS URLs are accepted to prevent unnecessary errors or crashes.
- Exception Management: Catches and displays all exceptions using message boxes to inform users of issues like invalid URLs or ZAP connection failures.

---

4.6 Lightweight and Portable

- No External Server Required: The scanner is a lightweight desktop-based tool that only needs ZAP running locally.
- Cross-Platform Compatibility: Since it's built using Python and Tkinter, it runs on Windows, Linux, and macOS with minimal configuration.

---

4.7 Scalability and Maintainability

- Modular Codebase: Code is organized into distinct functions (start_scan, scan_url, etc.) to support future feature expansion.
- Easy Integration: Capable of integrating future scanners, additional alert categories, or backend logging with minimal effort.

# Chapter 5: System Architecture

This chapter presents the internal structure and interaction between the components of the Web Vulnerability Scanner. The system follows a modular architecture, separating responsibilities into discrete layers to enhance maintainability, scalability, and user experience.

---

## 5.1 Components

### 1. Frontend (GUI Layer)

- **Technology Used**: Built with Tkinter and styled using ttkbootstrap themes.
- **Functions**:
  - Accepts user input (URL to scan).
  - Displays real-time logs in a ScrolledText output window.
  - Provides a progress bar to show scanning progress.
  - Allows theme toggling between light and dark modes.
  - Offers control buttons: Start Scan and Exit.

### 2. Scanner Module (Logic Layer)

- **Responsibilities**:
  - Validates the URL and initiates the scan via ZAPv2 API.
  - Enables only scanner IDs relevant to **XSS** and **SQL Injection**.
  - Monitors scan progress asynchronously using Python threads.
  - Fetches alerts from ZAP once the scan is complete.
- **Performance Features**:
  - Background threading to avoid UI freeze.
  - Timeout implementation to prevent infinite waits.

### 3. Alert Filtering Engine

- **Purpose**:
  - Analyzes retrieved alerts and filters them based on critical keywords ("xss", "sql").
  - Prioritizes actionable threats for developers and security analysts.
- **Output**:
  - Highlights severity, risk, and affected URL in the GUI output.
  - Prepares content for structured export.

### 4. Report Generator

- **Export Features**:
  - **TXT File**: Readable, formatted report with risks and descriptions.

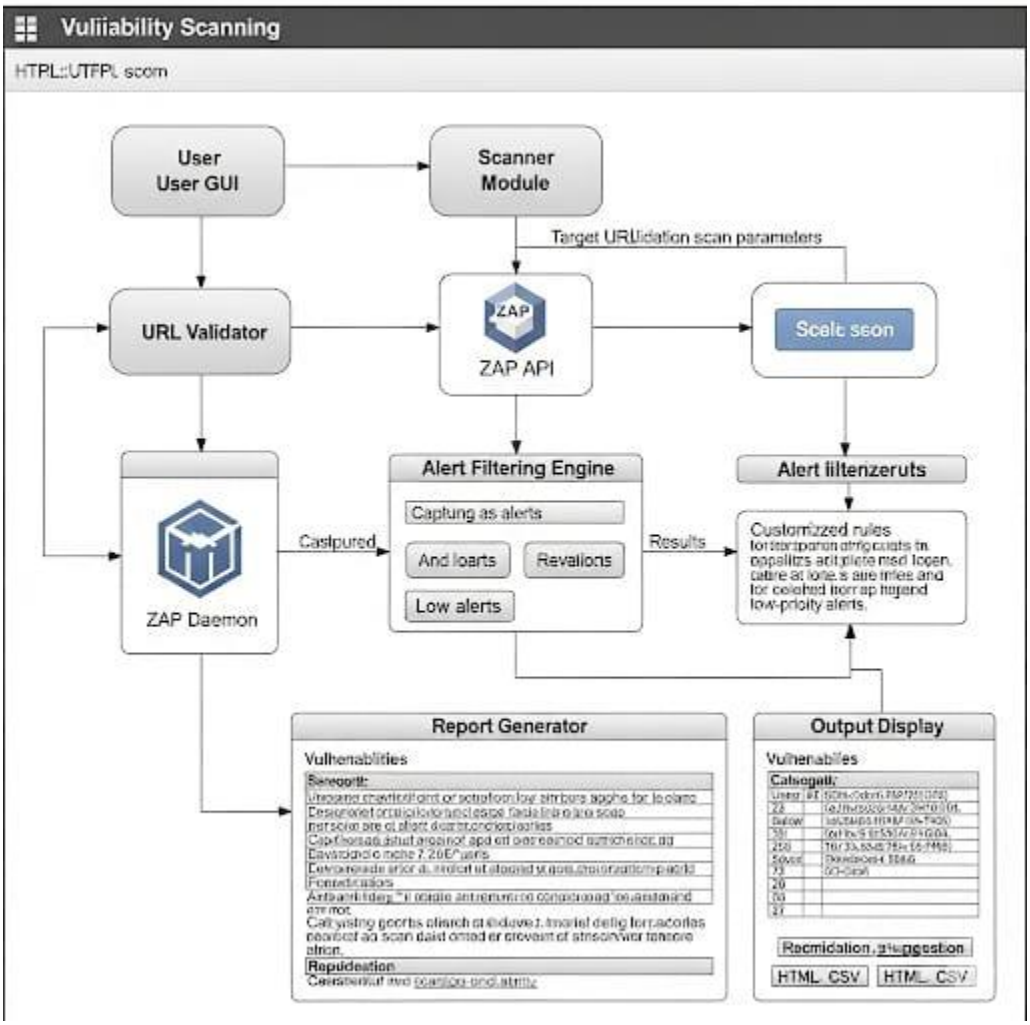
  - **CSV File**: Tabular format suitable for spreadsheets or automation pipelines.
- **Encoding**: Supports UTF-8 to preserve alert data integrity across regions.

