Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ДОПУСТИТИ ДО ЗАХИСТУ:
В.о. завідувача кафедри
кібербезпеки та захисту
інформації
________ Іван ПАРХОМЕНКО
«__» червня 2025 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи

галузь знань 12 Інформаційні технології
(шифр і назва галузі знань)

спеціальність 125 Кібербезпека
(код і назва спеціальності)

освітній ступень бакалавр

освітня програма Кібербезпека
(назва освітньо-професійної програми)

на тему: «Інструмент для виявлення вразливостей у вебдодатках»

Виконавець: студентка IV курсу, групи КБ-41

________________ **Владислава МАРИНИЧ**
(підпис) (ім’я, прізвище)

	Підпис	Ім’я, прізвище
Керівник		Олександр ТОРОШАНКО
Нормоконтроль		Юрій БАБЕНКО

Київ 2025

Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ЗАТВЕРДЖЕНО:
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
________ Іван ПАРХОМЕНКО
«29» листопада 2024 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності 125 Кібербезпека
(код і назва спеціальності)
освітньої програми Кібербезпека
(назва освітньо-професійної програми)

Студентці **КБ-41** **Маринич Владиславі Віталіївні**
(група) (прізвище ім'я по батькові)

Тема кваліфікаційної роботи Інструмент для виявлення вразливостей у вебдодатках

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Тема кваліфікаційної роботи затверджена на засіданні кафедри кібербезпеки та захисту інформації протокол №6 від 28.11.2024 р.

2. ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Вразливості вебдодатків, міжсайтовий скриптинг, технології headless браузерів

3. ЗМІСТ РОЗРАХУНКОВО-ПОЯСНЮВАЛЬНОЇ ЗАПИСКИ

Необхідно дослідити архітектуру вебдодатків та механізми виникнення міжсайтового скриптингу, проаналізувати сучасні сканери вразливостей, вивчити можливості headless браузерів для верифікації XSS-атак, розробити програмний модуль для виявлення XSS-вразливостей з оптимізованим використанням headless браузера.

4. ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Практична цінність Розроблений програмний модуль для автоматизованого виявлення XSS у вебдодатках з використанням headless браузера

5. ДАТА ВИДАЧІ ЗАВДАННЯ

Дата видачі завдання: 28 листопада 2024 року

Завдання видав ____________ Олександр ТОРОШАНКО
(підпис) (ім'я, прізвище)

Завдання прийняла до виконання ____________ Владислава МАРИНИЧ
(підпис) (ім'я, прізвище)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Найменування етапів робіт	Строки виконання робіт (початок-кінець)	Відмітка про виконання
1	Уточнення постановки задачі	29.11.2024 – 19.01.2025	*виконано*
2	Аналіз літератури	20.01.2025 – 03.02.2025	*виконано*
3	Обґрунтування вибору рішення	04.02.2025 – 09.02.2025	*виконано*
4	Дослідження архітектури вебдодатків та XSS-вразливостей	10.02.2025 – 23.02.2025	*виконано*
5	Огляд сучасних сканерів вразливостей	24.02.2025 – 09.03.2025	*виконано*
6	Дослідження використання headless браузера для для підтвердження наявності XSS	10.03.2025 – 31.03.2025	*виконано*
7	Дослідження можливих методів оптимізації швидкості сканування	01.04.2025 – 20.04.2025	*виконано*
8	Розробка та тестування інструменту	21.04.2025 – 18.05.2025	*виконано*
9	Оформлення пояснювальної записки	19.05.2025 – 01.06.2025	*виконано*
10	Підготовка до захисту кваліфікаційної роботи	02.06.2025 – 13.06.2025	*виконано*

Завдання видав ____________ Олександр ТОРОШАНКО
(підпис) (ім'я, прізвище)

Завдання прийняв до виконання ____________ Владислава МАРИНИЧ
(підпис) (ім'я, прізвище)

Термін подання кваліфікаційної роботи до ЕК 13 червня 2025 року

РЕФЕРАТ

Кваліфікаційна робота складається зі вступу, трьох розділів, загальних висновків, списку використаних джерел, додатків, має 65 сторінок основного тексту, 3 таблиці та 34 рисунки. Крім того, робота містить 5 додатків із загальною кількістю сторінок 16. Список використаних джерел містить 30 найменувань і займає 4 сторінки.

Метою роботи є розробка програмного модуля виявлення вразливостей типу XSS з використанням headless браузера.

Для досягнення зазначеної мети поставлено наступні завдання:

- дослідити архітектуру вебдодатків та механізми виникнення вразливостей типу XSS;
- проаналізувати сучасні сканери вразливостей та визначити критерії оцінки їх ефективності;
- забезпечити точність виявлення вразливостей за допомогою headless браузера;
- оптимізувати швидкість сканування за допомогою паралельної обробки, кешування та пакетної обробки запитів;
- розробити та протестувати інструмент для підтвердження наявності XSS-вразливостей.

Об'єктом дослідження є процес аналізу вразливостей вебдодатків.

Предметом дослідження є методи та інструменти виявлення вразливостей типу Cross-Site Scripting (XSS) у вебдодатках.

Практичною цінністю отриманих результатів є створення програмного модуля для комплексного виявлення XSS-вразливостей з використанням headless браузера та оптимізованою архітектурою сканування.

Ключові слова: вебдодаток, вразливості, міжстайтовий скриптинг, headless браузер.

ЗМІСТ

РЕФЕРАТ 4
ЗМІСТ 5
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ 7
ВСТУП 8
РОЗДІЛ 1 АНАЛІЗ ПРИЧИН ВИНИКНЕННЯ ВРАЗЛИВОСТЕЙ ТИПУ XSS У ВЕБДОДАТКАХ 10
1.1 Архітектура вебдодатків, принципи їх роботи та потенційні вектори атак 10
1.2 Вразливості типу XSS: характеристика, механізми виникнення 13
1.3 Класифікація вразливостей типу XSS 17
1.4 Методи захисту від XSS-вразливостей 29
Висновки за розділом 1 31
РОЗДІЛ 2 АНАЛІЗ І ПОРІВНЯННЯ ІНСТРУМЕНТІВ ДЛЯ ВИЯВЛЕННЯ XSS-ВРАЗЛИВОСТЕЙ 33
2.1 Аналіз підходів до виявлення XSS-вразливостей 33
2.2 Огляд сучасних сканерів вразливостей 37
2.3 Оцінка ефективності виявлення XSS-вразливостей сканерами 40
Висновки за розділом 2 43
РОЗДІЛ 3 РОЗРОБКА ПРОГРАМНОГО МОДУЛЯ ДЛЯ ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ ТИПУ XSS 44
3.2 Архітектура та реалізація програмного модуля 44
3.3 Тестування інструменту 48
3.4 Аналіз результатів 50
3.4. Можливі напрями вдосконалення програмного модуля 57
Висновки за розділом 3 58
ВИСНОВКИ 60
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 62

ДОДАТКИ 66
Додаток А Загальна схема роботи програмного модуля 66
Додаток Б Алгоритми з оптимізацією сканування 67
Додаток В Алгоритми з контекстним аналізом 70
Додаток Г Основні алгоритми виявлення 74
Додаток Д Результати тестування швидкодії інструменту до оптимізації 80

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

API	–	Application Programming Interface
CSP	–	Content Security Policy
CVE	–	Common Vulnerabilities and Exposures
DAST	–	Dynamic Application Security Testing
DOM	–	Document Object Model
HTML	–	HyperText Markup Language
HTTP	–	HyperText Transfer Protocol
IAST	–	Interactive Application Security Testing
OWASP	–	Open Web Application Security Project
SAST	–	Static Application Security Testing
SPA	–	Single Page Application
XSS	–	Cross Site Scripting

ВСТУП

Актуальність даної роботи обумовлено зростанням кількості вебдодатків та їх функціональної складності в сучасному інформаційному середовищі. Вразливості типу Cross-Site Scripting (XSS) залишаються в списку OWASP Top 10 найбільш небезпечних загроз для вебдодатків протягом багатьох років. Наявні інструменти виявлення таких вразливостей часто генерують значну кількість хибнопозитивних результатів через відсутність у них можливості реального виконання JavaScript-коду в контексті браузера. Наявність таких результатів знижує довіру до системи сканування та збільшує витрати на ручну верифікацію. Пропонований підхід із використання headless браузера дозволяє точно відтворювати умови виконання скриптів та забезпечувати більш достовірне підтвердження наявності XSS-вразливостей в автоматизованому режимі.

Метою роботи є розробка програмного модуля виявлення вразливостей типу XSS з використанням headless браузера.

Для досягнення поставленої мети були визначені наступні *завдання*:

- дослідити архітектуру вебдодатків та механізми виникнення вразливостей типу XSS;
- проаналізувати сучасні сканери вразливостей та визначити критерії оцінки їх ефективності;
- забезпечити точність виявлення вразливостей за допомогою headless браузера;
- оптимізувати швидкість сканування за допомогою паралельної обробки, кешування та пакетної обробки запитів;
- розробити та протестувати інструмент для підтвердження наявності XSS-вразливостей.

Об’єктом дослідження є процес аналізу вразливостей вебдодатків.

Предметом дослідження є методи та інструменти виявлення вразливостей типу Cross-Site Scripting (XSS) у вебдодатках.

Методи дослідження кваліфікаційної роботи бакалавра:

- аналіз літератури;
- аналіз технічної документації;
- порівняння;
- експериментальне дослідження.

Новизна роботи полягає у впровадженні комплексного підходу до виявлення XSS-вразливостей через поєднання технології headless браузера з контекстно-залежною генерацією пейлоадів та системою унікальних ідентифікаторів для точної класифікації типів вразливостей. На відміну від наявних рішень, розроблений програмний модуль забезпечує реальне виконання JavaScript-коду в браузерному середовищі з одночасною оптимізацією швидкості сканування через пакетну та паралельну обробку запитів, що дозволяє досягти оптимального балансу між точністю виявлення та продуктивністю сканування.

Галузь використання. Розроблений інструмент призначений для використання у сфері інформаційної безпеки, зокрема в процесах аудиту безпеки вебдодатків, пентестингу та автоматизованого тестування в рамках циклу розробки програмного забезпечення.

Практична цінність полягає у тому, що створений програмний модуль надає можливість комплексного виявлення XSS-вразливостей з використанням headless браузера для точного підтвердження виконання JavaScript-коду та оптимізованою архітектурою для ефективного сканування вебресурсів.

РОЗДІЛ 1
АНАЛІЗ ПРИЧИН ВИНИКНЕННЯ ВРАЗЛИВОСТЕЙ ТИПУ XSS У ВЕБДОДАТКАХ

1.1 Архітектура вебдодатків, принципи їх роботи та потенційні вектори атак

Вебдодатки – це програмне забезпечення, яке виконується на вебсервері та доступне користувачам через веббраузер, на відміну від традиційних настільних програм, які запускаються безпосередньо операційною системою. Архітектура вебдодатків базується на клієнт-серверній моделі обміну даними, що передбачає розподіл функціональності між клієнтською та серверною частинами. Серверна частина розміщується на віддалених серверах, тоді як клієнтська виконується безпосередньо на пристрої користувача.

Основою функціонування вебдодатків є протокол HTTP (HyperText Transfer Protocol) або його захищена версія HTTPS (HTTP Secure), що забезпечує стандартизований механізм обміну даними між клієнтом та сервером. Протокол базується на концепції запитів та відповідей, де клієнт формує запит до сервера, а сервер опрацьовує цей запит та повертає відповідь. Запити можуть виконуватися різними HTTP-методами, наприклад, GET для отримання ресурсів, POST для створення нових даних, PUT для оновлення існуючих ресурсів та DELETE для видалення даних тощо. Кожен із них може нести потенційні загрози, зокрема, уразливості, пов'язані з обробкою користувацького вводу, наприклад, XSS чи SQL-ін'єкції.

Сучасні вебдодатки зазвичай базуються на трирівневій архітектурі [1]:

- Рівень презентації (фронтенд) – інтерфейс користувача, реалізований за допомогою HTML, CSS та JavaScript. Сучасні додатки використовують фреймворки як React, Angular, Vue.js для створення динамічних інтерфейсів.

- Рівень додатку (бекенд) – містить бізнес-логіку, реалізується мовами програмування PHP, Python, Java, Node.js з різними фреймворками (Django, Laravel, Spring тощо).
- Рівень зберігання даних – бази даних: реляційні (MySQL, PostgreSQL), NoSQL (MongoDB, Redis) або графові (Neo4j).

З точки зору архітектурних підходів, вебдодатки можуть реалізовуватися за різними моделями. Монолітна архітектура передбачає розробку єдиного додатку, де всі функціональні компоненти тісно пов'язані між собою та виконуються в рамках одного процесу. Така архітектура спрощує розробку та розгортання, але може створювати проблеми при масштабуванні та підтримці складних систем [2].

Сервіс-орієнтована архітектура (SOA) поділяє функціональність на незалежні сервіси, що взаємодіють через стандартизовані інтерфейси за допомогою механізмів обміну повідомленнями [3]. Це дозволяє розробляти, масштабувати та підтримувати окремі компоненти системи незалежно один від одного, що підвищує гнучкість та стійкість вебдодатку. Однак, недоліками можуть бути складність реалізації, потреба у спеціалізованій інфраструктурі та висока вартість.

Мікросервісна архітектура є розвитком сервіс-орієнтованого підходу та передбачає розробку системи як набору невеликих, незалежних сервісів, кожен з яких відповідає за окрему бізнес-функцію. Сервіси мають свої власні процеси та можуть бути реалізовані різними технологіями, комунікуючи між собою за допомогою таких механізмів як API (Application Programming Interface) на основі HTTP або системи обміну повідомленнями. Мікросервісна архітектура покращує масштабованість, стійкість та гнучкість системи, забезпечує незалежне розгортання та ізоляцію даних, але збільшує складність розробки, тестування та управління всією системою [4].

Також важливим аспектом архітектури сучасних вебдодатків є взаємодія між клієнтською та серверною частинами. Традиційний підхід, заснований на багатосторінкових додатках (MPA), передбачає серверний рендеринг, коли за

кожним запитом генерується нова HTML-сторінка. Однак, сучасні вебдодатки часто використовують підхід Single Page Application (SPA), де на початку завантажується базова HTML-сторінка, а подальша взаємодія відбувається за допомогою JavaScript та AJAX-запитів, що оновлюють лише частини сторінки без її перезавантаження.

