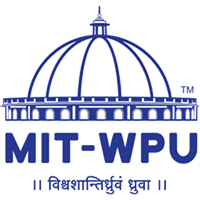
****

**Project Report**

on

Machine Learning powered Facial Recognition

based Attendance System

Submitted by

**Project Members**

Krishnaraj Thadesar 1032210888

Sourab Karad 1032211150

Saubhagya Singh 1032211573

Parth Zarekar 1032210846

**Under the Guidance of**

**Dr. Sharmishta Desai**

**School of Computer Engineering and Technology**

**MIT World Peace University, Kothrud,**

**Pune 411 038, Maharashtra - India**

**2024-2025**

### 

**DEPARTMENT OF COMPUTER ENGINEERING AND TECHNOLOGY**

**C E R T I F I C A T E**

This is to certify that,

Krishnaraj Thadesar 1032210888

Sourab Karad 1032211150

Saubhagya Singh 1032211573

Parth Zarekar 1032210846

of BTech.( Computer Science & Engineering) have completed their project titled Machine Learning powered Facial Recognition based Attendance System and have submitted this Capstone Project Report towards fulfillment of the requirement for the Degree-Bachelor of Computer Science & Engineering Cyber Security & Forensics (BTech-CSE CSF) for the academic year 2024-2025

**Dr. Sharmishta Desai Dr. Balaji M Patil**

Project Guide Program Coordinator

School of CET School of CET

MIT World Peace University, Pune MIT World Peace University, Pune

Internal Examiner:

External Examiner:

**Date: 06 May 2025**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Introduction | 11 |
| 2 |  | Literature Survey | 13 |
| 3 |  | Problem Statement | 17 |
|  | 3.1 | Project Objectives | 17 |
|  | 3.2 | Project Scope | 18 |
|  | 3.2.1 | Target Users | 18 |
|  | 3.2.2 | Vulnerability Coverage | 18 |
|  | 3.2.3 | Functionality | 19 |
|  | 3.3 | Project Assumptions | 20 |
|  | 3.3.1 | User-Related Assumptions | 20 |
|  | 3.3.2 | Technical Assumptions | 21 |
|  | 3.3.3 | Environmental Assumptions | 21 |
|  | 3.4 | Project Limitations | 21 |
|  | 3.4.1 | Functional Limitations | 21 |
|  | 3.4.2 | Technical Limitations | 22 |
|  | 3.4.3 | Environmental & User-Related Limitations | 22 |
|  | 3.4.4 | Scalability & Performance Limitations | 22 |
| 4 |  | Project Requirements | 23 |
|  | 4.1.1 | Human Resources | 23 |
|  | 4.1.2 | Reusable Software | 23 |
|  | 4.1.3 | Software & Hardware | 23 |
|  | 4.1.4 | Functional Specifications | 26 |
|  | 4.2.1 | Interfaces | 26 |
|  | 4.2.2 | External Interfaces | 27 |
|  | 4.2.3 | Internal Interfaces | 27 |
|  | 4.2.4 | Communication Interfaces | 28 |
|  | 4.2.5 | Graphical User Interfaces | 28 |
|  | 4.3 | Interactions | 29 |
|  | 4.3.1 | End User Interactions | 29 |
|  | 4.3.2 | Interface Interactions | 29 |
|  | 4.4 | Sustainability | 29 |
|  | 4.5 | Quality Management | 30 |
|  | 4.6 | Security | 30 |
| 5 |  | System Analysis and Design of the Project | 31 |
|  | 5.1 | Assumptions | 32 |
|  | 5.2 | Dependencies | 33 |
|  | 5.2.1 | Libraries for Backend | 33 |
|  | 5.3 | General Constraints | 33 |
|  | 5.4 | Diagrams |  |
|  | 5.5 | Modules of the Project | 35 |
|  | 5.5.1 | Module 1 | 41 |
|  | 5.5.2 | Module 2 | 41 |
|  | 5.5.3 | Module 3 | 42 |
|  | 5.5.4 | Module 4 | 43 |
| 6 |  | Project Plan | 44 |
| 7 |  | Implementation |  |
|  | 7.1 | Methodology |  |
|  | 7.2 | Algorithm |  |
|  | 7.3 | Workflows implementation |  |
| 8 |  | Performance Evaluation and Testing |  |
| 9 |  | Deployment Strategies |  |
|  | 9.1 | Chrome Extension Deployment |  |
|  | 9.2 | Backend Deployment |  |
|  | 9.3 | Security Aspects |  |
|  | 9.3.1 | Passive Scanning Only |  |
|  | 9.3.2 | Minimum Required Permissions |  |
|  | 9.3.3 | Secure Communication |  |
|  | 9.3.4 | Local Processing |  |
| 10 |  | Result and Analysis |  |
|  | 10.1 | Detailed Findings |  |
|  | 10.2 | Summary of Results |  |
|  | 10.3 | Scan Overview |  |
|  | 10.4 | Detailed Findings |  |
|  | 10.5 | Summary of Results |  |
| 11 |  | Conclusion |  |
| 12 |  | Future prospects of the project |  |
| 13 |  | References |  |
| 14 |  | Publication Details |  |
|  | 14.1 | Conference Details |  |
|  | 14.2 | Plagiarism Report |  |
| 15 |  | Individual Contribution |  |
|  | 15.1 | Name of the Student: Manali Desai |  |
|  | 15.2 | Name of the Student: Nishad Wanjari |  |
|  | 15.3 | Name of the Student: Tanishq Bapna |  |
|  | 15.4 | Name of the Student: Aditya |  |

## List of Figures

|  |  |  |
| --- | --- | --- |
| 5.1 | Block Diagram | 35 |
| 5.2 | System Architecture Diagram | 36 |
| 5.3 | Use Case Diagram | 37 |
| 5.4 | Sequence Diagram | 38 |
| 5.5 | Component Diagram | 39 |
| 5.6 | Activity Diagram |  |
| 5.7 | Class Diagram | 40 |
| 7.1 | Workflow for generating pdf |  |
| 7.2 | Rest API Trigger Node |  |
| 7.3 | Text Generation Node |  |
| 7.4 | HTML to PDF Node |  |
| 7.5 | Return State Node |  |
| 14.1 | Turn-it-in Report Submission |  |
| 14.2 | Turn-it-in Similarity Report |  |

## 

## List of Tables

|  |  |  |
| --- | --- | --- |
| I | Comparison of different papers for Literature Review |  |
| II | Requirements & Rationale |  |
| III | Risk Management |  |
| IV | SDLC of the proposed project |  |
| V | Front-end Test Cases |  |
| VI | Vulnerability Scanning Engine |  |
| VII | Project Outcome Mapping |  |

## 

### 

### 1. Introduction

In today’s increasingly digital landscape, cyber threats have become a pervasive challenge, particularly as the use of web applications surges across both consumer and enterprise sectors. The rapid adoption of these platforms, driven by the demand for seamless and accessible online services, has unfortunately paralleled a significant increase in the sophistication and frequency of cyber-attacks. Malicious actors exploit vulnerabilities in web applications to perpetrate a wide range of attacks, from data breaches to complete system takeovers, resulting in financial losses, compromised sensitive information, and long-term damage to organizational reputations. The damage inflicted by such attacks can be extensive. Beyond immediate monetary losses, successful breaches undermine customer trust and may necessitate costly remediation efforts, including legal fees, regulatory fines, and the implementation of comprehensive security measures. Notably, common web application vulnerabilities—such as SQL injection, cross-site scripting (XSS), cross-site request forgery (CSRF), and insecure deserialization—are frequently exploited due to their prevalence and the often-insufficient safeguards implemented during the development phase. Addressing these vulnerabilities is paramount, especially for small enterprises that may not have the financial resources to invest millions in third-party security.

This research paper proposes an innovative approach to vulnerability detection through a Google Chrome extension, which aims to empower small businesses by providing a cost-effective, accessible, and immediate tool for scanning web applications. By integrating vulnerability scanning into a widely used web browser, this method allows organizations to proactively identify and mitigate potential security flaws, laying the foundation for robust application security practices without the heavy financial burden traditionally associated with advanced cybersecurity measures. Through detailed analysis and evaluation, this study explores the technical framework of the proposed extension, its effectiveness in detecting common vulnerabilities, and its potential to serve as an initial step towards more comprehensive cybersecurity strategies in resource-constrained environments.

Web applications are critical to modern digital infrastructure, yet they remain prime targets for cyberattacks due to vulnerabilities like cross-site scripting (XSS) and SQLi (SQL Injection). The Open Web Application Security Project (OWASP) reports that such flaws consistently rank among the top threats to web security. While the Linux OS stands as one stop shop for all security related tools, they require technical expertise and standalone execution, leaving casual users and small developers underserved. This research paper addresses the gap by developing a Google Chrome extension that passively scans websites for vulnerabilities during browsing and reports them for . By integrating real-time detection into a familiar browser environment, this tool aims to enhance accessibility and awareness of web security, offering a practical contribution to both education and practice.

## 

### 2. Literature Survey

The authors Akshay Mathur et al.[1] have focused on developing a browser extension to detect Cross-Site Scripting (XSS) vulnerabilities in HTML5-based web applications. XSS has remained one of the most persistent security threats, ranking in the OWASP Top 10 for over a decade. The study presented a methodology where the extension continuously monitors web pages for suspicious activities by analyzing specific HTML5 tags and attributes. With HTML5 introducing new tags and attributes, it was identified that attackers have new vectors to exploit vulnerabilities. A primary task in researching and developing a chromium extension was to identify attack vectors that would serve as an entry point to hacking into the users’ sessions. A repository of different attack vectors was maintained based on a XSS cheat sheet. The three significant attack vectors that were experimented on were <TAGS>,<SVG> and <ATTRS> .This paper proved to be relevant in ideating an extension that includes a much more comprehensive set of attacks rather than simply adhering to finding XSs vulnerabilities.

The authors Avinash Kumar and Sangita Roy [2] in their research for ‘A Network Based Vulnerability Scanner for Detecting SQLI Attacks in Web Applications’ have explored approaches towards developing a vulnerability scanner that particularly deals with the detection of SQLi inside web applications. They have reviewed multiple pre-existing tools that serve the same functionality yet each tool lacked completeness and was time-consuming. It was found that these scanners had high overheads since they introspected every SQL query that was raised through HTTP requests. Another significant flaw was the generation of false positives where the SQL attack had failed but a distinct response was obtained than the legitimate one. The network based SQLi vulnerability scanner was tested on a total of five separate web applications where some of the applications were designed to simulate the operations of commonly known websites and the other tests were done on three real websites. The result of a comparative analysis of the other tools against the authors’ scanner indicated that the NVs was the most lightweight scanner, taking the least time to generate a report. It also detected the highest number of vulnerabilities tied with Acunetix (85%) however Acunetix simultaneously had the highest number of false positives. That summed up the research on why newer scanners that provide a more comprehensive search ability, completeness and efficiency need to be developed.

The authors Jason Bau et al.[3] have explored the effectiveness and limitations of black-box web application vulnerability scanners. Through this study they utilized scanners to simulate external attacks to uncover vulnerabilities such as Cross-Site Scripting and Cross-Site Request Forgery through remote code execution. The research aimed to assess the state of the art in automated black-box testing by evaluating eight leading commercial scanners against both real-world applications and a custom-designed testbed. The experiments were performed in two phases where initially scanner performance was tested on popular websites. The second phase consisted of scanner testing on vulnerabilities that are in accordance with the NIST web Application Scanner Functional Specification. The execution time ranged from 66 to 473 minutes, and network traffic generated ranged from 80MB to nearly 1GB, showing significant differences in how each tool approached vulnerability detection. Active content such as Flash, Java applets, and AJAX-based interactions were poorly understood by most scanners, leading to incomplete security assessments. The research indicated that while all XSS vulnerabilities were flagged correctly with almost minimal false positives ,the results for first-order SQL vulnerability classification reflected poorly. Similarly stored vulnerabilities and consequent malware was not identified.

The authors Haibo Chen[4] et al. were motivated to automate the vulnerability scanning process and expand the scope of scanning by not restricting it to a single target but instead providing a more comprehensive overview. The research yet again focused on the common web application vulnerabilities such as XSS and SQL injection. To battle these vulnerabilities the functionality of the proposed scanner was divided into three subcategories, each equivalently important. The first subcategory identified was related database and management module where all the assets and their corresponding information will be stored and managed, which included details pertinent to target websites as well as target ports. The subsequent category was an information collection module that focused on exercising a high-speed DNS stub resolver named Massdns to boost the capabilities and perform an elaborate enumeration. The third function was vulnerability detection which was mapped against OWASP Zap to get a comparative analysis of effectiveness. The OWASP ZAP scanner revealed almost half the total number of vulnerabilities detected by the automated vulnerability scanner however the authors have made no mention of any false

positives that may have been encountered. The research paper mentions a successful SQL injection attack that led to the exposure of a database and its tables.

Kandula et al. [5] (2024) conducted a study on malicious URL analysis, employing machine learning algorithms to enhance cybersecurity measures. The research centered on evaluating the performance of three classifiers: Random Forest, LightGBM, and XGBoost, in detecting malicious URLs. Their findings revealed that the XGBoost classifier achieved the highest accuracy, scoring 94.33%. In comparison, Random Forest and LightGBM achieved accuracies of 85% and 87.98% respectively. This outcome emphasizes XGBoost's potential as a potent tool for predicting and identifying malicious URLs, thereby contributing to more effective cybersecurity strategies against ever-evolving cyber threats. The study underscores the importance of leveraging machine learning to create adaptive and robust defenses in the digital landscape.