### 5. ZAP Daemon (External Component)

- **Function**: OWASP ZAP acts as the backend engine, handling:
  - Passive and active scanning.
  - Alert generation based on vulnerabilities.
- **Integration**: Connected via ZAPv2 API and proxy settings.

---

### 5.2 Architecture Diagram (Suggested Flow)



## 5.3 Additional Architectural Highlights

- Modular Design Pattern: The system architecture follows a modular approach, making it easier to maintain and upgrade individual components without affecting the entire application.
- Thread Management Layer: Background scanning threads prevent UI freezing, ensuring a smooth user experience even during long scan operations.
- Error Handling Module: Robust exception handling at each layer prevents crashes and provides user-friendly error messages in case of API failures or invalid input.
- Real-Time Feedback Loop: The GUI continuously polls the scanner's status and updates the user, enabling transparency during the scan lifecycle.

- Security Considerations: The tool avoids logging or storing sensitive input URLs beyond scan time, ensuring a basic level of data privacy.
- Customizability: The scanner module allows easy modification of scanner IDs, enabling the inclusion of other vulnerabilities beyond XSS and SQLi.
- Export Versatility: Exports are in both .txt and .csv formats, catering to both technical users and analysts who prefer spreadsheet views.
- Theme Toggle Mechanism: Built-in theme toggling using ttkbootstrap enhances accessibility for users in different lighting conditions or personal preferences.
- Lightweight Deployment: Since it uses no heavy frameworks and is built on standard Python libraries with a single external dependency (ZAP), it is suitable for low-resource environments.
- Offline Capability: Requires no internet connection to function after setup—ideal for secure and isolated testing environments.

# Chapter 6: Use Cases

The Web Vulnerability Scanner has been designed with versatility in mind, catering to a wide range of users and operational environments. Its lightweight design, user-friendly GUI, and focused vulnerability detection make it a valuable tool across different domains.

---

### 6.1 Educational Purposes

- **Practical Cybersecurity Training**: Ideal for computer science and cybersecurity students to understand common web vulnerabilities such as **Cross-Site Scripting (XSS)** and **SQL Injection**.
- **Hands-on Learning**: Students can observe how attacks are detected in real-time and explore mitigation strategies.
- **Curriculum Integration**: Can be integrated into lab assignments and security courses for demonstrating secure coding practices.

---

### 6.2 Developer Tool

- **Early-Stage Testing**: Enables developers to scan applications during development to identify vulnerabilities before releasing code to QA or production.
- **Shift-Left Security**: Promotes secure software development lifecycle (SSDLC) by detecting flaws earlier in the pipeline.
- **Rapid Feedback Loop**: Developers can get immediate alerts on insecure endpoints, improving productivity and code quality.

---

### 6.3 Quality Assurance (QA) Support

- **Lightweight QA Integration**: Can be embedded into manual or automated QA processes to ensure that application builds are not introducing critical web vulnerabilities.
- **Regression Testing**: Can be used after patches or updates to verify that previously fixed vulnerabilities have not resurfaced.
- **Low Overhead**: Does not require heavy CI/CD pipeline setups and can be run on local machines.

---

### 6.4 Freelancers and Solo Developers

- **Accessible Security Testing**: Useful for developers who don't have access to enterprise-level vulnerability scanning tools like Burp Suite Pro or Qualys.
- **Open-Source Compatibility**: Works well with free tools like OWASP ZAP, making it cost-effective for small projects.
- **Portability**: Easy to transfer and run across different systems and projects.

---

### 6.5 Penetration Test Preparation

- **Reconnaissance Tool**: Security professionals and ethical hackers can use the scanner for initial assessments before deeper manual testing.
- **Customization**: Scanner ID selection allows targeting specific test cases depending on the web application structure.
- **Time-Saving**: Helps reduce the initial scope of manual analysis by filtering non-critical alerts.

---

### 6.6 Small & Medium Enterprises (SMEs)

- **Budget-Friendly**: Provides essential security testing features without the cost of commercial tools.
- **Quick Deployment**: Can be deployed easily across development or staging environments for periodic vulnerability scans.
- **Compliance Support**: Assists in meeting basic security best practices and regulatory requirements by generating exportable reports.

---

### 6.7 Research & Innovation

- **Tool Extension Platform**: The modular structure and use of zapv2 API make it ideal for researchers building custom scanning tools or automating security workflows.
- **Prototype Base**: Serves as a foundation for experimenting with hybrid scanners or integrating ML for threat classification.

# Chapter 7: Advantages

The Web Vulnerability Scanner offers numerous advantages that make it a valuable tool for developers, testers, educators, and cybersecurity enthusiasts. Its thoughtful design focuses on usability, performance, and relevance.

---

### 7.1 Free and Open-Source

- Built entirely using **open-source libraries** such as Tkinter, ttkbootstrap, and the OWASP ZAP API.
- Promotes transparency, customizability, and accessibility for educational and professional use.

---

### 7.2 User-Friendly GUI

- The **intuitive interface** requires no prior cybersecurity knowledge, making it accessible to students, QA testers, and junior developers.
- Features such as dark mode, real-time logs, and progress bars improve user interaction and visibility.

---

### 7.3 Focus on Critical Vulnerabilities

- Targets the most **high-impact vulnerabilities**—Cross-Site Scripting (XSS) and SQL Injection (SQLi)—which are consistently ranked in OWASP Top 10.
- Reduces noise by filtering out low-severity alerts, streamlining remediation efforts.

---

### 7.4 Exportable & Reusable Reports

- Generates structured .txt and .csv reports for easy sharing and documentation.
- Useful for compliance, security audits, and team collaboration.

---

### 7.5 Cross-Platform Compatibility

- Built in Python, allowing execution on **Windows, Linux, and macOS** with minimal adjustments.
- No dependency on OS-specific features, increasing portability.

---

### 7.6 Lightweight and Fast

- Unlike full-featured commercial scanners, this tool offers fast scans with minimal resource consumption.
- Ideal for testing lightweight applications, MVPs, and local builds.