На основі підходу SPA також розвиваються прогресивні вебдодатки (PWA), що розширюють їх функції. PWA поєднують вебтехнології з функціональністю, притаманною повноцінним мобільним застосункам, зокрема підтримкою офлайн-режиму, push-сповіщеннями, механізмами кешування ресурсів та можливістю інсталяції додатка безпосередньо на пристрій користувача [5]. Ці архітектурні рішення, хоча й покращують користувацький досвід, водночас відкривають нові вектори атак.

Проєкт OWASP (Open Web Application Security Project) регулярно публікує список з 10 найбільш критичних вразливостей вебдодатків, який є важливим орієнтиром для розробників. Його остання версія була опублікована у 2021 році, де визначені такі категорії [6]:

- A01:2021-Broken Access Control
- A02:2021-Cryptographic Failures
- A03:2021-Injection
- A04:2021-Insecure Design
- A05:2021-Security Misconfiguration
- A06:2021-Vulnerable and Outdated Components
- A07:2021-Identification and Authentication Failures
- A08:2021-Software and Data Integrity Failures
- A09:2021-Security Logging and Monitoring Failures
- A10:2021-Server-Side Request Forgery (SSRF)

Вразливості типу XSS, що є фокусом даного дослідження, входять до категорії A03:2021-Injection та представляють собою один з найбільш поширених та небезпечних векторів атак на вебдодатки. Їх ефективність пояснюється особливостями архітектури вебдодатків, яка через свій

розподілений характер створює широкий спектр потенційних точок входу для атак. Поверхня атаки суттєво розширюється за рахунок виконання коду як на стороні клієнта, так і на стороні сервера. Вразливими можуть бути будь-які компоненти – від клієнтського браузера до серверних баз даних, – що відкриває додаткові можливості для експлуатації з боку зловмисників.

На клієнтській стороні основними векторами атак є маніпуляції з DOM-структурою сторінки, перехоплення користувацьких даних, експлуатація вразливостей у JavaScript-коді та атаки на локальні сховища даних (cookie, localStorage). Особливо небезпечними є вразливості типу XSS, які дозволяють виконувати довільний JavaScript-код у контексті вебдодатку та отримувати доступ до інформації з обмеженим доступом.

На серверній стороні вектори атак включають ін'єкції в бази даних, атаки на механізми автентифікації, експлуатацію вразливостей у вебсерверах та компонентах бізнес-логіки. Недостатня валідація вхідних даних, некоректна обробка помилок та помилки конфігурації можуть призвести до повної компрометації серверної частини.

Канали комунікації між клієнтом і сервером також є потенційними векторами атак. Однією із них є атака «людини посередині» (MITM), що дозволяє перехоплювати та модифікувати дані під час передачі. Відсутність належного шифрування трафіку, недоліки в реалізації HTTPS або використання незахищених протоколів становлять значну загрозу для конфіденційності та цілісності даних.

1.2 Вразливості типу XSS: характеристика, механізми виникнення

XSS (міжсайтовий скриптинг) – це тип вразливості, коли зловмисник впроваджує шкідливий JavaScript-код у вебсторінку, яку потім переглядають інші користувачі. Проблема виникає, коли дані, що надходять від користувача, виводяться на сторінку без перевірки або належної фільтрації. У результаті браузер не відрізняє шкідливий скрипт від звичайного коду сайту і виконує його

у контексті довіреного ресурсу, що призводить до порушення політики того самого походження (Same-Origin Policy) – одного з основних механізмів захисту браузера, який зазвичай обмежує доступ скриптів до даних із інших доменів.

XSS-вразливості є особливо небезпечними, оскільки вони дозволяють виконувати JavaScript-код у контексті вебдодатку, що надає зловмиснику доступ до всіх даних, доступних у цьому контексті. Незалежно від типу XSS-атаки, наслідки можуть варіюватися від завдання незручностей до повної компрометації облікового запису користувача. Найбільш критичні атаки включають викрадення сесійних cookie для захоплення сесії користувача, розкриття конфіденційних файлів, встановлення шкідливого ПЗ, перенаправлення на зловмисні ресурси та модифікацію контенту. Варто зазначити, що навіть «інформаційні» сайти без можливості редагування контенту можуть страждати від серйозних XSS-атак, які здатні завдати суттєвої шкоди компанії або навіть просто користувачу, наприклад, через зміну дозування ліків на фармацевтичних сайтах [7].

З огляду на масштабність можливих наслідків, XSS стабільно входить до переліку найнебезпечніших вразливостей за версією OWASP. За останніми даними міжсайтовий скриптинг належить категорії A03:2021-Injection. Якщо подивитися, як змінювалася його позиція у цьому списку з 2004 по 2021 рік (рис. 1.1), видно, що раніше він був на першому місці (у 2007 році), потім опустився (до 7 місця у 2017), але знову піднявся до третьої позиції у 2021 році. Також згідно з CWE Top 25 Most Dangerous Software Weaknesses за 2024 рік, міжсайтовий скриптинг (CWE-79) посідає друге місце серед найнебезпечніших вразливостей програмного забезпечення [8]. Крім того, три CVE, пов'язані з XSS, включено до каталогу Known Exploited Vulnerabilities (KEV), що підтверджує активне використання даних вразливостей у реальних кіберінцидентах. Хоча про XSS давно відомо і існують способи захисту, ця проблема досі залишається актуальною. Це пов'язано з тим, що вхідних точок для атаки дуже багато, складно повністю «почистити» всі небезпечні дані, а зловмисники постійно вигадують нові методи обходу захисту.

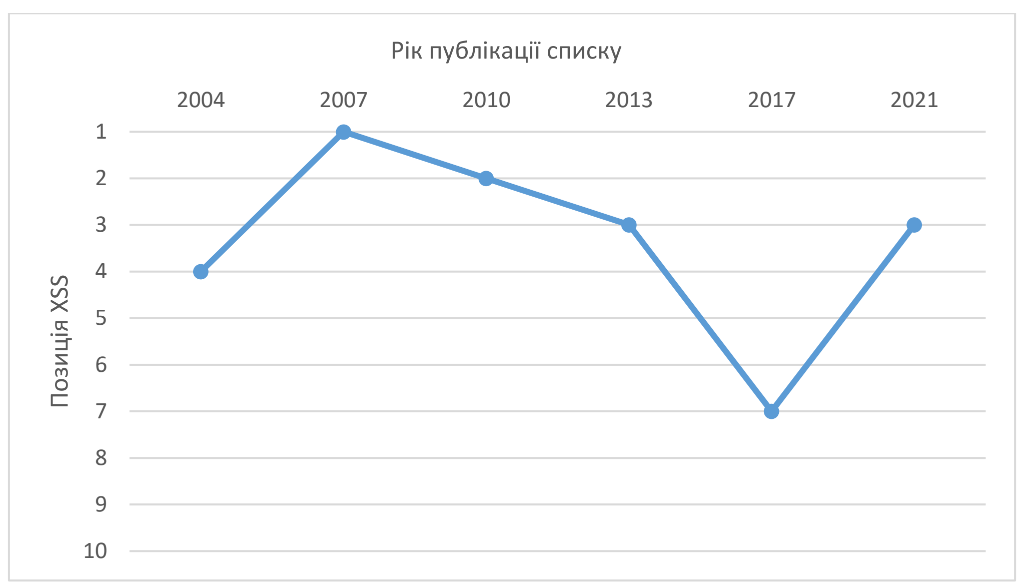


Рисунок 1.1 – Динаміка позицій вразливості XSS у списку OWASP Top 10
у 2004–2021 роках

Механізми виникнення XSS-вразливостей безпосередньо пов'язані з особливостями обробки даних у вебдодатках. Найбільш поширеними причинами появи XSS є недостатня валідація вхідних даних, неналежна санітизація виводу та неправильне використання DOM-маніпуляцій на стороні клієнта.

Недостатня валідація вхідних даних виникає, коли вебдодаток не перевіряє або недостатньо перевіряє дані, що надходять від користувача або інших зовнішніх джерел. Це дозволяє зловмисникам передавати шкідливі скрипти через різноманітні вхідні точки, такі як параметри URL, поля форм, заголовки HTTP-запитів або cookies. Наприклад, якщо вебдодаток дозволяє користувачам залишати коментарі, які згодом відображаються іншим відвідувачам без належної перевірки, зловмисник може включити в коментар JavaScript-код, що буде виконуватися у браузерах всіх користувачів, які переглядають сторінку з цим коментарем.

Неналежна санітизація виводу відбувається, коли дані, отримані з ненадійних джерел, включаються у вихідний HTML-код без належного екранування або кодування спеціальних символів. HTML є мовою розмітки, де певні символи та послідовності мають спеціальне значення. Якщо дані, що містять такі символи (наприклад, "<", ">", "&", """ або "'"), безпосередньо

включаються у вихідний HTML-код, вони можуть змінити структуру документа та дозволити впровадження шкідливого коду. Наприклад, якщо ім'я користувача відображається на сторінці без належного екранування, зловмисник може використати ім'я, що містить HTML-теги та JavaScript-код, який буде виконаний при завантаженні сторінки.

Неправильне використання DOM-маніпуляцій є особливо актуальним для сучасних вебдодатків, що активно використовують JavaScript для побудови та модифікації сторінок на стороні клієнта. Якщо JavaScript-код використовує ненадійні дані для побудови DOM-структури сторінки без належної перевірки, це може призвести до виконання шкідливого коду. Наприклад, такі функції, як innerHTML, document.write або jQuery .html() небезпечні, якщо вони використовують дані з ненадійних джерел. Те саме стосується і eval(), setTimeout(), setInterval() або Function() з динамічно сформованими рядками, що містять потенційно небезпечний контент, може призвести до виконання шкідливого коду [9].

Значну роль у виникненні XSS-вразливостей відіграють особливості обробки HTML та JavaScript браузерами. Браузери намагаються виправляти неправильно сформований HTML-код, що може призводити до неочікуваної інтерпретації контенту. Крім того, існують численні техніки обходу фільтрів та захисних механізмів, такі як використання різних кодувань символів, нестандартних форматів тегів або властивостей подій JavaScript.

Варто також враховувати, що XSS-вразливості можуть виникати на різних рівнях архітектури вебдодатків, які були визначені у підрозділі 1.1. На фронтенді вони з’являються при небезпечному використанні функцій, наприклад, dangerouslySetInnerHTML у React або bypassSecurityTrustHtml в Angular, а також при необережній обробці даних з localStorage, sessionStorage або URL-параметрів. На бекенді XSS зазвичай виникає через відсутність належної перевірки вводу, особливо під час формування HTML- або JSON-відповідей. Крім того, загрозу становить зберігання шкідливого коду в базах даних, який пізніше може бути виведений у браузер без фільтрації – це призводить до Stored

XSS. Додатково, вразливості можуть виникати у сторонніх бібліотеках або фреймворках, що використовуються у проєкті.

Ще одним важливим аспектом розуміння XSS-вразливостей є їх відмінність від інших типів атак. На відміну від SQL-ін’єкцій чи командних ін’єкцій, які націлені на серверну частину вебдодатку, XSS-атаки спрямовані на клієнтську сторону – тобто на самих користувачів. Вони не забезпечують безпосереднього доступу до серверних ресурсів і не дозволяють виконувати код на сервері, однак можуть бути використані для компрометації облікових записів, у тому числі адміністраторів. У результаті це опосередковано створює ризик компрометації й серверної інфраструктури.

XSS-вразливості також значно відрізняються від атак типу Cross-Site Request Forgery (CSRF). CSRF експлуатує довіру вебдодатку до користувача, змушуючи його виконати дію, якої він не мав наміру, при цьому нападник не бачить результату. XSS, навпаки, порушує довіру користувача до самого вебдодатку, дозволяючи впровадити та виконати шкідливий JavaScript безпосередньо у браузері, що відкриває можливості для збору та викрадення даних – саме тому наслідки XSS зазвичай серйозніші.

1.3 Класифікація вразливостей типу XSS

XSS-вразливості являють собою широкий клас вразливостей, що відрізняються механізмами впровадження шкідливого коду, тривалістю його дії, контекстом виконання та іншими характеристиками. Класифікація цих вразливостей дозволяє краще зрозуміти їхню природу, методи виявлення та підходи до захисту. Зазвичай вразливості типу XSS поділяються на три основні типи: відображені (Reflected XSS), збережені (Stored XSS) та DOM-based XSS. Кожен з цих типів має свої особливості та потребує специфічних методів захисту.

1.3.1 Reflected XSS

Відображений міжсайтовий скриптинг (або непостійний XSS) є одним із найбільш поширених різновидів XSS-вразливостей, за якого шкідливий код передається у складі запиту до вразливого вебдодатку, після чого безпосередньо відображається у відповіді сервера. Ця вразливість виникає, коли дані, отримані з ненадійних джерел (зазвичай із запиту користувача), включаються у відповідь сервера без належної перевірки, екранування або санітизації.

Головною особливістю відображеного XSS є його непостійний характер, оскільки шкідливий код не зберігається на вебсервері, а виконується лише під час обробки конкретного запиту. Для успішної експлуатації такої вразливості зловмисник повинен спонукати жертву відвідати спеціально сформоване посилання або виконати інші дії, що запускають шкідливий запит [10]. Тобто використовуються методи соціальної інженерії, такі як фішингові електронні листи, повідомлення в соціальних мережах або скорочені URL, що приховують шкідливий вміст.

Механізм реалізації відображеної XSS-атаки зазвичай має послідовність, що представлена на рис. 1.2.

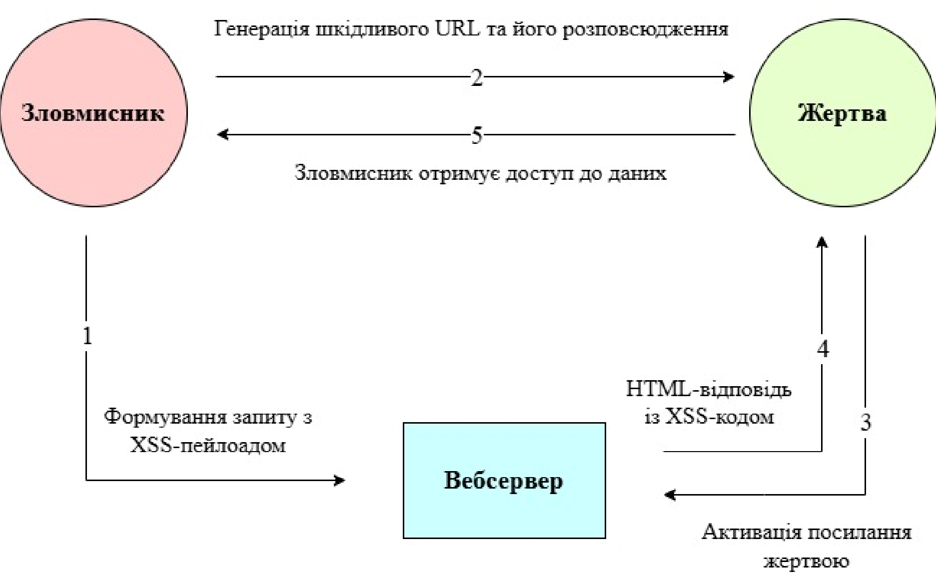


Рисунок 1.2 – Схема реалізації атаки типу Reflected XSS

Відповідно до представленої схеми (рис. 1.2), процес реалізації Reflected XSS складається з п'яти основних етапів:

1) Зловмисник надсилає спеціально сформований запит до вебсервера, що містить шкідливий JavaScript-код. Цей код зазвичай розміщується в параметрах URL, полях форм або інших компонентах HTTP-запиту. На цьому етапі зловмисник досліджує вразливості серверної обробки введених даних.

