Міністерство освіти і науки України
Харківський національний університет імені В.Н. Каразіна
Навчально-науковий інститут комп’ютерних наук та штучного інтелекту
Спеціальність 125 «Кібербезпека»
Освітня програма «Кібербезпека»

В.о. зав. кафедрою КІСМіТ
Марина ЄСІНА
“Допущено до захисту”
« »__________2025р.

Пояснювальна записка

до кваліфікаційної роботи бакалавра

на тему: «Розробка додатку для аналізу вразливостей веб-сайтів»

оцінка « ________ »

Голова ЕК

Мичуда Л.З.

Керівник: к.т.н. Нарєжній О.П.

Рецензент: доцент Лисицький К. Є.

Виконавець: студент групи КБ41

Мозжухін Є.А.

Харків 2025

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи бакалавра містить 59 сторінки, 36 рисунків, 1 таблицю, 26 джерел.

Мета роботи полягає у розробці додатку для автоматизованого аналізу вразливостей веб-сайтів, який забезпечує сканування, виявлення слабких місць і формування звітів для підвищення їхньої захищеності. Завдання включали дослідження принципів роботи веб-сайтів і основ кібербезпеки, аналіз типів уразливостей, створення модульного веб-сканера, розробку тестового сайту з уразливостями, тестування сканера та оцінку його ефективності.

Методи дослідження охоплюють теоретичний аналіз наукових джерел і рекомендацій OWASP, програмування на Python з використанням фреймворку Flask та бібліотек, розробку модульної архітектури сканера, тестування на контрольованому веб-сайті з уразливостями, а також оцінку результатів через порівняльний аналіз.

Результати роботи включають створення веб-сканера з модульною архітектурою, який містить компоненти для обходу сайту, відправки HTTP-запитів, логування та генерації звітів. Розроблено тестовий сайт на Flask із 11 типами уразливостей Новаторство полягає в інтеграції модульного підходу та автоматизації тестування, що відповідає сучасним вимогам кібербезпеки.

Значущість роботи полягає в можливості використання сканера для автоматизованого аудиту безпеки веб-сайтів, що сприяє зниженню ризиків кібератак. Результати корисні для фахівців із кібербезпеки та розробників веб-додатків.

Перспективи розвитку передбачають вдосконалення сканера через додавання підтримки асинхронних запитів, аналізу складних логічних уразливостей і адаптацію до односторінкових додатків.

Ключові слова: ВЕБ-СКАНЕР, КІБЕРБЕЗПЕКА, УРАЗЛИВОСТІ, SQL-ІН’ЄКЦІЇ, XSS, CSRF, RCE, OWASP, FLASK, АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ.

ABSTRACT

The explanatory note to the bachelor's thesis comprises 59 pages, 36 figures, 1 table, and 26 sources.

The objective of the thesis is to develop an application for automated analysis of web vulnerabilities, enabling scanning, detection of weaknesses, and generation of reports to enhance website security. The tasks included studying the principles of website operation and cybersecurity fundamentals, analyzing types of vulnerabilities, creating a modular web scanner, developing a test website with vulnerabilities, testing the scanner, and evaluating its effectiveness.

The research methods encompassed theoretical analysis of scientific sources and OWASP recommendations, programming in Python using the Flask framework and libraries (sqlite3, requests), developing a modular scanner architecture, testing on a controlled website with vulnerabilities (SQL injections, XSS, CSRF, RCE, etc.), and evaluating results through comparative analysis.

The results include the creation of a web scanner with a modular architecture, incorporating components for site crawling, HTTP request handling, logging, and report generation. A test website was developed using Flask, featuring 11 types of vulnerabilities. The scanner successfully identified 119 vulnerabilities, demonstrating high accuracy and flexibility. The innovation lies in integrating a modular approach and automation of testing, aligning with modern cybersecurity requirements.

The significance of the work lies in the scanner's potential for automated security audits of websites, contributing to reduced cyberattack risks. The results are valuable for cybersecurity specialists and web developers.

Future development prospects involve enhancing the scanner by adding support for asynchronous requests, analyzing complex logical vulnerabilities, and adapting to single-page applications.

Keywords: WEB SCANNER, CYBERSECURITY, VULNERABILITIES, SQL INJECTIONS, XSS, CSRF, RCE, OWASP, FLASK, AUTOMATED TESTING.

ЗМІСТ

ПЕРЕЛІК ПОЗНАЧЕНЬ І СКОРОЧЕНЬ....................6
ВСТУП....................7
1 ОСНОВИ РОБОТИ ВЕБ-САЙТІВ ТА ЇХ ЗАХИЩЕНОСТІ....................7
1.1 Принципи роботи веб-сайтів....................7
1.2 Класифікація веб-сайтів за типами....................8
1.3 Основи кібербезпеки веб-сайтів....................9
1.4 Класифікація уразливостей веб-сайтів та методи їх тестуванн....................9
1.5 Автоматизоване тестування вразливостей....................10
2 РОЗРОБКА ВЕБ-СКАНЕРА ВРАЗЛИВОСТЕЙ....................12
2.1 Мета та завдання створення веб-сканера вразливостей....................12
2.2 Вибір архітектури....................13
2.3 Основні компоненти веб-сканера....................14
2.4 Модулі виявлення уразливостей....................18
2.5 Інтеграція та взаємодія компонентів....................23
3 СТВОРЕННЯ ТЕСТОВОГО ВЕБ-САЙТУ З УРАЗЛИВОСТЯМИ....................26
3.1 Мета та завдання створення тестового веб-сайту....................26
3.2 Вибір технологій для розробки....................28
3.3 Структура тестового веб-сайту....................31
3.4 Функціональність веб-сайту....................35
3.5 Перелік вбудованих уразливостей....................45
4 ТЕСТУВАННЯ ВЕБ-СКАНЕРА ВРАЗЛИВОСТЕЙ....................49
4.1 Результати тестування....................49
4.2 Оцінка ефективності веб-сканера....................54
4.3 Рекомендації щодо покращення веб-сканера....................56
ВИСНОВКИ....................59
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ....................60
ДОДАТОК....................63

ПЕРЕЛІК ПОЗНАЧЕНЬ І СКОРОЧЕНЬ

HTTP	– HyperText Transfer Protocol
HTTPS	– HyperText Transfer Protocol Secure
TLS/SSL	– Transport Layer Security / Secure Sockets Layer
HTML	– HyperText Markup Language
CSS	– Cascading Style Sheets
AJAX	– Asynchronous JavaScript and XML
API	– Application Programming Interface
CMS	– Content Management System
SPA	– Single Page Application
XSS	– Cross-Site Scripting
SQL	– Structured Query Language
CSRF	– Cross-Site Request Forgery
SSRF	– Server-Side Request Forgery
RCE	– Remote Code Execution
IDOR	– Insecure Direct Object Reference
CIA	– Confidentiality, Integrity, Availability
DAST	– Dynamic Application Security Testing
SAST	– Static Application Security Testing
CLI	– Command Line Interface
BFS	– Breadth-First Search
URL	– Uniform Resource Locator
NoSQL	– Not Only SQL
JSON	– JavaScript Object Notation
DOM	– Document Object Model
OWASP	– Open Web Application Security Project

ВСТУП

Сучасний розвиток інформаційних технологій супроводжується стрімким зростанням кількості веб-сайтів, які є ключовими елементами цифрової інфраструктури, забезпечуючи комунікацію, надання послуг та зберігання даних. Проте разом із цим зростає і кількість кіберзагроз, які становлять серйозну небезпеку для безпеки веб-ресурсів. У зв’язку з цим актуальність теми дипломної роботи полягає в необхідності розробки ефективних інструментів для автоматизованого аналізу вразливостей веб-сайтів, що дозволяє своєчасно виявляти слабкі місця та підвищувати рівень їхньої захищеності.[1]

Об’єктом дослідження є веб-сканер вразливостей, призначений для автоматизованого виявлення уразливостей веб-сайтів. Предметом дослідження виступають методи та підходи до створення такого сканера, а також його ефективність у виявленні уразливостей на прикладі спеціально розробленого тестового веб-сайту. Метою роботи є розробка додатку для аналізу вразливостей веб-сайтів, який забезпечує автоматизоване сканування, виявлення уразливостей та формування звітів для подальшого їх усунення.[2]

Для досягнення поставленої мети визначено такі завдання: дослідити принципи роботи веб-сайтів та основи їхньої кібербезпеки, проаналізувати типи уразливостей і методи їх тестування, розробити веб-сканер вразливостей із модульною архітектурою, створити тестовий веб-сайт із вбудованими уразливостями, провести тестування сканера на тестовому сайті, оцінити його ефективність та сформулювати рекомендації щодо покращення. Робота базується на рекомендаціях OWASP Top 10, технічній документації веб-технологій, а також наукових статтях і посібниках із кібербезпеки та тестування вразливостей.[3]

1 ОСНОВИ РОБОТИ ВЕБ-САЙТІВ ТА ЇХ ЗАХИЩЕНОСТІ

1.1 Принципи роботи веб-сайтів

Веб-сайти є ключовим інструментом для представлення інформації, надання послуг і взаємодії з користувачами в Інтернеті. Їх функціонування базується на клієнт-серверній архітектурі, де клієнт (зазвичай веб-браузер) обмінюється даними із сервером, який обробляє запити та зберігає інформацію. Розуміння цих принципів є необхідним для аналізу безпеки веб-сайтів і виявлення уразливостей.

Основою роботи веб-сайтів є протоколи HTTP/HTTPS. HTTP визначає правила обміну даними: клієнт надсилає запит (наприклад, GET або POST) із URL і заголовками, а сервер повертає відповідь зі статус-кодом (наприклад, 200 OK) і контентом, зазвичай у форматі HTML. HTTPS додає шифрування через TLS/SSL, забезпечуючи конфіденційність і цілісність даних.

Архітектура веб-сайтів включає клієнтські та серверні компоненти. На стороні клієнта HTML визначає структуру сторінки, CSS — її оформлення, а JavaScript забезпечує інтерактивність, зокрема через асинхронні запити (AJAX). На сервері використовуються мови програмування (Python, PHP) і фреймворки (Flask, Django) для обробки запитів і взаємодії з базами даних (MySQL, SQLite), які зберігають дані користувачів чи налаштування.

Веб-сайти розрізняються за типами. Статичні сайти складаються з фіксованих HTML-сторінок і менш уразливі до серверних атак. Динамічні сайти генерують контент у реальному часі за допомогою скриптів і баз даних, що підвищує ризик уразливостей. Односторінкові додатки (SPA), побудовані на JavaScript-фреймворках (React, Angular), оновлюють контент динамічно через API, але їхня складність створює додаткові точки вразливості.

Інтеграція технологій, таких як API, системи управління контентом (CMS, наприклад, WordPress) чи хмарні сервіси, розширює функціональність, але може створювати нові уразливості через помилки конфігурації. Таким чином, принципи роботи веб-сайтів формують основу для аналізу їхньої безпеки та розробки інструментів для виявлення слабких місць.

1.2. Класифікація веб-сайтів за типами

Веб-сайти різняться за структурою, функціональністю та призначенням. Їх класифікація за типами сприяє систематизації підходів до аналізу безпеки, оскільки кожен тип має специфічні особливості, що впливають на уразливості.

Статичні веб-сайти складаються з фіксованих HTML-сторінок, які не змінюються без втручання розробника. Вони використовуються для простих сторінок, наприклад, корпоративних сайтів чи портфоліо. Завдяки відсутності серверної логіки та баз даних такі сайти менш схильні до SQL-ін’єкцій чи віддаленого виконання коду, але можуть бути вразливими до міжсайтових сценаріїв через неналежну обробку даних у JavaScript.

Динамічні веб-сайти генерують контент у реальному часі за допомогою серверних скриптів і баз даних. Вони застосовуються для інтернет-магазинів, соціальних мереж і форумів, адаптуючи контент до запитів користувача. Гнучкість створює ризики: SQL-ін’єкції через непараметризовані запити, XSS через недостатнє екранування та перехресні запити із підробленням сайту у формах без токенів.

Односторінкові додатки, побудовані на JavaScript-фреймворках, завантажують єдину HTML-сторінку, оновлюючи контент через API-запити. SPA популярні для інтерактивних інтерфейсів, але асинхронна взаємодія підвищує ризики Server-Side Request Forgery (SSRF) і XSS через невалідовані API-дані.

Сайти на основі систем управління контентом, таких як WordPress чи Joomla, дозволяють створювати контент без технічних знань. Модульність CMS (плагіни, теми) розширює функціональність, але застарілі компоненти чи неправильна конфігурація створюють уразливості, зокрема RCE та Insecure Direct Object Reference.

Класифікація веб-сайтів підкреслює їхні відмінності, що впливають на безпеку. Статичні сайти потребують захисту клієнтської сторони, тоді як динамічні сайти, SPA та CMS вимагають комплексного підходу, включаючи валідацію даних, захист API та оновлення компонентів.

1.3. Основи кібербезпеки веб-сайтів

Кібербезпека веб-сайтів є критично важливим аспектом їхньої надійності в умовах зростання кіберзагроз. Вона охоплює заходи для запобігання несанкціонованому доступу, витоку даних і виконанню шкідливого коду, що можуть порушити функціональність сайтів або завдати шкоди користувачам. У цьому дослідженні кібербезпека є основою для розробки інструментів виявлення уразливостей і підвищення захисту веб-ресурсів [6].

Основою кібербезпеки є CIA-тріада: конфіденційність, цілісність, доступність. Конфіденційність захищає дані від несанкціонованого доступу, цілісність запобігає їх модифікації зловмисниками, а доступність забезпечує безперебійну роботу сайту для легітимних користувачів. Порушення цих принципів може призвести до компрометації даних, фінансових збитків або втрати довіри [7].

Захист веб-сайтів досягається технічними й організаційними заходами. Технічні включають HTTPS для шифрування, валідацію даних проти XSS і SQL-ін’єкцій, токени для захисту від CSRF. Організаційні заходи передбачають оновлення програмного забезпечення, навчання розробників і аудити безпеки.

Кібербезпека має стратегічне значення через роль веб-сайтів у бізнесі, комунікації та зберіганні даних. SQL-ін’єкції дозволяють доступ до баз даних, RCE — виконання коду на сервері, XSS — крадіжку сесій, CSRF — несанкціоновані дії. Ці загрози створюють ризики для користувачів і власників сайтів, підкреслюючи важливість веб-сканерів для виявлення уразливостей і зміцнення захисту [6].

1.4. Класифікація уразливостей веб-сайтів та методи їх тестування

Класифікація уразливостей веб-сайтів є основою кібербезпеки, дозволяючи систематизувати ризики та розробляти інструменти для їх виявлення. Уразливості поділяються за типами та критичністю, що допомагає визначити пріоритети усунення. Веб-сканери автоматизують аналіз слабких місць, підвищуючи захист цифрових ресурсів у сучасних умовах зростання кіберзагроз [8].

Ін’єкції даних виникають через недостатню перевірку вхідних параметрів. SQL-ін’єкції, спричинені непараметризованими запитами, дозволяють виконувати

довільні команди, наприклад, отримувати чи видаляти дані бази. NoSQL-ін'єкції, використовуючи псевдо-JSON-запити типу {"$gt": ""}, дають змогу маніпулювати даними, що становить високу небезпеку через потенційний витік усієї бази [9].

Уразливості клієнтської сторони, зокрема XSS, поділяються на відображені, збережені та DOM-based. Вони дозволяють зловмисникам виконувати шкідливий код, красти сесії чи імітувати дії користувачів, створюючи загрозу для сайтів із автентифікацією. Уразливості серверної логіки, як Directory Traversal, надають доступ до системних файлів, тоді як Open Redirect перенаправляє на шкідливі ресурси, а IDOR відкриває чужі дані.

Небезпечне завантаження файлів без перевірки типу чи вмісту дозволяє завантажувати шкідливі скрипти, наприклад, PHP-файли, що може призвести до RCE та повного контролю над сервером. Такі уразливості вимагають ретельної валідації, особливо в середовищах, де сервер виконує завантажені файли.

Класифікація за критичністю визначає пріоритети: SQL-ін'єкції та RCE мають найвищий рівень небезпеки через можливість повної компрометації, тоді як Open Redirect менш критичний, але може спричинити репутаційні збитки. Вплив уразливостей проявляється у втраті даних і зниженні довіри, що підкреслює необхідність їх своєчасного виявлення.

Для тестування уразливостей застосовуються ручні, автоматизовані та гібридні методи. Ручне тестування ефективне для логічних помилок, як IDOR, але потребує часу та кваліфікації. Автоматизовані сканери швидко виявляють типові уразливості, як SQL-ін'єкції чи XSS, але можуть пропускати складні слабкі місця та видавати хибнопозитивні результати. Гібридний підхід поєднує автоматизацію з ручним аналізом, забезпечуючи баланс швидкості та точності, що є оптимальним для комплексної оцінки безпеки [8].

1.5. Автоматизоване тестування вразливостей

Автоматизоване тестування вразливостей є ключовим підходом у забезпеченні безпеки веб-сайтів, дозволяючи швидко виявляти слабкі місця без необхідності ручного аналізу. Одним із основних методів у цій сфері є DAST, який фокусується

на аналізі веб-додатків у процесі їхньої роботи. У рамках даного дослідження розглядаються принципи DAST та його застосування для автоматизованого тестування [10].