Fonseca et al. [6] (2007) presented a method for evaluating and comparing web application vulnerability scanners using software fault injection techniques. By injecting common software faults into web applications, they assessed the scanners' ability to detect resulting vulnerabilities, specifically SQL injection and Cross-Site Scripting (XSS). The evaluation of three commercial scanners revealed considerable differences in their performance, with a notable percentage of vulnerabilities remaining undetected and a high false positive rate, ranging from 20% to 77%. The study highlights the importance of using multiple scanners and manual code reviews for critical web applications, while also suggesting the potential to improve scanner quality by identifying and addressing their limitations through the proposed benchmarking approach.

TABLE I. COMPARISON OF DIFFERENT RESEARCH PAPERS FOR LITERATURE SURVEY

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Sr. No. | Citation and Author | Title | Methodology | Pros | Cons |
| 1. | P. Sivanesan, A. Mathur, A. Y. Javaid, IEEE EIT 2018 | A Google Chromium Browser Extension for Detecting XSS Attack in HTML5 Based Websites | Developed a Google Chrome browser extension that scans HTML5-based web applications for Cross-Site Scripting (XSS) vulnerabilities by monitoring suspicious HTML tags and attributes. | - Real-time detection of XSS attack vectors  - Browser extension offers lightweight security without complex setup | - Limited to XSS vulnerabilities, does not detect SQL Injection, CSRF, or other security issues.  - Only applies to HTML5-specific attack vectors. |
| 2. | A. K. Singh, S. Roy, IEEE RAIT 2012 | A Network Based Vulnerability Scanner for Detecting SQLI Attacks in Web Applications | Developed a network-based SQL Injection (SQLi) scanner that automatically crawls web applications, injects attack payloads, and analyzes responses to detect SQLi vulnerabilities | - Reduces false positives compared to other scanners.  - Database-specific attack rule library improves accuracy | - Scanning time is high compared to modern tools.  - Does not address other web vulnerabilities, only SQLi. |
| 3. | J. Bau, E. Bursztein, D. Gupta, J. Mitchell, IEEE Symposium on Security & Privacy 2010 | State of the Art: Automated Black-Box Web Application Vulnerability Testing | Evaluated eight commercial black-box web vulnerability scanners using real-world web applications and a controlled testbed with known vulnerabilities. Measured detection rate, false positives, and scanning efficiency. | - Highlights strengths and weaknesses of commercial scanners.  - Identifies gaps in stored XSS and SQLi detection. | - Many black-box scanners fail to detect stored vulnerabilities (e.g., stored XSS and second-order SQLi).  - Modern technologies like AJAX, Flash, and Java Applets are poorly handled. |
| 4. | H. Chen, J. Chen, J. Chen, S. Yin, Y. Wu, J. Xu, IEEE TrustCom 2020 | An Automatic Vulnerability Scanner for Web Applications | Developed an automated scanner integrating information collection and vulnerability detection to improve web security assessments. Tested on real-world applications. | - Comprehensive scanning approach combining data collection with attack simulation. - Detects multiple OWASP Top 10 vulnerabilities. | - No mention of false positive rates, making effectiveness hard to assess.  - Performance vs. OWASP ZAP scanner not fully compared. |
| 5. | A. R. Kandula, K. P. Kumar, A. B. Prakash, K. S. Kumar, IEEE ICECA 2024 | Malicious URL Analysis using Machine Learning Algorithm | Used machine learning (ML) with XGBoost classifier to detect malicious URLs, achieving 94.33% accuracy on a dataset of 651,191 URLs. Applied lexical feature engineering and honeypots to monitor attacker behavior. | - High classification accuracy (94.33%) using ML.  - Integrates honeypots to study attacker behavior. | - Lexical data conversion may lose information, limiting attack pattern detection.  - Honeypot observations may not generalize well to real-world threats. |
| 6. | J. Fonseca, M. Vieira, H. Madeira, PRDC 2007 | Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks | Used software fault injection to evaluate three commercial web vulnerability scanners, assessing their ability to detect SQL Injection (SQLi) and XSS vulnerabilities. Measured detection coverage and false positive rates. | - First study to systematically  benchmark commercial scanners.  - Identifies detection gaps and false positive rates. | - Manual inspection of vulnerabilities is time consuming.  -Commercial scanner names are anonymized, limiting direct tool comparison |

### 3. Problem Statement

As the digital world increasingly relies on web applications for daily operations and services, cyber threats targeting these platforms have become more frequent and sophisticated. Small businesses, independent developers, an

d general users often lack the financial resources and technical expertise to employ advanced security tools, leaving their applications and data vulnerable to attacks such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF). Existing professional security solutions are often complex, costly, or require standalone environments like Linux, which can be inaccessible to non-technical users. There is a critical need for an accessible, cost-effective, and easy-to-use tool that enables real-time vulnerability awareness directly within the browsing environment. This project addresses the gap by proposing a Google Chrome extension that passively scans websites for common web vulnerabilities, helping users proactively detect and understand security risks without needing deep cybersecurity knowledge or dedicated infrastructure.

### 3.1 Project Objectives:-

1. **Develop a User-Friendly Chrome Extension for Web Vulnerability Scanning**Create a lightweight Chrome extension named "WebSafe Sentinel" that integrates seamlessly into the browser toolbar, allowing users to initiate scans with a single click.
2. **Enable Manual Scanning of Websites for Common Vulnerabilities**Implement manual scanning functionality using Chrome’s webRequest and tabs APIs to detect vulnerabilities like XSS, outdated libraries, and misconfigured headers on user-specified websites.
3. **Provide Feedback and Downloadable Reports on Scanned Websites**Design a popup interface that displays real-time scan results with severity levels and offers a downloadable JSON report summarizing detected vulnerabilities for user reference.
4. **Bridge the Gap Between Technical Security Tools and Non-Technical Users**Simplify complex security concepts into intuitive alerts and visual indicators, ensuring non-technical users can understand and act on vulnerability findings without prior expertise.
5. **Promote Security Awareness Among Non-Technical Audiences**Include recommendations and links within the report that explain each vulnerability in plain language, encouraging non-technical users to adopt safer browsing and development practices.
6. **Evaluate the Effectiveness of the Chrome Extension Through Manual Testing**Conduct manual testing on websites, including vulnerable testbeds to assess detection accuracy, false positive rates, and overall usability for non-technical users.

### *3.2* Project Scope

#### 3.2.1 Target Users

1. **Small Business Owners**: Individuals managing websites with limited cybersecurity resources, seeking basic security insights to protect their online presence.
2. **Students**: Learners studying web development or cybersecurity, using the tool to understand vulnerabilities and practice safe coding.
3. **Web Developers**: Professionals looking for a quick, accessible way to check their sites for common issues during development.
4. **General Internet Users with Minimal Cybersecurity Expertise**: Non-technical users who want to assess the security of websites they visit, without needing advanced knowledge.

#### 

#### 3.2.2 Vulnerability Coverage

1. **Cross-Site Scripting (XSS)**: Targets reflected XSS vulnerabilities where untrusted input is echoed back in the webpage, potentially allowing malicious script execution.
2. **SQL Injection (SQLi)**: Focuses on SQL injection risks where improperly sanitized user inputs could manipulate database queries, leading to unauthorized data access.
3. **Insecure HTTP Headers**: Addresses vulnerabilities from missing or weak security headers like Content-Security-Policy, which can expose sites to attacks like XSS or clickjacking.
4. **Unsanitized Input Fields**: Identifies risks in input fields that reflect unsanitized data, increasing susceptibility to XSS by allowing harmful code injection.
5. **SSL Certificate Issues**: Highlights vulnerabilities from invalid or expired SSL certificates, which can undermine secure communication and expose data to interception.

#### 

#### 3.2.3 Functionality

1. **Manual Scanning of Visited Websites**: Users can input a URL in the popup to initiate a scan, with results displayed in real-time (currently active).
2. **User-Friendly Interface with Reports and Tips**: Features a styled popup with a results panel, including timestamps, issue counts, and recommendations (e.g., "Sanitize user input") for security awareness.
3. **Lightweight and Resource-Efficient**: Designed with a React frontend (via Vite) and Flask backend to minimize resource usage, though optimization is ongoing.
4. **Passive Scanning of Visited Websites**: Planned feature to automatically scan websites for vulnerabilities as users browse, using content scripts to analyze page elements.
5. **Desktop Versions of Google Chrome**: Supports Windows, macOS, and Linux, leveraging Chrome’s cross-platform availability. Mobile Chrome support is not included due to API limitations.

### 

### 3.3 Project Assumptions

#### 3.3.1 User-Related Assumptions

1. **Basic Knowledge of Browser Extensions**: Users can install the extension from chrome://extensions/ and interact with the popup interface.
2. **Non-Offensive Use**: The tool is used solely for educational or personal security purposes, not for exploiting websites.
3. **Testing Environments**: Scans are performed on user-owned sites or authorized test environments (e.g., OWASP Juice Shop,Acunetix), respecting legal and ethical boundaries.

#### 3.3.2 Technical Assumptions

1. **Chrome API Support**: The Chrome browser will continue to support necessary WebExtensions APIs (e.g., activeTab, storage) and permissions for manual scanning.
2. **Passive Detection Feasibility**: Future passive scanning will be possible with Chrome’s content script and messaging APIs, assuming no significant API deprecations.
3. **User Consent**: Users will grant permissions (e.g. "activeTab") to allow the extension to access site data for scanning purposes.
4. **Security Restrictions**: Chrome’s security policies, such as Content Security Policy (CSP) on websites, will not overly restrict the extension’s ability to inject content scripts or access DOM elements for scanning.
5. **Cross-Origin Requests**: The extension can make cross-origin requests to external APIs (e.g., Worqhat API) without being blocked, assuming proper CORS configurations are in place on the server side.

#### 

#### 3.3.3 Environmental Assumptions

1. **Local Development Environment**: A Flask server running on http://localhost:5000 is accessible during development and testing.
2. **Network Availability**: Users have a stable internet connection to access the Flask backend and external APIs (e.g. Worqhat API).
3. **Compatible Dependencies**: Node.js, npm, Python, and required libraries (e.g., Flask, flask-cors) are installed and functional on the user’s system.
4. **Firewall and Proxy Settings**: Users’ network environments will not block communication between the extension, the Flask backend, or external APIs due to firewalls, proxies, or other network restrictions.

### 3.4 Project Limitations

#### 3.4.1 Functional Limitations

#### Manual Scanning Only: The extension relies on user-initiated scans via the popup, excluding active penetration testing or exploitation to verify vulnerabilities, adhering to ethical guidelines and avoiding legal risks.

1. **Narrow Vulnerability Scope**: Targets common issues like Cross-Site Scripting (XSS), SQL Injection (SQLi), insecure HTTP headers, unsanitized inputs, and SSL certificate problems, but excludes complex vulnerabilities such as logic flaws or race conditions due to project scope and expertise limitations.
2. **Variable Detection Accuracy**: May generate false positives (e.g., flagging secure sites) or false negatives (e.g., missing obfuscated vulnerabilities) and these vulnerabilities must be further investigated through a thorough security audit.

#### 3.4.2 Technical Limitations

1. **Educational Tool, Not a Professional Audit**: Functions as a preliminary scanning tool for non-technical users, lacking the comprehensive analysis and depth of professional solutions like Burp Suite or OWASP ZAP.
2. **Challenges with Dynamic Content**: Real-time detection may struggle with heavily dynamic or obfuscated client-side code, reducing effectiveness against certain vulnerabilities.
3. **Dependency on Local Backend**: Requires a separate Flask server (e.g., http://localhost:5000) for scanning, which may not be practical for all users without additional setup.

#### 3.4.3 Environmental and User-Related Limitations

1. **Local Server Requirement**: Users must set up and maintain the Flask backend, posing a barrier for non-technical users without guidance.
2. **Internet Dependency**: External API calls (e.g., Worqhat) require internet access, and failures may disrupt report generation.
3. **User Error**: Incorrect URL inputs or lack of consent to permissions may prevent scanning, relying on user cooperation.

#### 3.4.4 Scalability and Performance Limitations

1. **Resource Constraints**: Designed to be lightweight, but extensive scanning across multiple sites could strain system resources without further optimization.
2. **Limited Scalability**: Tailored for individual or small-scale use, making it unsuitable for large-scale enterprise security assessments.

### 4. PROJECT REQUIREMENTS

### 4.1 Resources

##### 4.1.1 Human Resources

1. **Project Team**: A small team of 4 students with skills in JavaScript, React-vite, Python, and basic cybersecurity concepts to design, develop, and test the extension.
2. **Advisor/Supervisor**: An academic or industry mentor with expertise in web security to provide guidance on vulnerability detection and ethical considerations.
3. **Tester**: At least one individual to manually test the extension on diverse websites, ensuring usability and accuracy for non-technical users.

##### 4.1.2 Reusable Software Components

1. **Data Preprocessing Module**: A reusable JavaScript function to sanitize and normalize webpage data (e.g., removing noise, parsing DOM elements) applied across XSS detection, input field checks, and HTTP header analysis to streamline vulnerability scanning.

##### 4.1.3 Software & Hardware Requirements

##### (i) Software

1. **Google Chrome**: Version 90 or later for WebExtensions API support (e.g., activeTab, webRequest).
2. **Node.js and npm**: For managing Chrome extension development dependencies.
3. **Python 3.x**: For running the Flask backend server.
4. **Flask and flask-cors**: Libraries for backend API development and cross-origin resource sharing.
5. **Visual Studio Code or Similar IDE**: For coding and debugging the extension and backend.

##### (ii) Hardware

1. **Development Machine**: A computer with at least 4GB RAM, 2GHz CPU, and 10GB free storage for running Chrome, the Flask server, and development tools.
2. **Testing Devices**: Multiple devices (e.g., laptops, desktops) with Chrome installed to simulate diverse user environments.