2) Після виявлення вразливості зловмисник формує спеціальне URL-посилання, яке містить зловмисний код, і передає його потенційній жертві.

3) Жертва, не підозрюючи про небезпеку, переходить за отриманим посиланням, надсилаючи таким чином запит до вебсервера. Цей запит містить шкідливий код, вбудований зловмисником.

4) Вебсервер, отримавши запит від жертви, обробляє його, але через недостатню валідацію та екранування вхідних даних, включає шкідливий код до HTML-відповіді, яка надсилається назад браузеру жертви.

5) Браузер жертви, обробляючи HTML-сторінку, виконує шкідливий JavaScript у контексті вебдодатку. Це дозволяє зловмиснику отримати доступ до даних сесії, cookie або виконати дії від імені користувача – наприклад, змінити налаштування облікового запису або надіслати запити до бекенду.

Для глибшого розуміння технічної реалізації атак типу Reflected XSS доцільно розглянути конкретні приклади пейлоадів різної складності та проаналізувати їх функціональне призначення. Пейлоади являють собою ту частину запиту, яка містить зловмисний код і є ключовим елементом на етапах 1 та 2 описаного вище механізму атаки. Зазвичай у вигляді простого прикладу наводиться такий пейлоад що представлений на рис. 1.3:

```
<script>alert('Вас зламано!')</script>
```

Рисунок 1.3 – Приклад базового XSS-пейлоада із застосуванням тега <script>

Він використовується переважно для демонстрації наявності вразливості. Якщо такий код вставити у параметр запиту, що не проходить належної

перевірки, він спричинить появу діалогового вікна в браузері користувача. Це підтверджує можливість виконання JavaScript-коду. Такий тестовий пейлоад не завдає реальної шкоди, але наочно демонструє потенційну загрозу – якщо система дозволяє виконати цей нешкідливий код, то аналогічним чином може бути виконаний і більш небезпечний скрипт.

Прикладом складного пейлоаду, що динамічно формує код для виконання, наведено на рис. 1.4:

```
html<svg onload="eval(atob('YWxlcnQoJ9CS0LDRgSDQt9C70LDQvNCw0L3Q
viEnKTs='))">
```

Рисунок 1.4 – Приклад обфускованого пейлоаду з декодуванням Base64

Тут використовується функція atob() для декодування Base64-рядка, який містить скрипт alert('Вас зламано!'), що ускладнює виявлення вразливості автоматизованими сканерами. Такий підхід є прикладом обфускації – техніки, яка дозволяє приховати шкідливий код до моменту його виконання, що значно ускладнює виявлення за допомогою звичайного сигнатурного аналізу.

Щоб оцінити реальний вплив Reflected XSS доцільно звернутися до конкретних випадків вразливостей, задокументованих у базі CVE (Common Vulnerabilities and Exposures). Такі приклади демонструють, як невірна обробка введених користувачем даних призводила до виконання зловмисного коду в браузері. Наприклад:

- CVE-2024-13918 – Laravel (версії 11.9.0–11.35.1)

У режимі налагодження (APP_DEBUG=true) Laravel відображав параметри запиту без належного екранування. Вразливість усунено у версії 11.36.0.

- CVE-2023-29457 – Zabbix

Вразливість дозволяла виконати Reflected XSS через поля форми дій, що відображалися без належної перевірки введених даних.

- CVE-2024-11846 – Плагін Travel Tour для WordPress (до версії 5.2.4)

Вразливість дозволяла виконати Reflected XSS проти користувачів з більшими привілеями (наприклад, адміністраторів) через некоректну обробку введених даних.

Відображені XSS-атаки, хоча і вимагають соціальної інженерії для експлуатації, залишаються одним з найбільш поширених типів XSS-вразливостей через їх відносну простоту та широкий спектр потенційних вхідних точок у більшості вебдодатків.

1.3.2 Stored XSS

Збережений міжсайтовий скриптинг (або постійний XSS) є одним із найбільш небезпечних різновидів XSS-вразливостей, за якого шкідливий код зберігається на сервері вразливого вебдодатку та згодом відтворюється у браузері кожного користувача, який отримує доступ до сторінки чи ресурсу, де цей код зберігається. Ця вразливість виникає, коли дані, отримані з ненадійних джерел, зберігаються у базі даних, файловій системі або іншому постійному сховищі без належної перевірки, екранування або санітизації та потім включаються у відповіді сервера [11]. На відміну від відображеного XSS, збережений міжсайтовий скриптинг залишається в системі протягом тривалого часу, що дозволяє атакувати множину користувачів. Тут не використовуються методи соціальної інженерії. Будь-який користувач, який відвідує сторінку зі збереженим шкідливим кодом, автоматично стає жертвою. Це може бути особливо небезпечно, якщо ними стануть адміністратори: тоді зловмисник може отримати доступ до інформації з обмеженим доступом, змінювати налаштування системи або розширювати масштаб атаки на інші компоненти інфраструктури.

Загальний механізм реалізації збереженої XSS-атаки можна побачити на рис. 1.5.

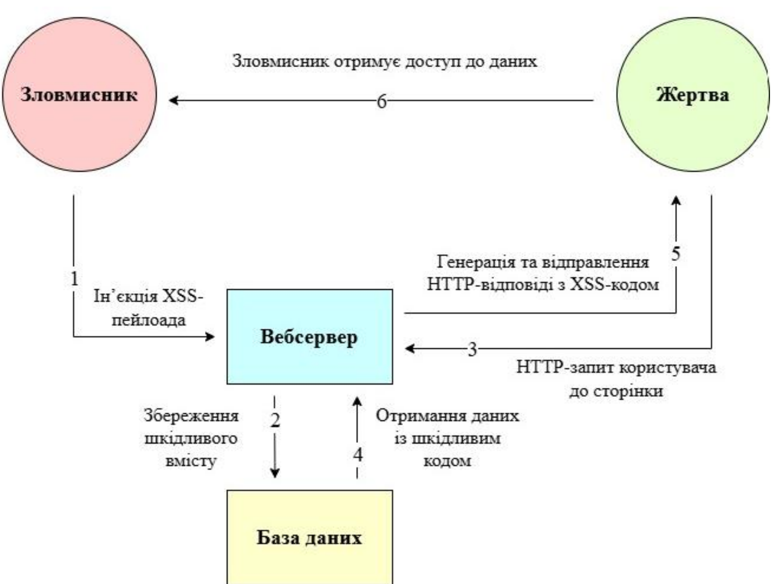


Рисунок 1.5 – Схема реалізації атаки типу Stored XSS

Схема вище (рис. 1.5) містить у собі 6 основних етапів:

1) Зловмисник впроваджує шкідливий JavaScript-код через один із інтерактивних елементів вебзастосунку (коментарі, профілі користувачів, поля форм), який приймає дані від користувачів. На цьому етапі відбувається формування корисного навантаження, розрахованого на експлуатацію недоліків валідації на стороні сервера.

2) Вебсервер отримує дані з шкідливим кодом і, через відсутність належних механізмів санітизації та валідації вхідних даних, зберігає шкідливе навантаження у систему зберігання даних (реляційну базу даних, NoSQL-сховище або файлову систему).

3) Жертва здійснює легітимний HTTP-запит до сторінки вебдодатку, де було збережено шкідливий контент. На цьому етапі жертва не виконує жодних підозрілих дій та використовує стандартну функціональність застосунку.

4) Вебсервер, обробляючи запит жертви, звертається до бази даних для отримання необхідного контенту, який включає збережений шкідливий JavaScript-код. База даних повертає інформацію без жодної додаткової обробки.

5) Вебсервер генерує HTTP-відповідь для клієнта, інтегруючи отримані з бази даних елементи (включно зі шкідливим кодом) у динамічно сформований

HTML-документ. Через відсутність коректного кодування спеціальних символів, шкідливий код залишається інтерпретованим як легітимний JavaScript.

6) Браузер жертви, отримавши HTTP-відповідь, обробляє HTML-документ та автоматично виконує вбудований JavaScript-код у межах довіреного домену. Це надає шкідливому коду доступ до об'єктів DOM, сесійних даних, токенів автентифікації та можливість здійснювати запити з привілеями автентифікованого користувача, передаючи викрадені дані безпосередньо зловмиснику.

Крім класичного пейлоаду, що наводився у пункті 1.3.1, для демонстрації вразливості типу Stored XSS можна представити інше шкідливе навантаження, яке використовує альтернативні вектори впровадження коду (рис. 1.6):

```
<img src="зображення.jpg" onerror="alert('Вас зламали!');">.
```

Рисунок 1.6 – Приклад XSS-пейлоада через обробник події onerror

Цей пейлоад використовує обробник події onerror елемента img для виконання JavaScript-коду при спробі завантаження неіснуючого зображення. Такий підхід дозволяє обійти фільтри захисту, які блокують лише теги script, але пропускають інші HTML-елементи з обробниками подій. При збереженні такого коду в базі даних, кожен відвідувач сторінки отримає спливаюче вікно, що підтверджує успішну експлуатацію постійного XSS.

Більш складний та небезпечний пейлоад може використовувати техніки обфускації, приховувати свою присутність та виконувати шкідливі дії. Наприклад (див. рис. 1.7):

```
<img src="x" onerror="(function(){var i=document.createElement('img');
i.src='https://attacker-server.com/steal?cookie='+encodeURIComponent(document.
cookie);var s=document.createElement('script');s.src='https://attacker-server.com/
backdoor.js';document.body.appendChild(s);})()">
```

Рисунок 1.7 – Приклад складного пейлоада із викраденням cookies та завантаженням додаткового скрипта

Цей пейлоад використовує техніку обфускації та кілька векторів атаки одночасно: спочатку він створює невидимий елемент зображення з неіснуючим джерелом, що викликає подію помилки; в обробнику цієї події запускається анонімна функція, яка надсилає cookies жертви на сервер зловмисника, а потім завантажує додатковий шкідливий скрипт, який може встановити постійний «бекдор» для подальших атак. Такий пейлоад не тільки викрадає конфіденційні дані, але й дозволяє зловмиснику розширювати масштаб атаки та підтримувати довготривалий доступ до скомпрометованих сесій користувачів.

Вразливості типу Stored XSS призвели до кількох серйозних інцидентів безпеки, задокументованих у системі CVE. У Ghost CMS (CVE-2024-23724) було виявлено вразливість, що дозволяла завантаження шкідливих SVG-файлів, які зберігались у профілях користувачів і могли виконувати JavaScript при перегляді, створюючи ризик захоплення облікових записів адміністраторів. У системі моніторингу Cacti (CVE-2024-43365) зловмисник міг вставити шкідливий код у параметр consolenewsection, що призводило до автоматичного виконання скрипту при відкритті відповідної сторінки, створюючи загрозу внутрішнім адміністративним панелям. В ERP-системі Holded (CVE-2025-1076) вразливість у полях name та icon дозволяла зберігати та виконувати JavaScript, що могло бути використано для викрадення даних або впровадження довільного контенту в інтерфейс програми. Усі ці випадки демонструють, як постійні XSS-вразливості можуть мати серйозні наслідки для безпеки, коли шкідливий код стає частиною довіреного інтерфейсу та впливає на широке коло користувачів.

Процес виявлення Stored XSS часто ускладнюється їхньою відкладеною природою, оскільки ефект від впровадження шкідливого коду може проявитися не одразу, а через деякий час при перегляді контенту іншими користувачами. Це створює додаткові проблеми для тестування безпеки вебдодатків, оскільки автоматизовані інструменти не завжди можуть відстежити виконання коду в різних контекстах та часових проміжках.

1.3.3 DOM-based XSS

DOM-based XSS є особливим типом міжсайтового скриптингу, що суттєво відрізняється від інших різновидів за механізмом виникнення та реалізації. Ця вразливість виникає, коли клієнтський JavaScript-код модифікує DOM сторінки небезпечним чином, використовуючи дані, що контролюються користувачем, без належної перевірки та санітизації. Особливо часто такі вразливості зустрічаються в SPA, про які йшла мова у підрозділі 1.1, де значна частина логіки обробки даних виконується на стороні клієнта.

На відміну від відображеного та збереженого XSS, які виникають внаслідок вставки шкідливого коду у відповідь сервера, DOM-based XSS повністю виконується на стороні клієнта [12]. Вразливість виникає, коли скрипти на сторінці динамічно обробляють дані з ненадійних джерел (наприклад, URL-параметри, фрагменти URL, дані з localStorage чи sessionStorage) і небезпечно вставляють їх у DOM, що дозволяє виконувати довільний JavaScript-код. Це створює додаткові складнощі для виявлення такого типу вразливостей стандартними інструментами статичного аналізу коду або звичайними сканерами безпеки, які не виконують JavaScript.

Механізм реалізації DOM-based XSS можна зобразити як на рис. 1.8.

Рисунок 1.8 – Схема реалізації DOM-based XSS-атаки

Детальніше кожен етап (див. рис.1.8) можна пояснити таким чином:

1) Зловмисник створює спеціально сформований URL, який містить XSS-пейлоад у фрагменті (наприклад, після #) або параметрах запиту. Метою є передача даних, які згодом будуть оброблені клієнтським JavaScript-кодом без належної валідації. Даний URL поширюється за допомогою методів соціальної інженерії.

2) Жертва переходить за отриманим посиланням, не підозрюючи про його шкідливу природу. Після цього браузер формує HTTP-запит до вебдодатку, водночас фрагмент URL (наприклад, #payload) не передається серверу, залишаючись доступним лише на стороні клієнта.