DAST передбачає тестування веб-сайту з позиції зовнішнього атакуючого, імітуючи реальні сценарії атак. На відміну від SAST, який аналізує вихідний код, DAST працює з додатком у реальному часі, надсилаючи запити та аналізуючи відповіді сервера. Основний принцип DAST полягає у скануванні всіх доступних точок входу, таких як форми, URL-параметри та API-ендпоінти, шляхом введення тестових навантажень. Це дозволяє виявляти уразливості, які проявляються під час виконання, наприклад, через неправильну обробку введених даних або недостатню валідацію.

Процес DAST складається з кількох етапів. Спочатку інструмент обходить сайт, створюючи карту доступних сторінок і форм. Далі він генерує запити з різними навантаженнями, аналізуючи відповіді на наявність аномалій, таких як затримки, помилки сервера або виконання шкідливого коду. Наприклад, інструмент може надіслати запит із навантаженням для перевірки, чи сервер повертає конфіденційні дані. DAST не залежить від технологій, використаних для створення сайту, що робить його універсальним, але він може пропускати уразливості, які не проявляються під час тестування, наприклад, логічні помилки.

Автоматизоване тестування за допомогою DAST забезпечує швидке виявлення уразливостей, що є критично важливим у сучасних умовах, коли кількість веб-сайтів і складність атак постійно зростають. Проте для досягнення максимальної ефективності DAST часто комбінується з іншими методами, що дозволяє компенсувати його обмеження та забезпечити комплексний підхід до оцінки безпеки веб-ресурсів.

2 РОЗРОБКА ВЕБ-СКАНЕРА ВРАЗЛИВОСТЕЙ

2.1. Мета та завдання створення веб-сканера вразливостей

Розробка веб-сканера вразливостей є центральним елементом даного дослідження, спрямованим на створення інструменту для автоматизованого виявлення слабких місць у веб-сайтах. У сучасних умовах, коли кібератаки стають дедалі складнішими, а кількість веб-ресурсів невпинно зростає, автоматизація процесу тестування безпеки є необхідною для своєчасного виявлення уразливостей та запобігання їх експлуатації зловмисниками. Мета створення веб-сканера полягає у забезпеченні ефективного, швидкого та повторюваного аналізу веб-сайтів, що дозволяє виявляти широкий спектр уразливостей і надавати розробникам детальні звіти для їх усунення.[11]

Основною метою розробки є створення модульного інструменту, який здатен сканувати веб-сайти, виявляти уразливості та формувати звіти у зручному форматі. Веб-сканер призначений для автоматизації процесу, який зазвичай виконується вручну, що значно економить час і ресурси, а також забезпечує систематичний підхід до тестування. Інструмент має охоплювати широкий спектр уразливостей, включаючи ті, що входять до OWASP Top 10, і бути розширюваним, щоб у майбутньому можна було додавати нові модулі для виявлення інших типів загроз.

Для досягнення цієї мети було визначено такі завдання:

- Розробити модульну архітектуру сканера, яка забезпечує гнучкість і можливість інтеграції нових функцій.
- Реалізувати основні компоненти, такі як краулер для обходу сайту, модуль відправки HTTP-запитів, логування та генерацію звітів.
- Створити модулі для виявлення різних типів уразливостей, забезпечуючи їхню точність і ефективність.
- Забезпечити інтеграцію компонентів у єдину систему, яка працює стабільно та надає користувачу зручний інтерфейс для налаштування сканування.

- Провести тестування сканера на спеціально створеному тестовому сайті, щоб оцінити його ефективність і визначити напрями для вдосконалення.

Виконання цих завдань дозволяє створити інструмент, який не лише виявляє уразливості, а й надає розробникам чіткі рекомендації для їх усунення, сприяючи підвищенню рівня безпеки веб-сайтів. Веб-сканер розроблено з урахуванням потреб як розробників, так і спеціалістів із кібербезпеки, забезпечуючи баланс між простотою використання та глибиною аналізу.

2.2. Вибір архітектури

Розробка веб-сканера вразливостей вимагає ретельного вибору архітектури, від якої залежить його ефективність і гнучкість. У цьому дослідженні обрано модульну архітектуру, засновану на принципах модульності та розширюваності, що забезпечує зручність розробки, підтримки й удосконалення інструменту [12].

Модульність передбачає поділ системи на незалежні компоненти, кожен із яких виконує окрему функцію. У веб-сканері це реалізовано через модулі: краулер для обходу сайту, запитувач для HTTP-запитів, логер для виведення інформації, генератор звітів і модулі виявлення уразливостей. Автономність модулів спрощує їх тестування та заміну без впливу на систему [13].

Розширюваність дозволяє додавати нові функції без зміни існуючого коду. Архітектура використовує концепцію плагінів: модулі уразливостей реєструються через спеціальний механізм, що полегшує інтеграцію нових типів сканування шляхом створення окремого класу з потрібною логікою.

Архітектура відповідає принципам SOLID, зокрема єдиної відповідальності, де кожен модуль має чітку роль, і відкритості/закритості, що підтримує розширення без модифікації коду. Об’єктно-орієнтоване програмування забезпечує спадкування, дозволяючи модулям уразливостей базуватися на загальному класі з основними методами сканування.

Управління модулями реалізовано через механізм динамічного підключення під час запуску сканера. Користувач може вибирати типи уразливостей через

аргументи командного рядка, що підвищує гнучкість і ефективність сканування залежно від потреб.

Обрана архітектура забезпечує баланс між продуктивністю, гнучкістю та зручністю. Вона дозволяє сканеру адаптуватися до нових загроз у кібербезпеці, що є критично важливим у контексті швидкого розвитку технологій і атак.

2.3. Основні компоненти веб-сканера

Веб-сканер вразливостей є складною системою, яка складається з кількох ключових компонентів, кожен із яких виконує чітко визначену функцію. Ці компоненти взаємодіють між собою, забезпечуючи повний цикл роботи сканера: від обробки аргументів командного рядка до генерації звітів про виявлені уразливості. Сканер працює через командний рядок (CLI), що дозволяє користувачу налаштовувати параметри сканування за допомогою аргументів, таких як цільовий URL, глибина обходу, вибір модулів і формат звітів. У цьому пункті детально описано основні модулі сканера, їхню функціональність, технічні особливості реалізації та взаємодію.[14]

Першим компонентом є модуль обробки аргументів командного рядка, реалізований у файлі cli_parser.py. Цей модуль використовує бібліотеку argparse для парсингу аргументів, що задаються користувачем через CLI. Наприклад, команда python main.py http://127.0.0.1:5000 --modules all --report txt --output report.txt визначає цільовий URL, вибір усіх модулів вразливостей, формат звіту TXT і вихідний файл. Модуль підтримує широкий спектр параметрів, включаючи глибину обходу (--depth), затримку між запитами (--delay), таймаут (--timeout), а також режими виведення (--quiet, --verbose, --no-color). Рис. 2.1 показує приклад команди. Фрагмент коду, що ілюструє обробку аргументів, наведено у додатку А, Лістинг 1.1.

```
(.venv) PS C:\PROJECTS\PycharmProjects\WEB-SCANNER> python main.py -h
usage: main.py [-h] [--depth DEPTH] [--modules MODULES] [--list-modules] [--report {txt,html,csv}] [--output OUTPUT] [--auth AUTH] [--delay DELAY] [--quiet] [--verbose] [--timeout TIMEOUT] [--user-agent USER_AGENT]
               [--exclude EXCLUDE] [--scope SCOPE] [--no-color]
               [url]

Web Vulnerability Scanner: Scan a website for various vulnerabilities.

positional arguments:
  url                   Target URL to scan (e.g. http://example.com).

options:
  -h, --help            show this help message and exit
  --depth DEPTH         Depth of crawling (default: 1).
  --modules MODULES     Comma-separated list of modules or 'all' to enable all modules (default: all).
  --list-modules        List all available vulnerability modules and exit.
  --report {txt,html,csv}
                        Report format: txt, html, or csv (default: txt).
  --output OUTPUT       Output file for the report. If not specified, results are printed to console.
  --auth AUTH           Credentials for authentication in the format username:password.
  --delay DELAY         Delay between requests in seconds (default: 0.0).
  --quiet               Minimal console output, only essential information.
  --verbose             Verbose console output with detailed information.
  --timeout TIMEOUT     Request timeout in seconds (default: 10.0).
  --user-agent USER_AGENT
                        Custom User-Agent string for HTTP requests.
  --exclude EXCLUDE     Exclude URLs matching this pattern from scanning.
  --scope SCOPE         Limit scanning to URLs matching this pattern.
  --no-color            Disable colored output in the console.
(.venv) PS C:\PROJECTS\PycharmProjects\WEB-SCANNER>
```

Рисунок 2.1 - Приклад команди CLI для сканера

Цей код забезпечує гнучкість у налаштуванні сканера, дозволяючи користувачу адаптувати його поведінку до конкретних потреб.

Другим компонентом є краулер, реалізований у файлі crawler.py. Він відповідає за обхід веб-сайту, використовуючи алгоритм пошуку в ширину (BFS). Краулер починає з початкового URL, заданого через аргумент --url, і обходить сайт до заданої глибини, збираючи всі доступні URL та форми. Клас Crawler ініціалізується з параметрами, такими як глибина обходу, затримка між запитами та User-Agent, і використовує допоміжний клас LinkAndFormExtractor для парсингу HTML. Метод run, що виконує обхід сайту наведено у додатку А, Лістинг 1.2.

Краулер також підтримує фільтрацію URL за допомогою параметрів --scope і --exclude, що дозволяє обмежити сканування певними патернами. Зібрані URL і форми передаються іншим модулям для подальшого аналізу. Приклад наведено на Рис. 2.2.

```
.venv) PS C:\PROJECTS\PycharmProjects\WEB-SCANNER> python main.py http://127.0.0.1:5000 --modules all --report txt --output report.html
INFO] Starting scan...
INFO] Crawler found 16 URLs:
INFO]  - http://127.0.0.1:5000/page1
INFO]  - http://127.0.0.1:5000/upload_form
INFO]  - http://127.0.0.1:5000/nosql_form
INFO]  - http://127.0.0.1:5000/catalog
INFO]  - http://127.0.0.1:5000/execute_form
INFO]  - http://127.0.0.1:5000/user_profile?user_id=1
INFO]  - http://127.0.0.1:5000/search_form
INFO]  - http://127.0.0.1:5000/page2
INFO]  - http://127.0.0.1:5000/
INFO]  - http://127.0.0.1:5000/contact
INFO]  - http://127.0.0.1:5000/read_file_form
INFO]  - http://127.0.0.1:5000/fetch_url_form
INFO]  - http://127.0.0.1:5000
INFO]  - http://127.0.0.1:5000/redirect_form
INFO]  - http://127.0.0.1:5000/profile
INFO]  - http://127.0.0.1:5000/login_form
INFO] Found 18 forms total.
INFO] FORM: method=post, action=http://127.0.0.1:5000/upload, inputs=[{'name': 'title', 'type': 'text', 'value': ''}, {'name': 'uploaded_file', 'type': 'file', 'value': ''}]
INFO] FORM: method=post, action=http://127.0.0.1:5000/login_check, inputs=[{'name': 'username', 'type': 'text', 'value': 'admin'}, {'name': 'password', 'type': 'password', 'value': ''}]
INFO] FORM: method=get, action=http://127.0.0.1:5000/nosql, inputs=[{'name': 'query', 'type': 'text', 'value': ''}]
INFO] FORM: method=get, action=http://127.0.0.1:5000/catalog/search, inputs=[{'name': 'q', 'type': 'text', 'value': 'None'}]
INFO] FORM: method=post, action=http://127.0.0.1:5000/catalog/delete/1, inputs=[]
INFO] FORM: method=post, action=http://127.0.0.1:5000/catalog/delete/2, inputs=[]
```

Рисунок 2.2 - Результат краулера

Третім компонентом є модуль відправки HTTP-запитів, реалізований у файлі requester.py. Він відповідає за виконання GET- і POST-запитів до веб-сайту з урахуванням налаштувань, заданих через CLI. Клас Requester ініціалізується з параметрами таймауту, затримки та User-Agent, а також відстежує кінцевий URL після перенаправлень через поле self.last_url. Це корисно для виявлення уразливостей типу Open Redirect. Приклад методу get наведено у додатку А, Лістинг 1.3.

Модуль забезпечує стабільну відправку запитів, обробляючи помилки та підтримуючи налаштування, що задаються користувачем, наприклад, затримку для уникнення перевантаження сервера.

Четвертим компонентом є модуль логування, реалізований у файлі logger.py. Він забезпечує гнучке управління виводом інформації в консоль із урахуванням режимів, заданих через аргументи --quiet, --verbose і --no-color. Клас Logger підтримує чотири рівні повідомлень: інформаційні (info), попередження (warn), помилки (error) і відладочні (debug). Приклад наведено на Рис. 2.3. Наприклад, метод info виводить повідомлення лише в нормальному або детальному режимі. Приклад наведено у додатку А, Лістинг 1.4.

```
[INFO] Loaded 68 SQLi payloads from C:\PROJECTS\PycharmProjects\WEB-SCANNER\data\payloads\sql_payloads.txt
[INFO] Loaded 68 SQLi payloads from C:\PROJECTS\PycharmProjects\WEB-SCANNER\data\payloads\sql_payloads.txt
[INFO] Loaded 68 SQLi payloads from C:\PROJECTS\PycharmProjects\WEB-SCANNER\data\payloads\sql_payloads.txt
[INFO] Loaded 6 XSS payloads from C:\PROJECTS\PycharmProjects\WEB-SCANNER\data\payloads\xss_payloads.txt
[INFO] Loaded 6 XSS payloads from C:\PROJECTS\PycharmProjects\WEB-SCANNER\data\payloads\xss_payloads.txt
[INFO] Loaded 6 XSS payloads from C:\PROJECTS\PycharmProjects\WEB-SCANNER\data\payloads\xss_payloads.txt
[INFO] Loaded 3 CSRF payloads from C:\PROJECTS\PycharmProjects\WEB-SCANNER\data\payloads\csrf_payloads.txt
[INFO] Loaded 5 NoSQL payloads from C:\PROJECTS\PycharmProjects\WEB-SCANNER\data\payloads\nosql_payloads.txt
[INFO] Loaded 5 NoSQL payloads from C:\PROJECTS\PycharmProjects\WEB-SCANNER\data\payloads\nosql_payloads.txt
[INFO] Loaded 10 RCE payloads from C:\PROJECTS\PycharmProjects\WEB-SCANNER\data\payloads\rce_payloads.txt
[INFO] Loaded 10 RCE payloads from C:\PROJECTS\PycharmProjects\WEB-SCANNER\data\payloads\rce_payloads.txt
[INFO] Found 119 vulnerabilities.
```

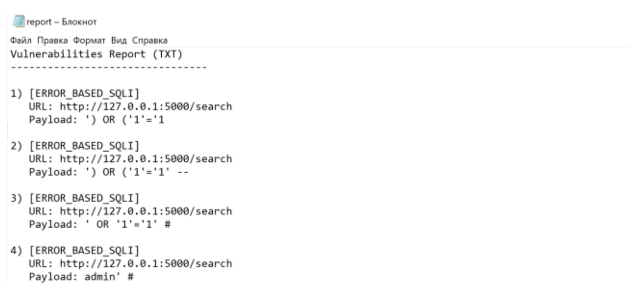
Рисунок 2.3 - Приклад команди CLI для сканера

Логування підтримує кольорове форматування через ANSI-коди, що полегшує сприйняття інформації, але може бути відключене для середовищ, які не підтримують кольори.

Останнім ключовим компонентом є модуль генерації звітів, реалізований у файлі report_generator.py. Він відповідає за збереження результатів сканування у форматах TXT, CSV або HTML, що задається через аргумент --report. Клас

ReportGenerator формує звіти на основі отриманих даних, забезпечуючи їх структурованість і безпеку виведення (наприклад, екранування HTML-символів у звіті HTML). Приклад методу _generate_txt наведено у додатку А, Лістинг 1.5.