##### TABLE II: REQUIREMENTS AND RATIONALE

|  |  |  |
| --- | --- | --- |
| **Sr. no.** | **Requirement** | **Rationale** |
| 1. | Chrome Extension API | Needed to develop and integrate features with the browser |
| 2. | Passive scanning for vulnerabilities | To avoid legal/ethical issues with active exploitation |
| 3. | Real-time alert system | To notify users immediately when a vulnerability is detected |
| 4. | Minimal CPU and memory usage | Ensure smooth browsing experience |
| 5. | GUI for displaying scan results | For user-friendly interaction and readability |
| 6. | No scan history logging for privacy purposes | Avoid storing user browsing data to allow them to maintain privacy |
| 7. | Support for basic vulnerabilities like XSS, SQLi,Directory Brute-forcing and header misconfigurations | These are most common and need to be addressed first |
| 8. | No third-party dependency for scanning logic | Maintain control over scanning logic and increase reliability |
| 9. | Regular updates and patching mechanism | Keep extension secure and relevant as web threats evolve |
| 10.. | Using Worqhat’s API for report generation by utilising HTML to PDF functionality | Allows users to save and share results, supporting education and documentation needs. |

##### TABLE 4.2: RISK MANAGEMENT

##### 

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Sr.no | Risk Factor | Description | Likelihood | Impact | Mitigation Strategy | Priority |
| 1. | API Deprecation | Chrome updates may deprecate WebExtensions APIs. | Medium | High | Monitor Chrome release notes and adapt code. | High |
| 2. | Flask Backend Failure | Local server crashes or misconfiguration. | Medium | Medium | Provide setup guide and test server stability. | Medium |
| 3. | False Positives/Negatives | Inaccurate vulnerability detection. | High | Medium | Conduct thorough testing on diverse sites. | High |
| 4. | User Permission Denial | Users block activeTab or other permissions. | Medium | Low | Include clear permission prompts in UI. | Medium |
| 5. | Internet Outage | Disrupts external API or report generation. | Low | Medium | Add offline mode for basic scans. | Medium |
| 6. | Resource Overload | Extensive scanning slows browser performance. | Medium | Low | Optimize scan logic and limit concurrent scans. | Low |
| **7.** | Outdated Vulnerability Rules | New vulnerabilities not detected over time. | Medium | Medium | Plan for periodic updates to run\_all\_scans. | Medium |

### 

### 

### 4.2 Functional Specifications

##### 4.2.1 Interfaces (Functions/Methods Exposed by the Component)

1. **Scan Initiation Interface:**
   1. Method: run\_all\_scans(url)
   2. Action: Executes a manual scan of the specified URL to detect vulnerabilities such as XSS, SQLi (placeholder), insecure headers (pending), unsanitized inputs, and SSL certificate issues.
   3. Interaction: Triggered when the user clicks the "Scan" button in the popup UI, sending the URL to the Flask backend for processing.
2. **Report Generation Interface:**
   1. Method: downloadReport()
   2. Action: Converts scan results from run\_all\_scans into a downloadable JSON file containing detected vulnerabilities and their details.
   3. Interaction: Called when the user clicks the "Download Report" button in the popup after a scan completes.

##### 4.2.2 External Interfaces Required

1. **Flask Backend API:**
   1. Endpoint: http://localhost:5000/scan
   2. Purpose: Receives the URL from run\_all\_scans, processes it for vulnerability detection, and returns results to the extension.
   3. Data Format: JSON (e.g., { "url": "example.com", "vulnerabilities": { "xss": true, "headers": "missing CSP" } }).
2. **WorqHat API:**
   1. Endpoint: External API for generating downloadable reports using workflows.
   2. Purpose: Converts the generated JSON report into a PDF format which is easily readable and no extra use of Puppeteer library is used.
   3. Data Format: PDF.

##### 4.2.3 Internal Interfaces Required

1. Content Script Interface:
   1. Method: extractPageData()
   2. Purpose: Extracts webpage data (e.g., DOM elements, HTTP headers) to support run\_all\_scans in detecting vulnerabilities like XSS and unsanitized inputs.
   3. Interaction: Communicates with the background script using Chrome’s messaging API to pass extracted data.
2. **Background Script Interface:**
   1. Method: handleScanResults(results)
   2. Purpose: Processes the output of run\_all\_scans from the Flask backend and prepares it for display in the popup UI.
   3. Interaction: Uses Chrome’s runtime.sendMessage to send results to the popup.

##### 4.2.4 Communication Interfaces

1. **Chrome Messaging API:**
   1. Enables communication between content scripts, background scripts, and the popup UI using chrome.runtime.sendMessage and chrome.runtime.onMessage.
   2. Example: Passes run\_all\_scans results from the background script to the popup for real-time feedback.
2. **HTTP Requests:**
   1. Facilitates communication with the Flask backend (http://localhost:5000) and WorqHat API using fetch.
   2. Example: Sends a POST request with the URL to the Flask endpoint for run\_all\_scans processing.

##### 4.2.5 Graphical User Interfaces (GUIs)

1. **Popup Interface:**
   1. Features a "Scan" button to trigger run\_all\_scans, a text field for URL input (or uses the current tab’s URL), and displays real-time results (e.g., "XSS Detected: High Severity").
   2. Includes a "Download Report" button to invoke downloadReport() and educational tooltips (e.g., "XSS: Allows attackers to inject malicious scripts").
2. **Alert Notifications:**
   1. Displays in-browser alerts in the popup for critical vulnerabilities (e.g., "Insecure Headers Detected") based on run\_all\_scans output.

#### 

#### 4.3 Interactions

##### 4.3.1 End-User Interaction:

* 1. The user installs "WebSafe Sentinel," pins it to the Chrome toolbar, and clicks the icon to open the popup.
  2. In the popup, the user enters a URL or uses the current tab’s URL, clicks "Scan" to trigger run\_all\_scans, and views results with severity indicators (e.g., red for high-risk issues).
  3. The user can download a JSON report via the "Download Report" button or hover over vulnerabilities for educational tooltips.

##### 4.3.2 Interface Interactions:

1. The popup UI triggers run\_all\_scans(url) when the user clicks "Scan," which sends the URL to the Flask backend via a POST request.
2. The content script (get\_all\_forms) extracts web page data and sends it to the background script for processing.
3. The background script uses handleScanResults to process the Flask response and sends the formatted results to the popup via runtime.sendMessage.
4. Clicking "Download Report" calls downloadReport() to generate a JSON file from the scan results.

#### 4.4 Sustainability

1. Code Maintenance: The run\_all\_scans logic is modular, allowing updates to vulnerability detection rules (e.g., adding SQLi detection) without major refactoring.
2. Scalability Considerations: The Flask backend can be deployed to a cloud server in the future to eliminate local setup, supporting broader adoption.
3. Energy Efficiency: Designed to minimize resource usage during scans, ensuring long-term usability on typical devices.

#### 4.5 Quality Management

1. Testing: Manual testing on 7 websites to validate run\_all\_scans accuracy, focusing on XSS, unsanitized inputs, and planned features like insecure headers.
2. Usability Standards: The popup UI prioritizes simplicity (e.g., one-click scanning, clear feedback) for non-technical users.
3. Performance Metrics:Scans the entire page under a minute’s time and generates report in downloadable format.

#### 4.6 Security

1. Data Privacy: Does not store user data; run\_all\_scans results are processed locally and sent securely to the Flask backend.
2. Secure Communication: Uses HTTPS for WorqHat API calls and enforces CORS on the Flask backend to prevent unauthorized access.
3. Ethical Constraints: Relies on passive scanning via run\_all\_scans, avoiding active exploitation to ensure ethical use.

### 5. SYSTEM ANALYSIS AND DESIGN OF THE PROJECT

##### 5.1 Assumptions

1. **Stable Chrome API Support**: Chrome will continue to support WebExtensions APIs (e.g., activeTab, webRequest, runtime.sendMessage) required for run\_all\_scans, content script injection, and communication between the popup, background, and content scripts, with no significant deprecations during the project timeline.
2. **User Permissions Granted**: Users will grant necessary permissions (e.g., activeTab) to enable the extension to access webpage data for scanning, ensuring run\_all\_scans can analyze DOM elements and HTTP headers.
3. **Feasibility of Future Features**: Planned features like passive scanning will be achievable using Chrome’s content scripts and messaging APIs, assuming no major changes to browser security policies like Content Security Policy (CSP) restrictions.
4. **Browser Compatibility**: The extension will function on Chrome version 90 or later, assuming consistent behavior of WebExtensions APIs across these versions.
5. **User Environment**: End users will have devices with sufficient resources (e.g., 4GB RAM, modern CPU) to run Chrome and the extension without significant performance issues during scans.

##### 5.2 Dependencies

1. **Chrome WebExtensions Framework**: Relies on Chrome’s WebExtensions APIs for core functionality, including webRequest for HTTP header analysis, tabs for DOM access, and runtime for messaging between components (e.g., sending run\_all\_scans results to the popup).
2. **Flask Backend**: Depends on a local Flask server (http://localhost:5000) to process run\_all\_scans logic and handle external API calls, requiring Python 3.x, Flask, and flask-cors libraries to be installed and operational.
3. **Worqhat API**: Utilizes the Worqhat API for supplementary vulnerability data (e.g., SSL validation), requiring a stable internet connection and proper CORS configuration to avoid cross-origin request issues.
4. **Development Tools**: Relies on Node.js and npm for managing extension dependencies (e.g., JavaScript libraries) and a code editor like Visual Studio Code for development and debugging.
5. **Network Connectivity**: Assumes a stable internet connection for users to access the Flask backend (if hosted remotely in the future) and external APIs, ensuring seamless report generation and scan functionality.

##### 5.2.1 Libraries for Backend:

1. **os**: Facilitates file and directory operations, used for managing local storage or log files generated by the Flask backend during run\_all\_scans processing.
2. **socket**: Enables network communication checks, supporting the Flask server’s ability to validate server responses or detect connectivity issues.
3. **select**: Assists in handling multiple network connections efficiently, potentially used for managing asynchronous requests in the Flask backend.
4. **requests**: Handles HTTP requests to external APIs (e.g., WorqHat API) and the Flask server, enhancing run\_all\_scans with additional vulnerability data.
5. **datetime**: Provides timestamp functionality for logging scan results and generating reports with downloadReport.
6. **OpenSSL**: Supports SSL certificate validation within run\_all\_scans, ensuring secure connections are assessed for vulnerabilities.
7. **re (Regular Expressions)**: Enables pattern matching for detecting vulnerabilities like XSS or unsanitized inputs in webpage content.
8. **certifi**: Provides a set of root certificates for secure HTTPS connections, ensuring reliable communication with external APIs.
9. **hashlib**: Used for generating hash values, potentially for verifying data integrity in scan results or reports.
10. **urllib.parse**: Parses URLs to extract components (e.g., domain, path) for targeted scanning by run\_all\_scans.
11. **BeautifulSoup**: Parses HTML and XML documents from webpages, aiding run\_all\_scans in analyzing DOM elements for vulnerabilities like XSS.
12. **colorama**: Enhances console output with colored text, useful for debugging or displaying scan results during development.
13. **time**: Manages timing operations, such as setting delays between scan requests to avoid overwhelming the Flask backend or target sites.

##### 5.3 General Constraints:

1. **Ethical Boundaries**: The extension is restricted to passive scanning via run\_all\_scans, avoiding active penetration testing or exploitation to comply with ethical guidelines and prevent legal risks, limiting its ability to confirm vulnerabilities.
2. **User Setup Requiremen**t: Relies on users running a local Flask server (http://localhost:5000) for run\_all\_scans processing, posing a barrier for non-technical users who may struggle with setup.
3. **Performance Trade-Offs**: The extension aims to be lightweight (e.g., under 10MB memory usage, scans under 3 seconds), but extensive scanning of dynamic or complex sites may impact browser performance, requiring careful optimization