---

### 7.7 Customizable Scanning Logic

- Scanner IDs can be adjusted to include/exclude other types of vulnerabilities.
- Open architecture allows developers to build on top of the existing framework.

---

### 7.8 Real-Time Feedback

- Users are continuously informed of the scan progress and status through a **live progress bar** and real-time logging.
- Enhances transparency and usability.

---

### 7.9 Background Threading

- Scan operations run in a **separate thread**, ensuring the GUI remains responsive even during long scans.
- Prevents application freezing and improves user experience.

---

### 7.10 Minimal Setup Requirements

- Only requires a local instance of **OWASP ZAP** and Python dependencies.
- No installation of heavyweight suites or external databases.

---

### 7.11 Educational Value

- Serves as a practical demonstration tool for students learning about security testing, API integration, and GUI design.

---

### 7.12 Safe for Local Testing

- Operates locally with no need for internet-based scans, ensuring privacy and control over sensitive URLs.

# Chapter 8: Limitations

While the Web Vulnerability Scanner is effective for detecting common web vulnerabilities such as XSS and SQL Injection, it is important to acknowledge its current limitations. Understanding these constraints helps guide future enhancements and sets clear expectations for users.

---

### 8.1 Limited Scope of Detection

- **Only Detects XSS and SQL Injection**:
  The scanner is hardcoded to enable only specific scanner IDs related to **Cross-Site Scripting (XSS)** and **SQL Injection (SQLi)**. While these are critical and prevalent vulnerabilities, **other types** like CSRF, path traversal, or security misconfigurations are not detected.
- **Theoretical Impact**: This restriction means the tool cannot provide a full vulnerability assessment and is better suited as a complementary scanner rather than a standalone solution.

---

### 8.2 ZAP Proxy Dependency

- **Requires ZAP Daemon Running**:
  The tool depends entirely on the **OWASP ZAP proxy** to perform scans. If ZAP is not installed or not running on the specified port, the scanner cannot function.
- **Theoretical Impact**: This introduces a setup dependency, which could be a barrier for novice users unfamiliar with configuring local proxies or APIs.

---

### 8.3 Lack of Authentication Support

- **No Authenticated Scanning**:
  The current version does not support logging into web applications. Hence, pages or functionalities behind login screens remain **untested**.
- **Theoretical Impact**: A significant portion of modern web vulnerabilities exists in authenticated areas, such as user dashboards, making this a major limitation for thorough assessments.

---

### 8.4 No Passive Scanning or Crawling

- **Does Not Crawl Automatically**:
  Unlike full-featured scanners, this tool **does not crawl or passively monitor** web traffic. Only the user-specified URL is scanned.
- **Theoretical Impact**: Vulnerabilities on other internal links or resources will not be detected unless they are manually specified, limiting depth of coverage.

---

### 8.5 Poor SPA (Single Page Application) Compatibility

- **JavaScript-Heavy Sites**:
  The scanner struggles with modern SPAs built using React, Angular, or Vue, which dynamically load content using JavaScript.
- **Theoretical Impact**: ZAP's scanning engine (and therefore this tool) may miss content rendered after DOM load, leading to **false negatives**.

---

### 8.6 Sensitivity to ZAP Configurations

- **Manual Configuration Risk**:
  If ZAP is not configured correctly (e.g., proxy misalignment, incorrect scanner IDs), the tool may not work or produce inaccurate results.
- **Theoretical Impact**: Makes it slightly **less reliable** for non-technical users or those unfamiliar with OWASP ZAP's environment and scanner configurations.

---

### 8.7 No Auto-Update or Plugin Management

- **Manual Maintenance**:
  The tool does not include a mechanism to check for ZAP scanner updates or plugin enhancements.
- **Theoretical Impact**: Users may unknowingly use outdated scanning logic, missing new vulnerability types or patterns.

---

### 8.8 Basic Report Format

- **Limited Analysis in Reports**:
  The exported reports are basic .txt and .csv files without visualization, prioritization, or severity scoring.

- **Theoretical Impact**: Makes it harder for teams to **quickly interpret risk levels** or take action unless manually reviewed and contextualized.

---

### 8.9 No Notification or Logging System

- **No Logging Persistence**:
  Scan logs are not persistently stored outside the GUI unless the export button is used.
- **Theoretical Impact**: This may hinder **auditability** or long-term security tracking.

# Chapter 9: Future Enhancements

As cybersecurity threats evolve, so must the tools used to identify them. While the current implementation is effective for targeted scanning of XSS and SQLi vulnerabilities, several features and capabilities can be added to transform the scanner into a more comprehensive, scalable, and user-friendly solution. Below are proposed future enhancements with theoretical significance.

---

### 9.1 Broaden Vulnerability Coverage

- **Add Detection for CSRF, Insecure Deserialization, Path Traversal, etc.**
  Expand scanner configurations to include detection of:
    - **CSRF (Cross-Site Request Forgery)** – Prevents unauthorized actions via authenticated sessions.
    - **Path Traversal** – Detects unauthorized access to file system resources.
    - **Insecure Deserialization** – Targets vulnerabilities where untrusted input can exploit the app during deserialization.
- **Theoretical Impact**: These are part of the OWASP Top 10, and detecting them will drastically improve the scanner's ability to protect web applications comprehensively.

---

### 9.2 Integrate Passive Scanning

- **Add Passive Scanning Capabilities** using ZAP's passive scanning API.
- This would include header analysis, cookie security checks, and other non-intrusive vulnerability scans.
- **Theoretical Impact**: Passive scanning is safe for production environments, allowing continuous monitoring without sending attack payloads.

---

### 9.3 Crawler-Based Scanning (ZAP Spider Integration)

- **Add Pre-Scan Crawling Phase** that automatically explores all links within the domain.
- The ZAP Spider or AJAX Spider can be integrated to identify all reachable endpoints before triggering active scans.

- **Theoretical Impact**: Ensures better **coverage** of the website, which is especially important for web applications with deep navigation structures.