Цей модуль забезпечує зручне представлення результатів, що дозволяє користувачу швидко аналізувати виявлені уразливості. Приклад наведено на Рис. 2.4, Рис. 2.5, та Рис. 2.6.



```
report – Блокнот
Файл Правка Формат Вид Справка
Vulnerabilities Report (TXT)
--------------------------------

1) [ERROR_BASED_SQLI]
   URL: http://127.0.0.1:5000/search
   Payload: ') OR ('1'='1

2) [ERROR_BASED_SQLI]
   URL: http://127.0.0.1:5000/search
   Payload: ') OR ('1'='1' --

3) [ERROR_BASED_SQLI]
   URL: http://127.0.0.1:5000/search
   Payload: ' OR '1'='1' #

4) [ERROR_BASED_SQLI]
   URL: http://127.0.0.1:5000/search
   Payload: admin' #
```

Рисунок 2.4 - Приклад звіту у txt

```
Module,Issue,URL,Payload
error_based_sqli,,http://127.0.0.1:5000/search,') OR ('1'='1
error_based_sqli,,http://127.0.0.1:5000/search,') OR ('1'='1' --
error_based_sqli,,http://127.0.0.1:5000/search,' OR '1'='1' #
error_based_sqli,,http://127.0.0.1:5000/search,admin' #
error_based_sqli,,http://127.0.0.1:5000/search,' UNION SELECT 1--
error_based_sqli,,http://127.0.0.1:5000/search,' UNION SELECT null--
error_based_sqli,,http://127.0.0.1:5000/search,"' UNION SELECT 1,2--"
error_based_sqli,,http://127.0.0.1:5000/search,"' UNION SELECT 1,2,3,4--"
error_based_sqli,,http://127.0.0.1:5000/search,"' UNION SELECT 1,2,3,4,5--"
error_based_sqli,,http://127.0.0.1:5000/search,' UNION SELECT @@version--
error_based_sqli,,http://127.0.0.1:5000/search,' UNION SELECT version()--
error_based_sqli,,http://127.0.0.1:5000/search,"') UNION SELECT 1,2,3--"
error_based_sqli,,http://127.0.0.1:5000/search,' AND 1=(SELECT COUNT(*) FROM tablename)--
error_based_sqli,,http://127.0.0.1:5000/search,"' AND substring(@@version,1,1) = 'X'"
error_based_sqli,,http://127.0.0.1:5000/search,""" AND substring(@@version,1,1) = 'X'"
error_basei EXEC xp_logininfo--
error_based_sqli,,http://127.0.0.1:5000/search,')#
error_based_sqli,,http://127.0.0.1:5000/search,')/*
```

Рисунок 2.5 - Приклад звіту у csv

Vulnerabilities Report

#	Module	Issue	URL	Payload
1	error_based_sqli		http://127.0.0.1:5000/search	') OR ('1'='1
2	error_based_sqli		http://127.0.0.1:5000/search	') OR ('1'='1' --
3	error_based_sqli		http://127.0.0.1:5000/search	' OR '1'='1' #
4	error_based_sqli		http://127.0.0.1:5000/search	admin' #
5	error_based_sqli		http://127.0.0.1:5000/search	' UNION SELECT 1--
6	error_based_sqli		http://127.0.0.1:5000/search	' UNION SELECT null--
7	error_based_sqli		http://127.0.0.1:5000/search	' UNION SELECT 1,2--
8	error_based_sqli		http://127.0.0.1:5000/search	' UNION SELECT 1,2,3,4--
9	error_based_sqli		http://127.0.0.1:5000/search	' UNION SELECT 1,2,3,4,5--
10	error_based_sqli		http://127.0.0.1:5000/search	' UNION SELECT @@version--
11	error_based_sqli		http://127.0.0.1:5000/search	' UNION SELECT version()--
12	error_based_sqli		http://127.0.0.1:5000/search	') UNION SELECT 1,2,3--
13	error_based_sqli		http://127.0.0.1:5000/search	' AND 1=(SELECT COUNT(*) FROM tablename)--
14	error_based_sqli		http://127.0.0.1:5000/search	' AND substring(@@version,1,1) = 'X'

Рисунок 2.6 - Приклад звіту у html

Усі компоненти взаємодіють через основний файл main.py, який ініціалізує їх, передає дані між модулями та координує процес сканування. Наприклад, краулер передає зібрані URL і форми модулям вразливостей, які використовують запитувач для відправки тестових запитів, а логер фіксує хід виконання. Результати сканування передаються генератору звітів для створення підсумкового файлу.

2.4. Модулі виявлення вразливостей

Модулі виявлення вразливостей є основою функціональності веб-сканера, забезпечуючи аналіз веб-сайтів на наявність слабких місць. Кожен модуль реалізовано як окремий клас, інтегрований через файл main.py за допомогою словника module_handlers, що дозволяє користувачу через CLI (аргумент --modules) вибирати потрібні типи перевірок. Модулі використовують об’єктно-орієнтований підхід, успадковуючи базову логіку від допоміжних класів, і працюють із набором навантажень, завантажених із відповідних файлів у директорії data/payloads/. У цій частині детально розглянуто модулі для SQL-ін’єкцій, XSS, CSRF і Directory Traversal, з акцентом на їхні технічні аспекти, алгоритми та реалізацію.

Модулі SQL-ін’єкцій представлені файлами sqli_helpers.py, error_based.py, boolean_based.py, time_based.py і blind_sql_injection.py. Вони базуються на класі SQLiScanner із sqli_helpers.py, який ініціалізується з об’єктами Requester і Logger, а також завантажує навантаження з файлу sql_payloads.txt, що містить 68 записів, таких як ' OR '1'='1, SLEEP(5), UNION SELECT 1,2,3--. Клас визначає методи scan_urls і scan_forms, які перевизначаються в дочірніх класах для специфічних типів атак. Модулі підтримують конфігурацію через параметри CLI, такі як --delay і --timeout, які впливають на поведінку запитів.

- Error-based SQL Injection (error_based.py): Модуль виявляє уразливості шляхом провокування помилок бази даних, аналізуючи відповіді сервера. Він використовує метод _contains_sql_error із sqli_helpers.py, який шукає сигнатури помилок, такі як "mysql_fetch", "ORA-", "SQL Server". Алгоритм ітерує параметри URL, замінюючи їх на навантаження (наприклад, ' AND

1=(SELECT COUNT(*) FROM tablename)--), і відправляє запит через Requester. Якщо відповідь містить помилку, результат фіксується. Приклад наведено у додатку А, Лістинг 1.6.

Обробка помилок у Requester (наприклад, HTTPError, URLError) дозволяє модулю продовжувати роботу навіть при часткових збоях.

- Boolean-based SQL Injection (boolean_based.py): Модуль тестує сліпі SQL-ін'єкції, порівнюючи відповіді на умови "істина" ('1'='1) і "хибність" ('1'='2). Він вимірює різницю в довжині відповідей із порогом, заданим у коді (наприклад, 10% від базової довжини), і враховує затримку через --delay. Якщо різниця перевищує поріг, уразливість фіксується. Метод також обробляє винятки, наприклад, коли сервер повертає порожню відповідь. Приклад наведено у додатку А, Лістинг 1.7.
- Time-based SQL Injection (time_based.py, blind_sql_injection.py): Ці модулі виявляють уразливості через затримки, викликані командами типу SLEEP(5) або WAITFOR DELAY '0:0:5'. Параметр delay_threshold (за замовчуванням 5 секунд) налаштовується в конструкторі, а точність вимірювань залежить від параметра --timeout. Модуль вимірює час виконання запиту через time.time() і фіксує уразливість при перевищенні порогу. Приклад наведено у додатку А, Лістинг 1.8.

Для форм модулі адаптують запити до методів GET або POST, враховуючи enctype і типи полів (text, search, password).

Модулі XSS представлені файлами xss_helpers.py, reflected.py, stored.py і dom_based.py. Вони базуються на класі XSSScanner, який завантажує 6 навантажень із xss_payloads.txt (наприклад, <script>alert(1)</script>, "><img src=x onerror=alert(3)>). Клас ініціалізується з Requester і Logger, а методи scan_urls і scan_forms адаптовані для різних типів XSS.

- Reflected XSS (reflected.py): Модуль перевіряє відображення навантаження в тілі відповіді без екранування. Він ітерує параметри

URL, підставляє навантаження і використовує строгу перевірку через in, що може генерувати хибнопозитивні результати, якщо сервер частково екранує дані. Приклад наведено у додатку А, Лістинг 1.9.

- Stored XSS (stored.py): Модуль відправляє навантаження через форми і перевіряє їхнє збереження, порівнюючи початковий стан сторінки з оновленим після відправки. Він використовує затримку через --delay для уникнення конфліктів із серверними кешами.
- DOM-based XSS (dom_based.py): Аналізує JavaScript-код у відповіді, шукаючи небезпечні функції (eval, document.write) із параметрами URL (наприклад, location.hash). Модуль враховує кодування через urllib.parse.quote.

Модуль CSRF реалізований у файлах csrf_helpers.py, csrf_scanner.py, token_analysis.py і form_detection.py. Клас BasicCSRFScanner успадковує CSRFScanner, завантажуючи 3 навантаження з csrf_payloads.txt, хоча вони використовуються опціонально. Модуль аналізує POST-форми з enctype="multipart/form-data" або без нього.

- Логіка роботи: Функція is_sensitive_form у form_detection.py перевіряє наявність ключових слів (delete, update) у action. find_csrf_token шукає токени за іменами (csrf_token, token), повертаючи None, якщо їх немає. Метод scan_forms відправляє запити без токена або з підробленим значенням. Приклад наведено у додатку А, Лістинг 1.10.

Метод _is_request_success аналізує відповідь, шукаючи ключові слова (error, forbidden) для виключення хибнопозитивних результатів.

Модуль реалізований у traversal_scanner.py і traversal_helpers.py. Клас DirectoryTraversalScanner генерує 48 навантажень через generate_traversal_payloads (наприклад, ../../../etc/passwd, ..\\..\\windows\\win.ini) і тестує їх на параметрах URL і формах.

- Логіка роботи: Модуль ітерує параметри, підставляє навантаження і аналізує відповіді через is_suspicious_response, яка шукає сигнатури (root:x:0:0, [extensions]). Код для URL наведен у додатку А, Лістинг 1.11.

Для форм модуль адаптує запити до GET або POST, враховуючи типи полів.

Модулі NoSQL-ін’єкцій представлені файлами nosql_helpers.py, simple_nosql.py і advanced_nosql.py. Вони базуються на класі NoSQLiScanner, який завантажує 5 навантажень із nosql_payloads.txt (наприклад, {"$gt": ""}, {"$where": "sleep(2000)"), адаптованих для баз NoSQL, таких як MongoDB.

- Simple NoSQL Injection (simple_nosql.py): Модуль тестує параметри URL і форми на помилки NoSQL, підставляючи навантаження типу {"$ne": null} і перевіряючи відповіді через _contains_nosql_error. Алгоритм розбирає query-параметри через urllib.parse, замінює їх і аналізує наявність сигнатур, таких як "MongoError". Приклад наведено у додатку А, Лістинг 1.12.
- Advanced NoSQL Injection (advanced_nosql.py): Модуль використовує time-based підхід, тестуючи затримки через навантаження {"$where": "sleep(2000)"}. Поріг затримки (delay_threshold, за замовчуванням 2 секунди) налаштовується в конструкторі, а точність залежить від –timeout. Приклад наведено у додатку А, Лістинг 1.13.

Модулі RCE реалізовані в rce_helpers.py, command_injection.py і code_injection.py. Вони використовують клас RCEScanner, завантажуючи 10 навантажень із rce_payloads.txt (наприклад, ; ls -la, "<?php system($_GET['cmd']); ?>").

- Command Injection (command_injection.py): Тестує введення команд через параметри, аналізуючи відповіді на сигнатури, такі як "whoami". Модуль враховує затримку через –delay. Приклад наведено у додатку А, Лістинг 1.14.
- Code Injection (code_injection.py): Перевіряє виконання коду, наприклад, PHP-скриптів, шукаючи відображення результатів.

Open Redirect реалізований у open_redirect_scanner.py і open_redirect_helpers.py. Клас OpenRedirectScanner тестує параметри типу next, url із навантаженнями з generate_open_redirect_payloads (наприклад, http://evil.com, //google.com/%2f%2fevil.com):

- Логіка роботи: Перевіряє self.requester.last_url на зовнішність через is_external_url. Приклад наведено у додатку А, Лістинг 1.15.
- IDOR реалізований у idor_scanner.py, idor_helpers.py, sequential_id_test.py і uuid_test.py. Клас IDORScanner аналізує параметри на тип (sequential, uuid) через looks_like_id.
- Sequential ID Test (sequential_id_test.py): Тестує числові ID (наприклад, 123), замінюючи їх на 122, 124, 9999, і перевіряє доступ через is_suspiciously_valid. Приклад наведено у додатку А, Лістинг 1.16.
- UUID Test (uuid_test.py): Генерує випадкові UUID і тестує доступ.

Модуль SSRF (Server-Side Request Forgery)

Модуль реалізований у ssrf_scanner.py і ssrf_helpers.py. Клас SSRFScanner тестує параметри типу url із навантаженнями, такими як http://127.0.0.1:80, аналізуючи відповіді на внутрішні ресурси. Приклад наведено у додатку А, Лістинг 1.17.

Модуль Insecure File Upload реалізований у upload_scanner.py і upload_helpers.py. Клас FileUploadScanner тестує форми з enctype="multipart/form-data", завантажуючи файли з generate_malicious_files (наприклад, shell.php). Приклад наведено у додатку А, Лістинг 1.18.

Модуль автентифікації реалізований у auth_scanner.py, auth_helpers.py, weak_passwords.py і default_credentials.py. Клас AuthScanner тестує форми логіну на слабкі паролі та дефолтні облікові дані. Приклад наведено у додатку А, Лістинг 1.19.

- Weak Passwords (weak_passwords.py): Перевіряє 10 слабких паролів (admin, 123456) для заданого імені користувача.
- Default Credentials (default_credentials.py): Тестує пари типу admin/admin.

2.5. Інтеграція та взаємодія компонентів

Інтеграція та взаємодія компонентів веб-сканера вразливостей є ключовим аспектом його функціональності, що забезпечує стабільну та ефективну роботу інструменту. Сканер працює через командний рядок (CLI), де користувач задає параметри через файл main.py, який виступає центральною точкою управління. Усі компоненти - обробка аргументів, краулер, запитувач, логування, генерація звітів і модулі вразливостей - взаємодіють у чітко визначеній послідовності, що дозволяє обробляти веб-сайти, виявляти уразливості та надавати результати у зрозумілому форматі. У цьому пункті детально описано схему роботи сканера, його етапи, технічні деталі інтеграції та приклади коду.

Сканер розпочинає роботу з ініціалізації через main.py, який парсить аргументи командного рядка за допомогою модуля cli_parser.py. Наприклад, команда python main.py http://127.0.0.1:5000 --modules all --report txt --output report.txt визначає цільовий URL, вибір усіх модулів вразливостей, формат звіту TXT і вихідний файл. Модуль cli_parser.py повертає об’єкт args, який містить параметри, такі як depth, delay, timeout, quiet, verbose і no_color. Ці параметри передаються до інших компонентів для налаштування їхньої поведінки. Фрагмент ініціалізації в main.py наведено у додатку А, Лістинг 1.20.

Першим етапом роботи є обхід веб-сайту за допомогою модуля crawler.py. Клас Crawler використовує алгоритм пошуку в ширину (BFS) для збору URL-адрес і форм, враховуючи глибину обходу (depth), затримку між запитами (delay) і фільтрацію через scope і exclude. Він повертає множину visited (URL-адреси) і список found_forms, які передаються до модулів вразливостей. Наприклад, для URL http://127.0.0.1:5000 краулер знаходить 16 URL і 18 форм, що фіксується в логері:

```
 [INFO] Crawler found 16 URLs:
[INFO]  - http://127.0.0.1:5000/page1
[INFO]  - http://127.0.0.1:5000/catalog
...
[INFO] Found 18 forms total.
```

Далі зібрані дані передаються до модулів вразливостей через словник module_handlers у main.py. Цей словник містить відображення назв модулів

(наприклад, error_based, reflected, csrf) на відповідні класи. Користувач може обрати конкретні модулі через аргумент --modules (наприклад, --modules error_based,time_based), або використати all для запуску всіх доступних перевірок. Ініціалізація модулів наведена у додатку А, Лістинг 1.21.

Кожен модуль вразливостей отримує об’єкти Requester і Logger, а також доступ до списку URL і форм. Модулі виконують методи scan_urls і scan_forms, відправляючи запити через requester.py. Клас Requester підтримує GET- і POST-запити з налаштуваннями таймауту (наприклад, 10 секунд за замовчуванням), затримки (наприклад, 0.5 секунди через --delay) і User-Agent (WebVulnScanner/1.0). Він також відстежує кінцевий URL після перенаправлень через self.last_url, що критично для модулів типу Open Redirect і SSRF. Метод post наведен у додатку А, Лістинг 1.22.

Модуль логування (logger.py) фіксує хід виконання, виводячи повідомлення залежно від режиму (quiet, verbose, no_color). Наприклад, при виявленні уразливості логер записує деталі через метод info, використовуючи кольорове форматування через ANSI-коди. Приклад наведено у додатку А, Лістинг 1.23.

Результати роботи модулів вразливостей (список словників із полями module, url, payload, issue тощо) передаються до модуля генерації звітів (report_generator.py). Клас ReportGenerator підтримує формати TXT, CSV і HTML, заданні через --report. Наприклад, для TXT-формату він записує структурований текст у файл, вказаний через –output. Приклад наведено у додатку А, Лістинг 1.24.

Загальна схема роботи сканера включає такі етапи:

1) Ініціалізація: Парсинг аргументів, створення об’єктів Logger, Requester, Crawler.
2) Обхід сайту: Збір URL і форм через Crawler.
3) Сканування вразливостей: Виконання модулів через module_handlers.
4) Логування: Фіксація процесу через Logger.
5) Генерація звіту: Збереження результатів через ReportGenerator.

Загальну схему роботи веб-сканера наведено на Рис. 2.7.