3) Вебдодаток повертає звичайну HTML-сторінку без урахування або обробки шкідливого фрагмента. Таким чином, серверна частина залишається поза межами атаки.

4) Після отримання відповіді браузер виконує клієнтський JavaScript-код, який зчитує дані з URL (location.href, location.hash, document.URL тощо) та вставляє їх у DOM без достатньої санітизації. Внаслідок цього пейлоад виконується у контексті вебсторінки, що відкриває доступ до об'єктів document, cookie та інших конфіденційних даних.

5) Виконаний XSS-пейлоад ініціює відправлення перехопленої інформації (наприклад, сесійних токенів, введених облікових даних тощо) на зовнішній сервер, контрольований зловмисником.

Для розуміння технічних аспектів DOM-based XSS можна розглянути приклади вразливого JavaScript-коду та відповідних пейлоадів. Припустимо, вебсторінка очікує параметр default у URL, який початково використовується для встановлення мови за замовчуванням, наприклад, http://www.some.site/page.html?default=Ukrainian. Зловмисник може експлуатувати цю вразливість, надіславши жертві модифіковане посилання (див. рис. 1.9), яке впроваджує шкідливий JavaScript-код [12]:

```
http://www.some.site/page.html?default=<script>alert(document.cookie)</script>
```

Рисунок 1.9 – Приклад експлуатації DOM-based XSS через URL-параметр

У цьому випадку при завантаженні сторінки виконається скрипт, що викликає спливаюче вікно з cookies поточної сесії, демонструючи потенційну можливість несанкціонованого доступу до конфіденційних даних користувача.

Також можна навести складніший приклад, пов'язаний з JSON-парсингом даних з hash-фрагменту URL, продемонстрований на рис. 1.10:

```
$(document).ready(function() {
  var userData = JSON.parse(decodeURIComponent(location.hash.substring(1)));
  $("#userProfile").html("Користувач: " + userData.username);
});
```

Рисунок 1.10 – Приклад вразливого коду з небезпечною обробкою location.hash

Цей код намагається вставити властивість username у DOM. Зловмисник може експлуатувати цю вразливість, створивши посилання з шкідливим hash-значенням, наведеним на рис. 1.11.

```
https://example.com/profile.html#{"username":"<img src='x' onerror='fetch(\"https:
//attacker.com/steal?cookie=\"+document.cookie)'>"}
```

Рисунок 1.11 – Приклад DOM-based XSS пейлоада через hash-фрагмент URL

При відвідуванні такого URL, вставка небезпечного HTML-контенту призведе до спроби завантаження зображення, якого не існує, і, як наслідок, до виконання функції-обробника помилки, яка відправить cookies користувача на сервер зловмисника.

Вразливості типу DOM-based XSS становлять значну загрозу для сучасних вебдодатків, особливо з розвитком SPA та фреймворків, що активно

використовують клієнтський JavaScript. Кілька задокументованих випадків у базі CVE підтверджують актуальність цієї проблеми. Наприклад:

- CVE-2023-2318

У редакторі Markdown MarkText (версії до 0.17.1) виявлено DOM-based XSS у модулі pasteCtrl.js. Вразливість дозволяла виконання JavaScript-коду при вставці скопійованого зловмисного тексту, що могло призвести до виконання коду в контексті програми.

- CVE-2023-6790

У вебінтерфейсі PAN-OS виявлено DOM-based XSS, що дозволяло віддаленому зловмиснику виконати JavaScript-код у браузері адміністратора при перегляді спеціально сформованого посилання. Вразливість усунено у версіях PAN-OS 8.1.25 та новіших.

- CVE-2023-2317

У редакторі Typora (версії до 1.6.7) виявлено DOM-based XSS, що дозволяло виконання JavaScript-коду при відкритті спеціально сформованого markdown-файлу. Це могло призвести до виконання шкідливого коду в контексті програми.

Виявлення DOM-based XSS у вебдодатках вимагає динамічного аналізу з урахуванням специфіки виконання клієнтського JavaScript-коду та механізмів передачі даних між джерелами і приймачами в DOM. Це створює додаткові виклики для автоматизованих засобів тестування безпеки, які повинні враховувати контекст виконання скриптів у браузері. З метою підвищення точності виявлення таких вразливостей доцільним є використання headless браузерів, що імітують реальне середовище виконання вебдодатку та дозволяють підтвердити можливість практичної експлуатації, мінімізуючи кількість хибнопозитивних результатів.

1.4 Методи захисту від XSS-вразливостей

Захист від вразливостей типу Cross-Site Scripting потребує комплексного підходу, який охоплює різні рівні вебдодатку та базується за принципом глибокої оборони (Defense in Depth). Цей підхід гарантує, що навіть при компрометації одного рівня інші шари продовжуватимуть забезпечувати безпеку, знижуючи ймовірність успішної атаки на всіх етапах взаємодії з вебдодатком [13]. Нижче наведено основні методи, що є актуальними у сучасній практиці веброзробки.

Санітизація вхідних даних є одним з основних методів запобігання XSS-атакам. Цей процес включає в себе модифікацію або видалення небезпечних символів у введених даних, щоб вони не могли бути інтерпретовані як частина шкідливого коду. Один із способів санітизації – це екранування символів, наприклад, заміна символу "<" на його HTML-сутність "<", що дозволяє браузеру відобразити цей символ, а не інтерпретувати його як початок HTML-тегу [14]. Важливим аспектом є те, що санітизація повинна враховувати контекст використання даних (HTML, JavaScript, URL, CSS), оскільки кожен з них має свої вимоги щодо екранування. Особливо критичним даний метод є для Reflected XSS, коли шкідливий код передається через параметри URL, та для Stored XSS, коли зловмисний код зберігається в базі даних і згодом відображається всім користувачам.

Валідація вхідних даних слугує додатковим механізмом перевірки введеної інформації на відповідність очікуваним форматам. Наприклад, якщо поле форми повинно містити лише числове значення, валідація забезпечує, що будь-які нечислові символи будуть відхилені. Валідація може виконуватися як на стороні клієнта (для покращення зручності користувача), так і на стороні сервера (для забезпечення безпеки). Перша може бути легко обійдена, наприклад, шляхом вимкнення JavaScript або використання проксі-інструментів, як Burp Suite, тому для ефективного захисту від XSS серверна валідація є обов'язковою [15]. Для DOM-based XSS, який виникає при небезпечній обробці

даних безпосередньо в DOM-структурі, необхідна додаткова валідація даних на клієнтській стороні перед їх обробкою JavaScript-кодом.

Кодування вихідних даних (output encoding) забезпечує безпечне відображення даних у відповіді сервера шляхом кодування потенційно небезпечних символів відповідно до контексту їх використання. Наприклад, для HTML-контексту застосовується HTML-кодування, для JavaScript – JavaScript-кодування і т.д. Сучасні фреймворки та бібліотеки часто надають вбудовані функції для автоматичного кодування вихідних даних, що зменшує ризик помилок розробників. Цей метод особливо ефективний проти Reflected та Stored XSS.

Content Security Policy (CSP) є потужним інструментом для пом'якшення наслідків XSS-атак. Політика CSP реалізується шляхом передачі відповідного HTTP-заголовка або за допомогою HTML-метатега, містить набір директив, кожна з яких регламентує дозволені джерела для певних типів контенту (наприклад, скриптів, таблиць стилів, зображень тощо).

Основною перевагою CSP є її здатність нейтралізувати ключові вектори XSS-атак, зокрема завантаження скриптів із неавторизованих джерел, виконання inline-коду без явного дозволу, використання javascript:-URL, inline-обробників подій, а також виклики потенційно небезпечних API, як-от eval(). Навіть у випадку успішного обходу механізмів санітизації, CSP виступає як додатковий рівень захисту, що значно ускладнює реалізацію атак.

У сучасних практиках реалізації CSP широко застосовуються криптографічні nonce-токени або геш-суми, які дозволяють авторизувати виконання лише визначених скриптів. Наприклад, директива script-src 'nonce-...' 'strict-dynamic' забезпечує запуск лише тих скриптів, що містять відповідний nonce-атрибут, згенерований динамічно для кожного запиту [16]. У результаті, навіть впроваджений сторонній код буде заблокований браузером.

CSP є ефективним захистом від різних типів XSS: для Reflected та Stored XSS вона обмежує можливість завантаження та виконання шкідливого коду, навіть якщо його вдалося розмістити на сторінці; для DOM-based XSS

застосування директив, що забороняють unsafe-inline та unsafe-eval, значно зменшує ризики виконання динамічно згенерованого шкідливого JavaScript-коду.

У загальному для ефективного захисту від XSS-вразливостей необхідно комбінувати різні методи та підходи, створюючи багаторівневу систему захисту. Жоден окремий метод не забезпечує повного захисту, але їх комбінація значно підвищує безпеку вебдодатку та ускладнює успішну експлуатацію потенційних вразливостей.

Висновки за розділом 1

У результаті написання 1 розділу було встановлено основні причини виникнення XSS-вразливостей у сучасних вебдодатках та визначено їх характерні особливості.

1. Дослідження архітектури вебдодатків показало, що трирівнева структура (рівень презентації, рівень додатку та рівень зберігання даних) створює множинні точки входу для потенційних атак. Особливо вразливими є інтерфейси взаємодії між рівнями, де відбувається обробка користувацьких даних без належної валідації. Еволюція архітектурних підходів від монолітних систем до мікросервісної архітектури, хоча й покращує масштабованість та гнучкість, водночас розширює поверхню атаки через збільшення кількості API-інтерфейсів та точок взаємодії між сервісами.

2. Аналіз механізмів виникнення XSS-вразливостей виявив три основні причини: недостатню валідацію вхідних даних, неналежну санітизацію виводу та неправильне використання DOM-маніпуляцій. Особливу увагу привертає той факт, що XSS залишається актуальною загрозою, посідаючи друге місце у рейтингу CWE Top 25 Most Dangerous Software Weaknesses за 2024 рік та входячи до категорії A03:2021-Injection у списку OWASP Top 10, що підтверджує необхідність розробки ефективних методів їх виявлення.

3. Класифікація XSS-вразливостей на три основні типи – Reflected, Stored та DOM-based – дозволила ідентифікувати специфічні особливості кожного різновиду. Reflected XSS характеризується непостійністю та вимагає соціальної інженерії для експлуатації, тоді як Stored XSS створює найбільшу загрозу через можливість атакувати множину користувачів без додаткових зусиль зловмисника. DOM-based XSS набуває особливої актуальності у контексті SPA, де значна частина логіки обробки даних виконується на стороні клієнта, що створює додаткові виклики для традиційних методів виявлення вразливостей.

4. Дослідження методів захисту від XSS-атак продемонструвало необхідність комплексного підходу, що включає санітизацію вхідних даних, валідацію на серверній стороні, кодування вихідних даних та впровадження CSP. Особливо важливим є контекстно-залежне екранування, оскільки різні контексти (HTML, JavaScript, URL, CSS) мають специфічні вимоги до обробки потенційно небезпечних символів.

5. Виявлено, що сучасні тенденції веброзробки, зокрема широке використання JavaScript-фреймворків та прогресивних вебдодатків, створюють нові вектори атак та ускладнюють процес виявлення DOM-based XSS-вразливостей. Це обумовлює необхідність розробки спеціалізованих інструментів, здатних аналізувати динамічну поведінку JavaScript-коду в реальному браузерному середовищі для підвищення точності виявлення та зменшення кількості хибнопозитивних результатів.

РОЗДІЛ 2
АНАЛІЗ І ПОРІВНЯННЯ ІНСТРУМЕНТІВ ДЛЯ ВИЯВЛЕННЯ XSS-ВРАЗЛИВОСТЕЙ

2.1 Аналіз підходів до виявлення XSS-вразливостей

Виявлення міжстайтового скриптингу потребує комплексного підходу, який включає різні методи аналізу вебзастосунків. Сучасні способи пошуку таких вразливостей зазвичай поділяють залежно від того, як саме аналізується код і в якому середовищі проводиться тестування.

Статичний аналіз коду (SAST, англ. Static Application Security Testing) представляє собою процес дослідження вихідного коду додатка без його фактичного виконання. Цей підхід дозволяє виявляти потенційні вразливості на етапі розробки шляхом аналізу структури коду, потоків даних та можливих шляхів виконання програми. SAST-інструменти, такі як SonarQube, CxSAST та Veracode, сканують статичний код та ідентифікують місця, де користувацькі дані можуть потрапити в HTML-вихід без належної валідації. Основною перевагою SAST є можливість раннього виявлення проблем безпеки, що значно знижує вартість їх усунення [17]. Проте статичний аналіз має суттєві обмеження при роботі з динамічними вебтехнологіями, оскільки не може врахувати поведінку додатка під час виконання та взаємодію з браузером користувача.

Динамічний аналіз (DAST, англ. Dynamic Application Security Testing) базується на тестуванні додатка шляхом надсилання спеціально сформованих запитів та аналізу отриманих відповідей. Традиційні DAST-інструменти аналізують HTTP-трафік та шукають ознаки успішного впровадження пейлоада в HTML-відповідях сервера [18]. Проте такий підхід має обмеження при виявленні DOM-based XSS-вразливостей, які виконуються виключно в контексті браузера без участі сервера. Прикладами сучасних DAST-сканерів є Acunetix, Burp Suite та Invicti.

Headless браузери як розгалуження сучасного DAST-підходу представляють собою повнофункціональні веббраузери без графічного інтерфейсу, які можуть виконувати JavaScript-код, обробляти DOM, взаємодіяти з CSS та імітувати всі аспекти роботи звичайного браузера. Основною перевагою таких браузерів є їх здатність виконувати JavaScript-код в реальному контексті DOM, що дозволяє точно визначити, чи призводить впровадження шкідливого навантаження до фактичного спрацювання скрипта. Це суттєво відрізняє їх від традиційних DAST-сканерів. Відомими інструментами для автоматизованої роботи з headless браузерами є:

- Playwright, розроблений командою Microsoft, підтримує автоматизацію Chromium, Firefox та WebKit браузерів через спільний API. Цей інструмент забезпечує надійну роботу з сучасними вебтехнологіями, включаючи Shadow DOM, SPA та PWA [19]. Playwright автоматично очікує завантаження елементів та підтримує паралельне виконання тестів, що значно підвищує швидкість сканування великих додатків.
- Puppeteer, розроблений командою Google Chrome, тісно інтегрується з Chrome DevTools Protocol. Це дозволяє отримати докладну інформацію про виконання JavaScript-коду, мережеву активність і зміни в DOM [20]. Також можна відслідковувати повідомлення в консолі та JavaScript-помилки, що допомагає виявляти успішне виконання XSS-атаки. Puppeteer підтримує емуляцію різних пристроїв і умов мережі.
- Selenium WebDriver може працювати в headless режимі з різними браузерами через стандартизований W3C WebDriver протокол [21]. Завдяки великій кількості бібліотек і розширень Selenium легко інтегрується в CI/CD-процеси, що робить його зручним для використання в реальних умовах розробки.