##### 5.4 Diagrams:

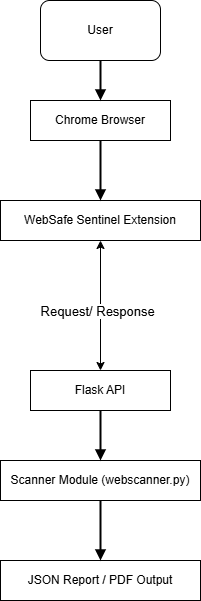


Figure 5.1: Block diagram

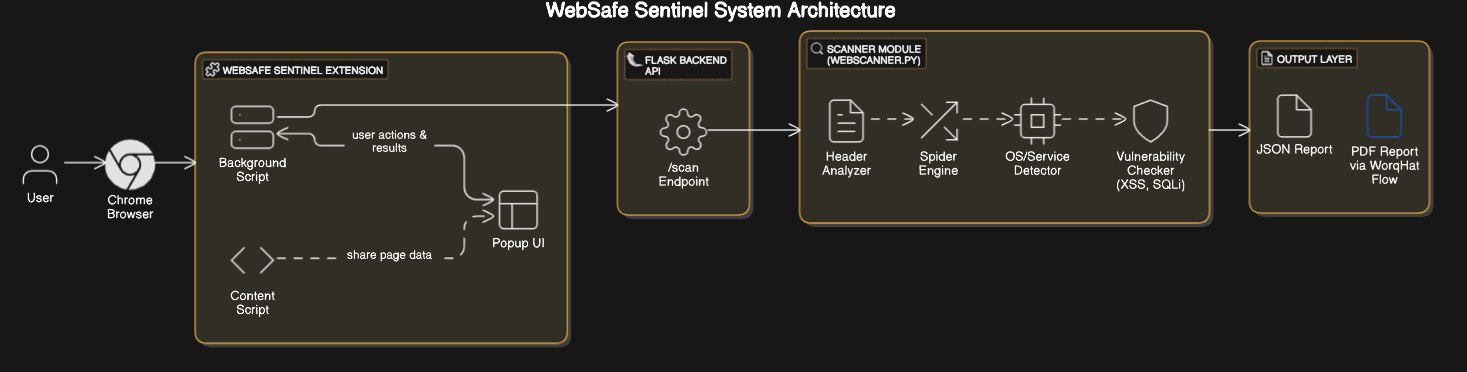


Figure 5.2: System Architecture

##### 

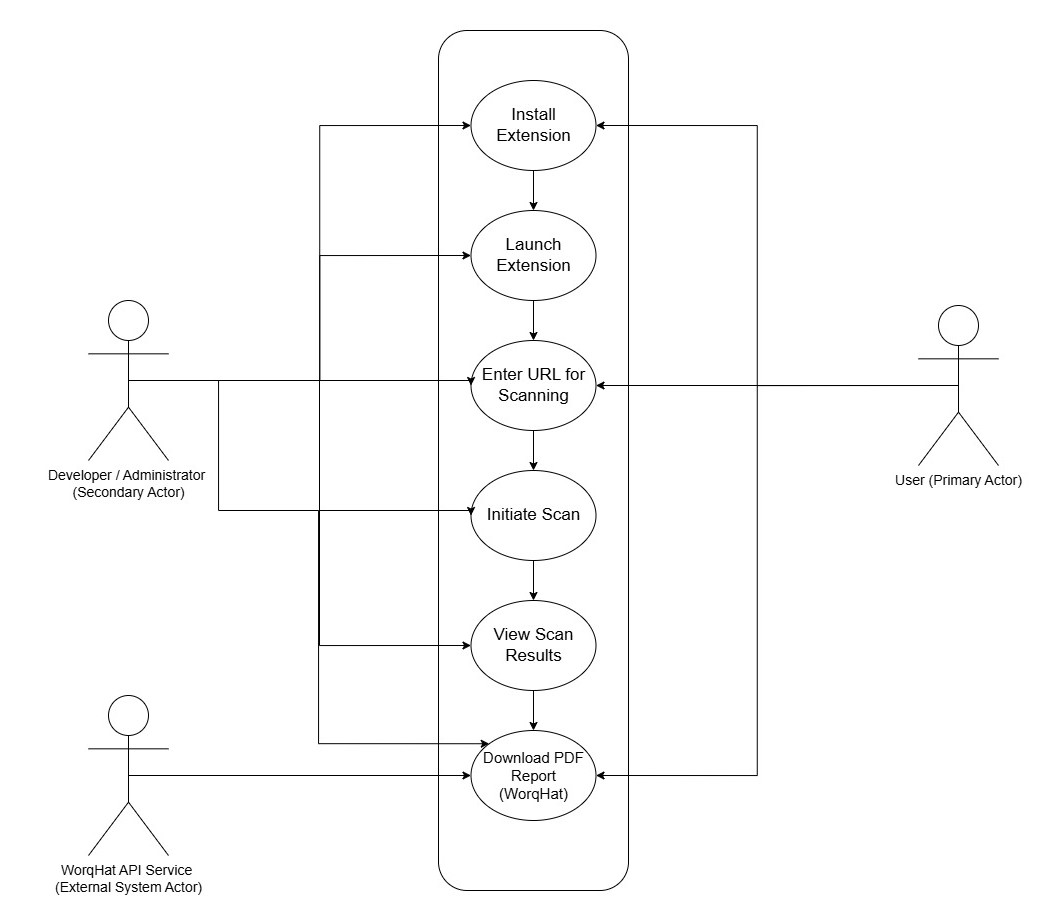


Figure 5.3: Use Case

##### 

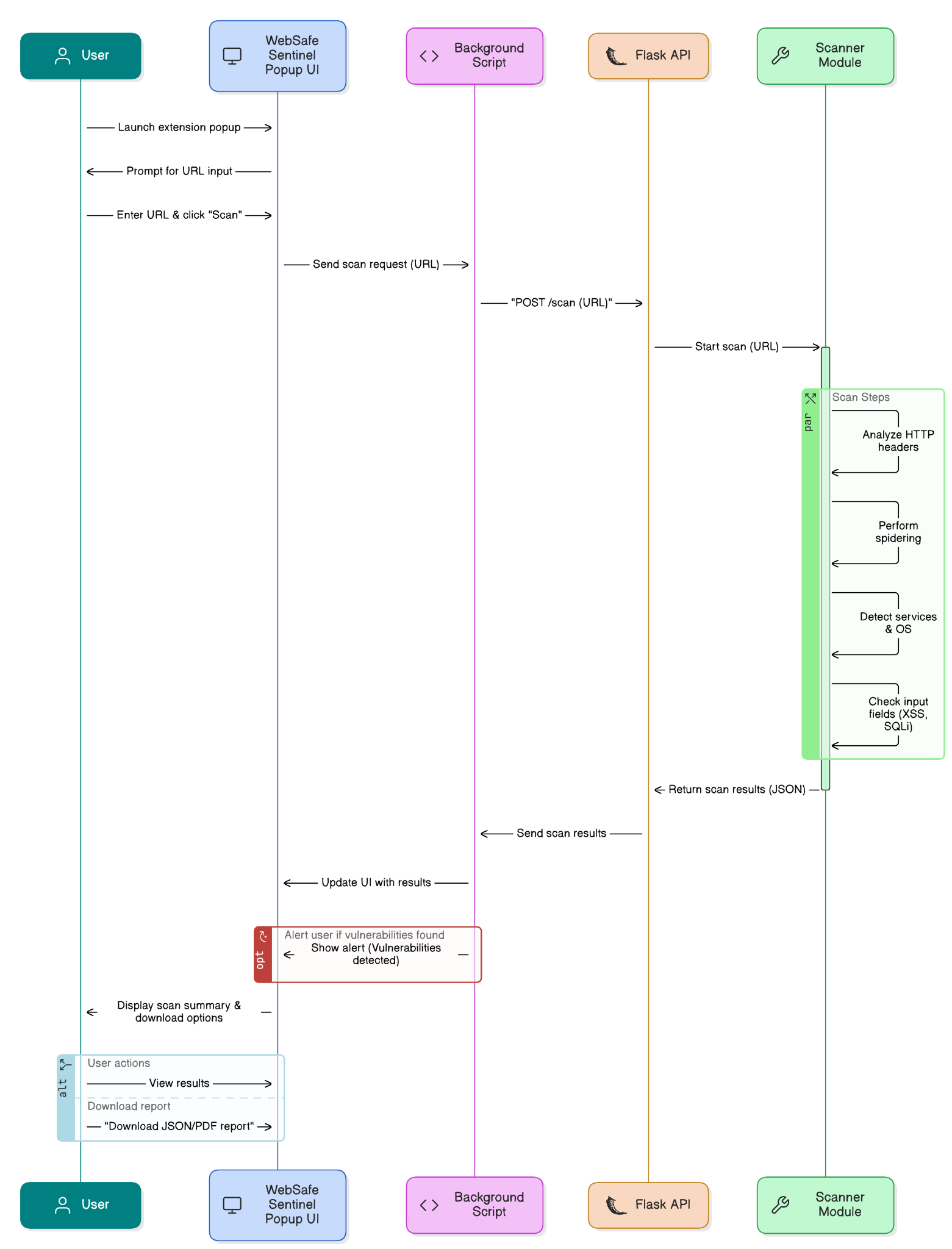


Figure 5.4: Sequence diagram

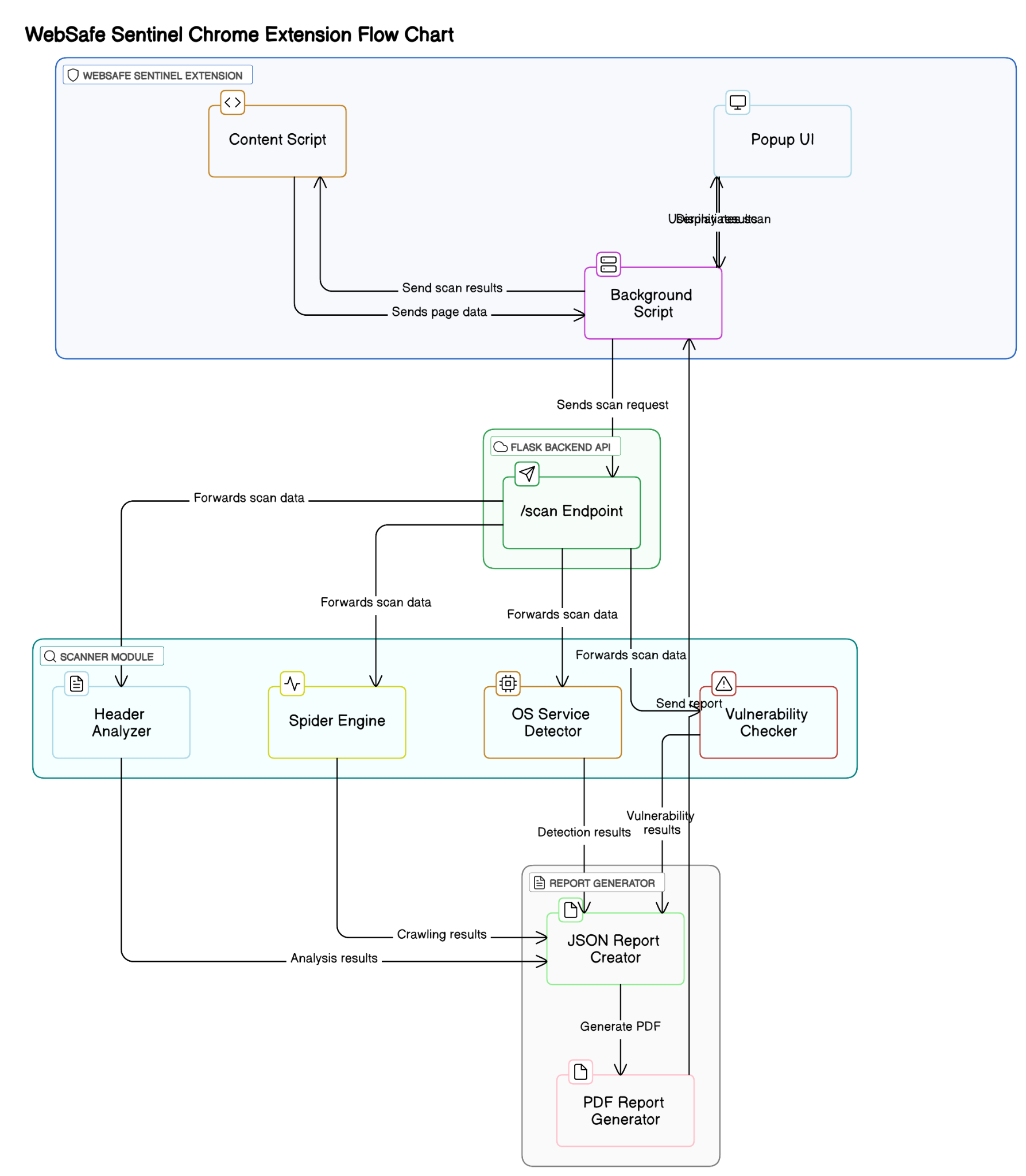


Figure 5.5: Component diagram

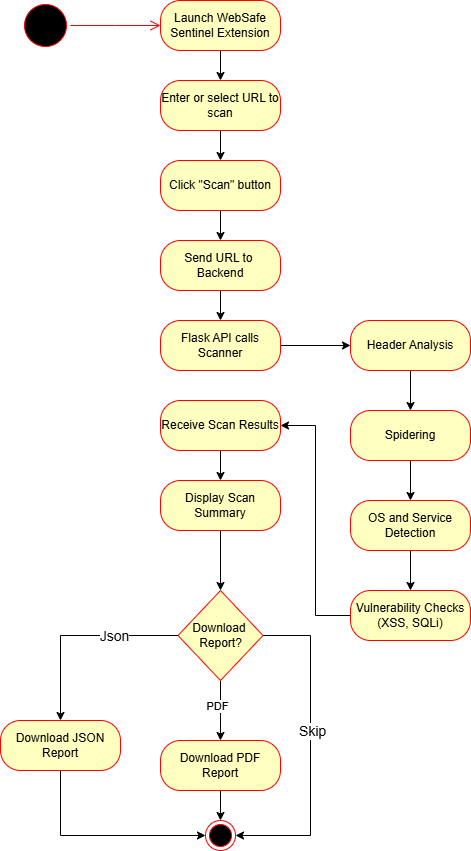


Figure 5.6: Activity diagram

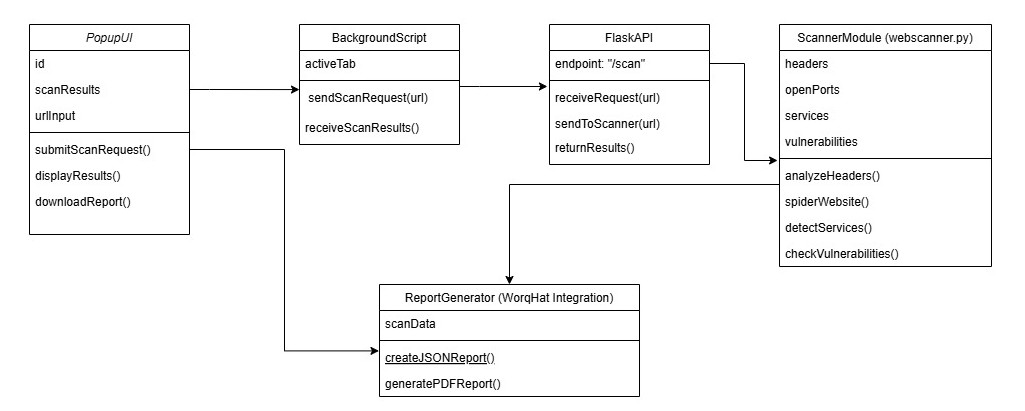


Figure 5.7: Class diagram

### 5.5 Modules of the Project

The "WebSafe Sentinel" Chrome Extension is composed of four distinct modules that work together to deliver a user-friendly web vulnerability scanning experience. These modules encapsulate the core functionalities of vulnerability detection, backend processing, frontend interaction, and external API integration, ensuring modularity and maintainability.