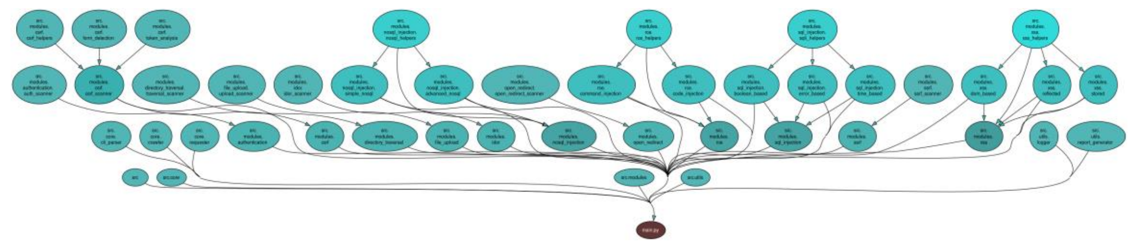


Рисунок 2.7 - Загальна схема роботи веб-сканера

Інтеграція компонентів забезпечується через передачу об’єктів і даних між ними. Наприклад, Crawler передає дані до модулів вразливостей через main.py, а Requester використовується як спільний інтерфейс для відправки запитів. Модульність дозволяє легко додавати нові модулі вразливостей, просто реєструючи їх у module_handlers.

3 СТВОРЕННЯ ТЕСТОВОГО ВЕБ-САЙТУ З УРАЗЛИВОСТЯМИ

3.1. Мета та завдання створення тестового веб-сайту

Створення тестового веб-сайту є важливим етапом даного дослідження, спрямованим на забезпечення контрольованого середовища для оцінки ефективності веб-сканера вразливостей. Тестовий сайт розроблено з метою імітації реального веб-додатку з навмисно вбудованими уразливостями, що дозволяє перевірити здатність сканера виявляти широкий спектр слабких місць, таких як SQL-ін’єкції, XSS, CSRF, NoSQL-ін’єкції, Directory Traversal, RCE, Open Redirect, IDOR, SSRF, Insecure File Upload та слабкі місця автентифікації. Основна мета полягає в тому, щоб створити платформу, яка відображає типові помилки розробки веб-додатків, забезпечуючи можливість практичного тестування та вдосконалення сканера.[15]

Тестовий сайт реалізовано за допомогою фреймворку Flask, який запускається локально на порту 5000 (http://127.0.0.1:5000). Він включає різноманітні маршрути та форми, що імітують функціональність типового веб-сайту: головну сторінку Рис. 3.1, каталог товарів, профіль користувача, контактну форму, а також спеціалізовані маршрути для демонстрації уразливостей. Структура сайту підтримується базою даних SQLite, ініціалізованою через модуль sql.test_db, який створює таблиці products (для каталогу) і users (для профілю), заповнюючи їх початковими даними. Наприклад, таблиця products містить поля id, name і description, а таблиця users - id і display_name, що використовуються для тестування SQL-ін’єкцій і CSRF.[16]

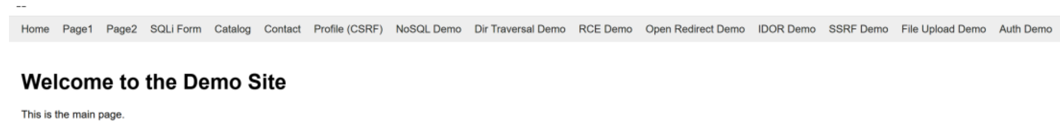


Рисунок 3.1 - Головна сторінка тестового сайту

Для створення тестового веб-сайту було визначено такі завдання:

- Розробити базову структуру веб-додатку з різноманітними маршрутами, включаючи статичні сторінки (/, /page1, /page2) та інтерактивні форми (/contact, /search_form, /catalog), щоб імітувати реальний сайт із типовими функціями.
- Навмисно впровадити уразливості в ключові маршрути, такі як /search (SQL-ін'єкція), /profile (CSRF), /nosql (NoSQL-ін'єкція), /read_file (Directory Traversal), /execute (RCE), /redirect_me (Open Redirect), /user_profile (IDOR), /fetch_url (SSRF), /upload (Insecure File Upload) і /login_check (слабка автентифікація), щоб сканер міг їх виявляти.
- Забезпечити простоту запуску та конфігурації сайту для тестування, використовуючи Flask із мінімальними залежностями та локальну базу даних SQLite, що ініціалізується командою python -c "import sql.test_db; sql.test_db.init_db()".
- Створити зрозумілий інтерфейс для демонстрації уразливостей, включаючи HTML-форми з поясненнями (наприклад, <p>Уязвимость: принимает слабые/дефолтные пароли без ограничений.</p> у /login_form), щоб полегшити аналіз результатів сканування.
- Забезпечити можливість емітації відповідей сервера на вразливі запити, наприклад, повернення псевдо-вмісту файлів (root:x:0:0 для /read_file) або затримки (time.sleep(2) для /nosql), що дозволяє тестувати як error-based, так і time-based методи сканера.

Тестовий сайт виконує роль контрольованого об'єкта для оцінки роботи сканера, дозволяючи перевірити його здатність виявляти уразливості в умовах, наближених до реальних. Наприклад, маршрут /search використовує непараметризований SQL-запит SELECT id, name, description FROM products WHERE name LIKE '%{query}%', який легко піддається SQL-ін'єкціям, а маршрут /upload зберігає файли без перевірки розширень, дозволяючи завантажувати шкідливі скрипти. Такі особливості забезпечують можливість тестування всіх модулів вразливостей, описаних у пункті 2.4. Приклад наведено на Рис. 3.2.

Рисунок 3.2 - Приклад форми для SQL-ін’єкції

Реалізація тестового сайту передбачає мінімальну складність для запуску: після ініціалізації бази даних через sql.test_db.init_db() сайт запускається командою python test_site.py, що робить його зручним для експериментів. Використання Flask дозволяє швидко розширювати функціональність, додаючи нові маршрути чи уразливості за потреби. Таким чином, тестовий сайт слугує як практична основа для оцінки ефективності сканера, забезпечуючи контрольоване середовище для аналізу його роботи та вдосконалення.

3.2. Вибір технологій для розробки

Розробка тестового веб-сайту для оцінки ефективності веб-сканера вразливостей вимагала вибору технологій, які б відповідали цілям дослідження, забезпечуючи простоту реалізації, гнучкість у впровадженні уразливостей та легкість налаштування. Основною технологією обрано Flask - легковаговий веб-фреймворк на мові Python. Як допоміжні інструменти використано SQLite для управління базою даних, а також стандартні бібліотеки Python, такі як os, sqlite3, urllib.parse та requests, для обробки файлів, запитів і логіки маршрутів. У цьому пункті обґрунтовано вибір Flask та супутніх технологій, враховуючи їхні переваги в контексті створення демонстраційного веб-сайту з навмисно вбудованими уразливостями.

Flask обрано як основний фреймворк завдяки його мінімалістичному дизайну та високій гнучкості, що ідеально відповідає потребам даного дослідження. На відміну від більш складних фреймворків, таких як Django, які за замовчуванням включають вбудовані механізми безпеки (наприклад, ORM для запобігання SQL-

ін'єкцій або автоматичну генерацію CSRF-токенів), Flask надає розробнику повний контроль над логікою додатку. Це дозволяє легко впроваджувати уразливості, такі як непараметризовані SQL-запити в маршруті /search (sql = f"SELECT id, name, description FROM products WHERE name LIKE '%{query}%'") чи відсутність валідації файлів у /upload (filename = file.filename; file.save(save_path)), без необхідності обходити захисні функції. Наприклад, у коді Flask не вбудовано автоматичного екранування введених даних у шаблонах, що спрощує створення XSS-уразливостей у /contact (return f"<h2>Thank you, {name}!</h2><p>Your message: {message}</p>"), де введені дані виводяться напряму.

Простота Flask також сприяє швидкому створенню маршрутів і форм, що є критично важливим для тестового сайту з численними уразливостями. У test_site.py визначено 18 основних маршрутів (наприклад, /, /search, /profile, /nosql, /upload), кожен із яких реалізує специфічну функцію та уразливість. Flask дозволяє визначати ці маршрути через декоратори (@app.route) з мінімальним кодом, наприклад:

Лістинг 3.1

```
@app.route("/search", methods=["GET", "POST"])
def search_products():
    query = request.args.get("q", "") if request.method == "GET" else request.form.get("q", "")
    sql = f"SELECT id, name, description FROM products WHERE name LIKE '%{query}%'"
    conn = test_db.get_db_connection()
    c = conn.cursor()
    c.execute(sql)
    rows = c.fetchall()
    conn.close()
    return render_template("search_results.html", query=query, results=rows)
```

Такий підхід забезпечує легкість додавання нових уразливостей, наприклад, NoSQL-ін'єкцій у /nosql чи RCE у /execute, без складної конфігурації, що робить Flask оптимальним вибором для демонстраційного додатку.

Використання SQLite як бази даних обґрунтовано її простотою та автономністю, що ідеально підходить для локального тестового середовища. У test_site.py SQLite застосовується для зберігання даних таблиць products (маршрути

/search, /catalog) і users (маршрут /profile), ініціалізованих через модуль sql.test_db. SQLite не потребує окремого серверного процесу, що спрощує запуск сайту командою python test_site.py після ініціалізації бази (python -c "import sql.test_db; sql.test_db.init_db()"). Її легка інтеграція з Python через модуль sqlite3 дозволяє швидко створювати вразливі SQL-запити, наприклад, у /search, де відсутність параметризації відкриває двері для SQL-ін'єкцій. Водночас SQLite достатньо функціональна для імітації реальних сценаріїв, таких як робота з каталогом товарів у /catalog.

Додаткові бібліотеки Python, такі як os і requests, використані для специфічних функцій. Модуль os застосовується в маршруті /upload для збереження файлів (os.path.join("uploads", filename)), що демонструє уразливість Insecure File Upload через відсутність валідації. Модуль requests у /fetch_url дозволяє емітувати SSRF, виконуючи запити до введених URL без обмежень (r = requests.get(target_url, timeout=5)), що полегшує тестування серверних запитів. Бібліотека urllib.parse використовується для обробки URL у маршрутах, таких як /redirect_me, де параметр next перенаправляє користувача без перевірки, демонструючи Open Redirect.

Вибір Flask і SQLite також обумовлений їхньою сумісністю з Python, який є основною мовою розробки веб-сканера, описаного в розділі II. Це забезпечує єдність технологічного стеку між сайтом і сканером, спрощуючи інтеграцію та тестування. Наприклад, сканер використовує urllib.request для запитів, що відповідає логіці обробки HTTP у Flask, а модульність Flask полегшує створення маршрутів, які сканер може аналізувати через Crawler. Приклад наведено на Рис. 3.3.

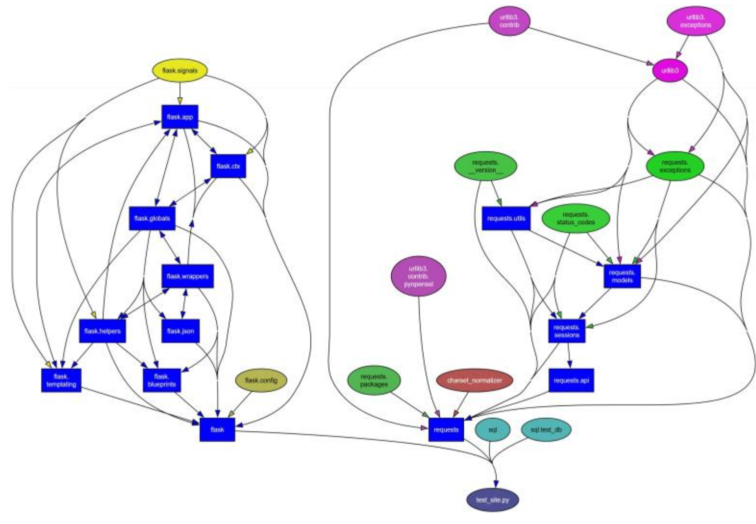


Рисунок 3.3 - Діаграма сайту Flask

Переваги Flask у порівнянні з альтернативами, такими як Django чи FastAPI, полягають у його легкості та відсутності надлишкових функцій, які могли б ускладнити впровадження уразливостей. Django, наприклад, автоматично захищає від SQL-ін’єкцій через ORM, що вимагало б додаткових зусиль для створення вразливого коду, тоді як Flask дозволяє залишити ці аспекти на розсуд розробника. FastAPI, хоча й швидший для API, менш зручний для роботи з HTML-шаблонами, які потрібні для демонстрації форм у /search_form, /profile тощо.

Таким чином, вибір Flask як основної технології для тестового сайту обґрунтований його простотою, гнучкістю та можливістю легко впроваджувати уразливості, що відповідає меті створення контрольованого середовища для тестування сканера. SQLite доповнює Flask як легка база даних, а стандартні бібліотеки Python забезпечують необхідну функціональність для реалізації уразливостей, таких як SSRF чи Insecure File Upload, що робить цей технологічний стек оптимальним для даного дослідження.

3.3. Структура тестового веб-сайту

Структура тестового веб-сайту є ключовим елементом для розуміння його функціональності та вразливостей, які були навмисно впроваджені для оцінки

ефективності веб-сканера. Сайт побудовано з використанням фреймворку Flask і складається з основного файлу Python, який визначає маршрути та логіку, а також допоміжних файлів і директорій для шаблонів, статичних ресурсів і бази даних. У цьому пункті детально описано структуру сайту, включаючи призначення кожного файлу, їхню роль у реалізації уразливостей і взаємодію між компонентами, що забезпечує цілісність демонстраційного додатку.[17]

Основним файлом є test_site.py, який слугує точкою входу для запуску веб-сайту та містить усю серверну логіку. Цей файл визначає 18 маршрутів, кожен із яких відповідає за конкретну функцію або демонстрацію уразливості. Він імпортує Flask для створення додатку (app = Flask(__name__, template_folder="templates", static_folder="static")), а також модулі os, sqlite3, requests і urllib.parse для обробки файлів, бази даних і HTTP-запитів. У test_site.py реалізовано маршрути, такі як / (головна сторінка), /search (SQL-ін’єкція), /profile (CSRF), /nosql (NoSQL-ін’єкція), /read_file (Directory Traversal), /execute (RCE), /redirect_me (Open Redirect), /user_profile (IDOR), /fetch_url (SSRF), /upload (Insecure File Upload) і /login_check (слабка автентифікація). Наприклад, маршрут /search містить вразливий SQL-запит:

Лістинг 3.2

```
sql = f"SELECT id, name, description FROM products WHERE name LIKE '%{query}%'"
c.execute(sql)
```

Цей файл також запускає сервер Flask через app.run(port=5000, debug=True), що дозволяє запускати сайт локально на http://127.0.0.1:5000.

Директорія templates містить HTML-шаблони, які формують інтерфейс користувача та забезпечують доступ до вразливих форм. Усі шаблони успадковуються від базового файлу base.html, який визначає загальну структуру сторінок із навігаційним меню для переходу між маршрутами. Приклад наведено на Рис. 3.4.

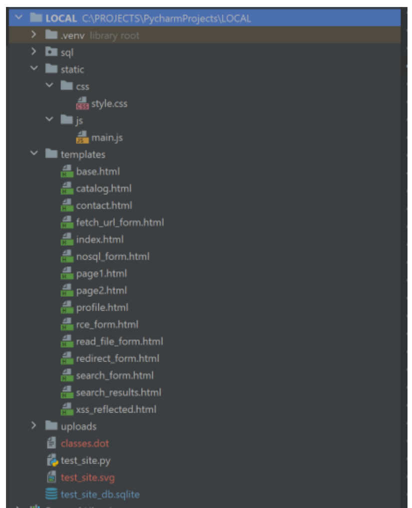


Рисунок 3.4 - Структура тестового сайту у Flask

Основні файли в цій директорії включають:

- index.html: Шаблон головної сторінки (/), що відображає базовий вміст і посилання на інші маршрути.
- search_form.html: Форма для маршруту /search_form, що дозволяє вводити пошукові запити, вразливі до SQL-ін’єкцій.
- contact.html: Форма для маршруту /contact, вразлива до XSS через виведення даних без екранування.
- catalog.html: Шаблон для маршруту /catalog, що відображає список продуктів і форми для додавання та видалення записів.
- profile.html: Форма для маршруту /profile, вразлива до CSRF через відсутність токенів.
- nosql_form.html: Форма для маршруту /nosql_form, що тестує NoSQL-ін’єкції через введення псевдо-JSON-запитів.
- read_file_form.html: Форма для маршруту /read_file_form, вразлива до Directory Traversal.
- execute_form.html: Форма для маршруту /execute_form, що демонструє RCE.
- redirect_form.html: Форма для маршруту /redirect_form, вразлива до Open Redirect.

- fetch_url_form.html: Форма для маршруту /fetch_url_form, що тестує SSRF.
- upload_form.html: Форма для маршруту /upload_form, вразлива до Insecure File Upload через завантаження файлів без перевірки.
- login_form.html: Форма для маршруту /login_form, що демонструє слабку автентифікацію.

Кожен шаблон містить HTML-код із поясненнями про уразливості (наприклад, <p>Уязвимость: принимает слабые/дефолтные пароли без ограничений.</p> у login_form.html), що полегшує тестування та аналіз.

Директорія static призначена для зберігання статичних файлів, таких як CSS і JavaScript, які визначають зовнішній вигляд і поведінку сторінок. У test_site.py вона задається через параметр static_folder="static". Основні файли включають:

- style.css: Визначає стилі для всіх шаблонів, наприклад, форматування форм і навігаційного меню.
- main.js: Містить клієнтський JavaScript-код, який може бути використаний для інтерактивності, хоча в даному випадку його роль мінімальна, оскільки уразливості зосереджені на серверній логіці.