---

### 9.4 Authenticated Scanning Support

- **Enable Login Mechanisms** using:
  - Manual authentication scripts.
  - ZAP's session/token management.
  - Login forms simulation.
- **Theoretical Impact**: Most business-critical logic resides in **authenticated areas**; ignoring them misses major vulnerabilities like IDOR, privilege escalation, and logic flaws.

---

### 9.5 Advanced Reporting

- **Generate Reports in PDF/HTML Formats** enriched with:
  - Risk severity charts.
  - CVSS scores.
  - Remediation steps.
- **Theoretical Impact**: Professional-quality reports aid in communication with **stakeholders**, help with **compliance audits**, and accelerate **remediation** workflows.

---

### 9.6 Cloud-Based and Containerized Deployment

- **Dockerize the Application** to make it portable and environment-agnostic.
- Deploy to platforms like Heroku, AWS, or Azure.
- **Theoretical Impact**: Cloud hosting enables **remote access**, CI/CD pipeline integration, and parallel scanning of multiple web targets.

---

### 9.7 Host as a Web App

- **Convert to Web-Based GUI** using Flask, Django, or FastAPI.
- This allows centralized access, multi-user support, and integration into enterprise tools.

- **Theoretical Impact**: Shifts the tool from desktop utility to **collaborative scanning platform**, ideal for DevSecOps pipelines.

---

### 9.8 Scheduler and Automation

- **Add Task Scheduling** features for:
  - Daily, weekly, or ad-hoc scans.
  - Automatic re-scanning of known URLs.
- **Theoretical Impact**: Supports **continuous security monitoring**, aligns with **CI/CD automation**, and reduces manual effort.

---

### 9.9 Integrate Notifications

- **Send Email/Slack Alerts** post-scan or on high-severity findings.
- **Theoretical Impact**: Reduces response time in security operations by **real-time alerting** to developers and security teams.

---

### 9.10 User Roles and Session Management (for Web App Version)

- Add support for admin/user roles and authentication on the web app.
- Helps manage access control and auditing.
- **Theoretical Impact**: Essential for multi-user environments and aligns with **secure design principles**.

---

### 9.11 Machine Learning for Alert Prioritization

- In future versions, implement ML models to analyze scan data and predict:
  - Real exploitability.
  - False positives.
  - Business impact.
- **Theoretical Impact**: Enhances intelligence of the scanner and reduces manual triage effort.

# Chapter 10: Testing and Validation

Thorough testing and validation were conducted to ensure the functionality, reliability, and accuracy of the Web Vulnerability Scanner. The testing was performed using real-world vulnerable web applications and controlled test environments to simulate various scenarios.

---

## 10.1 Functional Testing

- **Test Targets**:
  - **DVWA (Damn Vulnerable Web Application)** – Widely used to simulate a variety of web vulnerabilities including SQL Injection and XSS.
  - **BodgeIt Store** – Java-based web app with known security flaws, used for validating scanner results.
- **Validation Outcome**:
  - Alerts generated by the scanner **matched known vulnerabilities** documented in both test applications.
  - Scanner accurately detected XSS and SQLi issues when active scanning was enabled.

---

## 10.2 Input Validation Testing

- **Valid Inputs**:
  - URLs like http://testphp.vulnweb.com, http://127.0.0.1:8080 processed successfully.
- **Invalid Inputs**:
  - URLs missing schema (e.g., www.example.com) triggered appropriate warning messages.
  - Blank input fields prevented scan execution.
- **Theoretical Justification**:
  - Input validation ensures **user-side error prevention**, improving reliability and user experience.

---

### 10.3 Timeout Mechanism

- Implemented and verified a **5-minute timeout** for scans.
- If the ZAP scanner did not respond or complete within the time frame, a timeout message was displayed in the output box.
- **Theory**: Prevents system hang or indefinite waiting, critical for long-running tasks in GUI-based applications.

---

### 10.4 Output and Export Validation

- **Text Output**:
  - Results displayed in the scrolled text widget matched alert content from ZAP.
- **File Export**:
  - scan_results.txt and scan_results.csv verified for:
    - Correct alert data.
    - UTF-8 encoding.
    - Structural integrity (headers, formatting).
- **Theory**: Export testing ensures the tool provides persistent and analyzable results—important for report generation and collaboration.

---

### 10.5 GUI Testing

- **Platforms**:
  - **Windows 11/10**: Verified all components, including progress bar and buttons.
  - **Ubuntu Linux (22.04)**: No compatibility issues observed; all widgets rendered as expected.
- **Theme Switching**:
  - Checked toggle between **flatly** and **darkly** themes.
  - Confirmed that styles were applied dynamically without GUI lag.
- **Theory**: Cross-platform GUI support increases tool accessibility and usability.

---

### 10.6 Threading Behavior

- Confirmed that scanning runs in a **separate thread**:
    - The main window remains responsive during scans.
    - User can cancel or interact without freezing the UI.
- **Theory**: Proper multithreading improves user experience and supports **non-blocking operations**, which is vital in desktop applications.

---

### 10.7 ZAP Daemon Dependency Check

- Tested behavior when:
    - ZAP is **not running** – shows connection error message.
    - ZAP is **misconfigured** – appropriate exceptions raised and handled.
- **Theory**: Ensures stability under missing dependency conditions.

---

### 10.8 Alert Filtering Logic

- Verified that only **XSS** and **SQLi-related** alerts are displayed and exported.
- Manual inspection confirmed that other noise (e.g., cookie flags, CSP) was successfully ignored.
- **Theory**: Improves focus and reduces manual triage time, especially in educational and developer settings.

---

### 10.9 Limit Testing (Edge Cases)

- Scanned **unreachable IPs or non-existent hosts** – proper error messages triggered.
- Tested **extremely long URLs or parameters** – no crashes or unexpected behavior.
- Tested **duplicate scans** – verified stable behavior under repeated usage.