##### 5.5.1 Module 1: Vulnerability Scanner Engine (webscanner.py)

1. **Purpose**: Serves as the core engine for detecting web vulnerabilities, implementing the run\_all\_scans logic to identify issues like XSS, unsanitized inputs, and SSL certificate problems.
2. **Functionality**:
   1. Executes run\_all\_scans(url) to scan a given URL for vulnerabilities, leveraging libraries like BeautifulSoup for DOM parsing, re for pattern matching (e.g., XSS detection), and OpenSSL for SSL validation.
   2. Analyzes webpage content, input fields, and HTTP headers, returning a structured JSON output (e.g., { "xss": true, "headers": "missing CSP" }).
   3. Logs scan timestamps using datetime and supports debugging with colorama for colored console output during development.
3. **Dependencies**:
   1. Python libraries: BeautifulSoup, re, OpenSSL, datetime, colorama, requests, urllib.parse.
   2. Input: URL provided by the Flask backend or frontend.
   3. Output: JSON object containing vulnerability scan results.
4. **Interaction**:
   1. Receives scan requests from the Flask backend via HTTP POST to http://localhost:5000/scan.
   2. Returns scan results to the Flask backend for further processing and forwarding to the frontend.

##### 5.5.2 Module 2: Flask Backend Configuration

1. **Purpose**: Manages server-side processing, routing scan requests, and coordinating with external APIs, acting as a bridge between the frontend and the vulnerability scanner engine.
2. **Functionality**:
   1. Hosts a Flask server at http://localhost:5000 to handle HTTP requests from the Chrome extension.
   2. Defines an endpoint /scan that accepts POST requests with a URL, invokes run\_all\_scans in webscanner.py, and returns the results as JSON.
   3. Uses flask-cors to enable cross-origin requests from the Chrome extension, ensuring seamless communication.
   4. Integrates with the WorqHat API using the requests library to fetch additional vulnerability data (e.g., library version checks).
3. **Dependencies**:
   1. Python libraries: Flask, flask-cors, requests.
   2. Input: URL and scan parameters from the Chrome extension.
   3. Output: JSON response with vulnerability scan results.
4. **Interaction**:
   1. Receives scan requests from the frontend via HTTP POST.
   2. Calls the Vulnerability Scanner Engine (webscanner.py) to perform the scan.
   3. Forwards results to the frontend and coordinates with the Worqhat API module for report enhancement.

##### 5.5.3 Module 3: Frontend in React-Vite

1. **Purpose**: Provides a user-friendly interface within the Chrome extension popup, built using React-Vite for efficient development and rendering.
2. **Functionality**:
   1. Renders a popup UI with a "Scan" button, URL input field, real-time scan results display, and a "Download Report" button.
   2. Displays educational tooltips (e.g., "XSS: Allows attackers to inject malicious scripts") when users hover over detected vulnerabilities.
   3. Sends scan requests to the Flask backend by invoking fetch to POST the URL to http://localhost:5000/scan.
   4. Receives and displays run\_all\_scans results in real-time, with severity indicators (e.g., red for high-risk issues).
3. **Dependencies**:
   1. React: For building a dynamic and component-based UI.
   2. Vite: For fast development and bundling of the React frontend.
   3. Chrome WebExtensions APIs: tabs to access the current URL, runtime.sendMessage to communicate with the background script.
   4. Input: User-provided URL or current tab URL.
   5. Output: Visual display of scan results and downloadable report trigger.
4. **Interaction**:
   1. Communicates with the background script to extract page data (e.g., current URL).
   2. Sends HTTP requests to the Flask backend and receives scan results.
   3. Triggers the Worqhat API module for report downloading when the user clicks "Download Report".

##### 5.5.4 Module 4: API Integration through Worqhat for Report Downloading

1. **Purpose**: Facilitates report generation and download functionality by leveraging the no code workflows functionality.
2. **Functionality**:
   1. Integrates with the Worqhat API to convert the JSON report into a structured PDF format.
   2. Accurately provides the results of the vulnerability scan in a set template.
   3. Triggered by the downloadReport function in the frontend.
   4. Handles the download process by creating a downloadable file link in the browser.
3. **Dependencies**:
   1. Python libraries: requests for API calls.
   2. WorqHat API: For generating a pdf report.
   3. Input: Scan results from the Flask backend.
   4. Output: Enhanced pdf report file for download.
4. **Interaction**:
   1. Called by the Flask backend to present scan results with the pdf.
   2. Triggered by the frontend’s "Download Report" button to generate and serve the final report.
   3. Communicates with the Worqhat API via HTTP requests, ensuring CORS compliance.

### 6. Project Plan

TABLE IV : SDLC OF THE PROPOSED PROJECT

|  |  |  |  |
| --- | --- | --- | --- |
| **Phase No.** | **SDLC Phase** | **Proposed Timeline (Weeks)** | **Description** |
| 1 | Planning & Research | Week 1 | Identify project goals, define scope, and research relevant technologies and tools (Chrome Extension, Flask, Web Vulnerabilities). |
| 2 | Requirement Analysis | Week 2 | Gather detailed functional and non-functional requirements. Decide scanning methods, UI features, and reporting needs. |
| 3 | System Design | Week 3 | Design system architecture, define modules, create data flow diagrams, and plan integration of frontend, backend, and APIs. |
| 4 | Frontend + Backend Development | Week 4 - Week 5 | Develop Chrome Extension UI (React + Vite), create Flask API backend, and implement scanner module (webscanner.py). |
| 5 | Integration & Testing | Week 6 - Week 7 | Integrate frontend with backend, test API communication, test scanner module, and validate end-to-end functionality. |
| 6 | Final Implementation | Week 8 | Complete remaining features, optimize performance, fix bugs, and finalize all functionalities. |
| 7 | Documentation & Report Writing | Week 9 | Prepare blackbook report, create diagrams, write user manual, and document the whole project process. |
| 8 | Submission & Presentation | Week 10 | Final review, prepare presentation, submit report, and demonstrate the project. |

### 7. Implementation

##### 7.1 Methodology:

1. The process begins when the user opens the extension’s popup in Chrome, designed to detect common web vulnerabilities such as Cross-Site Scripting (XSS), SQL Injection (SQLi), and insecure HTTP headers, aligning with the project’s educational and security awareness goals.
2. The popup interface, built using React—a JavaScript library for dynamic UI components—and compiled with Vite for optimized development, resides in the front-vite directory, with core components defined in App.jsx.
3. The user enters a URL (e.g., https://worqhat.com) into a styled input field, enhanced with index.css and App.css, and clicks the "Scan" button, triggering the handleScan function within App.jsx.
4. handleScan validates the URL, ensuring it starts with http:// or https://, sets the loading state using useState hooks, clears prior errors or results for a smooth experience, and sends a POST request to the /scan endpoint using the fetch API, with the URL in a JSON payload.
5. Detailed logging (e.g., console.log for the URL being fetched) is implemented in handleScan to aid debugging, capturing any issues during the request process.
6. The request is sent to the backend, a Flask-based API defined in api.py, running locally on http://localhost:5000 in debug mode, requiring manual startup by the user, with print statements providing visibility into server operations like startup and request handling.
7. Flask, a lightweight Python web framework, processes the request at the /scan endpoint, extracting the URL from the JSON payload and validating it—ensuring it begins with http:// or https:// and removing trailing slashes to standardize input.
8. CORS is enabled using the flask-cors library, allowing cross-origin requests from the Chrome extension, ensuring compatibility with Chrome’s security policies.
9. The Flask backend delegates the vulnerability detection to the scanner module, preparing to return the results to the frontend.
10. The scanner module, a separate Python script (webscanner.py), executes the detection logic via the run\_all\_scans function, starting with a basic XSS check by sending a test request with requests (e.g., ?q=<script>alert(1)</script>) to check for reflected, unsanitized input.
11. A 1-second delay in run\_all\_scans simulates processing time, after which the function compiles a structured JSON object with a summary (e.g., status, issues found), timestamp, and scans object detailing the XSS check’s outcome, including status, result, details, and recommendations (e.g., “Sanitize user input”).
12. The scanner module includes placeholders for SQLi and CSRF detection, with the current focus on XSS, and plans to expand to additional vulnerabilities like insecure HTTP headers in future iterations.
13. Flask receives the JSON output from run\_all\_scans, formats it with jsonify, and sends it back to the frontend as a response to the /scan request.
14. The handleScan function processes the response, updating the state with setResult to store the data and setLoading to false, signaling the scan’s completion, or logs errors (e.g., console.error for HTTP status issues) and displays messages like “Failed to scan site: HTTP error!” in the popup via the error state.
15. Upon a successful scan, the results are rendered in the popup’s results panel, showing the timestamp, issue count, and detailed outcomes (e.g., XSS detection status) with recommendations, promoting security awareness among users.
16. The output delivery offers two reporting options: the handleDownload function generates a downloadable JSON report using the Blob API, formatting the results with indentation and triggering a download with a filename like websafe\_report\_2025-04-24.json.
17. Simultaneously, the triggerFlow function sends the scan results to the Worqhat API at https://api.worqhat.com/flows/trigger/76135102-bc2e-4bd3-a1a8-7e41477dd5aa, authenticating with a bearer token and passing the results as a JSON string, storing the returned PDF report URL in the reportUrl state.
18. Users can download the PDF via a “Download Full Report” button, with an alert notifying them to wait if the report isn’t ready, though this step depends on the Worqhat API’s availability and response format.
19. Development followed an iterative cycle: starting with a basic popup, resolving CSS loading issues by configuring web\_accessible\_resources for hashed filenames, addressing fetch failures with CORS and absolute URLs, and adding JSON downloads and Worqhat integration.
20. Tools used include React, Vite, Flask, Python’s requests, Chrome DevTools for debugging, and curl for API testing, with Git for version control (assumed).
21. Validation involved manually scanning URLs like https://worqhat.com to verify XSS detection, monitoring resource usage in Chrome Task Manager for lightweight operation, and simulating non-technical user interactions to confirm accessibility.
22. Challenges, such as the local server dependency, were mitigated by documenting setup steps, with future plans to deploy the backend to a cloud service for broader usability.
23. This methodology establishes a foundation for future enhancements, including passive scanning via content scripts, expanded vulnerability detection, and real-time alerts, aligning with the project’s goals of accessibility and security education.

##### 7.2 Algorithm:

Here is the high-level algorithm followed for passive scanning:

|  |
| --- |
| BEGIN WebSafeSentinel\_ScanWorkflow   // Step 1: User Interaction (Presentation Layer)  1. User opens Chrome extension popup  2. User enters URL (e.g., "https://worqhat.com") into input field  3. User clicks "Scan" button  4. IF URL is empty OR does not start with "http://" or "https://" THEN  Display error message ("Please enter a valid URL")  RETURN  END IF  5. Set loading state to TRUE  6. Clear previous error and result states   // Step 2: Extension Logic (Send Request)  7. Prepare JSON payload with { "url": enteredURL }  8. Send POST request to "http://localhost:5000/scan" using fetch API  9. Log request details (e.g., "Fetching http://localhost:5000/scan with URL: " + enteredURL)   // Step 3: Backend Processing (Flask API)  10. Flask API (api.py) receives POST request at /scan endpoint  11. Extract URL from JSON payload  12. Validate URL:  IF URL is empty THEN  Return JSON { "error": "URL cannot be empty" } with status 400  EXIT  END IF  IF URL does not start with "http://" or "https://" THEN  Return JSON { "error": "URL must start with http:// or https://" } with status 400  EXIT  END IF  Remove trailing slash from URL if present  13. Enable CORS using flask-cors to allow cross-origin request  14. Call run\_all\_scans(url) from webscanner.py  15. Receive scan results as JSON object  16. Return JSON response to frontend   // Step 4: Scanner Module Processing (webscanner.py)  17. FUNCTION run\_all\_scans(url)  Initialize result object with { "summary": { "status": "Completed", "issues\_found": 0, "total\_scans": 3 }, "timestamp": currentTimestamp, "scans": {} }  Simulate 1-second delay to mimic processing  // XSS Detection  Send test request to url + "?q=<script>alert(1)</script>" using requests  IF response contains "<script>alert(1)</script>" THEN  Update result.scans["xss\_check"] with { "status": "Vulnerable", "result": "Reflected XSS detected", "details": ["Input is reflected without sanitization"], "recommendation": "Sanitize user input" }  Increment result.summary.issues\_found by 1  ELSE  Update result.scans["xss\_check"] with { "status": "Passed", "result": "No XSS detected", "details": [], "recommendation": "N/A" }  END IF  // Placeholder for SQLi and CSRF  Update result.scans["sqli\_check"] with { "status": "Not Implemented", "result": "N/A", "details": [], "recommendation": "Implement SQLi detection" }  RETURN result  18. END FUNCTION   // Step 5: Frontend Processing and Output Delivery  19. Frontend (App.jsx) receives response  20. IF response status is not OK THEN  Extract error text  Log error (e.g., "Server response: " + errorText) using console.error  Set error state to "Failed to scan site: HTTP error! Status: " + response.status + ", Response: " + errorText  Set loading state to FALSE  Display error message in popup  RETURN  END IF  21. Parse JSON response into data  22. Log received data (e.g., "Received data: " + data) using console.log  23. Set result state to data  24. Set loading state to FALSE  25. Render results panel with timestamp, issue count, and scan details (e.g., XSS status)  26. Call triggerFlow(data) to initiate report generation  27. FUNCTION triggerFlow(report)  Set endpoint to "https://api.worqhat.com/flows/trigger/76135102-bc2e-4bd3-a1a8-7e41477dd5aa"  Set API key to "THE API KEY IS KEPT SECRET"  Log "Triggering flow with report: " + report  Send POST request to endpoint with headers { "Authorization": "Bearer " + apiKey, "Content-Type": "application/json" } and body { "report": JSON.stringify(report) }  IF response status is not OK THEN  Log error (e.g., "Flow error: HTTP error! Status: " + response.status) using console.error  RETURN  END IF  Parse response JSON into data  Log "Flow response: " + data  IF data.data?.fileurl exists THEN  Set reportUrl state to data.data.fileurl  END IF  28. END FUNCTION  29. IF user clicks "Download JSON Report" button THEN  Create Blob from JSON.stringify(result, null, 2) with type "application/json"  Create downloadable link with URL from Blob and filename "websafe\_report\_" + currentDate + ".json"  Trigger link click to download file  30. IF user clicks "Download Full Report" button THEN  IF reportUrl is not empty THEN  Create downloadable link with href set to reportUrl  Trigger link click to download file  ELSE  Display alert "Report not ready yet. Please wait..."  END IF   // Step 6: Validation and Iterative Refinement  31. Manually test with URLs (e.g., "https://worqhat.com") to verify XSS detection and UI responsiveness  END |