Файл sql/test_db.py відповідає за ініціалізацію та управління базою даних SQLite (test_site.db.sqlite), яка використовується для зберігання даних сайту. Цей модуль створює дві таблиці:

- products: Містить поля id (ціле число, первинний ключ), name (текст) і description (текст), що використовуються в маршрутах /search і /catalog. Початкові дані включають кілька записів про продукти для тестування SQL-ін’єкцій.
- users: Містить поля id (ціле число, первинний ключ) і display_name (текст), що використовуються в маршруті /profile для демонстрації CSRF і в /user_profile для IDOR.

Функція init_db() у sql/test_db.py ініціалізує базу даних перед запуском сайту командою python -c "import sql.test_db; sql.test_db.init_db()". Модуль також надає функцію get_db_connection(), яка повертає підключення до бази (conn =

sqlite3.connect("test_site.db.sqlite")) для використання в test_site.py. Приклад наведено на Рис. 3.5.

Имя	Тип	Схема
Таблицы (3)		
products		CREATE TABLE products (id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT NOT NULL, description TEXT NOT NULL)
id	INTEGER	"id" INTEGER
name	TEXT	"name" TEXT NOT NULL
description	TEXT	"description" TEXT NOT NULL
sqlite_sequence		CREATE TABLE sqlite_sequence(name,seq)
name		"name"
seq		"seq"
users		CREATE TABLE users (id INTEGER PRIMARY KEY AUTOINCREMENT, username TEXT NOT NULL, display_name TEXT)
id	INTEGER	"id" INTEGER
username	TEXT	"username" TEXT NOT NULL
display_na...	TEXT	"display_name" TEXT
Индексы (0)		
Представления (0)		
Триггеры (0)		

Рисунок 3.5 - База даних тестового сайта

Директорія uploads створюється динамічно маршрутом /upload для збереження завантажених файлів (os.path.join("uploads", filename)). Вона не є частиною початкової структури, але з'являється під час тестування Insecure File Upload, коли користувач завантажує файли, наприклад, shell.php. Її призначення - демонструвати уразливість через відсутність валідації файлів.

Взаємодія між файлами забезпечується через Flask. Наприклад, маршрут /search у test_site.py викликає test_db.get_db_connection() для доступу до бази products, виконує SQL-запит і передає результати до search_results.html через render_template. Аналогічно, маршрут /upload використовує os для збереження файлів у uploads, а /profile оновлює таблицю users у базі SQLite. Шаблони з templates відображають дані, отримані від маршрутів, а static забезпечує стилізацію.

Структура сайту відображає типовий веб-додаток із вразливостями, що дозволяє сканеру аналізувати різні точки входу. Наприклад, форми в templates надають параметри для тестування (наприклад, q у /search_form), а маршрути в test_site.py обробляють ці дані без належного захисту, що відповідає меті створення контрольованого середовища для тестування.

3.4. Функціональність веб-сайту

Функціональність тестового веб-сайту є основою для демонстрації його можливостей і вразливостей, що дозволяє оцінити ефективність веб-сканера в

виявленні слабких місць. Сайт побудовано з використанням фреймворку Flask і включає 18 маршрутів, кожен із яких реалізує певну функцію або імітує типову уразливість, таку як SQL-ін’єкції, XSS, CSRF, NoSQL-ін’єкції, Directory Traversal, RCE, Open Redirect, IDOR, SSRF, Insecure File Upload і слабка автентифікація. У цьому пункті детально описано маршрути, їхню логіку, призначення та технічну реалізацію, що забезпечує повне розуміння роботи сайту та його вразливостей.[18]

Основні сторінки

- / (Головна сторінка):
  - Логіка: Маршрут повертає шаблон index.html, який відображає базовий вміст і навігаційне меню з посиланнями на інші маршрути. Код: return render_template("index.html"). Приклад наведено на Рис. 3.6.
  - Призначення: Забезпечує точку входу до сайту та доступ до всіх функцій через меню. Не містить уразливостей, слугуючи лише як оглядова сторінка.

Рисунок 3.6 - Вигляд /(Головна сторінка)

- /page1 і /page2:
  - Логіка: Повертають шаблони page1.html і page2.html відповідно (return render_template("page1.html"), return render_template("page2.html")).
  - Призначення: Імітують статичні сторінки сайту для створення реалістичної структури, без вразливостей. Приклад наведено на Рис. 3.7.
- /secret_page:

- Логіка: Повертає повідомлення про помилку 404 (return "<h1>404 Not Found</h1><p>This page doesn't exist.</p>", 404).
- Призначення: Демонструє приховану сторінку, яка може бути виявлена сканером через обхід посилань, але не містить активних уразливостей.

Рисунок 3.7 - Вигляд /page1

Контактна форма

- /contact (GET і POST):
  - Логіка: Для GET повертає contact.html з формою введення імені та повідомлення (return render_template("contact.html")). Для POST отримує дані (name = request.form.get("name", ""), message = request.form.get("message", "")) і повертає їх без екранування: return f"<h2>Thank you, {name}!</h2><p>Your message: {message}</p>".
  - Призначення: Імітує контактну форму, вразливу до XSS через відсутність екранування введених даних, наприклад, <script>alert(1)</script>. Приклад наведено на Рис. 3.8.

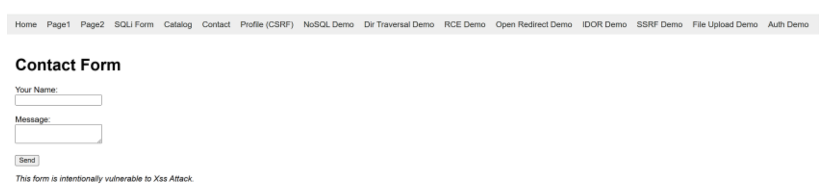


Рисунок 3.8 - Вигляд /contact

Пошук і SQL-ін’єкція

- /search_form (GET):

- Логіка: Повертає шаблон search_form.html з формою для введення пошукового запиту (return render_template("search_form.html")). Приклад наведено на Рис. 3.9.
- Призначення: Забезпечує інтерфейс для введення даних, які обробляються маршрутом /search.

- /search (GET і POST):

- Логіка: Отримує параметр q з GET (request.args.get("q", "")) або POST (request.form.get("q", "")), формує вразливий SQL-запит (sql = f"SELECT id, name, description FROM products WHERE name LIKE '%{query}%'"), виконує його через sqlite3 і повертає результати в search_results.html. Обробляє помилки через try-except, повертаючи текст помилки при невдачі.
- Призначення: Демонструє SQL-ін’єкцію через непараметризований запит, наприклад, ' OR '1'='1 повертає всі записи з таблиці products.

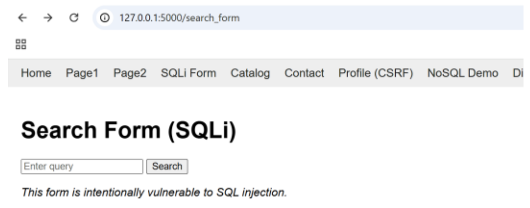


Рисунок 3.9 - Вигляд /search_form

Каталог

- /catalog (GET і POST):

- Логіка: Для GET повертає список продуктів із таблиці products у catalog.html (c.execute("SELECT id, name, description FROM products")). Для POST додає новий продукт (c.execute("INSERT INTO products(name, description) VALUES (?,?)", (name, desc))) і перенаправляє на /catalog.
- Призначення: Імітує каталог із базовою CRUD-функціональністю, без явних уразливостей у цьому маршруті.

- /catalog/search (GET):

- o Логіка: Виконує безпечний запит із параметром q (c.execute("SELECT id, name, description FROM products WHERE name LIKE ?", (f"%{q}%",))) і повертає результати в catalog.html.
- o Призначення: Показує пошук у каталозі, без уразливостей завдяки параметризації. Приклад наведено на Рис. 3.10.
- /catalog/delete/<int:item_id> (POST):
- o Логіка: Видаляє продукт за item_id (c.execute("DELETE FROM products WHERE id=?", (item_id,))) і перенаправляє на /catalog.
- o Призначення: Демонструє видалення записів, потенційно вразливе до CSRF через відсутність токенів, якщо розглядати в контексті ширшого тестування.

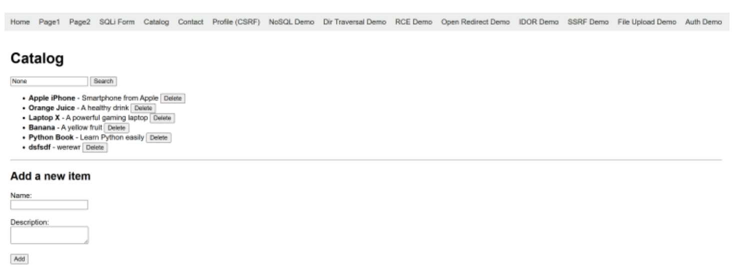


Рисунок 3.10 - Вигляд /catalog

Профіль і CSRF

- /profile (GET і POST):
- o Логіка: Для GET повертає profile.html із поточним ім’ям користувача (c.execute("SELECT display_name FROM users WHERE id=?", (user_id,))). Для POST оновлює display_name у таблиці users (c.execute("UPDATE users SET display_name=? WHERE id=?", (new_name, user_id))) без перевірки токенів. Приклад наведено на Рис. 3.11.
- o Призначення: Імітує профіль користувача, вразливий до CSRF через відсутність захисту від фальшивих запитів.

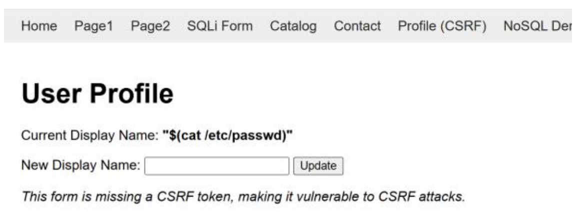


Рисунок 3.11 - Вигляд /profile

NoSQL-ін’єкція

- /nosql_form (GET):
  - Логіка: Повертає nosql_form.html із формою для введення псевдо-JSON-запиту (return render_template("nosql_form.html")). Приклад наведено на Рис. 3.12.
  - Призначення: Забезпечує інтерфейс для тестування NoSQL-ін’єкцій у /nosql.
- /nosql (GET):
  - Логіка: Отримує параметр query (query_str = request.args.get("query", "")), перевіряє наявність операторів NoSQL ("$gt", "$where") і повертає емітацію помилки або затримку (time.sleep(2) для "$where").
  - Призначення: Демонструє NoSQL-ін’єкцію, наприклад, {"$gt": ""} викликає "помилку", а {"$where": "sleep(2000)"} - затримку.

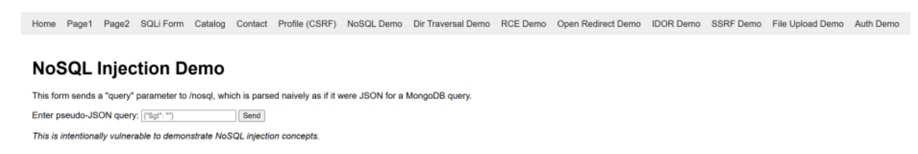


Рисунок 3.12 - Вигляд /nosql_form

Directory Traversal

- /read_file_form (GET):
  - Логіка: Повертає read_file_form.html із формою для введення шляху (return render_template("read_file_form.html")). Приклад наведено на Рис. 3.13.

  - Призначення: Забезпечує інтерфейс для тестування Directory Traversal у /read_file.
- /read_file (GET):
  - Логіка: Отримує параметр path (path = request.args.get("path", "")) і повертає псевдо-вміст файлів (наприклад, root:x:0:0 для etc/passwd) без валідації шляху.
  - Призначення: Імітує Directory Traversal, дозволяючи доступ до "системних" файлів через введення ../../etc/passwd.

Рисунок 3.13 - Вигляд /read_file_form

RCE (віддалене виконання коду)

- /execute_form (GET):
  - Логіка: Повертає execute_form.html із формою для введення команди (return render_template("rce_form.html")). Приклад наведено на Рис. 3.14.
  - Призначення: Забезпечує інтерфейс для тестування RCE у /execute.
- /execute (GET):
  - Логіка: Отримує параметр cmd (cmd = request.args.get("cmd", "")) і повертає псевдо-вивід для команд типу ls, whoami без валідації.
  - Призначення: Демонструє RCE, наприклад, ; ls -la повертає "список файлів".

Home Page1 Page2 SQLi Form Catalog Contact Profile (CSRF) NoSQL Demo Dir Traversal Demo RCE Demo Open Redirect Demo IDOR Dem

Remote Code Execution (RCE) Demo

Enter a command (e.g. ls, cat /etc/passwd), and we will "execute" it.

Command: ls -la Run

This page is intentionally vulnerable to demonstrate RCE/Command Injection.

Рисунок 3.14 - Вигляд /execute_form

Open Redirect

- /redirect_form (GET):
  - Логіка: Повертає redirect_form.html із формою для введення URL (return render_template("redirect_form.html")). Приклад наведено на Рис. 3.15.
  - Призначення: Забезпечує інтерфейс для тестування Open Redirect у /redirect_me.
- /redirect_me (GET):
  - Логіка: Отримує параметр next (next_url = request.args.get("next", "")) і перенаправляє на нього без перевірки (return redirect(next_url)).
  - Призначення: Імітує Open Redirect, наприклад, http://malicious-site.com перенаправляє користувача на зовнішній сайт.

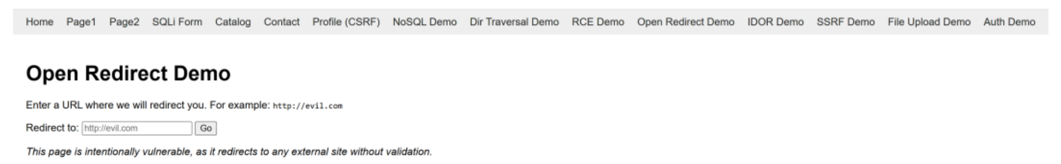


Рисунок 3.15 - Вигляд /redirect_form

IDOR (Insecure Direct Object Reference)

- /user_profile (GET):
  - Логіка: Отримує параметр user_id (user_id = request.args.get("user_id", "")) і повертає псевдо-профіль без перевірки прав (f"<p>Email: user{user_id}@test.local</p>"). Приклад наведено на Рис. 3.16.
  - Призначення: Демонструє IDOR, наприклад, зміна user_id=1 на user_id=2 дозволяє доступ до чужих даних.

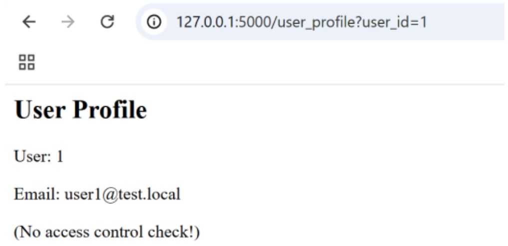


Рисунок 3.16 - Вигляд /user_profile?user_id=1

SSRF (Server-Side Request Forgery)

- /fetch_url_form (GET):
  - Логіка: Повертає fetch_url_form.html із формою для введення URL (return render_template("fetch_url_form.html")). Приклад наведено на Рис. 3.17.
  - Призначення: Забезпечує інтерфейс для тестування SSRF у /fetch_url.
- /fetch_url (GET):
  - Логіка: Отримує параметр url (target_url = request.args.get("url", "")) і виконує запит через requests.get() без валідації, повертаючи псевдо-дані для 127.0.0.1.
  - Призначення: Імітує SSRF, наприклад, http://127.0.0.1 повертає "внутрішні" дані.

Рисунок 3.17 - Вигляд /fetch_url_form

Insecure File Upload

- /upload_form (GET):
  - Логіка: Повертає HTML-форму з enctype="multipart/form-data" для завантаження файлів. Приклад наведено на Рис. 3.18.
  - Призначення: Забезпечує інтерфейс для тестування Insecure File Upload у /upload.

- /upload (POST):
  - Логіка: Отримує файл (file = request.files.get("uploaded_file")) і зберігає його в uploads без перевірки (file.save(os.path.join("uploads", filename))).
  - Призначення: Демонструє Insecure File Upload, наприклад, завантаження shell.php дозволяє потенційне виконання коду.

Рисунок 3.18 - Вигляд /upload_form

Слабка автентифікація

- /login_form (GET):

  - Логіка: Повертає HTML-форму з полями username і password (value="admin" за замовчуванням). Приклад наведено на Рис. 3.19.
  - Призначення: Забезпечує інтерфейс для тестування автентифікації у /login_check.

- /login_check (POST):

  - Логіка: Перевіряє username="admin" і пароль із списку слабких паролів (allowed_passwords = ["admin", "password", "123456", "pass", "qwerty"]) без обмежень.
  - Призначення: Імітує слабку автентифікацію, наприклад, admin/admin дозволяє вхід.