# Chapter 11: Code Implementation

This chapter provides the full implementation of the Web Vulnerability Scanner built with Python, Tkinter for GUI, and the OWASP ZAP API for scanning. The code is modularized into functional components for better understanding and maintainability.

## 11.1 Imports and Setup

The application uses various Python libraries including tkinter, threading, csv, and zapv2. It uses the ttkbootstrap library to enhance the GUI appearance.

```
import tkinter as tk
from tkinter import messagebox, scrolledtext
from ttkbootstrap import Style, Entry, Button, Progressbar, Checkbutton
from ttkbootstrap.constants import *
from zapv2 import ZAPv2
import threading
import time
import csv
```

## 11.2 ZAP API Setup

Initializes the ZAP API client with the API key and local proxy URL.

```
# ZAP setup
API_KEY = 'n3m6jha6edpiesf1hbtf7egnhs'
ZAP_PROXY = 'http://127.0.0.1:8080'
zap = ZAPv2(apikey=API_KEY, proxies={'http': ZAP_PROXY, 'https': ZAP_PROXY})
```

## 11.3 Exporting Results

Exports the filtered alerts to both a text file and a CSV file.

```
def export_results(alerts):
    with open("scan_results.txt", "w", encoding='utf-8') as txt_file, \
         open("scan_results.csv", "w", newline='', encoding='utf-8') as csv_file:
        writer = csv.writer(csv_file)
        writer.writerow(["Alert", "Risk", "URL", "Description"])
        for alert in alerts:
            if 'xss' in alert['alert'].lower() or 'sql' in alert['alert'].lower():
                txt_file.write(f"[{alert['alert']}]\nRisk: {alert['risk']}\nURL: {alert['url']}\nDescription: {alert['description']}\n\n")
                writer.writerow([alert['alert'], alert['risk'], alert['url'], alert['description']])
```

### 11.4 Scanning Function

Handles the scanning process using ZAP. It runs in a separate thread to keep the GUI responsive.

```
def scan_url(url, output, progress_bar):
    try:
        zap.urlopen(url)
        time.sleep(2)
        zap.ascan.disable_all_scanners()
        zap.ascan.enable_scanners("40012,40014,40016,40017,40018,40019,40020,40021")

        output.insert(tk.END, "🔍 Starting active scan (XSS + SQL Injection only)...\n")
        scan_id = zap.ascan.scan(url)

        start_time = time.time()
        timeout = 300
        last_progress = -1
```

```
        while True:
            progress = int(zap.ascan.status(scan_id))
            if progress != last_progress:
                output.insert(tk.END, f"⏳ Scan progress: {progress}%\n")
                progress_bar['value'] = progress
                output.see(tk.END)
                last_progress = progress
            if progress >= 100:
                break
            if time.time() - start_time > timeout:
                output.insert(tk.END, "\n⏰ Scan timed out after 5 minutes.\n")
                return
            time.sleep(3)

        output.insert(tk.END, "✅ Scan complete! Fetching alerts...\n")
        alerts = zap.core.alerts(baseurl=url)

        found = False
        for alert in alerts:
            if 'xss' in alert['alert'].lower() or 'sql' in alert['alert'].lower():
                found = True
                output.insert(tk.END, f"🚨 [{alert['alert']}]\nRisk: {alert['risk']}\nURL:
{alert['url']}\nDescription: {alert['description']}\n\n")

        if not found:
            output.insert(tk.END, "✔ No XSS or SQL Injection vulnerabilities found.\n"
        export_results(alerts)
        output.insert(tk.END, "📂 Results saved to scan_results.txt and scan_results.csv\n")
    except Exception as e:
        messagebox.showerror("Scan Error", str(e))
```

## 11.5 Scan Launcher

This function validates the URL and starts the scanning process in a separate thread.

```
def start_scan(entry, output, progress_bar):
    url = entry.get().strip()
    if not url.startswith("http"):
        messagebox.showwarning("Invalid URL", "URL must start with http:// or https://")
        return
    output.delete("1.0", tk.END)
    progress_bar['value'] = 0
    threading.Thread(target=scan_url, args=(url, output, progress_bar), daemon=True).start()
```

## 11.6 Theme Toggle Function

Switches the GUI theme between light and dark modes.

```
current_theme = "flatly"
def toggle_theme(style, toggle_btn):
    global current_theme
    current_theme = "darkly" if current_theme == "flatly" else "flatly"
    style.theme_use(current_theme)
    toggle_btn.config(text="🌙 Dark Mode" if current_theme == "flatly" else "🌞 Light Mode")
```

### 11.7 GUI Initialization Function

Main GUI function that sets up all the components: input field, buttons, output box, progress bar, and theme toggle.

```
def main_gui():
    style = Style(theme=current_theme)
    root = style.master
    root.title("🛡️ Web Vulnerability Scanner")
    root.geometry("800x600")
    root.minsize(600, 500)

    frame = tk.Frame(root, bg=style.colors.bg)
    frame.pack(padx=20, pady=20, fill=tk.BOTH, expand=True)

    tk.Label(frame, text="Enter Website URL:", font=("Segoe UI", 12),
bg=style.colors.bg).pack(anchor=tk.W)

    url_entry = Entry(frame, width=60)
    url_entry.pack(pady=10)

    progress_bar = Progressbar(frame, bootstyle="info-striped", length=500)
    progress_bar.pack(pady=5)

    output_box = scrolledtext.ScrolledText(frame, height=20, font=("Consolas", 10))
    output_box.pack(fill=tk.BOTH, expand=True, pady=10)

    btn_frame = tk.Frame(frame, bg=style.colors.bg)
    btn_frame.pack(pady=10)

    scan_btn = Button(btn_frame, text="🔍 Start Scan", bootstyle=SUCCESS,
                      command=lambda: start_scan(url_entry, output_box, progress_bar))
    scan_btn.pack(side=tk.LEFT, padx=10)
```

```
    exit_btn = Button(btn_frame, text="❌ Exit", bootstyle=DANGER, command=root.quit)
    exit_btn.pack(side=tk.LEFT)
    toggle_btn = Checkbutton(frame, text="🌙 Dark Mode",
                             command=lambda: toggle_theme(style, toggle_btn))
    toggle_btn.pack(anchor=tk.NE)
    root.mainloop()
```