##### 7.3 Workflows implementation:

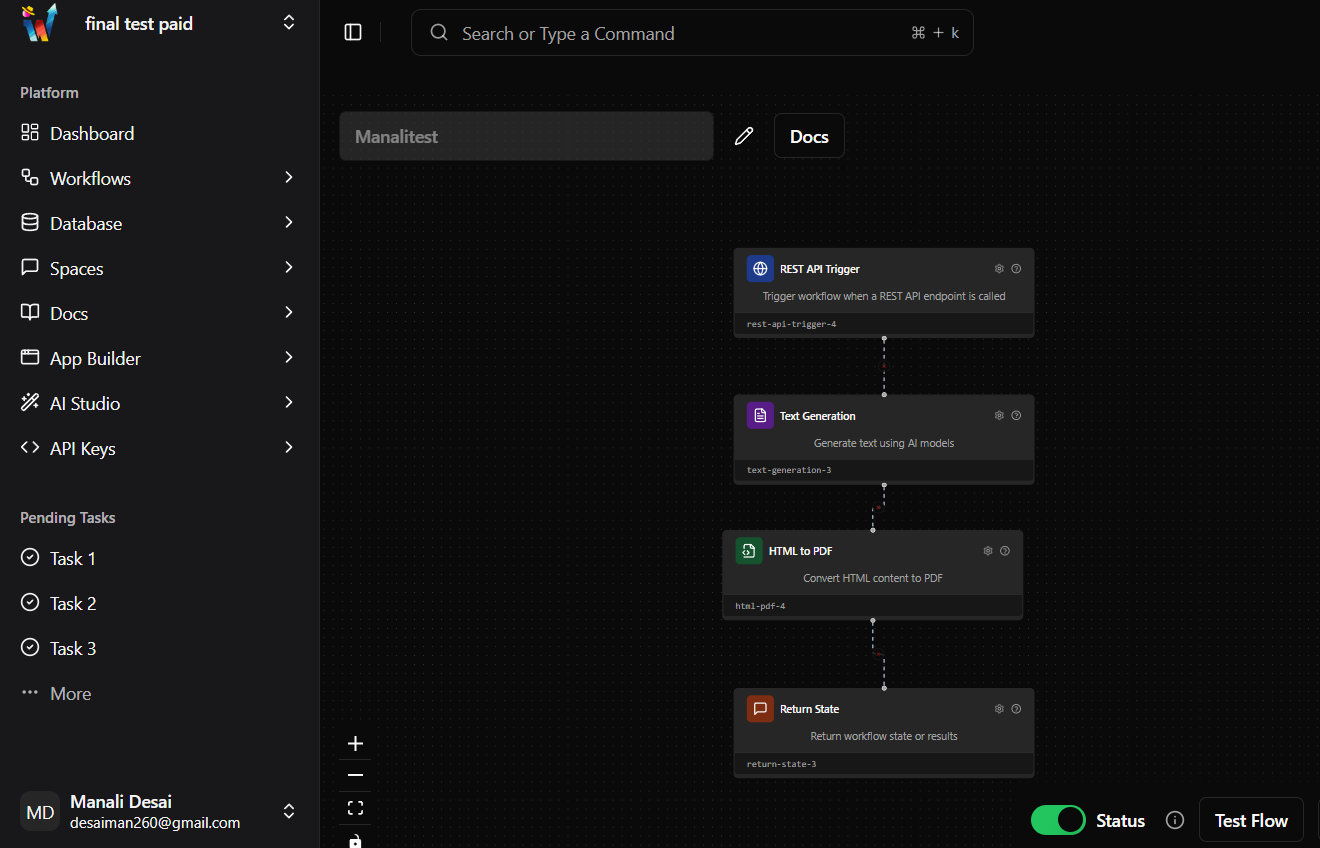


Figure 7.1: Workflow for generating pdf

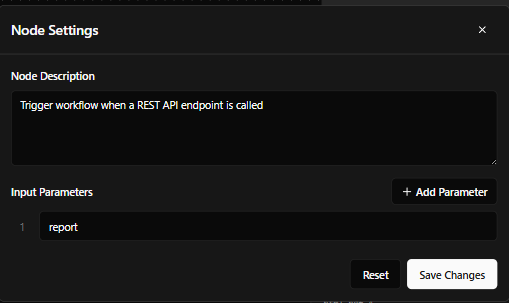


Figure 7.2: Rest API Trigger Node

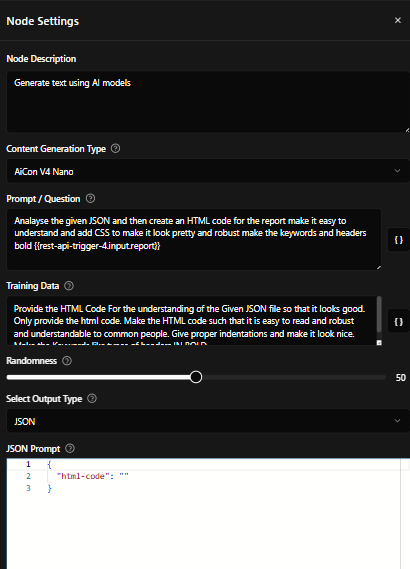


Figure 7.3: Text Generation Node

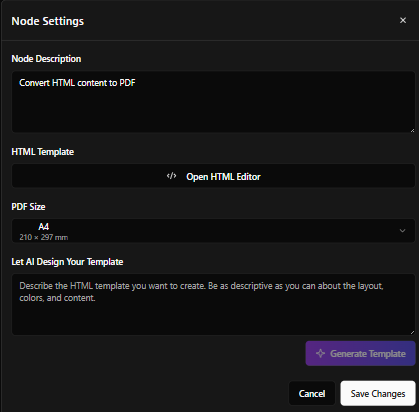


Figure 7.4: Html to Pdf Node

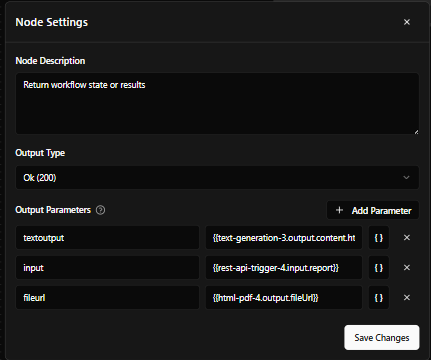


Figure 7.5: Return State Node

### 

### 8. Performance Evaluation and Testing:

##### 8.1 Time Complexity

1. The run\_all\_scans function in webscanner.py exhibits a time complexity of O(f⋅w⋅n) O(f \cdot w \cdot n) O(f⋅w⋅n), where f f f is the number of forms on the target site, w w w is the number of payloads (up to 1000), and n n n is the HTTP response size.
2. This is dominated by the check\_sql\_injection and detect\_xss functions, which test forms with multiple payloads.
3. Other scans (port\_scan: O(1) O(1) O(1), check\_headers: O(n) O(n) O(n), check\_tls\_cert: O(1) O(1) O(1), directory\_bruteforce: O(d⋅n) O(d \cdot n) O(d⋅n) with d≤1000 d \leq 1000 d≤1000, scrape\_contacts: O(n) O(n) O(n)) contribute lesser terms.
4. Real-world runtime is constrained by 5-second timeouts and a 0.1-second delay, but the theoretical complexity highlights scalability challenges with complex sites.

##### TABLE V: FRONT-END TEST CASES

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case No** | **Description** | **Input** | **Desired Output** |
| TC-UI-01 | Verify URL input validation | Empty URL ("") | Error message displayed: "Please enter a valid URL" |
| TC-UI-02 | Verify URL input validation | Invalid URL ("example") | Error message displayed: "Please enter a valid URL" |
| TC-UI-03 | Verify URL input validation | Valid URL ("<https://worqhat.com>") | No error message; "Scan" button remains enabled |
| TC-UI-04 | Verify loading state during scan | Click "Scan" with URL "<https://worqhat.com>" | Loading state displayed (e.g., spinner or "Loading..." text); UI remains responsiv |
| TC-UI-05 | Verify results panel rendering | Scan URL "<https://worqhat.com>" (mock XSS vulnerable response) | Results panel shows timestamp, issue count (e.g., 1), and details (e.g., "XSS Detected") |
| TC-UI-06 | Verify JSON report download | Click "Download JSON Report" after scan | File websafe\_report\_[date].json downloaded with scan results in JSON format |
| TC-UI-07 | Verify full report download (Worqhat API) | Click "Download Full Report" after scan | If reportUrl available, PDF downloaded; else, alert: "Report not ready yet. Please wait..." |
| TC-UI-08 | Verify UI responsiveness after error | Scan URL with Flask server off | Error message: "Failed to scan site: HTTP error!"; UI remains usable for new scans |

##### 

##### TABLE VI: VULNERABILITY SCANNING ENGINE

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case No** | **Description** | **Input** | **Desired Output** |
| TC-SCAN-01 | Verify port scan functionality | port\_scan("worqhat.com", 20, 100) | Returns list of open ports (e.g., [80]); scans["port\_scan"] includes ports and recommendation |
| TC-SCAN-02 | Verify header check functionality | check\_headers("https://worqhat.com") | scans["check\_headers"] lists headers (e.g., missing CSP), issues count, and recommendations |
| TC-SCAN-03 | Verify TLS certificate check | check\_tls\_cert("worqhat.com") | scans["check\_tls\_cert"] includes cert details (e.g., "Valid until", "Issuer") and status |
| TC-SCAN-04 | Verify SQL injection check (no forms) | check\_sql\_injection("https://worqhat.com") (mock no forms) | scans["check\_sql\_injection"]: "vulnerable\_forms": 0, "details": "No forms found..." |
| TC-SCAN-05 | Verify SQL injection check (vulnerable) | check\_sql\_injection("http://test.com") (mock form with SQL error) | scans["check\_sql\_injection"]: "vulnerable\_forms": 1, "details": "Form 1 potentially vulnerable" |
| TC-SCAN-06 | Verify XSS detection (vulnerable) | detect\_xss("http://test.com") (mock form reflecting payload) | scans["detect\_xss"]: "vulnerable": true, "result": "XSS Detected", recommendation provided |
| TC-SCAN-07 | Verify XSS detection (no vulnerabilities) | detect\_xss("https://worqhat.com") (mock no reflection) | scans["detect\_xss"]: "vulnerable": false, "result": "XSS Not Detected" |
| TC-SCAN-08 | Verify directory brute-forcing | directory\_bruteforce("https://worqhat.com") (mock dir /admin) | scans["directory\_bruteforce"]: "found\_directories": ["/admin"], recommendation to secure dirs |
| TC-SCAN-09 | Verify contact scraping | scrape\_contacts("http://test.com") (mock email "test@example.com") | scans["scrape\_contacts"]: "emails\_count": 1, "details": "Emails found: 1" |
| TC-SCAN-10 | Verify error handling (invalid URL) | run\_all\_scans("") | scans["error"]: "status": "Failed", "details": "Error: No valid URL provided" |

##### TABLE 8.2: Vulnerability Scanning Engine

### 

### 9. Deployment Strategies:

##### 9.1 Chrome Extension Deployment:

1. The extension is packaged and tested using the Chrome Extension Manifest v3 standard.
2. The extension includes a manifest.json that defines permissions, background scripts, content scripts, and icons.
3. During development, the extension is tested using Chrome’s Developer Mode by loading the unpacked folder.
4. The extension is not deployed on a cloud as you need to invest money while hosting it hence for the time being it is deployed locally.

##### 9.2 Backend Deployment:

1. The Flask API is hosted locally during development (localhost:5000).
2. For production, it can be:
   * Deployed on cloud platforms like Render, Heroku, or AWS EC2.
   * Secured using HTTPS and proper CORS configuration.
3. The backend can also be containerized using Docker for scalable deployment.

##### 9.3 Security Aspects:

##### 9.3.1 Passive Scanning Only:

1. The tool does not perform any active exploitation.
2. This avoids legal issues and ensures ethical use.

##### 9.3.2 Minimum Required Permissions:

1. The extension only asks for:
   * Access to tabs and active website URLs

##### 9.3.3 Secure Communication:

1. Communication between the extension and Flask API is handled using fetch() over HTTP/HTTPS.
2. In production, HTTPS will be enforced with valid SSL certificates.