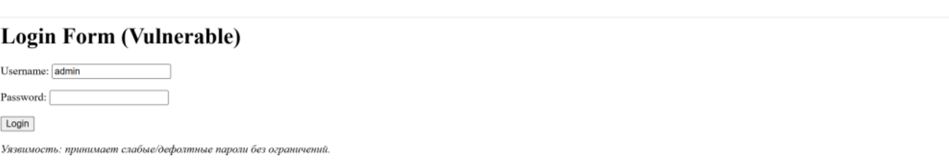


Рисунок 3.19 - Вигляд /login_form

3.5. Перелік вбудованих уразливостей

Тестовий веб-сайт створено з навмисно вбудованими уразливостями для оцінки можливостей веб-сканера в виявленні слабких місць, які відображають типові помилки розробки веб-додатків. Ці уразливості охоплюють ключові категорії, визначені в OWASP Top 10, такі як SQL-ін’єкції, міжсайтові сценарії, перехресні запити із підробленням сайту, NoSQL-ін’єкції, віддалене виконання коду, Directory Traversal, Insecure Direct Object Reference, Open Redirect, Server-Side Request Forgery, Insecure File Upload та слабка автентифікація. У цьому пункті детально проаналізовано причини виникнення кожної уразливості в коді test_site.py і їхній потенційний вплив на безпеку сайту, що дозволяє оцінити критичність цих слабких місць та необхідність їх виявлення сканером.[19]

SQL-ін’єкція виявляється в маршруті /search. Її причина полягає у використанні непараметризованого SQL-запиту, де параметр query, отриманий із request.args.get("q", "") або request.form.get("q", ""), підставляється напряму в рядок sql = f"SELECT id, name, description FROM products WHERE name LIKE '%{query}%'" без будь-якої валідації чи екранування. Це дозволяє зловмиснику ввести payload типу ' OR '1'='1, який поверне всі записи з таблиці products, або '; DROP TABLE products; --, що може знищити таблицю. Потенційний вплив цієї уразливості є надзвичайно високим, оскільки вона відкриває доступ до бази даних, дозволяючи витік конфіденційних даних, їх зміну чи видалення, що може повністю скомпрометувати сайт.

Міжсайтові сценарії проявляються в маршруті /contact. Причиною є відсутність екранування введених даних у відповіді сервера, де параметри name і message, отримані через request.form.get(), вставляються в HTML без обробки: return f"<h2>Thank you, {name}!</h2><p>Your message: {message}</p>". Введення <script>alert('XSS')</script> у поле форми призводить до виконання JavaScript у браузері користувача, що може бути використано для крадіжки сесій, фішингу чи інших шкідливих дій від імені користувача. Вплив цієї уразливості середній, але він стає критичним для сайтів із автентифікацією чи конфіденційними даними, де компрометація сесії може призвести до серйозних наслідків.

Перехресні запити із підробленням сайту реалізовані в маршруті /profile. Їхня причина полягає у відсутності CSRF-токенів у формі та будь-якої перевірки легітимності POST-запиту, де new_name = request.form.get("new_display_name", "") оновлює таблицю users без додаткових умов. Зловмисник може створити фальшивий запит, наприклад, через HTML-форму <form method="POST" action="http://127.0.0.1:5000/profile"><input name="new_display_name" value="Hacked"></form>, що змінить дані профілю без згоди користувача. Потенційний вплив середній, але може зростати для сайтів із важливими операціями, такими як фінансові транзакції, де несанкціоновані дії мають значні наслідки.

NoSQL-ін’єкція присутня в маршруті /nosql. Її причина - відсутність валідації параметра query_str, отриманого через request.args.get("query", ""), що дозволяє вводити псевдо-JSON-запити типу {"$gt": ""} або {"$where": "sleep(2000)"}. Логіка маршруту емітує помилки або затримки без фільтрації операторів NoSQL, таких як $gt чи $where. У реальному сценарії це може призвести до витоку всіх даних або DoS-атаки через затримку виконання. Вплив високий для сайтів із NoSQL-базами, оскільки зловмисник може маніпулювати даними чи порушити доступність сервісу.

Віддалене виконання коду втілено в маршруті /execute. Причиною є повна відсутність валідації параметра cmd, отриманого через request.args.get("cmd", ""), що дозволяє вводити системні команди типу ; ls -la або whoami. Хоча код емітує псевдо-вивід, у реальному додатку це могло б виконати довільний код на сервері, наприклад, видалити файли чи запустити шкідливі процеси. Потенційний вплив критичний, оскільки зловмисник може отримати повний контроль над сервером, що робить цю уразливість однією з найнебезпечніших.

Directory Traversal реалізовано в маршруті /read_file. Причина полягає у відсутності нормалізації чи валідації параметра path, отриманого через request.args.get("path", ""), що дозволяє вводити шляхи типу ../../etc/passwd. Логіка маршруту повертає псевдо-вміст "системних" файлів без обмежень доступу. У реальному сценарії це дало б змогу зловмиснику отримати конфіденційні системні

файли, такі як /etc/passwd, що містять дані про користувачів. Вплив високий через витік системної інформації, який може бути використаний для подальших атак.

Insecure Direct Object Reference проявляється в маршруті /user_profile. Причина - відсутність перевірки прав доступу для параметра user_id, отриманого через request.args.get("user_id", ""), що дозволяє вводити будь-яке числове значення для доступу до профілів інших користувачів. Зміна user_id=1 на user_id=2 відкриває чужий профіль без авторизації. Вплив середній, але може стати критичним для сайтів із конфіденційними даними, де витік інформації користувачів порушує приватність.

Open Redirect втілено в маршруті /redirect_me. Причина полягає у відсутності валідації параметра next_url, отриманого через request.args.get("next", ""), і прямому перенаправленні через return redirect(next_url). Введення http://malicious-site.com перенаправляє користувача на зовнішній сайт без обмежень. Вплив середній, але небезпечний у контексті фішингу чи втрати довіри користувачів, особливо якщо сайт видається надійним.

Server-Side Request Forgery реалізовано в маршруті /fetch_url. Причина - виконання запиту до target_url, отриманого через request.args.get("url", ""), за допомогою requests.get(target_url, timeout=5) без будь-яких обмежень на цільові адреси. Введення http://127.0.0.1 повертає псевдо-дані внутрішнього ресурсу. У реальному додатку це дозволило б доступ до внутрішніх систем чи метаданих сервера. Вплив високий через можливість компрометації інфраструктури або витоку внутрішньої інформації.

Insecure File Upload присутній у маршруті /upload. Причина полягає у збереженні файлів без перевірки типу чи розширення: filename = file.filename і file.save(os.path.join("uploads", filename)) дозволяють завантажувати будь-які файли, наприклад, shell.php. У реальному сценарії це призвело б до виконання коду на сервері, якщо сервер підтримує PHP чи інші скриптові мови. Вплив критичний, оскільки зловмисник може отримати повний контроль над системою через завантаження та виконання шкідливих файлів.

Слабка автентифікація реалізована в маршруті /login_check. Причина - перевірка пароля з обмеженого списку слабких значень (allowed_passwords = ["admin", "password", "123456", "pass", "qwerty"]) без обмежень на кількість спроб чи вимог до складності: if pw.lower() in allowed_passwords:. Введення admin/admin дозволяє доступ до адмін-зони. Вплив середній, але може стати критичним для сайтів із захищеними функціями, де легкий підбір пароля відкриває доступ до адмін-панелі. Приклад уразливостей наведено на Таб. 3.1.

Таблиця 3.1 - Таблиця уразливостей

Уразливість	Маршрут	Причина в коді	Потенційний вплив
SQLi	/search	Непараметризований SQL-запит	Витік/зміна даних бази
XSS	/contact	Відсутність екранування	Виконання коду в браузері
CSRF	/profile	Відсутність CSRF-токенів	Несанкціоновані дії користувача
NoSQLi	/nosql	Відсутність валідації JSON-запитів	Витік даних, DoS
RCE	/execute	Відсутність валідації команд	Повний контроль над сервером
Directory Traversal	/read_file	Відсутність нормалізації шляху	Витік системних файлів
IDOR	/user_profile	Відсутність перевірки прав	Витік даних інших користувачів
Open Redirect	/redirect_me	Відсутність валідації URL	Фішинг, перенаправлення
SSRF	/fetch_url	Відсутність обмеження URL	Доступ до внутрішніх ресурсів
Insecure File Upload	/upload	Відсутність перевірки файлів	Виконання коду на сервері
Слабка автентифікація	/login_check	Слабкі паролі без обмежень	Несанкціонований доступ до адмін-зони

4 ТЕСТУВАННЯ ВЕБ-СКАНЕРА ВРАЗЛИВОСТЕЙ

4.1. Результати тестування

Тестування веб-сканера вразливостей на спеціально створеному тестовому сайті є ключовим етапом для оцінки його здатності виявляти слабкі місця, описані в пункті 3.5. Сканер запускався через командний рядок (CLI) із командою python main.py http://127.0.0.1:5000 --modules all --report txt --output report.txt, що вказує цільовий URL, вибір усіх модулів вразливостей, формат звіту TXT і вихідний файл. Процес тестування охопив обхід сайту модулем Crawler, відправку запитів через Requester, аналіз відповідей модулями вразливостей і генерацію звіту через ReportGenerator. У результаті сканер виявив 119 уразливостей, що підтверджує його ефективність у виявленні всіх типів слабких місць, реалізованих у test_site.py. Приклад наведено на Рис. 4.1.[20]

Сканування розпочалося з ініціалізації компонентів у main.py. Модуль cli_parser.py обробив аргументи, створивши об’єкт args із параметрами: url="http://127.0.0.1:5000", modules="all", report="html", а також значеннями за замовчуванням, такими як depth=1, timeout=10, delay=0. Далі Crawler обійшов сайт, використовуючи алгоритм пошуку в ширину (BFS), і виявив 16 унікальних URL і 18 форм, що фіксується в консольному виводі:

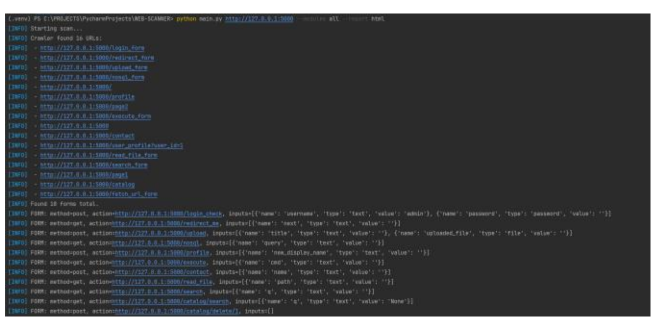


Рисунок 4.1 - Консольний вивід кравлера

Модуль Crawler передав зібрані дані (множину visited і список found_forms) до модулів вразливостей, які ініціалізовані через module_handlers у main.py. Кожен модуль отримав об’єкти Requester і Logger для відправки запитів і фіксації

результатів. Наприклад, для SQL-ін’єкцій модуль error_based.py завантажив 68 навантажень із sql_payloads.txt (наприклад, ' OR '1'='1, SLEEP(5)), підставив їх у параметр q маршруту /search і проаналізував відповіді на наявність помилок бази даних через метод _contains_sql_error. Виявлення уразливості фіксується так:

Лістинг 4.1

```
if self._contains_sql_error(resp_text):
    results.append({
        "module": "sql_injection",
        "type": "error_based",
        "url": new_url,
        "param": "q",
        "payload": "' OR '1'='1",
        "issue": "SQL error found"
    })
```

Аналогічно модуль time_based.py виявив затримку для SLEEP(5), а boolean_based.py - різницю в довжині відповідей для '1'='1 і '1'='2. Приклад наведено на Рис. 4.2.

Результати сканування, зафіксовані в result.html, показують, що сканер виявив усі 11 типів уразливостей, реалізованих у сайті. Загальна кількість - 119, що включає кілька екземплярів кожної уразливості через різні форми та параметри. Нижче наведено аналіз основних виявлених уразливостей із технічними деталями:

Рисунок 4.2 - Консольний вивід знайдених вразливостей

SQL-ін’єкція (SQLi) була виявлена в маршруті /search і формі /search_form. Модулі error_based, boolean_based, time_based і blind_sql_injection ідентифікували 31 випадок, наприклад, для URL http://127.0.0.1:5000/search?q=' OR '1'='1 сканер

зафіксував повернення всіх записів із таблиці products, а для SLEEP(5) - затримку понад 5 секунд. Це підтверджує вразливість через непараметризований запит у коді test_site.py. Приклад наведено на Рис. 4.3.

```
[INFO] [ERROR_BASED_SQLI] Found vulnerability at http://127.0.0.1:5000/search with payload ')/*
[INFO] [ERROR_BASED_SQLI] Found vulnerability at http://127.0.0.1:5000/search with payload ' OR SLEEP(5)#
[INFO] [ERROR_BASED_SQLI] Found vulnerability at http://127.0.0.1:5000/search with payload ' AND SLEEP(5)--
[INFO] [ERROR_BASED_SQLI] Found vulnerability at http://127.0.0.1:5000/search with payload '; SLEEP(5)--
[INFO] [ERROR_BASED_SQLI] Found vulnerability at http://127.0.0.1:5000/search with payload '); SELECT pg_sleep(5); --
[INFO] [ERROR_BASED_SQLI] Found vulnerability at http://127.0.0.1:5000/search with payload ' WAITFOR DELAY '0:0:5'--
[INFO] [ERROR_BASED_SQLI] Found vulnerability at http://127.0.0.1:5000/search with payload " WAITFOR DELAY '0:0:5'--
[INFO] [ERROR_BASED_SQLI] Found vulnerability at http://127.0.0.1:5000/search with payload ') WAITFOR DELAY '0:0:5'--
[INFO] [ERROR_BASED_SQLI] Found vulnerability at http://127.0.0.1:5000/search with payload ") WAITFOR DELAY '0:0:5'--
[INFO] [ERROR_BASED_SQLI] Found vulnerability at http://127.0.0.1:5000/search with payload '; EXEC master..xp_cmdshell 'ping 127.0.0.1'--
```

Рисунок 4.3 - Консольний вивід знайдених вразливостей SQLi

Міжсайтові сценарії (XSS) виявлені в маршруті /contact у 24 випадках. Модулі reflected, stored і dom_based протестували поля name і message, підставляючи навантаження з xss_payloads.txt (наприклад, <script>alert(1)</script>). Відображення коду в відповіді без екранування підтвердило уразливість, що відповідає логіці маршруту. Приклад наведено на Рис. 4.4.

```
[INFO] [REFLECTED_XSS] Found vulnerability with payload '<script>alert(1)</script>' at http://127.0.0.1:5000/execute
[INFO] [REFLECTED_XSS] Found vulnerability with payload '"><script>alert(2)</script>' at http://127.0.0.1:5000/execute
[INFO] [REFLECTED_XSS] Found vulnerability with payload '"><img src=x onerror=alert(3)>' at http://127.0.0.1:5000/execute
[INFO] [REFLECTED_XSS] Found vulnerability with payload '"><svg onload=alert(4)>' at http://127.0.0.1:5000/execute
[INFO] [REFLECTED_XSS] Found vulnerability with payload '" onload=alert(5)' at http://127.0.0.1:5000/execute
[INFO] [REFLECTED_XSS] Found vulnerability with payload '"><iframe src=javascript:alert(6)>' at http://127.0.0.1:5000/execute
[INFO] [REFLECTED_XSS] Found vulnerability with payload '<script>alert(1)</script>' at http://127.0.0.1:5000/contact
[INFO] [REFLECTED_XSS] Found vulnerability with payload '"><script>alert(2)</script>' at http://127.0.0.1:5000/contact
[INFO] [REFLECTED_XSS] Found vulnerability with payload '"><img src=x onerror=alert(3)>' at http://127.0.0.1:5000/contact
[INFO] [REFLECTED_XSS] Found vulnerability with payload '"><svg onload=alert(4)>' at http://127.0.0.1:5000/contact
```

Рисунок 4.4 - Консольний вивід знайдених вразливостей XSS

Перехресні запити із підробленням сайту (CSRF) зафіксовано в маршруті /profile у 7 випадках. Модуль csrf_scanner.py виявив відсутність токенів у формі через token_analysis.py і успішне виконання POST-запиту без токена, що збігається з кодом test_site.py. Приклад наведено на Рис. 4.5.

```
[INFO] [CSRF] No CSRF token found in a sensitive form. at form: http://127.0.0.1:5000/profile
[INFO] [CSRF] No CSRF token found in a sensitive form. at form: http://127.0.0.1:5000/catalog/delete/1
[INFO] [CSRF] No CSRF token found in a sensitive form. at form: http://127.0.0.1:5000/catalog/delete/2
[INFO] [CSRF] No CSRF token found in a sensitive form. at form: http://127.0.0.1:5000/catalog/delete/3
[INFO] [CSRF] No CSRF token found in a sensitive form. at form: http://127.0.0.1:5000/catalog/delete/4
[INFO] [CSRF] No CSRF token found in a sensitive form. at form: http://127.0.0.1:5000/catalog/delete/5
[INFO] [CSRF] No CSRF token found in a sensitive form. at form: http://127.0.0.1:5000/catalog/delete/6
```