На основі аналізу документацій даних інструментів було створено порівняльну таблицю, наведену нижче (див. табл. 2.1).

Таблиця 2.1

Порівняння headless інструментів для виявлення XSS-вразливостей

Характеристика	**Playwright**	**Puppeteer**	**Selenium WebDriver**
Браузери, що підтримуються	Chromium, Firefox, WebKit	Chromium/Chrome	Chrome, Firefox, Safari, Edge
Виконання JavaScript	Повна підтримка ES2022	Chrome V8 engine	Залежить від браузера
Консольні повідомлення	page.on('console')	page.on('console')	driver.get_log ('browser')
Робота з DOM	Нативний API для DOM	Chrome DevTools Protocol	WebDriver commands
Shadow DOM	Повна підтримка	Через DevTools API	Обмежена підтримка

З наведеного порівняння видно, що Playwright забезпечує найширшу підтримку сучасних вебтехнологій, включаючи повноцінну роботу з Shadow DOM та всіма основними браузерами. Puppeteer тісно інтегрований з Chromium і добре підходить для аналізу JavaScript-коду, хоча його підтримка інших браузерів обмежена. Selenium WebDriver залишається найгнучкішим з точки зору сумісності, однак має обмеження у роботі з новими технологіями та залежить від можливостей конкретного браузера.

Підходом, що поєднує елементи статичного та динамічного підходів є інтерактивний аналіз безпеки додатків (IAST, англ. Interactive Application Security Testing). IAST-інструменти, такі як Contrast Security та Synopsys Seeker, інтегруються безпосередньо в середовище виконання додатка та моніторять його поведінку в реальному часі під час тестування або експлуатації [22]. Цей підхід забезпечує більш точне виявлення вразливостей завдяки можливості спостереження за фактичним виконанням коду та потоками даних. Водночас інтеграція IAST-інструментів потребує втручання у внутрішню структуру

додатка, що не завжди є доцільним у умовах обмежень стабільності або безперервної роботи сервісу.

Ще одним підходом до виявлення вразливостей XSS є ручна перевірка Досвідчені фахівці з безпеки можуть виявляти складні логічні вразливості, які не детектуються автоматизованими засобами. Ручне тестування особливо ефективне для виявлення контекстно-залежних XSS-вразливостей, які вимагають глибокого розуміння бізнес-логіки додатка та специфічних сценаріїв використання. Аналітики використовують спеціалізовані пейлоади для різних контекстів, включаючи HTML-атрибути, JavaScript-рядки та CSS-стилі.

Різні підходи до тестування безпеки демонструють різну ефективність при виявленні XSS-вразливостей, що пов’язано з особливостями їх реалізації у вебдодатках. Узагальнюючи, у таблиці нижче (табл. 2.2) подано відповідність між типами XSS і методами, які найкраще підходять для їх виявлення.

Таблиця 2.2

Ефективність підходів до виявлення XSS-вразливостей

Тип XSS	**Рекомендовані методи**	**Причина ефективності**
Reflected XSS	SAST, DAST, ручне тестування	SAST виявляє уразливі фрагменти коду; DAST виявляє ін'єкції у відповідях сервера; ручне тестування враховує складні та контекстні випадки
Stored XSS	SAST, IAST, ручне тестування	SAST знаходить місця збереження даних без валідації; IAST відстежує потоки введення під час виконання; ручне тестування – для логічних вразливостей

Продовження табл. 2.2

DOM-based XSS	DAST з headless браузерами, ручне тестування	Headless-браузери виявляють клієнтські ін'єкції в DOM; ручний підхід дозволяє аналізувати складну взаємодію з JavaScript та динамічним контентом

У практичній площині, вибір підходу до виявлення XSS-вразливостей має ґрунтуватися на специфіці додатка, що досліджується, доступних ресурсах та вимогах до точності результатів. Комплексний підхід, що поєднує різні методи аналізу та враховує особливості різних типів XSS-вразливостей, дозволяє досягти оптимального балансу між покриттям кодової бази, мінімізацією хибнопозитивних результатів та ефективним використанням ресурсів.

2.2 Огляд сучасних сканерів вразливостей

Сучасні вебзастосунки мають складну архітектуру та часто виконують важливі функції в бізнес-процесах організацій. Через це питання їх інформаційної безпеки стає все більш актуальним. У цьому контексті важливу роль відіграють сканери вразливостей – програми, які автоматично перевіряють вебдодатки на наявність потенційних загроз, зокрема XSS-вразливостей. Наразі на ринку представлено широкий спектр подібних інструментів – від комерційних продуктів із розширеним функціоналом до безкоштовних рішень із відкритим кодом, які зазвичай орієнтовані на виконання окремих завдань. Їх можна класифікувати таким чином:

- Комерційні сканери – надають комплексне рішення для виявлення широкого спектра вразливостей, включаючи XSS. До цієї категорії належать, напркилад, Acunetix, Burp Suite Professional, Invicti, AppScan та Veracode.

- Відкриті (open-source) рішення – безкоштовні рішення з відкритим кодом, які можна інтегрувати в CI/CD-конвеєри для регулярного тестування безпеки. Прикладами є OWASP ZAP, Wapiti, Nikto та w3af.
- Спеціалізовані інструменти для виявлення XSS – зосереджені саме на пошуку XSS-вразливостей, що часто забезпечує більшу точність при виявленні складних випадків. Прикладами є XSStrike, Dalfox, XSSer.

2.2.1 Аналіз комерційних сканерів вразливостей

Комерційні сканери вразливостей демонструють різні рівні ефективності у виявленні XSS-атак, маючи як суттєві переваги, так і обмеження. Acunetix відзначається високою точністю виявлення DOM-based XSS завдяки технології DeepScan, яка здатна аналізувати JavaScript-фреймворки Angular, React та Vue.js. Основною перевагою цього рішення є низький рівень хибнопозитивних результатів та комплексний підхід до тестування, що підтверджується найвищим балом 0,81 у дослідженні WASSEC [23]. Проте недоліком Acunetix є висока вартість ліцензування та складність налаштування для специфічних середовищ.

Invicti характеризується унікальною технологією Proof-Based Scanning, яка автоматично верифікує виявлені XSS-вразливості, значно зменшуючи кількість хибних спрацьовувань [24]. Ця особливість робить інструмент особливо цінним для великих організацій, де ручна перевірка результатів потребує значних ресурсів. Водночас недоліком є обмежена гнучкість у налаштуванні правил сканування та складність інтеграції з нестандартними системами розробки.

Burp Suite Professional забезпечує високу гнучкість завдяки модульній архітектурі та можливості використання спеціалізованих плагінів, зокрема DOM Invader для покращеного виявлення DOM-based XSS. Серед основних переваг варто відзначити розширені можливості налаштування та активну підтримку з боку спільноти розробників. Водночас використання цього інструменту потребує відповідної технічної підготовки, а швидкість автоматизованого сканування може поступатися деяким альтернативним рішенням.

2.2.2 Аналіз інструментів з відкритим вихідним кодом

Інструменти з відкритим кодом демонструють суперечливі результати у виявленні XSS-вразливостей, що підтверджується дослідженнями Мбурано та Сі [25]. OWASP ZAP виявляє значні переваги у вигляді безкоштовного використання та активної підтримки спільноти, досягаючи 76% ефективності у виявленні XSS за OWASP Benchmark та 100% за бенчмарком WAVSEP [26]. Проте основним недоліком ZAP є складність налаштування для новачків та обмежені можливості виявлення DOM-based XSS без додаткових налаштувань.

Arachni демонструє нижчі показники – 64% за OWASP Benchmark та 91% за WAVSEP, що пояснюється його архітектурними особливостями [26]. Перевагою Arachni є модульна структура та можливість розподіленого сканування, що дозволяє обробляти великі вебдодатки. Недоліками є нестабільна робота з додатками зі складною клієнтською логікою та припинення активної розробки проекту, що ставить під сумнів довгострокову підтримку.

Суттєва розбіжність результатів між бенчмарками OWASP та WAVSEP свідчить про обмеження автоматизованих інструментів у виявленні складних XSS-сценаріїв. Це підкреслює необхідність комбінування автоматизованого сканування з ручним тестуванням для досягнення комплексного покриття безпеки.

2.2.3 Аналіз спеціалізованих інструментів для виявлення XSS

XSStrike вирізняється своєю здатністю до глибокого аналізу відповіді сервера за допомогою ручних парсерів HTML та JavaScript, що дозволяє йому виявляти складні XSS-уразливості, включаючи DOM-based XSS [27]. Основною перевагою є адаптивність до різних фільтрів та захисних механізмів, що дозволяє успішно виявляти вразливості, незважаючи на застосування сучасних засобів протидії XSS. Недоліком є вузька спеціалізація, що не дозволяє виявляти інші типи вразливостей, та залежність від Python-середовища.

Dalfox демонструє технологічні переваги завдяки використанню Go та комбінації фазингу з мутаційним тестуванням, забезпечуючи 100% ефективність для Reflected та Stored XSS. Його перевагами є висока швидкість роботи та мінімальні системні вимоги. Проте критичним недоліком є низька ефективність виявлення DOM-based XSS, що обмежує його застосування у тестуванні сучасних JavaScript-додатків.

Враховуючи викладене вище, спеціалізовані інструменти доцільно використовувати як доповнення до універсальних сканерів для підвищення загальної ефективності виявлення XSS-вразливостей. Це особливо важливо з огляду на їх обмежену функціональну область покриття.

2.3 Оцінка ефективності виявлення XSS-вразливостей сканерами

Розглянувши різноманітні підходи до виявлення XSS-вразливостей та проаналізувавши сучасні сканери безпеки, важливо встановити об'єктивні критерії для оцінки їх ефективності. На основі них були побудовані деякі дослідження розглянуті у підрозділі 2.2. Такий аналіз дозволяє визначити переваги та недоліки інструментів, що є критичним для вибору оптимальних рішень при забезпеченні безпеки вебдодатків.

Основними показниками є:

- True Positive (TP) – кількість правильно виявлених вразливостей
- False Positive (FP) – кількість хибнопозитивних результатів (помилкових спрацювань)
- True Negative (TN) – кількість правильно визначених захищених елементів
- False Negative (FN) – кількість пропущених вразливостей

На основі цих базових показників формуються ключові метрики для оцінки ефективності [28]:

- Точність (Precision) – відношення правильно виявлених вразливостей до загальної кількості виявлених вразливостей:

$$\text{Precision} = \frac{TP}{TP+FP} \tag{2.1}$$

- Повнота (Recall) – відношення правильно виявлених вразливостей до загальної кількості наявних вразливостей:

$$\text{Recall} = \frac{TP}{TP+FN} \tag{2.2}$$

- F-міра (F-measure) – гармонійне середнє між точністю та повнотою:

$$F = 2 \cdot \frac{\text{Precision}\cdot\text{Recall}}{\text{Precision}+\text{Recall}} = \frac{2\cdot TP}{2\cdot TP+FP+FN} \tag{2.3}$$

- Загальна точність (Accuracy) – відношення правильно класифікованих результатів до загальної кількості тестів:

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN} \tag{2.4}$$

- Коефіцієнт Йодена (Youden's index) – показник балансу між чутливістю та специфічністю:

$$J = \text{Sensitivity} + \text{Specificity} - 1 = \frac{TP}{TP+FN} + \frac{TN}{TN+FP} - 1 \tag{2.5}$$

Для забезпечення об'єктивності порівняння різних сканерів важливо використовувати стандартизовані бенчмарки. Одним з найбільш визнаних є OWASP Benchmark, який містить тисячі тестових випадків для різних типів вразливостей, включаючи XSS. Якраз у підрозділі 2.2 було зазначено, що на ній

проводилось тестування різних сканерів вразливостей. Ця методика базується на обчисленні показників True Positive Rate (TPR) та False Positive Rate (FPR) [29]:

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP+FN}} \tag{2.6}$$

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP+TN}} \tag{2.7}$$

Інтегральним показником в методології OWASP Benchmark є Triage Score, який обчислюється як різниця між TPR та FPR:

$$\text{Triage score} = \mathrm{TPR} - \mathrm{FPR} \tag{2.8}$$

Діапазон значень Triage Score знаходиться в межах від -1 до 1. Значення близькі до 1 свідчать про високу ефективність інструменту, значення близькі до 0 відповідають випадковому вгадуванню, а значення нижче 0 вказують на неефективність інструменту (гірше за випадкове вгадування).

Аналіз результатів тестування різних сканерів за методологією OWASP Benchmark показує значні розбіжності в ефективності виявлення XSS-вразливостей. Зокрема, традиційні DAST-сканери часто демонструють високий рівень FP результатів при аналізі DOM-based XSS, оскільки не можуть точно визначити, чи призводить впровадження пейлоада до фактичного виконання шкідливого коду в браузері.

Ця проблема особливо актуальна для сучасних вебдодатків, що використовують складні JavaScript-фреймворки та динамічно генерують контент (наприклад, SPA). Саме тому використання headless браузерів як частини процесу сканування може значно покращити точність виявлення вразливостей та зменшити кількість хибнопозитивних результатів завдяки можливості реального виконання JavaScript-коду в контексті браузера.

Висновки за розділом 2

У результаті проведення аналізу у 2 розділі було систематизовано сучасні підходи до виявлення XSS-вразливостей та оцінено ефективність наявних інструментів сканування безпеки вебдодатків.

1. Визначено, що статичний аналіз коду забезпечує раннє виявлення потенційних вразливостей на етапі розробки, однак має обмеження при роботі з динамічними вебтехнологіями. Динамічний аналіз ефективний для виявлення Reflected та Stored XSS, проте традиційні DAST-інструменти демонструють низьку точність при детекції DOM-based вразливостей через неможливість аналізу клієнтського JavaScript-коду.

2. Встановлено, що headless браузери представляють найбільш перспективний підхід для виявлення всіх типів XSS-вразливостей, оскільки забезпечують реальне виконання JavaScript-коду в контексті браузера. Серед аналізованих інструментів Playwright демонструє найширшу підтримку сучасних вебтехнологій, включаючи Shadow DOM та основні браузери, що робить його оптимальним вибором для створення сканера вразливостей.