##### 9.3.4 Local Processing:

1. All scanning logic executes on the locally hosted API.

### 

### 10. Result and Analysis:

The "WebSafe Sentinel" Chrome Extension performed a security scan on the target URL https://www.worqhat.com on 2025-05-05 at 11:59:09. The scan evaluated multiple aspects of web security, including port scanning, HTTP headers, TLS certificate validity, SQL Injection (SQLi), Cross-Site Scripting (XSS), directory brute-forcing, and contact scraping.

**Site**: https://worqhat.com

Scan Overview

* **Total Scans Performed**: 7 (Port Scan, Check Headers, Check TLS Certificate, Check SQL Injection, Detect XSS, Directory Brute Force, Scrape Contacts).
* **Overall Status**: Completed successfully for all scans.

#### 10.1 Detailed Findings

1. **Port Scan**
   1. **Result**: Open port detected: 80 (HTTP).
   2. **Analysis**: Port 80 is open, indicating the site is accessible via HTTP, which is unencrypted and vulnerable to interception attacks (e.g., man-in-the-middle).
   3. **Recommendation**: Close port 80 and enforce HTTPS (port 443) to ensure secure communication. Implement HTTP-to-HTTPS redirection.
2. **Check Headers**
   1. **Result**: 6 issues identified out of 8 headers checked.
   2. **Issues**:
      1. **Content-Security-Policy (CSP)**: Missing (CWE-693, Medium Risk).
         * Lacks protection against XSS and other injection attacks.
      2. **X-Frame-Options**: Missing (CWE-1021, Medium Risk).
         * Vulnerable to clickjacking attacks.
      3. **X-Content-Type-Options**: Missing (CWE-693, Low Risk).
         * Risks MIME-type sniffing, potentially leading to XSS.
      4. **Referrer-Policy**: Missing.
         * Risks referrer leakage, exposing sensitive data.
      5. **Permissions-Policy**: Missing.
         * Allows unrestricted access to browser features (e.g., geolocation).
      6. **X-XSS-Protection**: Missing.
         * Legacy header; modern browsers rely on CSP, but absence noted.
      7. **Strict-Transport-Security (HSTS)**: Present, properly configured (max-age=63072000).
      8. **Server Header**: Present, discloses "Vercel".
         * Risks information disclosure, aiding attackers in targeting.
   3. **Analysis**: The site lacks critical security headers, increasing vulnerability to attacks like XSS, clickjacking, and data leakage. HSTS is properly configured, ensuring HTTPS enforcement.
   4. **Recommendation**: Implement missing headers (e.g., CSP with default-src 'self', X-Frame-Options: DENY, X-Content-Type-Options: nosniff). Remove or minimize the Server header to reduce information disclosure.
3. **Check TLS Certificate**
   1. **Result**: No issues (valid until 2025-06-18).
   2. **Details**: Domain www.worqhat.com, RSA 2048-bit key, SHA-256 fingerprint, issued by R10.
   3. **Analysis**: The TLS certificate is valid and secure, ensuring encrypted communication.
   4. **Recommendation**: Continue monitoring certificate expiration; renew before 2025-06-18.
4. **Check SQL Injection (SQLi)**
   1. **Result**: No vulnerable forms detected.
   2. **Details**: No forms found to test for SQLi.
   3. **Analysis**: Absence of forms eliminates SQLi risk in this context, but this may indicate limited testing scope.
   4. **Recommendation**: Expand testing to include dynamic pages or API endpoints that may process user inputs.
5. **Detect XSS**
   1. **Result**: No vulnerabilities detected.
   2. **Analysis**: No reflected XSS vulnerabilities found, indicating good input sanitization. However, CSP absence (noted in headers) still poses a latent risk.
   3. **Recommendation**: Implement CSP header to further mitigate XSS risks.
6. **Directory Brute Force**
   1. **Result**: No accessible directories found (71 tested).
   2. **Analysis**: No sensitive directories exposed, reducing the risk of unauthorized access.
   3. **Recommendation**: Continue monitoring and ensure directory indexing is disabled.
7. **Scrape Contacts**
   1. **Result**: No emails or phone numbers found.
   2. **Analysis**: No contact information exposed, reducing the risk of data leakage or phishing.
   3. **Recommendation**: Maintain strict controls to prevent contact info exposure.

#### 10.2 Summary of Results

1. **Total Issues Identified**: 7 (1 open port, 6 header issues).
2. **Severity**:
   1. Medium Risk: 2 (CSP, X-Frame-Options).
   2. Low Risk: 1 (X-Content-Type-Options).
   3. Informational: 4 (Referrer-Policy, Permissions-Policy, X-XSS-Protection, Server header).
3. **Strengths**: Valid TLS certificate, no SQLi/XSS vulnerabilities, no exposed directories or contacts.
4. **Weaknesses**: Significant gaps in security headers and an open HTTP port, increasing exposure to attacks like clickjacking, XSS, and data interception.

**Site :** http://testphp.vulnweb.com

#### 10.3 Scan Overview

* **Total Scans Performed**: 7 (Port Scan, Check Headers, Check TLS Certificate, Check SQL Injection, Detect XSS, Directory Brute Force, Scrape Contacts).
* **Overall Status**: Completed successfully for all scans.

#### 10.4 Detailed Findings

1. **Port Scan**
   * **Result**: No open ports found in range 20-100.
   * **Analysis**: No open ports detected, reducing the attack surface. However, the scan range is limited and may miss common ports like 80 (HTTP), which is likely open given the HTTP URL.
   * **Recommendation**: Expand the port scan range (e.g., include 80, 443) to confirm the absence of open ports.
2. **Check Headers**
   * **Result**: 7 issues identified out of 9 headers checked.
   * **Issues**:
     + **Content-Security-Policy (CSP)**: Missing (CWE-693, Medium Risk).
       - Lacks protection against XSS and injection attacks.
     + **Strict-Transport-Security (HSTS)**: Missing (CWE-319).
       - Using HTTP instead of HTTPS; risks data interception.
     + **X-Frame-Options**: Missing (CWE-1021, Medium Risk).
       - Vulnerable to clickjacking attacks.
     + **X-Content-Type-Options**: Missing (CWE-693, Low Risk).
       - Risks MIME-type sniffing, potentially leading to XSS.
     + **Referrer-Policy**: Missing.
       - Risks referrer leakage, exposing sensitive data.
     + **Permissions-Policy**: Missing.
       - Allows unrestricted access to browser features (e.g., geolocation).
     + **X-XSS-Protection**: Missing.
       - Legacy header; modern browsers rely on CSP, but absence noted.
     + **Server**: Present, discloses "nginx/1.19.0".
       - Risks information disclosure, aiding attackers.
     + **X-Powered-By**: Present, discloses "PHP/5.6.40-38+ubuntu20.04.1+deb.sury.org+1".
       - Exposes outdated PHP version (5.6, end-of-life since 2018), increasing exploit risk.
   * **Analysis**: The site lacks critical security headers, uses HTTP (no HSTS), and discloses server details, significantly increasing vulnerability to attacks like XSS, clickjacking, and data interception. The outdated PHP version is a severe concern.
   * **Recommendation**: Enforce HTTPS with HSTS (max-age=31536000), implement missing headers (e.g., CSP with default-src 'self', X-Frame-Options: DENY), remove Server and X-Powered-By headers, and upgrade PHP to a supported version (e.g., 8.x).
3. **Check TLS Certificate**
   * **Result**: Unable to obtain certificate chain (timed out).
   * **Analysis**: The site uses HTTP, not HTTPS, so no TLS certificate was retrieved. The report’s "Certificate valid" recommendation is incorrect given the timeout error and HTTP usage.
   * **Recommendation**: Enable HTTPS with a valid TLS certificate to secure communication; resolve timeout issues by ensuring proper server configuration.
4. **Check SQL Injection (SQLi)**
   * **Result**: 1 vulnerable form detected.
   * **Details**: Form at http://testphp.vulnweb.com/search.php?test=query showed an SQL error, indicating potential SQLi vulnerability.
   * **Analysis**: The presence of an SQLi vulnerability suggests improper input validation, allowing attackers to manipulate database queries (e.g., unauthorized data access, data manipulation).
   * **Recommendation**: Sanitize all user inputs, use prepared statements or parameterized queries, and validate form data on both client and server sides.
5. **Detect XSS**
   * **Result**: Vulnerable (true).
   * **Analysis**: An XSS vulnerability was detected, likely due to reflected scripts in user inputs (e.g., via the vulnerable form). Combined with the missing CSP header, this poses a high risk of script injection attacks.
   * **Recommendation**: Implement input sanitization (e.g., escape HTML characters), enable CSP, and validate user inputs to prevent script execution.
6. **Directory Brute Force**
   * **Result**: No accessible directories found (71 tested).
   * **Analysis**: No sensitive directories exposed, reducing the risk of unauthorized access.
   * **Recommendation**: Continue monitoring and ensure directory indexing is disabled.
7. **Scrape Contacts**
   * **Result**: No emails or phone numbers found.
   * **Analysis**: No contact information exposed, reducing the risk of data leakage or phishing.
   * **Recommendation**: Maintain strict controls to prevent contact info exposure.

#### 10.5 Summary of Results

* **Total Issues Identified**: 9 (7 header issues, 1 SQLi vulnerability, 1 XSS vulnerability).
* **Severity**:
  + High Risk: 2 (SQLi, XSS vulnerabilities).
  + Medium Risk: 2 (CSP, X-Frame-Options).
  + Low Risk: 1 (X-Content-Type-Options).
  + Informational: 4 (HSTS, Referrer-Policy, Permissions-Policy, X-XSS-Protection, server disclosures).
* **Strengths**: No exposed directories or contact info, limited port exposure (though scan range is narrow).
* **Weaknesses**: Severe vulnerabilities in SQLi and XSS, lack of HTTPS (no HSTS), missing security headers, and outdated PHP version, making the site highly susceptible to attacks.

### 11. Conclusion

The development of the Google Chrome extension for web vulnerability scanning successfully demonstrates an effective approach to identifying common security flaws directly within the browser environment. This project highlights the importance of integrating security tools into the early stages of web application development, making vulnerability detection more accessible, especially for developers and testers who may lack advanced security expertise or resources.

Through real-time analysis of HTTP requests, responses, and input handling, the extension provides a simple yet efficient mechanism for detecting vulnerabilities such as Cross-Site Scripting (XSS), SQL Injection, and other common threats. The user-friendly design and ease of deployment further enhance its practicality for day-to-day use in development and testing workflows.

By bridging the gap between development and security assessment, this solution encourages proactive vulnerability identification and promotes secure coding practices. Although the extension is not a replacement for advanced penetration testing tools, it serves as a valuable supplementary tool that can assist in improving the overall security posture of web applications. Future enhancements may include expanding its scanning capabilities, supporting additional vulnerability types, and integrating reporting features for broader usability.

### 12. Future prospects of the project :

The development of the Google Chrome extension for web vulnerability scanning opens up several avenues for further enhancement, innovation, and scalability. While the current implementation successfully identifies common web vulnerabilities and provides real-time feedback, there is significant scope to expand its capabilities to meet the evolving demands of modern web security.

One of the primary prospects for future development is the integration of support for a wider range of vulnerability types, including more advanced issues such as Security Misconfigurations, and Sensitive Data Exposure. Incorporating checks for emerging vulnerabilities, based on the latest security research and updates from frameworks like OWASP Top 10, will enhance the tool’s relevance and effectiveness.

Another key area for advancement is the implementation of automated reporting and analytics features. Adding options for generating structured vulnerability reports in formats such as PDF or CSV can help developers and testers document identified issues systematically and track remediation efforts efficiently. This can be further extended by integrating the extension with popular issue-tracking platforms like Jira or GitHub, allowing seamless vulnerability management within existing workflows.

Introducing customizable scanning profiles and rule-based scanning mechanisms could allow users to tailor the scanning process according to the specific requirements of their applications, improving accuracy and flexibility. Additionally, providing visual dashboards and summary views within the extension could offer better insights into the overall security status of scanned web pages.

Incorporating machine learning and behavior-based analysis presents another exciting future direction. Leveraging these technologies could enable the tool to identify complex attack patterns, zero-day vulnerabilities, or unusual web behaviors that may not be detectable through signature-based scanning alone.

The extension lays a strong foundation for lightweight, accessible web vulnerability scanning, with ample opportunities for enhancement. Through continuous development and adaptation to emerging security threats, this tool has the potential to evolve into a comprehensive browser-based solution for improving web application security.

### 13. References :

[1] Z. Huang, Y. Zhang, R. Duan and R. Wang, "Research on Malicious URL Identification and Analysis for Network Security," in *Proc. 2021 7th IEEE Int. Conf. Network Intelligence and Digital Content (IC-NIDC)*, Beijing, China, 2021, pp. 418-422, doi: 10.1109/IC-NIDC54101.2021.9660440.

[2] S. A. Kumar and Y. U. Rani, "Implementation and analysis of Web application security measures using OWASP Guidelines," in *Proc. 2022 Int. Conf. Recent Trends in Microelectronics, Automation, Computing and Communications Systems (ICMACC)*, Hyderabad, India, 2022, pp. 182-187, doi: 10.1109/ICMACC54824.2022.10093657.

[3] L. Zhang, X. Liu and Y. Cheng, "Research on Risk Analysis and Protection of Short URL Security," in *Proc. 2024 IEEE 4th Int. Conf. Electronic Technology, Communication and Information (ICETCI)*, Changchun, China, 2024, pp. 277-282, doi: 10.1109/ICETCI61221.2024.10594054.