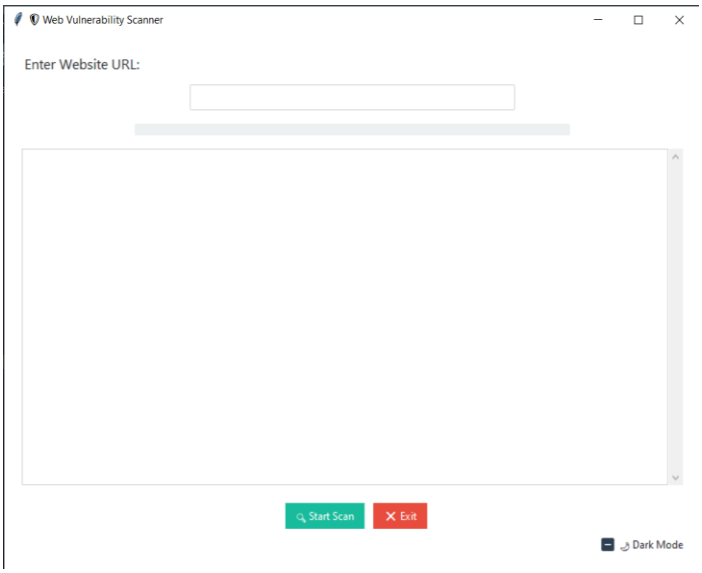
## 11.8 Main Entry Point

The application starts from this conditional block.

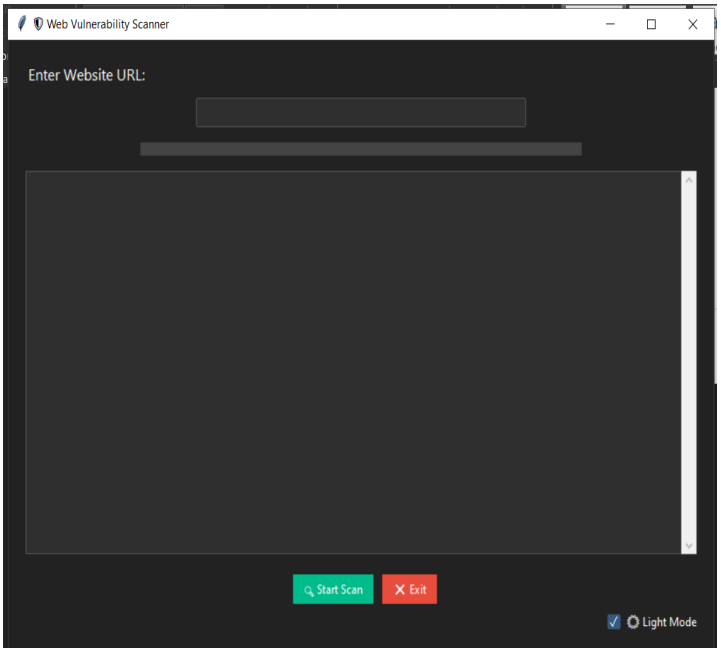
```
if __name__ == "__main__":
    main_gui()
```

# Chapter 12: screenshot

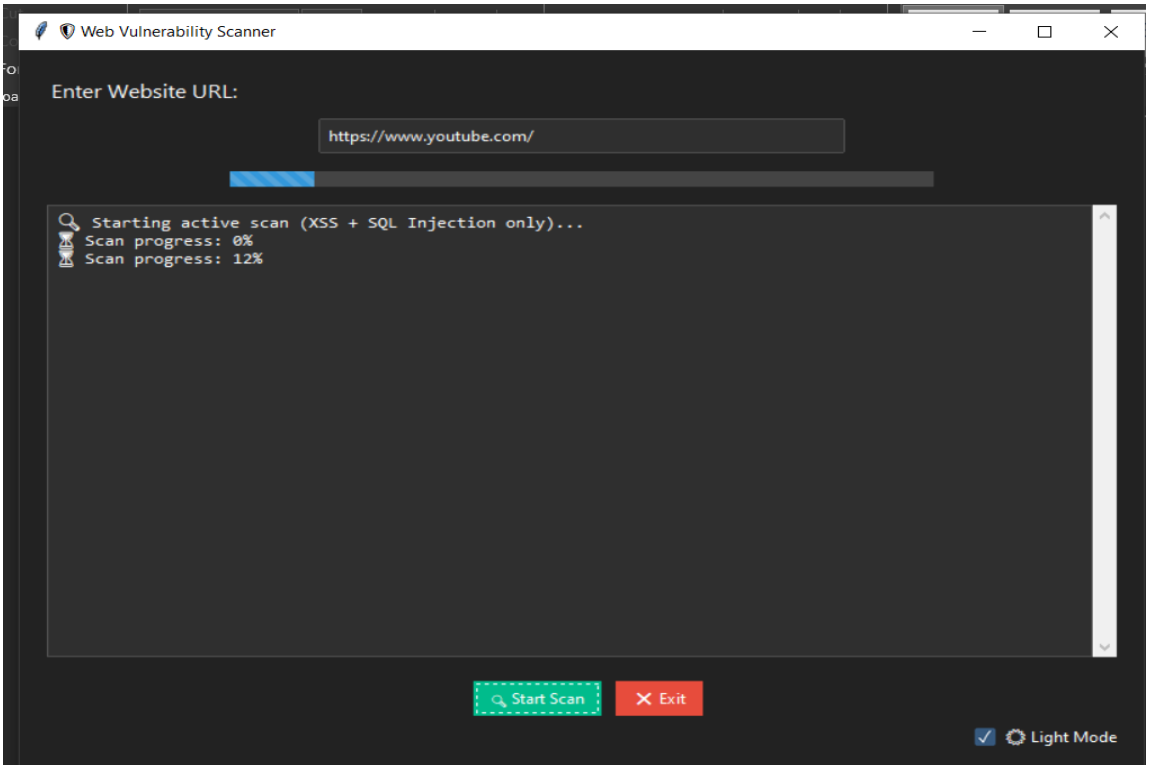
## In light mode:




## In dark mode:



# Scanning URL:



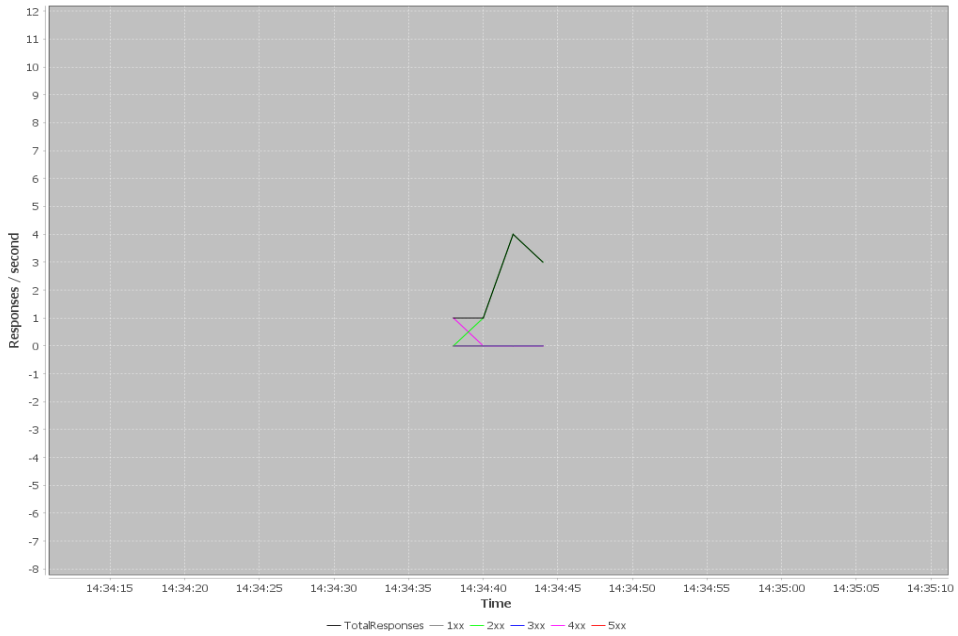

# Scan progress in backend:

Progress | Response Chart

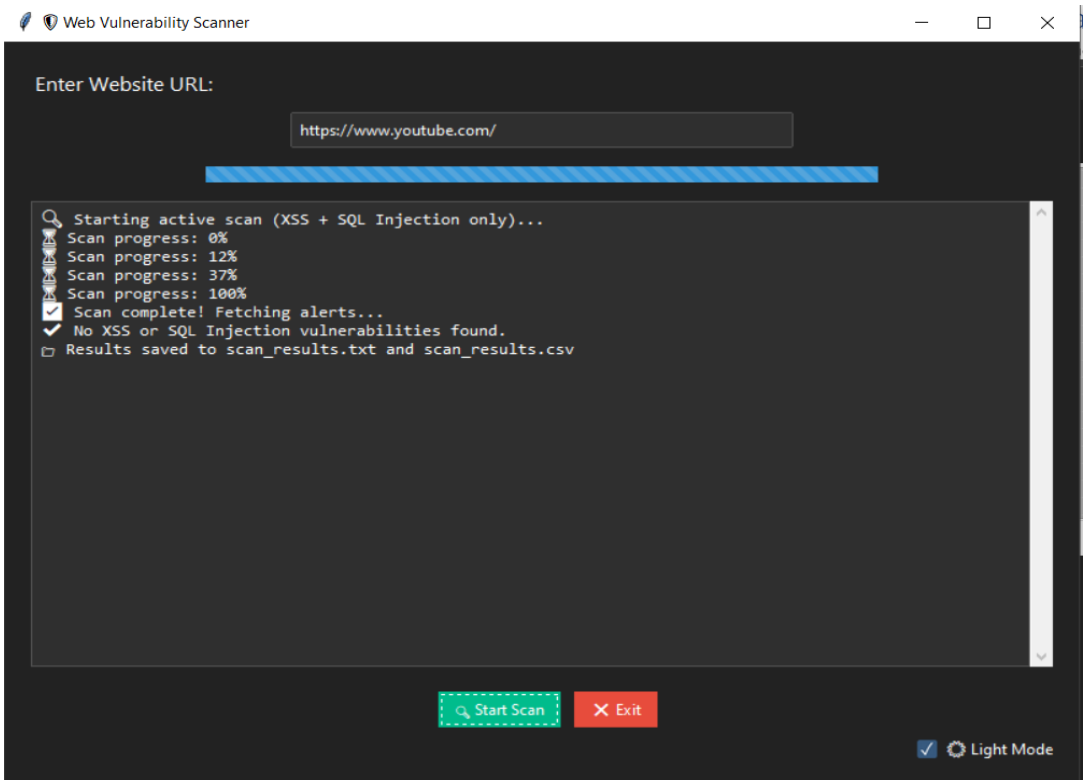
Host: https://www.youtube.com

| | Strength | Progress | Elapsed | Reqs | Alerts | Status |
|---|---|---|---|---|---|---|
| Analyser | | | 00:00.653 | 1 | | |
| Plugin | | | | | | |
| Cross Site Scripting (Reflected) | Medium | | 00:01.703 | 0 | 0 | ✔ |
| Cross Site Scripting (Persistent) - Prime | Medium | | 00:01.625 | 0 | 0 | ✔ |
| Cross Site Scripting (Persistent) - Spider | Medium | | 00:01.237 | 2 | 0 | ✔ |
| Cross Site Scripting (Persistent) | Medium | | 00:00.683 | 0 | 0 | ✔ |
| SQL Injection | Medium | | 00:00.634 | 0 | 0 | ✔ |
| SQL Injection - MySQL | Medium | | 00:00.720 | 0 | 0 | ✔ |
| SQL Injection - Hypersonic SQL | Medium | | 00:00.807 | 0 | 0 | ✔ |
| SQL Injection - Oracle | Medium | | 00:00.646 | 0 | 0 | ✔ |
| Totals | | | 00:09.096 | 11 | 0 | |