3. Проаналізовано сучасний ринок сканерів безпеки, який включає комерційні рішення з розширеним функціоналом, інструменти з відкритим кодом для інтеграції в CI/CD-процеси та спеціалізовані засоби для виявлення XSS. Виявлено, що жоден з існуючих інструментів не забезпечує оптимального балансу між швидкістю сканування та точністю результатів для всіх типів XSS-вразливостей.

4. Сформовано систему метрик для об'єктивної оцінки ефективності сканерів, включаючи точність, повноту, F-міру та коефіцієнт Йодена. Визначено методологію OWASP Benchmark як стандартизований підхід для порівняння різних інструментів, що дозволяє отримати об'єктивні результати через використання показника Triage Score.

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМНОГО МОДУЛЯ ДЛЯ ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ ТИПУ XSS

3.1. Опис програмного модуля

Програмний модуль для виявлення XSS-вразливостей був написаний мовою програмування Python. Вона є зручною для швидкої розробки, має зрозумілий синтаксис і підтримує велику кількість готових бібліотек для роботи з вебом та автоматизації браузера.

Основою сканера є використання headless браузера Chromium через бібліотеку Playwright, що дозволяє емулювати поведінку реального користувача та точніше відслідковувати виконання JavaScript-коду в вебдодатках. Програмний модуль містить забезпечує виявлення усіх трьох типів XSS, описаних у розділі 1, там, де вони дійсно існують.

3.2 Архітектура та реалізація програмного модуля

Архітектура програмного модуля побудована навколо центрального класу XSSScanner, який містить основну логіку сканування та керує взаємодією між іншими компонентами системи. Основні функціональні блоки включають модуль ініціалізації браузера, систему збору URL-адрес, аналізатор форм, генератор пейлоадів та систему підтвердження вразливостей. Загальна структура роботи сканера зображена у додатку А.

Метод start_scan() є головною точкою входу, яка ініціалізує Chromium з оптимізованими параметрами для підвищення швидкості роботи. Браузер запускається з відключеними функціями безпеки та фонової обробки, що дозволяє ефективніше тестувати потенційно небезпечний контент. Це показано на рис. 3.1.

```
async with async_playwright() as playwright:
    browser = await playwright.chromium.launch(
        headless=True,
        args=[
            '--disable-web-security',
            '--disable-features=TranslateUI',
            '--disable-ipc-flooding-protection',
            '--disable-renderer-backgrounding',
            '--disable-backgrounding-occluded-windows',
            '--disable-client-side-phishing-detection'
        ]
    )
```

Рисунок 3.1 – Ініціалізація браузера з оптимізованими параметрами

Збір URL-адрес виконується методом _collect_urls(), який використовує JavaScript-ін'єкції для отримання всіх посилань на сторінці. Це дозволяє виявляти як статичні посилання в HTML, так і динамічно генеровані через JavaScript. Метод включає захист від випадків, коли контекст сторінки може бути знищений під час асинхронних операцій.

Важливою метою даної роботи було оптимізувати швидкість сканування. Одним із підходів було застосування пакетної обробки URL-адрес у методі _crawl_website(). Замість послідовної обробки кожної сторінки окремо, система групує URL-адреси в пакети оптимального розміру та обробляє їх паралельно з використанням семафорів для контролю кількості одночасних з'єднань. Це представлено на рис. 3.2 та 3.3. Також необхідно було також впровадити кешування, щоб пришвидшити сканування. Це було реалізовано через внутрішній словник _dom_check_cache, який зберігає вже перевірені сторінки та дозволяє уникнути повторного аналізу однакових сторінок, а це зі свого боку прискорює роботу коду (див. рис. 3.4). Опис методів з даною оптимізацією представлений у додатку Б.

```
optimal_batch_size = min(self.concurrency * 3, 30)

while urls_to_scan and scanned_count < self.max_urls_to_scan:
    batch_size = min(optimal_batch_size, len(urls_to_scan),
                     self.max_urls_to_scan - scanned_count)
```

Рисунок 3.2 – Пакетна обробка URL-адрес з оптимальним розміром пакету

```
semaphore = asyncio.Semaphore(self.concurrency)

user
async def process_with_semaphore(url):
    async with semaphore:
        return await self._process_url(context, url, urls_to_scan, progress, scan_task)

tasks = [process_with_semaphore(url) for url in unique_batch_urls]
results = await asyncio.gather(*tasks, return_exceptions=True)
```

Рисунок 3.3 – Паралельна обробка з семафором

```
# Використання кешу для уникнення повторного аналізу
try:
    cache_key = f"dom_check_{hash(url)}"
    if hasattr(self, '_dom_check_cache') and cache_key in self._dom_check_cache:
        return self._dom_check_cache[cache_key]

    if not hasattr(self, '_dom_check_cache'):
        self._dom_check_cache = {}
```

Рисунок 3.4 – Кешування для прискорення DOM-аналізу

Аналіз форм здійснюється методом _check_forms_for_xss(), який спочатку збирає інформацію про всі форми на сторінці, включаючи типи полів вводу, методи відправки та атрибути. Система автоматично класифікує тип XSS-вразливості на основі методу форми та наявності пейлоаду в URL після відправки. Для GET-запитів з пейлоадом в URL вразливість класифікується як Reflected XSS, тоді як для POST-запитів проводиться додаткова перевірка на наявність Stored XSS.

Генерація шкідливого навантаження відбувається контекстно-орієнтованим способом через метод _generate_context_aware_payloads(). Система створює різні типи пейлоадів залежно від контексту: для форм використовуються пейлоади, призначені для тестування збережених вразливостей, для URL-параметрів – для відображених, а для DOM-тестування – спеціалізовані пейлоади, що спрацьовують через маніпуляції з DOM. Загальна концепція даної функції зображена на рис. 3.5, а повністю представлена у додатку В. Також там представлений метод _comprehensive_dom_xss_check(), що дозволяє комплексно перевірити DOM-вразливості.

```python
def _generate_context_aware_payloads(self, xss_type: str, context: str = "form") -> List[str]:

    if xss_type == "stored" or context == "form":
        stored_payloads = [
        ]
        return stored_payloads

    elif xss_type == "reflected":
        reflected_payloads = [
        ]
        return reflected_payloads

    elif xss_type == "dom":
        return self._create_targeted_dom_payloads({})
```

Рисунок 3.5 – Контекстно-орієнтована генерація пейлоадів

Щоб зменшити кількість помилкових спрацьовувань, система використовує додаткові методи _get_enhanced_detection_scripts() та _check_xss_execution() (див. додаток Г). У кожну сторінку впроваджується спеціальний JavaScript-код, який перехоплює виклики небезпечних функцій, таких як innerHTML, eval, document.write та alert(). Унікальний ідентифікатор в кожному пейлоаді дозволяє точно визначити, чи був виконаний саме тестовий код (див. рис. 3.6).

```javascript
if (value && typeof value === 'string' &&
value.includes(window.xssIdentifier)) {{
    logDetection('innerHTML', value);
}}
```

Рисунок 3.6 – Механізм виявлення виконання JavaScript-коду

Для виявлення DOM-based XSS окремо використовується метод _analyze_page_for_dom_vulnerabilities(). Він сканує JavaScript-код сторінки та шукає ознаки потенційно небезпечної обробки, зокрема, використання URL-фрагментів, параметрів запиту та функцій, які можуть призвести до XSS. На основі цього аналізу формуються цільові пейлоади для тестування конкретних векторів атаки.

Для виявлення Stored XSS використовується метод _enhanced_stored_xss_check(). Він після відправки даних через форму перевіряє виконання пейлоаду на інших сторінках сайту. Це дає змогу виявити випадки, коли введені дані записуються у базу даних і відображаються пізніше.

Наостанок результати сканування зберігаються у форматі JSON через метод _save_results_to_json(), що включає детальну інформацію про кожну виявлену вразливість, параметри сканування та статистику. Це забезпечує можливість подальшого аналізу та інтеграції з іншими інструментами безпеки.

3.3 Тестування інструменту

Відповідно до чинного українського законодавства, використання сканерів вразливостей не є забороненим, однак, будь-яке сканування вебресурсів без дозволу може кваліфікуватися як несанкціонований доступ або спроба втручання, що регулюється Кримінальним кодексом України, а саме статтею 361 [30]. Тестування допустиме лише на власних або на офіційно дозволених ресурсах. Таким чином для перевірки дієздатності розробленого інструменту було обрано такі тестові середовища:

- https://xss-game.appspot.com/ – інтерактивна навчальна платформа від компанії Google, створена спеціально для вивчення та практичного засвоєння методів виявлення міжсайтового скриптингу. Ресурс містить серію завдань різного рівня складності, що дозволяє поступово освоювати техніки пошуку XSS-вразливостей у контрольованому та безпечному середовищі. Для виконання даного дослідження було взято 1-3 рівні для демонстрації виявлення сканером різних типів XSS.
- http://testhtml5.vulnweb.com/ – спеціалізований тестовий вебдодаток компанії Acunetix, що являє собою SPA з навмисно впровадженими вразливостями різних типів. Ресурс містить SQL-ін'єкції, XSS, CSRF та інші поширені загрози веббезпеки.

Вибір саме цих тестових середовищ обумовлений тим, що вони офіційно створені для навчальних цілей і не потребують додаткових дозволів для тестування. Це дозволяє провести експериментальну частину роботи в повній відповідності з вимогами законодавства та етичними принципами досліджень у сфері інформаційної безпеки.

Спочатку сканер був протестований на xss-game.appspot.com. 1 рівень розглядає Reflected XSS, 2 – Stored XSS, 3 – DOM-based XSS. Якщо інструмент точно виявить кожну із вразливостей, це дозволить переконатись, що він коректно сканує і розрізняє різні типи міжсайтового скриптингу.

При запуску коду показується банер, а далі користувача просять увести URL, який вони хочуть просканувати та ім'я файлу, у який будуть зберігатись детальні результати сканування. Для 1, 2 та 3 рівнів XSS Game від Google результати виконання коду представлені відповідно на рис. 3.7, 3.8 та 3.9.

№	URL	Параметр	Тип XSS	Пейлоад
1	https://xss-game.a…	query	Reflected XSS	<img src=x onerror=conso...

```
Всього виявлено вразливостей: 1
Reflected XSS: 1
Stored XSS: 0
DOM-based XSS: 0
------------------------------------------------------------
Результати збережено у файл: rXSS.json
------------------------------------------------------------
[!] У таблиці показано скорочений вивід. Повну інформацію дивіться у JSON
файлі.
------------------------------------------------------------
Час виконання: 5.44 секунд
```

Рисунок 3.7 – Виявлені вразливості на першому рівні вебресурсу XSS game

№	URL	Параметр	Тип XSS	Пейлоад
1	https://xss-game.app…	content	Stored XSS	<img src=x onerror=conso...

```
Всього виявлено вразливостей: 1
Reflected XSS: 0
Stored XSS: 1
DOM-based XSS: 0
------------------------------------------------------------
Результати збережено у файл: sXSS.json
------------------------------------------------------------
[!] У таблиці показано скорочений вивід. Повну інформацію дивіться у JSON
файлі.
------------------------------------------------------------
Час виконання: 10.68 секунд
```

Рисунок 3.8 – Виявлені вразливості на другому рівні вебресурсу XSS game

№	URL	Параметр	Тип XSS	Пейлоад
1	https://xss-game…	DOM_fragment	DOM-based XSS	1' onerror='console.…

```
Всього виявлено вразливостей: 1
Reflected XSS: 0
Stored XSS: 0
DOM-based XSS: 1
------------------------------------------------------------
Результати збережено у файл: dXSS.json
------------------------------------------------------------
[!] У таблиці показано скорочений вивід. Повну інформацію дивіться у JSON
файлі.
------------------------------------------------------------
Час виконання: 7.56 секунд
```

Рисунок 3.9 – Виявлені вразливості на третьому рівні вебресурсу XSS game

Наступним був просканований SPA-вебзастосунок testhtml5.vulnweb.com. Враховуючи його архітектуру і більшу масштабованість, час витрачений на виявлення вразливостей збільшився. Особливо це можна порівняти, дивлячись на результатів сканування попередніх ресурсів, які мають простішу архітектуру. Було виявлено два типи XSS – збережений та DOM-інтегрований (див. рис. 3.10).

№	URL	Параметр	Тип XSS	Вектор
1	http://testhtml5…	username	Stored XSS	<details ontoggle=consol...
2	http://testhtml5…	DOM_fragment	DOM-based XSS	1' onerror='console.…

```
Всього виявлено вразливостей: 2
Reflected XSS: 0
Stored XSS: 1
DOM-based XSS: 1
------------------------------------------------------------
Результати збережено у файл:
xss_scan_results_testhtml5.vulnweb.com_20250603_121446.json
------------------------------------------------------------
[!] У таблиці показано скорочений вивід. Повну інформацію дивіться у JSON
файлі.
------------------------------------------------------------
Час виконання: 352.63 секунд
```

Рисунок 3.10 – Виявлені вразливості на вебресурсі vulnweb від Acunetix

3.4 Аналіз результатів

Щоб перевірити чи точно були виявлені правильні типи XSS на тестових сайтах, а також чи є дійсно ці вразливості у них, було вручну вставлено знайдені пейлоади у вебресурси та проаналізовано їх дію.