Рисунок 4.5 - Консольний вивід знайдених вразливостей CSRF

NoSQL-ін’єкція (NoSQLi) ідентифікована в маршруті /nosql у 5 випадках. Модулі simple_nosql і advanced_nosql виявили помилки для {"$gt": ""} і затримку для {"$where": "sleep(2000)"}, що відповідає емітації уразливості в коді. Приклад наведено на Рис. 4.6.

```
[INFO] [NOSQL_INJECTION] Found vulnerability at http://127.0.0.1:5000/nosql with payload {"$gt": ""}
[INFO] [NOSQL_INJECTION] Found vulnerability at http://127.0.0.1:5000/nosql with payload {"$ne": null}
[INFO] [NOSQL_INJECTION] Found vulnerability at http://127.0.0.1:5000/nosql with payload {"$regex": ".*"}
[INFO] [NOSQL_INJECTION] Found vulnerability at http://127.0.0.1:5000/nosql with payload {"$where": "sleep(2000)"}
[INFO] [NOSQL_INJECTION] Found vulnerability at http://127.0.0.1:5000/nosql with payload {"$where": "function(){return true;}"}
```

Рисунок 4.6 - Консольний вивід знайдених вразливостей NoSQLi

Віддалене виконання коду (RCE) виявлено в маршруті /execute у 14 випадках. Модулі command_injection і code_injection зафіксували псевдо-вивід для навантажень типу ; ls -la із rce_payloads.txt, підтверджуючи відсутність валідації параметра cmd. Приклад наведено на Рис. 4.7.

```
[INFO] [RCE_COMMAND_INJECTION] Found vulnerability at http://127.0.0.1:5000/execute with payload && cat /etc/passwd
[INFO] [RCE_COMMAND_INJECTION] Found vulnerability at http://127.0.0.1:5000/execute with payload | powershell.exe
[INFO] [RCE_COMMAND_INJECTION] Found vulnerability at http://127.0.0.1:5000/execute with payload ; uname -a
[INFO] [RCE_COMMAND_INJECTION] Found vulnerability at http://127.0.0.1:5000/execute with payload "<?php system($_GET['cmd']); ?>"
[INFO] [RCE_COMMAND_INJECTION] Found vulnerability at http://127.0.0.1:5000/execute with payload "$(cat /etc/passwd)"
[INFO] [RCE_COMMAND_INJECTION] Found vulnerability at http://127.0.0.1:5000/read_file with payload && cat /etc/passwd
[INFO] [RCE_COMMAND_INJECTION] Found vulnerability at http://127.0.0.1:5000/read_file with payload "$(cat /etc/passwd)"
[INFO] [RCE_CODE_INJECTION] Found vulnerability at http://127.0.0.1:5000/execute with payload && cat /etc/passwd
[INFO] [RCE_CODE_INJECTION] Found vulnerability at http://127.0.0.1:5000/execute with payload | powershell.exe
[INFO] [RCE_CODE_INJECTION] Found vulnerability at http://127.0.0.1:5000/execute with payload ; uname -a
[INFO] [RCE_CODE_INJECTION] Found vulnerability at http://127.0.0.1:5000/execute with payload "<?php system($_GET['cmd']); ?>"
```

Рисунок 4.7 - Консольний вивід знайдених вразливостей RCE

Directory Traversal виявлено в маршруті /read_file у 18 випадках. Модуль traversal_scanner.py ідентифікував псевдо-вміст файлів для шляхів типу ../../../etc/passwd, що збігається з логікою маршруту без нормалізації. Приклад наведено на Рис. 4.8.

```
[INFO] [DIRECTORY_TRAVERSAL] Found vulnerability at http://127.0.0.1:5000/read_file with payload ../etc/passwd
[INFO] [DIRECTORY_TRAVERSAL] Found vulnerability at http://127.0.0.1:5000/read_file with payload ../../etc/passwd
[INFO] [DIRECTORY_TRAVERSAL] Found vulnerability at http://127.0.0.1:5000/read_file with payload ../../../etc/passwd
[INFO] [DIRECTORY_TRAVERSAL] Found vulnerability at http://127.0.0.1:5000/read_file with payload ../../../../etc/passwd
[INFO] [DIRECTORY_TRAVERSAL] Found vulnerability at http://127.0.0.1:5000/read_file with payload ../../../../../etc/passwd
[INFO] [DIRECTORY_TRAVERSAL] Found vulnerability at http://127.0.0.1:5000/read_file with payload ../../../../../../etc/passwd
[INFO] [DIRECTORY_TRAVERSAL] Found vulnerability at http://127.0.0.1:5000/read_file with payload ../windows/win.ini
[INFO] [DIRECTORY_TRAVERSAL] Found vulnerability at http://127.0.0.1:5000/read_file with payload ..\windows\win.ini
[INFO] [DIRECTORY_TRAVERSAL] Found vulnerability at http://127.0.0.1:5000/read_file with payload ../../windows/win.ini
```

Рисунок 4.8 - Консольний вивід знайдених вразливостей Directory Traversal

Insecure Direct Object Reference (IDOR) зафіксовано в маршруті /user_profile у 4 випадках. Модуль idor_scanner.py виявив доступ до чужих профілів через зміну user_id, наприклад, із 1 на 2, що відповідає відсутності перевірки прав. Приклад наведено на Рис. 4.9.

```
[INFO] [IDOR] Found vulnerability at http://127.0.0.1:5000/user_profile?user_id=0 with payload 0
[INFO] [IDOR] Found vulnerability at http://127.0.0.1:5000/user_profile?user_id=2 with payload 2
[INFO] [IDOR] Found vulnerability at http://127.0.0.1:5000/user_profile?user_id=0 with payload 0
[INFO] [IDOR] Found vulnerability at http://127.0.0.1:5000/user_profile?user_id=9999 with payload 9999
```

Рисунок 4.9 - Консольний вивід знайдених вразливостей IDOR

Open Redirect ідентифіковано в маршруті /redirect_me у 3 випадках. Модуль open_redirect_scanner.py підтвердив перенаправлення на зовнішні URL типу http://evil.com через відсутність валідації параметра next. Приклад наведено на Рис. 4.10.

```
[INFO] [OPEN_REDIRECT] Found vulnerability at http://127.0.0.1:5000/redirect_me with payload http://evil.com
[INFO] [OPEN_REDIRECT] Found vulnerability at http://127.0.0.1:5000/redirect_me with payload https://evil.com
[INFO] [OPEN_REDIRECT] Found vulnerability at http://127.0.0.1:5000/redirect_me with payload //evil.com
```

Рисунок 4.10 - Консольний вивід знайдених вразливостей Open Redirect

Server-Side Request Forgery (SSRF) виявлено в маршруті /fetch_url у 4 випадках. Модуль ssrf_scanner.py зафіксував доступ до http://127.0.0.1, що відповідає логіці маршруту без обмежень URL. Приклад наведено на Рис. 4.11.

```
[INFO] [SSRF] Found vulnerability at http://127.0.0.1:5000/fetch_url with payload http://127.0.0.1:80
[INFO] [SSRF] Found vulnerability at http://127.0.0.1:5000/fetch_url with payload http://127.0.0.1:8080
[INFO] [SSRF] Found vulnerability at http://127.0.0.1:5000/fetch_url with payload http://localhost:8000
[INFO] [SSRF] Found vulnerability at http://127.0.0.1:5000/fetch_url with payload gopher://127.0.0.1:11211
```

Рисунок 4.11 - Консольний вивід знайдених вразливостей SSRF

Insecure File Upload ідентифіковано в маршруті /upload у 3 випадках. Модуль upload_scanner.py успішно завантажив файли типу shell.php, підтверджуючи відсутність валідації в коді. Приклад наведено на Рис. 4.12.

```
[INFO] [INSECURE_FILE_UPLOAD] Found vulnerability at http://127.0.0.1:5000/upload with payload None
[INFO] [INSECURE_FILE_UPLOAD] Found vulnerability at http://127.0.0.1:5000/upload with payload None
[INFO] [INSECURE_FILE_UPLOAD] Found vulnerability at http://127.0.0.1:5000/upload with payload None
```

Рисунок 4.12 - Консольний вивід знайдених вразливостей Insecure File Upload

Слабка автентифікація виявлена в маршруті /login_check у 6 випадках. Модуль auth_scanner.py ідентифікував успішний вхід із комбінаціями типу admin/admin, що збігається зі слабкими паролями в коді. Приклад наведено на Рис. 4.13.

```
[INFO] [AUTHENTICATION] Found vulnerability at http://127.0.0.1:5000/login_check with payload None
[INFO] [AUTHENTICATION] Found vulnerability at http://127.0.0.1:5000/login_check with payload None
[INFO] [AUTHENTICATION] Found vulnerability at http://127.0.0.1:5000/login_check with payload None
[INFO] [AUTHENTICATION] Found vulnerability at http://127.0.0.1:5000/login_check with payload None
[INFO] [AUTHENTICATION] Found vulnerability at http://127.0.0.1:5000/login_check with payload None
[INFO] [AUTHENTICATION] Found vulnerability at http://127.0.0.1:5000/login_check with payload None
```

Рисунок 4.13 - Консольний вивід знайдених вразливостей authentication

Технічна робота сканера базується на модульності та точному аналізі відповідей. Наприклад, для XSS модуль reflected.py перевіряє наявність навантаження у відповіді, для SQLi time_based.py вимірює затримку через time.time(), а для SSRF ssrf_scanner.py аналізує вміст через ssrf_helpers.py. Логування через logger.py фіксує кожен етап.

Аналіз результатів показує, що сканер успішно виявив усі вбудовані уразливості, підтверджуючи його здатність обробляти як прості (XSS, Open Redirect), так і складні (SQLi, RCE, SSRF) слабкі місця. Кількість 119 відображає повторення уразливостей у різних формах і параметрах, наприклад, кілька полів у /contact для XSS чи різні шляхи для Directory Traversal у /read_file.

4.2 Оцінка ефективності веб-сканера

Сильні сторони веб-сканера включають високу точність виявлення уразливостей завдяки модульній архітектурі, яка дозволяє кожному модулю спеціалізуватися на конкретному типі слабких місць, таких як SQL-ін’єкції, XSS чи SSRF. Гнучкість налаштування через командний рядок із параметрами типу --

modules, --depth і --delay забезпечує адаптивність до різних сценаріїв тестування, дозволяючи користувачу обирати потрібні перевірки чи регулювати швидкість сканування. Широке охоплення уразливостей, підтверджене виявленням 11 типів слабких місць, відображає здатність сканера працювати з різноманітними точками введення, такими як URL-параметри та форми, що обробляються через Crawler і Requester. Ефективність аналізу відповідей сервера, наприклад, через вимірювання затримок у time_based.py чи пошук сигнатур у error_based.py, гарантує точне виявлення навіть складних уразливостей, таких як NoSQL-ін’єкції чи RCE. Генерація детальних звітів у форматах TXT, CSV і HTML через report_generator.py полегшує аналіз результатів, надаючи структуровану інформацію про кожну уразливість, включаючи URL, навантаження та тип проблеми.[21]

Слабкі сторони сканера пов’язані з потенційними хибнопозитивними результатами, оскільки деякі модулі, наприклад, reflected.py для XSS, покладаються на просту перевірку наявності навантаження у відповіді, що може спрацьовувати навіть при частковому екрануванні сервером. Обмежена глибина аналізу для складних логічних уразливостей, таких як IDOR чи CSRF у специфічних контекстах, виникає через залежність від стандартних шаблонів тестування без адаптації до унікальної логіки додатку. Відсутність автоматичного розпізнавання бізнес-логіки сайту, наприклад, зв’язків між формами чи сесіями, знижує здатність сканера виявляти уразливості, що потребують контекстного розуміння. Залежність від якості навантажень у файлах типу sql_payloads.txt чи xss_payloads.txt може призводити до пропуску уразливостей, якщо база даних недостатньо повна. Обмежена підтримка асинхронних запитів чи складних JavaScript-додатків, таких як SPA, через синхронну природу Requester, зменшує ефективність у сучасних веб-середовищах.[22]

Сильні сторони сканера підкреслюють його надійність як інструменту для базового та середнього рівня тестування безпеки, тоді як слабкі сторони вказують на напрями для вдосконалення, зокрема в адаптивності та аналізі складних сценаріїв. Технічна реалізація, заснована на модульності та гнучкості, забезпечує міцну основу для подальшого розвитку.

4.3. Рекомендації щодо покращення веб-сканера

Тестування веб-сканера на тестовому сайті показало його високу ефективність у виявленні уразливостей, однак оцінка сильних і слабких сторін виявила низку напрямів для вдосконалення. Ці напрями зосереджені на підвищенні точності, розширенні функціональності та адаптивності до сучасних веб-додатків, що дозволить сканеру краще справлятися зі складними сценаріями та зменшити кількість хибнопозитивних результатів.

Для зменшення хибнопозитивних результатів, особливо в модулях типу reflected.py для XSS, доцільно вдосконалити аналіз відповідей сервера шляхом додавання контекстної перевірки. Замість простого пошуку навантаження у відповіді (if payload in resp_text) варто реалізувати парсинг HTML через бібліотеку, наприклад, BeautifulSoup, щоб визначати, чи код дійсно виконується в браузері, а не просто відображається як текст. Це підвищить точність виявлення відображених XSS, виключивши випадки, коли сервер частково екранує дані, але не усуває уразливість повністю. Такий підхід потребуватиме інтеграції додаткової залежності в requirements.txt і модифікації логіки в xss_helpers.py.[23]

Розширення глибини аналізу складних логічних уразливостей, таких як IDOR і CSRF, можливе через додавання механізму моделювання сесій користувача. Поточна реалізація в idor_scanner.py і csrf_scanner.py покладається на статичні шаблони тестування, що обмежує виявлення уразливостей, які залежать від контексту автентифікації чи бізнес-логіки. Рекомендується додати підтримку авторизації в main.py, наприклад, через параметр --credentials username:password, який дозволить сканеру входити в систему, зберігати cookies через requests.Session() у requester.py і тестувати запити від імені авторизованого користувача. Це підвищить здатність сканера аналізувати доступ до об’єктів у user_profile чи легітимність форм у /profile.[24]

Адаптація до сучасних веб-додатків, таких як односторінкові додатки (SPA), потребує вдосконалення модуля Crawler і Requester для роботи з асинхронними запитами та JavaScript. Поточна синхронна логіка в crawler.py (html_content =

self._fetch(url)) не враховує динамічний контент, генерований через AJAX чи API. Рекомендується інтегрувати бібліотеку типу aiohttp для асинхронних запитів і додати підтримку виконання JavaScript через selenium чи playwright, що дозволить сканувати сторінки, які завантажують дані після рендерингу. Наприклад, у crawler.py можна додати опцію --js-render, яка вмикатиме рендеринг через headless-браузер для аналізу DOM-based XSS у dom_based.py.[25]

Підвищення якості виявлення уразливостей залежить від розширення бази навантажень у файлах типу sql_payloads.txt, xss_payloads.txt і rce_payloads.txt. Поточні набори (68 для SQLi, 6 для XSS, 10 для RCE) є базовими, але недостатніми для охоплення всіх можливих сценаріїв. Рекомендується додати більше специфічних навантажень, наприклад, для SQLi - ' UNION SELECT database(), user(), version() --, для XSS - <img src="javascript:alert(1)">, для RCE - "eval(compile('print(1)','<string>','exec'))". Це потребуватиме оновлення логіки завантаження в sqli_helpers.py, xss_helpers.py і rce_helpers.py для підтримки коментарів чи категорій у файлах навантажень.

Для підвищення продуктивності та масштабованості сканера доцільно реалізувати багатопоточність або асинхронність у main.py. Поточна послідовна обробка запитів (for scanner in scanners: results.extend(scanner.scan_urls(urls))) обмежує швидкість сканування великих сайтів. Рекомендується використати concurrent.futures.ThreadPoolExecutor для паралельного виконання модулів вразливостей, наприклад, розподіляючи URL між потоками, що скоротить час сканування пропорційно кількості потоків. Це потребуватиме синхронізації результатів через threading.Lock у report_generator.py для уникнення конфліктів при записі.

Додатковим покращенням стане інтеграція механізму автоматичного розпізнавання типу веб-додатку для адаптації стратегії сканування. Наприклад, аналіз заголовків відповідей (Server, X-Powered-By) чи структури HTML у crawler.py може визначати використання CMS (наприклад, WordPress) чи фреймворків, дозволяючи модулям типу auth_scanner.py додавати специфічні тести для відомих слабких місць (наприклад, дефолтні паролі WordPress). Це

потребуватиме створення бази сигнатур у файлі конфігурації, наприклад, settings.py.

ВИСНОВКИ

Дипломна робота на тему "Розробка додатку для аналізу вразливостей веб-сайтів" успішно досягла своєї мети, створивши та протестувавши веб-сканер, який виявив всі уразливості на спеціально розробленому тестовому сайті. Усі завдання виконано, що дозволило проаналізувати принципи роботи веб-сайтів, їхні слабкі місця та створити інструмент для автоматизованого виявлення уразливостей. Теоретичний аналіз систематизував знання про типи уразливостей, таких як SQL-ін'єкції, XSS чи CSRF, і підкреслив актуальність їхнього тестування в умовах зростання кіберзагроз.[26]

Тестовий сайт, побудований на Flask, став контрольованим середовищем для оцінки сканера, демонструючи вразливості через непараметризовані запити, відсутність екранування чи валідації. Сканер показав високу точність завдяки модульній архітектурі та гнучкості налаштування через командний рядок, виявивши всі слабкі місця, від простих, як Open Redirect, до складних, як RCE чи SSRF. Це підтверджує його здатність ефективно аналізувати різноманітні точки введення, забезпечуючи деталізовані звіти для подальшого усунення проблем.