# Report chart:




# After scan complete:



# Chapter 13: Flowchart

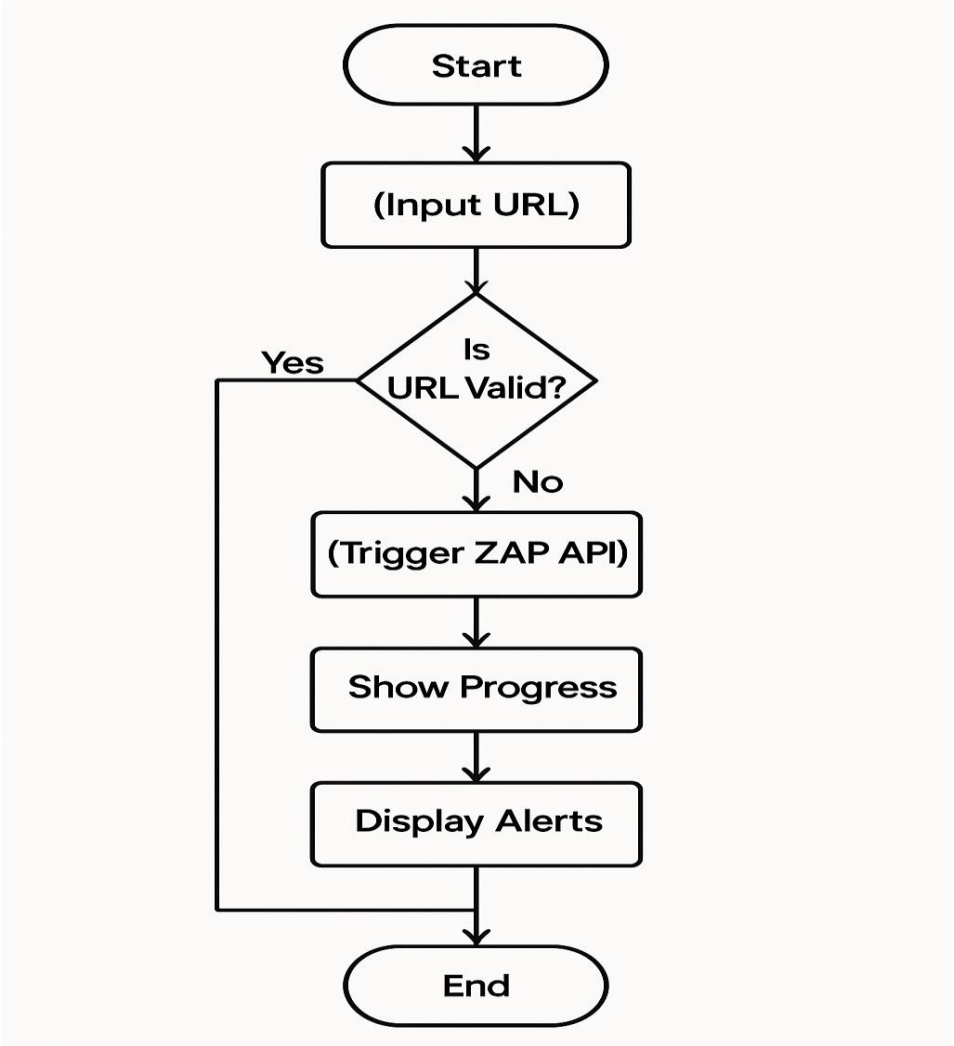
## Flowchart Steps and Shapes

1. **Start**
    - Shape: **Terminator (Oval)**
    - Label: "Start"
2. **Input URL**
    - Shape: **Input/Output (Parallelogram)**
    - Label: "Input URL"
3. **Validate URL**
    - Shape: **Decision (Diamond)**
    - Label: "Is URL Valid?"
4. **Trigger ZAP API**
    - Shape: **Process (Rectangle)**
    - Label: "Trigger ZAP API"
5. **Show Progress**
    - Shape: **Process (Rectangle)**
    - Label: "Show Progress"
6. **Display Alerts**
    - Shape: **Process (Rectangle)**
    - Label: "Display Alerts"
7. **Export**
    - Shape: **Process (Rectangle)**
    - Label: "Export"
8. **End**
    - Shape: **Terminator (Oval)**
    - Label: "End"

---

**Flow Logic**

- Start → Input URL → Validate URL
- If **No** (URL invalid), maybe loop back to Input URL or show error (optional)
- If **Yes**, proceed to Trigger ZAP API → Show Progress → Display Alerts → Export → End



# Chapter 13: Conclusion

This project delivers a user-centric web vulnerability scanner that simplifies the detection of common security threats, specifically Cross-Site Scripting (XSS) and SQL Injection vulnerabilities. By integrating the robust capabilities of the OWASP ZAP API with a clean and intuitive graphical user interface, the tool lowers the barrier to entry for both developers and learners in the cybersecurity domain. While designed to be lightweight and easy to use, this project lays a strong foundation for future enhancements, paving the way for more advanced security solutions and educational resources.

# Chapter 14: References


1. **OWASP ZAP Documentation** – https://www.zaproxy.org
2. **OWASP Top 10 Web Application Security Risks** – https://owasp.org/www-project-top-ten/
3. **Python tkinter Documentation** – https://docs.python.org/3/library/tkinter.html
4. **ttkbootstrap Project** – https://ttkbootstrap.readthedocs.io
5. **SQL Injection** – https://owasp.org/www-community/attacks/SQL_Injection
6. **Cross-Site Scripting (XSS)** – https://owasp.org/www-community/attacks/xss/