[4] T. Z. Difaizi, O. P. -W. L. Camille, T. C. Benhura and G. Gupta, "URL Based Malicious Activity Detection Using Machine Learning," in *Proc. 2023 Int. Conf. Disruptive Technologies (ICDT)*, Greater Noida, India, 2023, pp. 414-418, doi: 10.1109/ICDT57929.2023.10150899.

[5] A. R. Kandula, K. P. Kumar, A. B. Prakash and K. S. Kumar, "Malicious URL Analysis using Machine Learning Algorithm," in *Proc. 2024 8th Int. Conf. Electronics, Communication and Aerospace Technology (ICECA)*, Coimbatore, India, 2024, pp. 1160-1167, doi: 10.1109/ICECA63461.2024.10800903.

[6] T. Jaswanthi, T. Harshitha, S. Tanya and M. Belwal, "Malicious URL Detection: Comparative Study of Machine Learning Algorithms," in *Proc. 2024 15th Int. Conf. Computing Communication and Networking Technologies (ICCCNT)*, Kamand, India, 2024, pp. 1-5, doi: 10.1109/ICCCNT61001.2024.10725561.

[7] A. Joshi, A. Raturi and S. Kumar, "An Analysis of Vulnerability Scanners in Web Applications for VAPT," in *Proc. 2022 Int. Conf. Computational Intelligence and Sustainable Engineering Solutions (CISES)*, Greater Noida, India, 2022, pp. 278-283, doi: 10.1109/CISES54857.2022.9844365.

[8] K. Rahman and C. Izurieta, "A Mapping Study of Security Vulnerability Detection Approaches for Web Applications," in *Proc. 2022 48th Euromicro Conf. Software Engineering and Advanced Applications (SEAA)*, Gran Canaria, Spain, 2022, pp. 491-494, doi: 10.1109/SEAA56994.2022.00081.

[9] S. Nagpure and S. Kurkure, "Vulnerability Assessment and Penetration Testing of Web Application," in *Proc. 2017 Int. Conf. Computing, Communication, Control and Automation (ICCUBEA)*, Pune, India, 2017, pp. 1-6, doi: 10.1109/ICCUBEA.2017.8463920.

[10] E. Nowroozi, Abhishek, M. Mohammadi and M. Conti, "An Adversarial Attack Analysis on Malicious Advertisement URL Detection Framework," *IEEE Trans. Network and Service Management*, vol. 20, no. 2, pp. 1332-1344, Jun. 2023, doi: 10.1109/TNSM.2022.3225217.

[11] Y.-C. Chen, Y.-W. Ma and J.-L. Chen, "Intelligent Malicious URL Detection with Feature Analysis," in *Proc. 2020 IEEE Symp. Computers and Communications (ISCC)*, Rennes, France, 2020, pp. 1-5, doi: 10.1109/ISCC50000.2020.9219637.

[12] D. Yadav, D. Gupta, D. Singh, D. Kumar and U. Sharma, "Vulnerabilities and Security of Web Applications," in *Proc. 2018 4th Int. Conf. Computing Communication and Automation (ICCCA)*, Greater Noida, India, 2018, pp. 1-5, doi: 10.1109/CCAA.2018.8777558.

### 14. Publication Details :

##### 14.1 Conference Details

1. **Title of the Paper:** *A study on Formulating a Chrome Extension to Protect Against Web Application Vulnerabilities*
2. **Authors:** Ms. Manali Desai
   * Mr. Nishad Wanjari
   * Mr. Tanishq Bapna
   * Mr. Aditya Patil
   * Dr. Prashant Lahane
3. **Conference Name:** *ICT4SD 2025*
4. **Conference Dates**: *July 17, 2025*
5. **Location**: *Goa, India*
6. **Publication Status:** *Accepted for presentation*
7. **Publisher (Expected):** *Springer*
8. **Acceptance Date:** *April 22, 2025*

##### 14.2 Plagiarism Report



Fig 14.1 Turn-it-in Report Submission

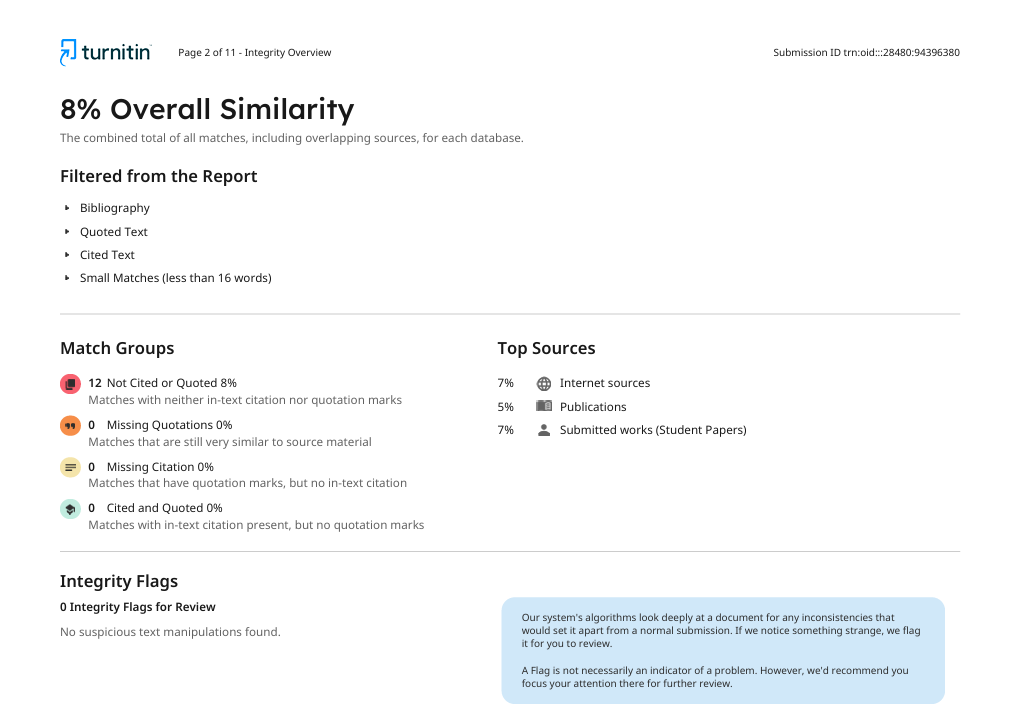


Fig 14.2 Turn-it-in Similarity Report

### 

### 15. Individual Contribution

##### Problem Statement: Developing "WebSafe Sentinel," a Chrome Extension to detect web vulnerabilities (XSS, SQLi, insecure headers) and promote security education for non-technical users.

**15.1 Name of the Student**: Manali Desai

**Module Title:** Vulnerability Scanner & PDF Report Downloading

**Project’s Module Objectives:**

1. Implement the webscanner.py script to detect XSS, SQLi, and insecure headers.
2. Integrate the Worqhat API for PDF report generation.  
    Project’s Module Scope:
3. Covers vulnerability scanning logic (run\_all\_scans, check\_sql\_injection, check\_headers) and Worqhat API calls (triggerFlow).

**Project’s Module(s):  
 Hardware & Software Requirements:**

1. Python 3.x, requests, BeautifulSoup, OpenSSL, certifi, colorama.
2. Worqhat API access for report generation

**Module Interfaces:**

1. **With Nishad:**
   * "Scan Result Pipeline": webscanner.py returns JSON results to Flask (api.py) for frontend rendering.
   * "Report Generation Sync": triggerFlow provides PDF report URL to React popup.

**Module Dependencies:**

1. Nishad’s Backend Module:
   * "Requires Flask /scan endpoint to invoke run\_all\_scans and return results."

**Module Design:**

1. Flowchart of run\_all\_scans process (URL → form analysis → vulnerability checks → JSON output).
2. triggerFlow workflow (send JSON → receive PDF URL).

**Module Implementation:**

1. Python scripts for scanning (e.g., if is\_response\_vulnerable(response): return True).
2. Worqhat API POST request in triggerFlow with JSON payload.

**Module Testing Strategies:**

1. Verify XSS/SQLi detection on testbeds (e.g., DVWA).
2. Validate PDF report generation via Worqhat API.

**Module Deployment:**

1. Integrated scanner and API logic into final build.

##### 15.2 Name of the Student: Nishad Wanjari

**Module Title**: Frontend Development & Backend Integration

**Project’s Module Objectives**:

1. Develop the React-Vite frontend for the popup UI (App.jsx).
2. Integrate the frontend with the Flask backend for scan requests and results.

**Project’s Module Scope**:

1. Covers popup UI (URL input, "Scan" button, results panel) and backend communication (fetch to /scan).

**Project’s Module(s)**:  
 **Hardware & Software Requirements**:

1. Node.js, npm, React, Vite, Google Chrome (Version 90+).  
    **Module Interfaces**:
2. With Manali:
   * "Scan Result Pipeline": Frontend receives JSON results from webscanner.py via Flask.
   * "Report Generation Sync": Displays Worqhat PDF report URL in popup.

**Module Dependencies**:

1. Manali’s Scanner Module:
   * "Requires run\_all\_scans JSON output for rendering in popup."

**Module Design**:

1. Flowchart of scan process (click "Scan" → POST /scan → display results).
2. UI wireframe for popup layout (input, buttons, results panel).

**Module Implementation**:

1. React components in App.jsx (e.g., handleScan for POST requests).
2. Backend integration with fetch to http://localhost:5000/scan.

**Module Testing Strategies**:

1. Verify UI responsiveness and result rendering on 10+ test URLs.
2. Validate backend-frontend communication with Flask server.

**Module Deployment**:

1. Deployed frontend as part of Chrome extension build.

##### 15.3 Name of the Student: Tanishq Bapna

**Module Title**: Backend Support & Research

**Project’s Module Objectives**:

1. Assist in developing the Flask backend script (api.py).
2. Research vulnerability detection techniques and headers logic

**Project’s Module Scope**:

1. Covers /scan endpoint logic in Flask and research for check\_headers (anti-CSRF token check).

**Project’s Module(s)**:  
 **Hardware & Software Requirements**:

1. Python 3.x, Flask, flask-cors.

**Module Interfaces**:

1. With Nishad:
   * "Backend-Frontend Sync": /scan endpoint processes requests from React popup.
2. With Manali:
   * "Scanner Integration": Routes requests to run\_all\_scans in webscanner.py.

**Module Dependencies**:

1. Manali’s Scanner Module:
   * "Requires run\_all\_scans to process scan requests."

**Module Design**:

1. Flowchart of Flask request handling (receive POST → invoke run\_all\_scans → return JSON).

**Module Implementation**:

1. Flask endpoint /scan in api.py (e.g., jsonify(run\_all\_scans(url))).
2. Researched CWE-352 for anti-CSRF token implementation.

**Module Testing Strategies**:

1. /scan endpoint with 5 URLs for correct JSON response.
2. Test Validate research findings with OWASP standards.

**Module Deployment**:

1. Integrated backend script into final build.

##### 15.4 Name of the Student: Adityap

**Module Title**: Testing and Documentation

**Project’s Module Objectives**:

1. Design and execute testing strategies for the Chrome extension’s components.

**Project’s Module Scope**:

1. Covers project report sections (e.g., methodology, requirements, testing) and testing of frontend, backend, scanner, and API integration.

**Project’s Module(s)**:  
 **Hardware & Software Requirements**:

1. Word processor (e.g., Microsoft Word, Google Docs).
2. Google Chrome (Version 90+), Python 3.x, Node.js (for testing environment).

**Module Interfaces**:

1. With All Team Members:
   * "Documentation Sync": Gathered inputs from Manali, Nishad, and Tanishq for methodology and testing sections.
   * "Testing Feedback Loop": Provided test results to the team for debugging and improvements.

**Module Dependencies**:

1. All Modules:
   * "Requires implementation details and test results for accurate documentation."
   * "Depends on frontend (App.jsx), backend (api.py), and scanner (webscanner.py) for comprehensive testing."

**Module Design**:

1. Outline of report structure (Introduction, Methodology, Results).
2. Test plan for vulnerability scanning, UI responsiveness, backend communication, and API integration.

**Module Implementation**:

1. Wrote sections like "Methodology" (layered architecture, challenges) and "Testing Strategies".
2. Conducted tests:
   * Tested XSS/SQLi detection on testbeds (e.g., DVWA) for webscanner.py.
   * Validated UI responsiveness and result rendering on 10+ test URLs in React popup.
   * Verified backend-frontend communication with Flask (/scan endpoint) using 5 URLs.

**Module Testing Strategies**:

1. Reviewed documentation for accuracy and completeness with team.
2. Validated test results against OWASP standards and user accessibility requirements.

**Module Deployment**:

1. Finalized report and test documentation for submission.

**Project to Outcome mapping**

Objectives:

1. Design and development of a Google Chrome extension for web vulnerability scanning
2. Integration of automated scanning features
3. Testing the extension on various sample web applications and test environments
4. Implementation of a user-friendly interface for the extension

TABLE VII:PROJECT OUTCOME MAPPING

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No.** | **PRN No.** | **Student Name** | **Individual Project Student Specific Objective** | **Learning Outcomes mapped ( To be filled by Guide )** |
| 1 | 1032211347 | Manali Desai | Integrate the Worqhat API(trigger flow) to allow PDF report downloads and generate the webscanner.py script logic for vulnerability scanning |  |
| 2 | 1032210680 | Nishad Wanjari | Design and develop the REact-vite popup UI(App.jsx) with url input ,”Scan” button,results panel and report download buttons ensuring an intuitive user experience |  |
| 3 | 1032211573 | Aditya Patil | Documented the UI implementation and tested its responsiveness to ensure usability |  |
| 4 | 1032212350 | Tanishq Bapna | Support the design of scanning script logic in webscanner.py and design header analysis |  |