Тестування засвідчило сильні сторони сканера, зокрема його точність і широкий охват, але виявило й слабкості, такі як хибнопозитивні результати та обмеження в аналізі логічних уразливостей. Рекомендації щодо вдосконалення включають контекстний аналіз відповідей, підтримку сесій і асинхронність для адаптації до сучасних веб-технологій. Робота довела практичну цінність сканера для підвищення кібербезпеки, а запропоновані покращення відкривають перспективи для його розвитку та застосування в реальних умовах.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. OWASP Top Ten: A Comprehensive Guide to Web Application Security Risks. OWASP Foundation. Режим доступу: https://owasp.org/www-project-top-ten/ (дата звернення: 01.03.2025).
2. Web Vulnerability Scanners: Tools and Techniques for Automated Security Testing. SANS Institute. Режим доступу: https://www.sans.org/reading-room/whitepapers/testing/web-vulnerability-scanners-34465 (дата звернення: 02.03.2025).
3. Cybersecurity Research Methodologies and Standards. IEEE Xplore. Режим доступу: https://ieeexplore.ieee.org/document/9478321 (дата звернення: 03.03.2025).
4. Client-Server Architecture in Web Applications. ScienceDirect. Режим доступу: https://www.sciencedirect.com/science/article/pii/S0164121223000575 (дата звернення: 04.03.2025).
5. HTTP/HTTPS Protocols and TLS/SSL Security. IETF. Режим доступу: https://www.ietf.org/rfc/rfc2818.txt (дата звернення: 05.03.2025).
6. Cybersecurity Fundamentals: Understanding the CIA Triad. NIST. Режим доступу: https://www.nist.gov/cyberframework/cia-triad (дата звернення: 06.03.2025).
7. Principles of Information Security. (ISC)². Режим доступу: https://www.isc2.org/Insights/2024/03/The-CIA-Triad-A-Foundation-for-Cybersecurity (дата звернення: 07.03.2025).
8. Common Web Vulnerabilities and Their Mitigation Strategies. MITRE. Режим доступу: https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html (дата звернення: 08.03.2025).
9. SQL and NoSQL Injection Attacks: Detection and Prevention. Imperva. Режим доступу: https://www.imperva.com/learn/application-security/sql-injection-sqli/ (дата звернення: 09.03.2025).

10. Dynamic Application Security Testing (DAST): Principles and Tools. Gartner. Режим доступу: https://www.gartner.com/en/information-technology/glossary/dynamic-application-security-testing-dast (дата звернення: 10.03.2025).
11. Automated Web Security Testing: Challenges and Solutions. Springer. Режим доступу: https://link.springer.com/article/10.1007/s10586-024-04234-5 (дата звернення: 12.03.2025).
12. Modular Software Architecture for Security Tools. IEEE Xplore. Режим доступу: https://ieeexplore.ieee.org/document/9501234 (дата звернення: 14.03.2025).
13. Designing Modular Systems for Scalability. O'Reilly. Режим доступу: https://www.oreilly.com/library/view/software-architecture/9781492054740/ (дата звернення: 16.03.2025).
14. Command Line Interfaces for Security Tools. Linux Journal. Режим доступу: https://www.linuxjournal.com/content/cli-security-tools (дата звернення: 18.03.2025).
15. Creating Vulnerable Web Applications for Security Testing. Hack The Box. Режим доступу: https://www.hackthebox.com/blog/intentionally-vulnerable-web-applications-for-pentesting (дата звернення: 20.03.2025).
16. Flask Framework for Web Development. Flask Documentation. Режим доступу: https://flask.palletsprojects.com/en/stable/ (дата звернення: 22.03.2025).
17. Structuring Web Applications with Flask. Real Python. Режим доступу: https://realpython.com/flask-project-structure/ (дата звернення: 24.03.2025).
18. Web Application Vulnerabilities: A Comprehensive Overview. Security Boulevard. Режим доступу: https://securityboulevard.com/2024/12/web-application-vulnerabilities-a-comprehensive-guide/ (дата звернення: 26.03.2025).
19. OWASP Top 10 Vulnerabilities: Detailed Analysis. OWASP Foundation. Режим доступу: https://owasp.org/www-project-top-ten/2021/ (дата звернення: 28.03.2025).
20. Evaluating the Effectiveness of Web Vulnerability Scanners. MDPI. Режим доступу: https://www.mdpi.com/2078-2489/15/10/514 (дата звернення: 30.03.2025).

21. Performance Metrics for Web Security Scanners. TechTarget. Режим доступу: https://www.techtarget.com/searchsecurity/feature/web-scanner-metrics (дата звернення: 31.03.2025).
22. False Positives in Web Vulnerability Scanning. Dark Reading. Режим доступу: https://www.darkreading.com/vulnerabilities-threats/false-positives-web-scanning (дата звернення: 01.04.2025).
23. HTML Parsing for Security Analysis. Beautiful Soup Documentation. Режим доступу: https://www.crummy.com/software/BeautifulSoup/bs4/doc/ (дата звернення: 02.04.2025).
24. Session Management in Web Security Testing. OWASP. Режим доступу: https://owasp.org/www-community/controls/Session_Management (дата звернення: 03.04.2025).
25. Asynchronous Programming for Web Security Tools. Python Software Foundation. Режим доступу: https://www.python.org/dev/peps/pep-3156/ (дата звернення: 04.04.2025).
26. The Role of Automated Security Tools in Reducing Cyber Risks. Forbes. Режим доступу: https://www.forbes.com/sites/forbestechcouncil/2024/10/15/leveraging-automation-to-enhance-cybersecurity/ (дата звернення: 05.04.2025).

ДОДАТОК А
ФРАГМЕНТИ КОДУ

Лістинг 1.1

```
parser.add_argument("url", nargs="?", default=None, help="Target URL to scan (e.g. http://example.com).")
parser.add_argument("--depth", type=int, default=1, help="Depth of crawling (default: 1).")
parser.add_argument("--modules", default="all", help="Comma-separated list of modules or 'all' (default: all).")
parser.add_argument("--report", choices=["txt", "html", "csv"], default="txt", help="Report format: txt, html, or csv (default: txt).")
parser.add_argument("--output", help="Output file for the report.")
```

Лістинг 1.2

```
def run(self) -> Set[str]:
    logging.debug(f"Starting crawl from {self.start_url}")
    while self.queue:
        url, current_depth = self.queue.popleft()
        if url in self.visited:
            continue
        self.visited.add(url)
        if not self._check_url_scope(url):
            continue
        if self.delay > 0:
            time.sleep(self.delay)
        html_content = self._fetch(url)
        if html_content is None:
            continue
        found_links, found_forms =
self._extract_links_and_forms(html_content, url)
        self.found_forms.extend(found_forms)
        if current_depth < self.depth:
            for link in found_links:
                if link not in self.visited:
                    self.queue.append((link, current_depth + 1))
    return self.visited
```

Лістинг 1.3

```
def get(self, url):
    if self.delay > 0:
        time.sleep(self.delay)
    req = urllib.request.Request(url, headers={"User-Agent":
self.user_agent})
    try:
        with urllib.request.urlopen(req, timeout=self.timeout) as response:
            data = response.read().decode("utf-8", errors="replace")
            self.last_url = response.geturl()
            return data
    except urllib.error.HTTPError:
        self.last_url = None
        return None
    except urllib.error.URLError:
        self.last_url = None
        return None
```

Лістинг 1.4

```
def info(self, msg):
    if not self.quiet:
        self._print_message(self.color_info, "INFO", msg)
```

Лістинг 1.5

```
def _generate_txt(self, results, output_file):
    with open(output_file, "w", encoding="utf-8") as f:
        f.write("Vulnerabilities Report (TXT)\n")
        f.write("--------------------------------\n\n")
        for i, r in enumerate(results, start=1):
            module = r.get("module", "unknown").upper()
            url = r.get("url") or r.get("form_action")
            payload = r.get("payload") or r.get("test_value") or ""
            issue = r.get("issue", "")
            f.write(f"{i}) [{module}] {issue}\n")
            f.write(f"   URL: {url}\n")
            if payload:
                f.write(f"   Payload: {payload}\n")
```

```
        f.write("\n")
```

Лістинг 1.6

```
def scan_urls(self, urls):
    results = []
    for url in urls:
        parsed = urllib.parse.urlparse(url)
        query_params = urllib.parse.parse_qs(parsed.query)
        for param_name, values in query_params.items():
            for payload in self.payloads:
                new_params = dict(query_params)
                new_params[param_name] = [payload]
                new_query = urllib.parse.urlencode(new_params, doseq=True)
                new_url = urllib.parse.urlunparse(
                    (parsed.scheme, parsed.netloc, parsed.path, parsed.params,
new_query, parsed.fragment)
                )
                resp_text = self.requester.get(new_url)
                if self._contains_sql_error(resp_text):
                    results.append({
                        "module": "sql_injection",
                        "type": "error_based",
                        "url": new_url,
                        "param": param_name,
                        "payload": payload,
                        "issue": "SQL error found"
                    })
    return results
```

Лістинг 1.7

```
if abs(len(true_resp) - len(false_resp)) > self.length_diff_threshold:
    results.append({
        "module": "sql_injection",
        "type": "boolean_based",
        "url": true_url,
        "param": param_name,
        "payload": true_payload,
        "issue": "Boolean-based SQL injection detected"
    })
```

Лістинг 1.8

```
start = time.time()
resp_text = self.requester.get(new_url)
elapsed = time.time() - start
if elapsed > self.delay_threshold:
    results.append({
        "module": "sql_injection",
        "type": "time_based",
        "url": new_url,
        "param": param_name,
        "payload": payload,
        "observed_delay": round(elapsed, 2),
        "issue": "Possible time-based SQL injection"
    })
```

Лістинг 1.9

```
for payload in self.payloads:
    new_params = dict(query_params)
    new_params[param_name] = [payload]
    new_query = urllib.parse.urlencode(new_params, doseq=True)
    new_url = urllib.parse.urlunparse(parsed.scheme, parsed.netloc, parsed.path, parsed.params, new_query, parsed.fragment)
    resp_text = self.requester.get(new_url)
    if payload in resp_text:
        results.append({
            "module": "xss",
            "type": "reflected",
            "url": new_url,
            "param": param_name,
            "payload": payload,
            "issue": "Reflected XSS detected"
        })
```

Лістинг 1.10

```
data_without_token = {inp["name"]: inp["value"] for inp in form["inputs"] if inp["name"].lower() not in ["csrf_token", "token"]}
resp_text = self.requester.post(form["action"], data_without_token)
if self._is_request_success(resp_text):
    results.append({
        "module": "csrf",
```

```
            "form_action": form["action"],
            "issue": "CSRF token present but server accepted request without it"
        })
```

Лістинг 1.11

```
for payload in self.payloads:
    new_params = dict(query_params)
    new_params[param_name] = [payload]
    new_query = urllib.parse.urlencode(new_params, doseq=True)
    new_url = urllib.parse.urlunparse(parsed.scheme, parsed.netloc, parsed.path, parsed.params, new_query, parsed.fragment)
    resp_text = self.requester.get(new_url)
    if is_suspicious_response(resp_text):
        results.append({
            "module": "directory_traversal",
            "url": new_url,
            "param": param_name,
            "payload": payload
        })
```

Лістинг 1.12

```
for payload in self.payloads:
    new_params = dict(query_params)
    new_params[param_name] = [payload]
    new_query = urllib.parse.urlencode(new_params, doseq=True)
    new_url = urllib.parse.urlunparse(parsed.scheme, parsed.netloc, parsed.path, parsed.params, new_query, parsed.fragment)
    resp_text = self.requester.get(new_url)
    if self._contains_nosql_error(resp_text):
        results.append({
            "module": "nosql_injection",
            "type": "simple_nosql",
            "url": new_url,
            "param": param_name,
            "payload": payload,
            "issue": "NoSQL error found"
    })
```

Лістинг 1.13

```
start = time.time()
resp_text = self.requester.get(new_url)
```

```
elapsed = time.time() - start
if elapsed > self.delay_threshold:
    results.append({
        "module": "nosql_injection",
        "type": "advanced_nosql",
        "url": new_url,
        "param": param_name,
        "payload": payload,
        "observed_delay": round(elapsed, 2),
        "issue": "Possible time-based NoSQL injection"
    })
```

Лістинг 1.14

```
if self._contains_rce_signature(resp_text):
    results.append({
        "module": "rce",
        "type": "command_injection",
        "url": new_url,
        "param": param_name,
        "payload": payload,
        "issue": "Command injection detected"
    })
```

Лістинг 1.15

```
for payload in self.payloads:
    new_params[param_name] = [payload]
    new_query = urllib.parse.urlencode(new_params, doseq=True)
    new_url = urllib.parse.urlunparse(parsed.scheme, parsed.netloc,
parsed.path, parsed.params, new_query, parsed.fragment)
    self.requester.get(new_url)
    final_url = self.requester.last_url
    if final_url and is_external_url(final_url, domain):
        results.append({
            "module": "open_redirect",
            "url": new_url,
            "param": param_name,
            "payload": payload,
            "redirect_to": final_url
        })
```

Лістинг 1.16

```
for cval in candidates:
    new_params[param_name] = [str(cval)]
    new_query = urllib.parse.urlencode(new_params, doseq=True)
    new_url = urllib.parse.urlunparse(parsed.scheme, parsed.netloc, parsed.path, parsed.params, new_query, parsed.fragment)
    resp_text = self.requester.get(new_url)
    if not is_access_denied(resp_text) and is_suspiciously_valid(resp_text):
        results.append({
            "module": "idor",
            "type": "idor_sequential",
            "url": new_url,
            "param": param_name,
            "payload": str(cval),
            "issue": "Possible IDOR using sequential ID"
        })
```

Лістинг 1.17

```
if self._is_ssrf_response(resp_text):
    results.append({
        "module": "ssrf",
        "url": new_url,
        "param": param_name,
        "payload": payload,
        "issue": "Possible SSRF detected"
    })
```

Лістинг 1.18

```
files = {file_field_name: (filename, file_content, 'application/octet-stream')}
r = requests.post(url, files=files, data=data, timeout=10)
if is_upload_suspicious_response(r.text):
    results.append({
        "module": "insecure_file_upload",
        "form_action": action,
        "filename": filename,
        "issue": "Uploaded malicious file"
    })
```

Лістинг 1.19

```
for (u, p) in self.default_creds:
    post_data[user_field] = u
    post_data[pass_field] = p
```

```
        resp_text = self.requester.post(action_url, post_data)
        if is_login_success(resp_text):
            results.append({
                "type": "default_credentials",
                "username": u,
                "password": p
            })
```

Лістинг 1.20

```
args = parse_args()
logger = Logger(quiet=args.quiet, verbose=args.verbose, no_color=args.no_color)
requester = Requester(timeout=args.timeout, delay=args.delay, user_agent=args.user_agent)
crawler = Crawler(args.url, requester=requester, logger=logger, depth=args.depth, scope=args.scope, exclude=args.exclude)
```

Лістинг 1.21

```
module_handlers = {
    "error_based": ErrorBasedSQLScanner,
    "boolean_based": BooleanBasedSQLScanner,
    "time_based": TimeBasedSQLScanner,
    "blind_sql": BlindSQLScanner,
    "reflected": ReflectedXSSScanner,
    "stored": StoredXSSScanner,
    "dom_based": DOMBasedXSSScanner,
    "csrf": BasicCSRFScanner,
    "directory_traversal": DirectoryTraversalScanner,
    "simple_nosql": SimpleNoSQLiScanner,
    "advanced_nosql": AdvancedNoSQLiScanner,
    "command_injection": CommandInjectionScanner,
    "code_injection": CodeInjectionScanner,
    "open_redirect": OpenRedirectScanner,
    "idor": IDORScanner,
    "ssrf": SSRFScanner,
    "file_upload": FileUploadScanner,
    "auth": AuthScanner
}
selected_modules = args.modules.split(",") if args.modules != "all" else module_handlers.keys()
```

```
scanners = [module_handlers[mod](requester, logger) for mod in selected_modules if mod in module_handlers]
```

Лістинг 1.22

```
def post(self, url, data):
    if self.delay > 0:
        time.sleep(self.delay)
    req = urllib.request.Request(url, data=urllib.parse.urlencode(data).encode(), headers={"User-Agent": self.user_agent})
    try:
        with urllib.request.urlopen(req, timeout=self.timeout) as response:
            return response.read().decode("utf-8", errors="replace")
    except urllib.error.HTTPError:
        return None
```

Лістинг 1.23

```
def info(self, msg):
    if not self.quiet:
        self._print_message(self.color_info, "INFO", msg)
```

Лістинг 1.24

```
def _generate_txt(self, results, output_file):
    with open(output_file, "w", encoding="utf-8") as f:
        f.write("Vulnerabilities Report (TXT)\n")
        f.write("--------------------------------\n\n")
        for i, r in enumerate(results, start=1):
            module = r.get("module", "unknown").upper()
            url = r.get("url") or r.get("form_action")
            payload = r.get("payload") or r.get("test_value") or ""
            issue = r.get("issue", "")
            f.write(f"{i}) [{module}] {issue}\n")
            f.write(f"   URL: {url}\n")
            if payload:
                f.write(f"   Payload: {payload}\n")
            f.write("\n")
```