Першим була перевірка на Reflected XSS для XSS Game від Google. Для цього рівня було використано пейлоад (рис. 3.11), який експлуатує поведінку HTML-елементу img при завантаженні неіснуючого зображення. Після вставки у поле вводу воно наче відображає те, що було введено користувачем. Його успішне виконання було підтверджене через появу повідомлення у консолі браузера (рис. 3.12). Це демонструє наявність Reflected XSS вразливості.


```
"vulnerabilities": [
  {
    "url": "https://xss-game.appspot.com/level1/frame",
    "parameter": "query",
    "payload": "<img src=x onerror=console.log(\"xss_found_112364\")>",
    "xss_type": "Reflected XSS",
    "is_vulnerable": true,
    "details": "Поле форми query вразливе до Reflected XSS (метод: GET)",
    "discovery_timestamp": "2025-06-03T02:21:53.990428"
  }
],
```


Рисунок 3.11 – JSON-вивід із пейлоадом для перевірки наявності Reflected XSS в XSS Game від Google

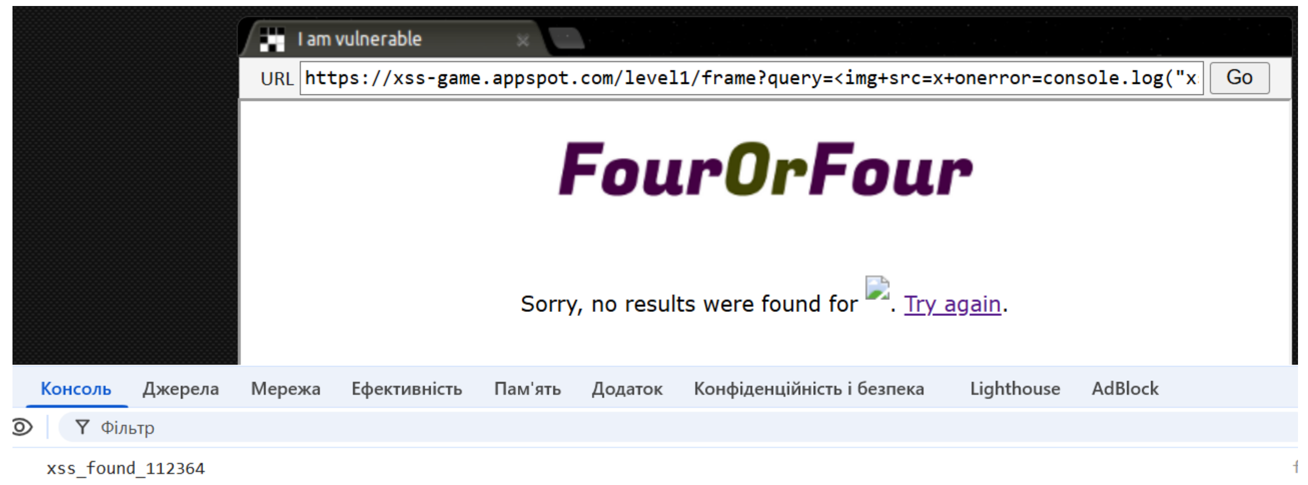


Рисунок 3.12 – Підтвердження наявності Reflected XSS в XSS Game від Google

Далі для цього ж вебресурсу було перевірено Stored XSS. Сканер виявив аналогічний пейлоад, що і у попередньому випадку, але з іншим ідентифікатором (див. рис. 3.13). Під час проведення тесту пейлоад був успішно збережений й базі даних сайту через форму коментарів. При подальших відвідуваннях сторінки

зловмисний код автоматично завантажувався та виконувався, що підтверджувалося появою відповідного повідомлення в консолі браузера. Там навіть було видно попередні спроби тестування даного сайту. Це свідчить про наявність критичної Stored XSS вразливості у ньому.

```
"vulnerabilities": [
  {
    "url": "https://xss-game.appspot.com/level2/frame",
    "parameter": "content",
    "payload": "<img src=x onerror=console.log(\"xss_found_758963\")>",
    "xss_type": "Stored XSS",
    "is_vulnerable": true,
    "details": "Поле форми content вразливе до Stored XSS",
    "discovery_timestamp": "2025-06-03T02:36:45.284849"
  }
],
```

Рисунок 3.13 – JSON-вивід із пейлоадом для перевірки наявності Stored XSS в XSS Game від Google

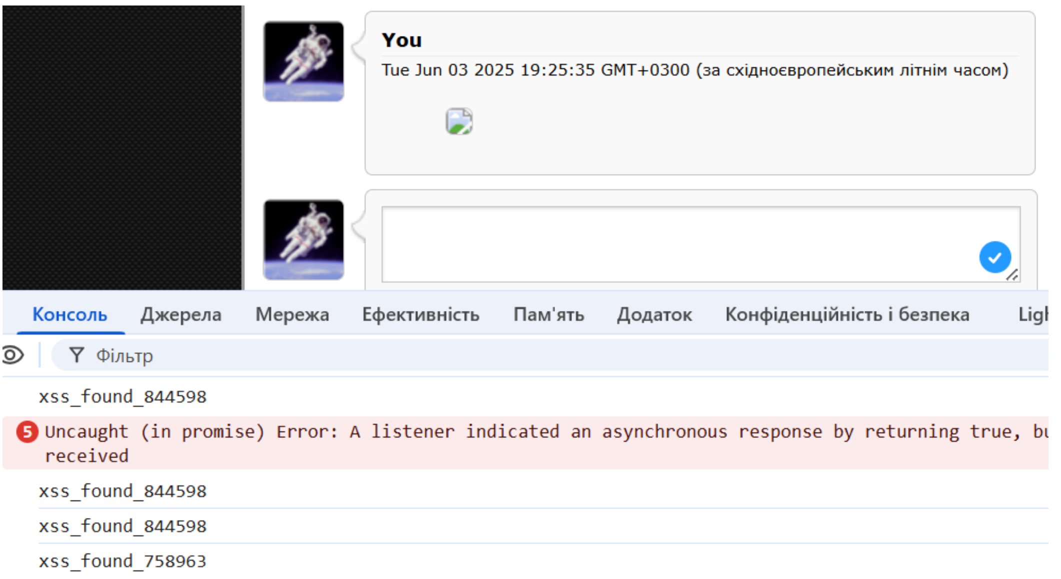


Рисунок 3.14 – Підтвердження наявності Stored XSS в XSS Game від Google

Наостанок для XSS Game було перевірено виявлення DOM-based XSS за допомогою пейлоаду, що представлений на рис. 3.15. Він експлуатує неправильну обробку параметрів URL або інших клієнтських даних, які використовуються для динамічного створення HTML-елементів. Під час

тестування було встановлено, що додаток неправильно обробляє параметри, дозволяючи ін'єкцію JavaScript-коду через атрибути HTML-елементів. Успішне виконання пейлоаду так само було підтверджене появою унікального повідомлення в консолі браузера, що демонструє наявність DOM-based XSS вразливості (рис. 3.16).


```
"vulnerabilities": [
  {
    "url": "https://xss-game.appspot.com/level3/frame",
    "parameter": "DOM_fragment",
    "payload": "1' onerror='console.log(\"xss_found_709568\")' src='x.jpg",
    "xss_type": "DOM-based XSS",
    "is_vulnerable": true,
    "details": "DOM XSS через URL фрагмент: https://xss-game.appspot.com/level3/frame#1' onerror='console.log(\"xss_found_709568\")' src=
    "discovery_timestamp": "2025-06-03T02:38:46.164756"
  }
],
```


Рисунок 3.15 – JSON-вивід із пейлоадом для перевірки наявності DOM-based XSS в XSS Game від Google

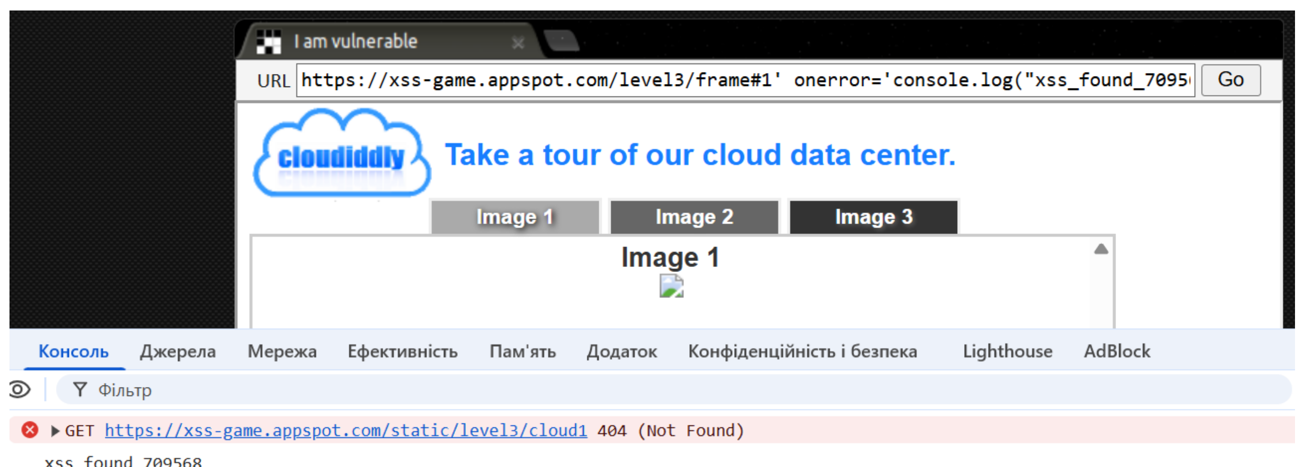


Рисунок 3.16 – Підтвердження наявності Stored XSS в XSS Game від Google

Для сайту testhtml5.vulnweb.com сканер виявив дві потенційні вразливості різних типів. При детальному аналізі результатів було встановлено наступне. Stored XSS вразливість у параметрі username для поля входу на сайт була підтверджена експериментально – пейлоад успішно виконався при збереженні та подальшому завантаженні сторінки, що свідчить про правильну класифікацію цієї вразливості розробленим сканером (див. рис. 3.17, 3.18 та 3.19).

```json
"vulnerabilities": [
  {
    "url": "http://testhtml5.vulnweb.com/#/popular",
    "parameter": "username",
    "payload": "<details ontoggle=console.log(\"xss_found_816286\") open>",
    "xss_type": "Stored XSS",
    "is_vulnerable": true,
    "details": "Поле форми username вразливе до Stored XSS (метод: POST)",
    "discovery_timestamp": "2025-06-03T12:29:00.099809"
  },
```


Рисунок 3.17 – JSON-вивід із пейлоадом для перевірки наявності Stored XSS у testhtml5.vulnweb.com

Login

Username

le=console.log("xss_found_816286") open>

Password

Forgot Pwd?

Login Close

Рисунок 3.18 – Введений пейлоад у полі входу на сайт testhtml5.vulnweb.com

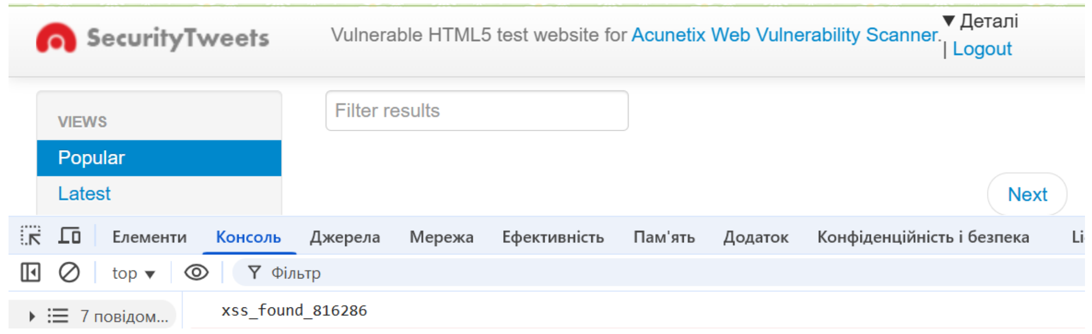


Рисунок 3.19 – Підтвердження наявності Stored XSS на testhtml5.vulnweb.com

Для DOM-based XSS сканер правильно ідентифікував наявність вразливості DOM-based типу, оскільки сайт дійсно має проблеми з обробкою

URL фрагментів та використовує небезпечні функції типу document.write, про що попереджується і у самій консолі браузера (рис. 3.20).

```
xss_found_816286
▶[Violation] Avoid using document.write(). https://developers.google.com/web/updates/2016/08/removing-document-write
▶[Violation] Avoid using document.write(). https://developers.google.com/web/updates/2016/08/removing-document-write
```

Рисунок 3.20 – Попередження про використання document.write() на testhtml5.vulnweb.com

Також було проаналізовано швидкість сканування власним інструментом тестових платформ. Для цього було проведене порівняння результатів роботи програмного модуля до оптимізації коду за допомогою паралельної обробки, кешування та пакетної обробки запитів та після неї. Варто зазначити, що такий підхід використовується у даному випадку з урахуванням того, що порівняння власної розробки із спеціалізованими інструментами для виявлення XSS-вразливостей, такими як XSSStrike та Dalfox, не є доцільним з кількох ключових причин.

По-перше, ці інструменти були обрані для аналізу саме тому, що вони є спеціалізованими рішеннями, зосередженими виключно на пошуку XSS-вразливостей, що дозволяє краще оцінити ефективність власного підходу в контексті цільового призначення. Однак практичне тестування показало істотні обмеження цих інструментів. Зокрема, XSStrike демонструє неповне виявлення вразливостей на тестових платформах, як це було продемонстровано на прикладі testhtml5.vulnweb.com, де інструмент не зміг повністю ідентифікувати наявні вразливості (див. рис. 3.21).

