🧾 Web Application Vulnerability Scanner - Project Report

📄 Page 1 of 2

🔹 Project Overview	
Project Title	Web Application Vulnerability Scanner
Technology	Python, Flask, BeautifulSoup4, Requests, lxml
Duration	2 Weeks (Internship Project)
Domain	Cybersecurity – Web Application Testing

💡 1. Introduction

In today's interconnected digital landscape, web applications are frequently targeted by malicious actors. Identifying and mitigating vulnerabilities is crucial for maintaining data integrity, user privacy, and system availability. This project aimed to develop a **"Web Application Vulnerability Scanner"** capable of automatically detecting common web vulnerabilities such as Reflected Cross-Site Scripting (XSS), SQL Injection (SQLi), and missing security headers. The scanner is designed as a Python-based tool with a user-friendly Flask web interface, providing a foundational understanding of automated vulnerability assessment.

🧠 2. Abstract

The Web Application Vulnerability Scanner is a Python 3 application leveraging **requests** for HTTP interactions, **BeautifulSoup4** for HTML parsing, and **Flask** for its graphical interface. It crawls a target web app, identifies forms, and injects non-destructive payloads to test for vulnerabilities. Detected issues (XSS, SQLi, missing headers) with severity and evidence are stored in a structured JSON report. Emphasis is placed on **ethical scanning practices** — polite delays and clear disclaimers ensure responsible educational use.

⚙️ 3. Tools Used

Tool / Library	Purpose
🐍 Python 3	Primary programming language
🌐 Flask	Web framework for the scanner's UI (app.py)
📡 requests	Handles HTTP requests for scanning
🧩 BeautifulSoup4	HTML parser to extract forms and links
⚡ lxml	High-performance XML/HTML parsing
🔎 re	Regular expressions for payload pattern matching

🧾 json	Report generation and data structuring
🗂️ os	File and directory operations

🧩 4. Steps Involved in Building the Project

- 🔧 **Environment Setup:** Created a virtual environment and installed Flask, requests, bs4, and lxml.
- 🧱 **Core Logic (core.py):**
  - 🛰️ **Request Handling:** Implemented GET/POST with exception handling and request throttling.
  - 🔗 **URL Crawling:** Recursively discovered same-domain links up to depth limits.
  - 📝 **Form Extraction:** Parsed HTML to locate form actions, methods, and fields.
- 🧪 **Testing Modules:**
  - ⚔️ **XSS Detection:** Injected safe `<script>` and `<img>` payloads; detected reflection in responses.
  - 💉 **SQL Injection:** Used harmless payloads like `' OR '1'='1` to detect SQL error messages or anomalies.
  - 🧰 **Header Analysis:** Checked for missing headers such as Content-Security-Policy and X-Frame-Options.
  - 🗃️ **Reporting:** Saved results as JSON with URL, payload, severity, and evidence snippet.

📄 Page 2 of 2 (Continued)

🌍 Web Interface Development (app.py, templates/)

- 🏗️ **Flask Application:** Served as the front-end to accept target URLs and display results.
- 🧾 **Input Form:** index.html allowed users to input a website URL for scanning.
- ⚙️ **Scan Execution:** /start-scan route invoked the core scanner and saved a JSON report.
- 📊 **Dynamic Reporting:** report.html presented findings in tables showing type, severity, payload, and evidence.
- ⚠️ **Ethical Disclaimer:** Displayed clear warning to scan only authorized systems.
- 🧪 **Example Target:** Built a simple vulnerable Flask app (search/login forms) for demo testing.

🏁 5. Conclusion

The **Web Application Vulnerability Scanner** successfully automates the detection of common web flaws such as XSS, SQLi, and missing security headers. By integrating crawling, form parsing, and safe payload injection within a Flask UI, the project demonstrates the fundamentals of web vulnerability assessment. It reinforces key cybersecurity principles — automation, ethical testing, and secure coding awareness. Future enhancements may include deeper scan logic (DOM-based XSS, blind SQLi), authentication support, and visual analytics dashboards.

✅ **Submitted By:** [FATEHALI ABBASALI MAKNOJIYA] 🎓 **Internship Project Duration:** 2 Weeks
📅 **Year:** 2025

Mentor: [ELEVATE LABS]

📄

🔐 Password Strength Analyzer with Custom Wordlist Generator — Final Project Report (Page 1)

1️⃣ Introduction

The Password Strength Analyzer with Custom Wordlist Generator is a comprehensive security tool designed to address modern password vulnerability challenges. It combines educational strength assessment with practical penetration-testing capabilities to serve both individual users and security professionals.

2️⃣ Abstract

This project delivers a robust Python application integrating password strength analysis (zxcvbn) with intelligent, customizable wordlist generation. The GUI (Tkinter) provides real-time feedback while the generation engine produces targeted dictionaries using personal data, leetspeak, and pattern variations.

🧩 Key Areas & Feature Summary

Area	Feature Summary
🔒 Password Analysis	Score (0-4), crack time estimates, pattern detection, suggestions
🧠 Wordlist Generation	Personal data integration, leetspeak, year suffixes, case/special char variations
📤 Export	Hashcat/John-compatible .txt, size management, progress tracking
💻 Compatibility	Windows, Linux, macOS

🛠️ 3. Tools Used

Tool	Purpose
Python 3.8+	Core programming language
Tkinter	GUI development
zxcvbn-python	Password strength estimation
NLTK	Word variations and NLP
argparse	CLI support
re, itertools, datetime	Pattern generation and utilities

🧭 4. Steps Involved in Building the Project

1.	Phase 1: Core Architecture Design — modular MVC structure and configuration system.
2.	Phase 2: Password Analysis Module — integrated zxcvbn for scoring and feedback.

3.	Phase 3: Wordlist Generation Engine — personal info parsing, leetspeak, year appending (1970–2024).
4.	Phase 4: GUI Development — tabbed interface, real-time analysis, export management.
5.	Phase 5: Advanced Features — color-coded strength meter, pattern detection, bulk export.
6.	Phase 6: Testing & Validation — unit tests, performance and compatibility checks.

📄

🔐 Password Strength Analyzer — Implementation, Examples & Deliverables (Page 2)

6️⃣ Example: Password Analysis Function (Python)

```
def analyze_password_strength(password):
    result = zxcvbn(password)
    return {
        'score': result['score'],
        'feedback': result['feedback'],
        'crack_time': result['crack_times_display'],
        'patterns': result['sequence']
    }
```

7️⃣ Advanced Features

- ⚡ Real-time strength meter with visual cues (implemented in GUI logic)
- 🔍 Pattern-based vulnerability detection (keyboard walks, repeated sequences)
- ⚙️ Customizable generation parameters and bulk export
- 🔗 Compatibility with common cracking tools and formats (Hashcat, John)

8️⃣ Conclusion

The project meets objectives by combining defensive education and offensive testing capability. The modular design supports future enhancements. Deliverables include a functional GUI application, CLI support, and exportable wordlists.

📦 Deliverables Achieved

Item	Description
✅ GUI application	Real-time analysis and wordlist generation
✅ CLI tool	Scriptable generation and export
✅ Export Formats	.txt compatible with Hashcat and John the Ripper

✅ Documentation	User guide and testing report

📌 **Project Status:**	Completed Successfully
📅 **Prepared on:**	October 25, 2025
📞 **Contact:**	fatehali_ (for follow-ups)
📝 **Submitted By:**	Fatehali Abbasali Maknojiya
🎓 **Internship Project Duration:**	2 Weeks
📅 **Year:**	2025
🧑‍🏫 **Mentor:**	Elevate Labs

📄

🔐 End-to-End Encrypted Chat — Architecture & Flow (Page 1)

📘 Overview

This document describes a minimal demonstration of true end-to-end encryption (E2EE) for a chat application built with Flask-SocketIO (server) and browser-based clients. The server acts solely as a relay and does not learn message contents or session keys.

🚦 High-level Message Flow

When Client A sends a message to Client B, the following steps occur:

1️⃣ **AES Session Key Generation (Client A)**: Client A generates a random 256-bit AES key (unique per message).

2️⃣ **Message Encryption (Client A)**: The plaintext is encrypted with the AES key producing ciphertext.

3️⃣ **AES Key Encryption (Client A)**: Client A encrypts the AES key using Client B's RSA public key.

4️⃣ **Transmission to Server (Client A)**: Client A sends two opaque blobs to the server: the RSA-encrypted AES key and the AES-encrypted message ciphertext.

5️⃣ **Message Relay (Server)**: The server relays both blobs to Client B without attempting decryption.

6️⃣ **AES Key Decryption (Client B)**: Client B decrypts the RSA-encrypted AES key with its RSA private key.

7️⃣ **Message Decryption (Client B)**: Client B decrypts the AES ciphertext with the recovered AES key and obtains the plaintext.

🔒 Security Guarantees

This design ensures the server never has access to plaintext or symmetric keys, providing true end-to-end confidentiality assuming clients protect their private keys.

⚠️ Assumptions & Limitations

• Clients correctly generate and protect RSA private keys.
• Public keys are exchanged via the server at registration but must be verified (out-of-band or fingerprint checks) to prevent active MITM attacks.
• This demo uses RSA for key-wrapping; production systems should consider authenticated encryption, MACs, and forward secrecy (e.g., ephemeral Diffie–Hellman) to strengthen security.

📄

🛠️ Implementation Details & Usage (Page 2)

💻 Technology Stack

Backend: Python 3 + Flask + Flask-SocketIO.
Frontend: HTML5 + JavaScript (client-side RSA & AES).
Cryptography: Python cryptography and browser libraries such as JSEncrypt (RSA) and CryptoJS or Web Crypto API (AES).

📁 File Layout (deliverables)

- server.py — Flask server + SocketIO event handlers (register, get_public_keys, send_message, relay).
- static/js/chat.js — Client-side key gen, encryption/decryption, UI logic.
- templates/index.html — Minimal page to register username, display users, send encrypted messages.
- utils/crypto.py — Optional server-side helper (conceptual).

⚙️ Getting Started (quick)

1️⃣ Install Python dependencies:

```
pip install Flask Flask-SocketIO python-engineio python-socketio cryptography
```

2️⃣ Place files in project structure and run:

```
python server.py
```

3️⃣ Open *http://127.0.0.1:5000* in multiple browser tabs, register unique usernames, and send encrypted messages.

📝 Minimal Server Pseudocode

```
from flask import Flask, render_template
from flask_socketio import SocketIO

app = Flask(__name__)
socketio = SocketIO(app)
users = {}  # username -> public_key

@socketio.on('register')
def handle_register(data):
    users[data['username']] = data['public_key']
    socketio.emit('public_keys', [{'username': u, 'public_key': users[u]} for u in users])

@socketio.on('send_message')
def handle_send(msg):
    socketio.emit('relay_message', msg, to=msg['to'])

if __name__ == '__main__':
    socketio.run(app, host='127.0.0.1', port=5000)
```

💡 Client-side Notes
• Generate an RSA key pair in the browser (e.g., JSEncrypt) and register the public key with the server.
• For each message generate a random 256-bit AES key, encrypt the plaintext with AES, then encrypt the AES key with the recipient's RSA public key.
• Send both encrypted components to the server which relays to the recipient.
• Recipient decrypts AES key with RSA private key and then decrypts AES ciphertext.

🚀 Recommended Enhancements
• Add message authentication (HMAC or AES-GCM) for integrity and authenticity.
• Add forward secrecy with ephemeral DH (X25519).
• Persist public keys and user registrations in a database and present key fingerprints to users for manual verification.
• For group chats adopt a group key management protocol (e.g., double-ratchet or MLS).

📝 Consultation / Demonstration & Future Enhancements

This demo effectively showcases the core principles of E2EE. Users register, exchange public keys, and send messages that are encrypted client-side and decrypted only by the intended recipient. The server remains oblivious to message content.

Potential Enhancements:
• Persistent User Data: Implement a database (SQLite, PostgreSQL) for storing registrations and public keys.
• Message Authentication Codes (MACs): Ensure message integrity and authenticity.
• Forward Secrecy: Use Diffie-Hellman for ephemeral keys.
• Group Chat: Extend E2EE for group messaging.
• Offline Messaging: Store encrypted messages for offline recipients.
• Improved UI/UX: Add message history, typing indicators, and better styling.
• Auditable Security: Log cryptographic operations for audits.

📝 Submitted By

✅ **Submitted By:**	FATEHALI ABBASALI MAKNOJIYA
🎓 **Internship Project Duration:**	2 Weeks
📅 **Year:**	2025
🧑‍🏫 **Mentor:**	ELEVATE LABS

🛡️ Browser Extension to Block Trackers - Part 1

📄 Abstract

The project aims to enhance user privacy by blocking known tracking scripts that monitor browsing behavior. It intercepts network requests, filters known tracker domains, and provides analytics to the user. A user-controlled whitelist ensures flexibility and transparency.

🔍 Introduction

Online tracking has become a significant privacy concern. Advertisers and data brokers use scripts to collect user data without consent. This project introduces a Chrome/Firefox extension that automatically blocks these scripts, helping users maintain anonymity and control over their data.

🛠️ Tools Used

🗂️ **Category**	🛠️ **Tools / Technologies**
💻 Languages	JavaScript, HTML, CSS
🌐 Browser API	Chrome Extension Manifest v3
📚 Libraries	WebRequest API, Storage API

📝 Steps Involved in Building the Project (Part 1)

🔢 **Step**	📋 **Description**
1️⃣ Define Tracker List	Maintain a list of known tracking domains (based on public tracker lists).
2️⃣ Intercept Requests	Use webRequest.onBeforeRequest to detect and cancel tracker requests.
3️⃣ Count Blocked Scripts	Each blocked request increases a badge counter.

🛡️ Browser Extension to Block Trackers - Part 2

📝 Steps Involved in Building the Project (Part 2)

🔢 **Step**	📋 **Description**
4️⃣ Popup Dashboard	Show total blocked requests and allow user whitelisting.
5️⃣ Storage Management	Save user whitelist/blacklist using chrome.storage.local.
6️⃣ Testing	Load unpacked extension in Chrome and verify blocking functionality across sites.

✅ Conclusion

The Tracker Blocker extension effectively improves user privacy by preventing data leakage through third-party trackers. With customizable whitelisting, it balances control and convenience, offering users a transparent, privacy-focused browsing experience.

🎯 Key Features

✨ **Feature**	📝 **Description**
🔒 Privacy Protection	Blocks trackers to prevent unauthorized data collection.
⚙️ User Control	Whitelist specific sites to allow trackers if desired.
📊 Analytics	Shows the number of blocked requests and tracking attempts.

🌟 Bonus Tips

- 📌 Users can customize their whitelist to allow trusted sites.
- 📌 Badge icon shows real-time blocked tracker counts.
- 📌 Compatible with both Chrome and Firefox browsers.

📝 Submitted By

✅ **Submitted By:**	FATEHALI ABBASALI MAKNOJIYA
🎓 **Internship Project Duration:**	2 Weeks

📅 **Year:**	2025
🧑‍🏫 **Mentor:**	ELEVATE LABS