По-друге, аналіз показав, що обидва інструменти ефективно працюють лише з Reflected XSS, проте є недоліки при роботі з іншими типами міжсайтового скриптингу, що можна побачити на рис. 3.22 та 3.23. Це обмежує їх практичну застосовність для комплексного аудиту безпеки, що робить пряме порівняння швидкодії з власним інструментом методологічно некоректним.

```
vladyslava@ubuntu:~/XSStrike$ python3 xsstrike.py -u https://xss-game.appspot.com/level3/frame#1

        XSStrike v3.1.5

[~] Checking for DOM vulnerabilities
[+] Potentially vulnerable objects found
------------------------------------------------------------
8   window.location.hash = num;
25  chooseTab(unescape(self.location.hash.substr(1)) || "1");
31  chooseTab(unescape(self.location.hash.substr(1)));
------------------------------------------------------------
[-] No parameters to test.
```

Рисунок 3.21 – Результати сканування testhtml5.vulnweb.com за допомогою XSStrike

```
Target            https://xss-game.appspot.com/level1/frame
Method            GET
Performance       100 worker / 1 cpu
Mining            true (Gf-Patterns, DOM Mining Enabled)
Timeout           10
FollowRedirect    false
Started at        2025-06-04 20:56:00

[*] ----------------------------------------------------------------------------------------------------
[*] Starting scan [SID:Single] / URL: https://xss-game.appspot.com/level1/frame
[I] Found 1 testing points in DOM-based parameter mining
[I] Content-Type is text/html; charset=utf-8
[I] Reflected query param => . [ = } ; $ ` ( , + { ' " \ ] > ) - < | :
    13 line:  s were found for <b>Dalfox</b>. <a href='?'>Try again
[W] Reflected Payload in HTML: query="><script/"<a"/src=data:=".<a,[].some(confirm)>
    13 line:  s were found for <b>"><script/"<a"/src=data:=".<a,[].some(confirm)></b>. <a href
[POC][R][GET][inHTML] https://xss-game.appspot.com/level1/frame?query=%22%3E%3Cscript%2F%22%3Ca%22%2Fsrc%3Ddata%3A%3D%22.%3Ca%2C%5B%5D.some%28confirm%29%3E
[W] Reflected Payload in HTML: query="><d3"<"/onclick=" class=dalfox>[confirm``]"<">z
    13 line:  s were found for <b>"><d3"<"/onclick=" class=dalfox>[confirm``]"<">z</b>. <a hre
[POC][R][GET][inHTML] https://xss-game.appspot.com/level1/frame?query=%22%3E%3Cd3%22%3C%22%2Fonclick%3D%22+class%3Ddalfox%3E%5Bconfirm%60%60%5D%22%3C%22%3Ez
[W] Reflected Payload in HTML: query='><a href=javas&#99;ript:alert(1)/class=dalfox>click
    13 line:  s were found for <b>'><a href=javas&#99;ript:alert(1)/class=dalfox>click</b>. <a
[POC][R][GET][inHTML] https://xss-game.appspot.com/level1/frame?query=%27%3E%3Ca+href%3Djavas%26%2399%3Bript%3Aalert%281%29%2Fclass%3Ddalfox%3Eclick
[V] Triggered XSS Payload (found DOM Object): query="><img/src/onerror=.1|alert`` class=dalfox>
    13 line:  s were found for <b>"><img/src/onerror=.1|alert`` class=dalfox></b>. <a href='?'
[POC][V][GET][inHTML] https://xss-game.appspot.com/level1/frame?query=%22%3E%3Cimg%2Fsrc%2Fonerror%3D.1%7Calert%60%60+class%3Ddalfox%3E
[*] ----------------------------------------------------------------------------------------------------
[*] [duration: 4.037249013s][issues: 4] Finish Scan!
```

Рисунок 3.22 – Результати сканування 1 рівня XSS Game за допомогою Dalfox

```
Target            https://xss-game.appspot.com/level2/frame
Method            POST
Performance       100 worker / 1 cpu
Mining            true (Gf-Patterns, DOM Mining Enabled)
Timeout           10
FollowRedirect    false
Started at        2025-06-04 21:13:49

[*] --------------------------------------------------------------------------
[*] Starting scan [SID:Single] / URL: https://xss-game.appspot.com/level2/frame
[I] Found 0 testing points in DOM-based parameter mining
[I] Content-Type is text/plain; charset=UTF-8
[*] --------------------------------------------------------------------------
[*] [duration: 2.46658254s][issues: 0] Finish Scan!
```

Рисунок 3.23 – Результати сканування 2 рівня XSS Game за допомогою Dalfox

Таким чином було проведено тестування швидкодії власного коду до його оптимізації та після. Результати перевірки були наведені у таблиці 3.1, а додаткові підтвердження щодо дійсного проведення сканування перед використанням методів збільшення швидкості представлені у додатку Д.

Таблиця 3.1

Порівняльний аналіз швидкодії сканера до та після оптимізації коду

Тестова платформа	**Час сканування до оптимізації (с)**	**Час сканування після оптимізації (с)**	**Коефіцієнт прискорення**
1 рівень XSS Game	11,44	5,44	2,10
2 рівень XSS Game	15,17	10,68	1,42
3 рівень XSS Game	17,06	7,56	2,26
testhtml5.vulnweb.com	440,76	352,63	1,25

Можна стверджувати, що швидкість роботи сканера була підвищена у середньому у 1,76 разів. Найкращі результати продемонстровано на XSS Game платформах, де прискорення сягає 2,10 рази для першого рівня та 2,26 рази для третього рівня.

Найменший коефіцієнт прискорення на testhtml5.vulnweb.com пояснюється SPA-архітектурою, яка динамічно генерує контент через JavaScript. Це вимагає від сканера очікування завершення рендерингу перед аналізом, що обмежує ефективність впроваджених покращень.

При оптимізації сканерів безпеки критично важливо дотримуватися балансу між швидкістю та точністю виявлення вразливостей. Надмірне прискорення може призвести до пропуску критичних XSS-вразливостей через недостатній час на аналіз динамічного контенту, особливо у складних вебдодатках з багаторівневою архітектурою.

3.4. Можливі напрями вдосконалення програмного модуля

Попри те, що розроблений сканер успішно виявляє більшість основних типів XSS-вразливостей, його функціональність можна покращити для підвищення точності та ефективності роботи. Одним із можливих напрямів удосконалення є розширення бази пейлоадів з урахуванням особливостей

сучасних вебтехнологій. Зокрема, було би доцільно додати спеціалізовані пейлоади для таких фреймворків, як React, Vue.js та Angular, а також для CSP та роботи з Web Components. Це дозволило б сканеру краще адаптуватися до специфіки окремих вебдодатків.

Другим важливим аспектом є покращення розпізнавання контексту виконання JavaScript-коду. Сканер може стати точнішим, якщо буде реалізована можливість глибшого аналізу DOM-структури та відстеження передачі даних між джерелами (sources) і точками виконання (sinks). Це особливо важливо для виявлення складних DOM-based XSS-атак.

Також перспективним напрямом є реалізація підтримки налаштування правил сканування відповідно до типу вебзастосунку (наприклад, інтернет-магазин, CMS чи SPA). Додатково, можна було би впровадити елементи машинного навчання для підбору найбільш релевантних пейлоадів на основі попередніх результатів сканування. При правильному виконанні це могло би точно покращити загальну ефективність інструменту.

Висновки за розділом 3

У третьому розділі було розроблено та протестовано програмний модуль для виявлення XSS-вразливостей з використанням headless браузера.

1. Створено архітектуру модуля на основі класу XSSScanner з використанням Python та бібліотеки Playwright. Модуль здатний виявляти три типи XSS-вразливостей через емуляцію дій користувача в браузері Chromium.

2. Впроваджено оптимізації для покращення швидкості роботи. Застосовано пакетну обробку URL-адрес, кешування результатів DOM-аналізу та налаштування параметрів браузера. Це дозволило прискорити процес сканування в середньому у 1,76 разів.

3. Розроблено механізм перевірки вразливостей через вставку спеціального JavaScript-коду з унікальними ідентифікаторами. Такий підхід

зменшує кількість помилкових спрацювань та забезпечує точнішу класифікацію типів вразливостей.

4. Проведено тестування на навчальних платформах XSS Game та testhtml5.vulnweb.com. Результати підтвердили здатність модуля коректно виявляти різні типи XSS-вразливостей, хоча виявлено деякі обмеження при роботі з односторінковими додатками.

5. Визначено напрямки подальшого розвитку інструменту, зокрема розширення бази тестових пейлоадів та покращення роботи з сучасними JavaScript-фреймворками.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи бакалавра було розроблено програмний модуль для виявлення XSS-вразливостей з використанням headless браузера, який демонструє переваги над традиційними методами статичного та динамічного аналізу. Основні результати роботи наступні.

1. Досліджено архітектуру сучасних вебдодатків та механізми виникнення XSS-вразливостей. Встановлено, що міжсайтовий скриптинг залишається критичною загрозою інформаційної безпеки, посідаючи друге місце у рейтингу CWE Top 25 Most Dangerous Software Weaknesses за 2024 рік. Проаналізовано специфічні особливості трьох основних типів XSS-вразливостей – Reflected, Stored, DOM-based – у контексті сучасних JavaScript-фреймворків та архітектури односторінкових додатків, що створюють нові можливості для атак.

2. Проаналізовано сучасні сканери вразливостей та визначено критерії оцінки їх ефективності. Дослідження показало, що статичний та динамічний аналіз мають значні недоліки при роботі з DOM-based XSS-вразливостями. Встановлено, що спеціалізовані інструменти типу XSStrike та Dalfox демонструють неповне виявлення вразливостей на складніших платформах та ефективно працюють переважно з Reflected XSS. Визначено ключові критерії ефективності, включаючи точність класифікації, мінімізацію хибнопозитивних спрацювань та швидкість обробки.

3. Забезпечено високу точність виявлення XSS-вразливостей через використання headless браузера Chromium з бібліотекою Playwright. Було розроблено механізм верифікації через ін'єкцію JavaScript-коду з унікальними ідентифікаторами, який перехоплює виклики небезпечних функцій. Впроваджено систему контекстно-залежної генерації пейлоадів та спеціалізовані методи для виявлення кожного типу XSS, що дозволяє точно класифікувати вразливості відповідно до механізму їх реалізації.

4. Оптимізовано швидкість сканування за допомогою паралельної обробки, кешування та пакетної обробки запитів, що дозволило досягти середнього коефіцієнта прискорення у 1,76 разів. Впроваджено пакетну обробку URL-адрес з використанням семафорів, систему кешування DOM-аналізу через словник _dom_check_cache та оптимізовані параметри запуску браузера. Найкращі результати продемонстровано на платформах XSS Game з прискоренням до 2,26 разів, тоді як для SPA-додатків коефіцієнт становив 1,25 рази.

5. Розроблено та успішно протестовано інструмент для підтвердження наявності XSS-вразливостей на офіційних навчальних платформах. Експериментальне тестування на XSS Game від Google та testhtml5.vulnweb.com від Acunetix підтвердило здатність сканера точно виявляти всі три типи XSS-вразливостей з мінімальною кількістю хибнопозитивних спрацювань. Ручна верифікація знайдених пейлоадів продемонструвала правильність класифікації вразливостей.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Технологічний стек роботи веб-додатків [Електронний ресурс] / Максим Непийвода [та ін.] // Modern engineering and innovative technologies. – 2019. – № 22-01. – С. 104–112. – Режим доступу: https://doi.org/10.30890/2567-5273.2022-22-01-038

2. Blinowski G. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation [Електронний ресурс] / Grzegorz Blinowski, Anna Ojdowska, Adam Przybylek // IEEE Access. – 2022. – Т. 10. – С. 20357–20374. – Режим доступу: https://doi.org/10.1109/access.2022.3152803

3. Музичук І. В. Сервіс-орієнтована архітектура [Електронний ресурс] / І. В. Музичук // *Сучасні інфокомунікаційні технології* : Науково-техн. конф., Київ, 5 груд. 2019 р. – Київ, 2019. – С. 161–162. – Режим доступу: https://duikt.edu.ua/uploads/n_7726_26577046.pdf

4. Microservice architecture style - Azure Architecture Center [Електронний ресурс] // Microsoft Learn: Build skills that open doors in your career. – Режим доступу: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

5. Progressive web apps | MDN [Електронний ресурс] // MDN Web Docs. – Режим доступу: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps

6. OWASP Top Ten [Електронний ресурс] // OWASP Foundation, the Open Source Foundation for Application Security. – Режим доступу: https://owasp.org/www-project-top-ten/

7. Cross Site Scripting (XSS) [Електронний ресурс] // OWASP Foundation, the Open Source Foundation for Application Security. – Режим доступу: https://owasp.org/www-community/attacks/xss/

8. CWE - 2024 CWE Top 25 Most Dangerous Software Weaknesses [Електронний ресурс] // CWE - Common Weakness Enumeration. – Режим доступу: https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html

9. Cross Site Scripting Prevention - OWASP Cheat Sheet Series [Електронний ресурс] // Introduction - OWASP Cheat Sheet Series. – Режим доступу: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

10. Detection of Web Cross-Site Scripting (XSS) Attacks [Електронний ресурс] / Mohammad Alsaffar [та ін.] // Electronics. – 2022. – T. 11, № 14. – 2212. – Режим доступу: https://doi.org/10.3390/electronics11142212

11. Types of XSS (Cross-site Scripting) [Електронний ресурс] // Acunetix. – Режим доступу: https://www.acunetix.com/websitesecurity/xss/

12. DOM Based XSS [Електронний ресурс] // OWASP Foundation, the Open Source Foundation for Application Security. – Режим доступу: https://owasp.org/www-community/attacks/DOM_Based_XSS

13. What is the defense-in-depth approach to mitigating XSS attacks and why is it important to implement multiple layers of security controls? - EITCA Academy [Електронний ресурс] // EITCA Academy. – Режим доступу: https://eitca.org/cybersecurity/eitc-is-wasf-web-applications-security-fundamentals/cross-site-scripting/cross-site-scripting-xss/examination-review-cross-site-scripting-xss/what-is-the-defense-in-depth-approach-to-mitigating-xss-attacks-and-why-is-it-important-to-implement-multiple-layers-of-security-controls/

14. HTML, CSS and JavaScript in Easy Steps. – [Б. м.] : In Easy Steps Limited, 2020. – 480 с.

15. Input Validation - OWASP Cheat Sheet Series [Електронний ресурс] // Introduction - OWASP Cheat Sheet Series. – Режим доступу: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html

16. Strict-dynamic explained [Електронний ресурс] // Content-Security-Policy (CSP) Header Quick Reference. – Режим доступу: https://content-security-policy.com/strict-dynamic/

17. Source Code Analysis Tools [Електронний ресурс] // OWASP Foundation, the Open Source Foundation for Application Security. – Режим доступу: https://owasp.org/www-community/Source_Code_Analysis_Tools

18. Dynamic Application Security Testing [Електронний ресурс] // OWASP Foundation, the Open Source Foundation for Application Security. – Режим доступу: https://owasp.org/www-project-devsecops-guideline/latest/02b-Dynamic-Application-Security-Testing

19. Playwright [Електронний ресурс] // Fast and reliable end-to-end testing for modern web apps. – Режим доступу: https://playwright.dev/

20. What is Puppeteer? [Електронний ресурс] // Puppeteer. – Режим доступу: https://pptr.dev/guides/what-is-puppeteer

21. WebDriver [Електронний ресурс] // Selenium. – Режим доступу: https://www.selenium.dev/documentation/webdriver/

22. Interactive Application Security Testing [Електронний ресурс] // OWASP Foundation, the Open Source Foundation for Application Security. – Режим доступу: https://owasp.org/www-project-devsecops-guideline/latest/02c-Interactive-Application-Security-Testing

23. An empirical comparison of commercial and open-source web vulnerability scanners [Електронний ресурс] / Richard Amankwah [та ін.] // Software: Practice and Experience. – 2020. – Т. 50, № 9. – С. 1842–1857. – Режим доступу: https://doi.org/10.1002/spe.2870

24. Avoid False Positives with Proof-Based Scanning | Invicti [Електронний ресурс] // Invicti. – Режим доступу: https://www.invicti.com/features/proof-based-scanning/

25. Mburano B. Evaluation of Web Vulnerability Scanners Based on OWASP Benchmark [Електронний ресурс] / Balume Mburano, Weisheng Si // 2018 26th International Conference on Systems Engineering (ICSEng), Sydney, Australia, 18–20 груд. 2018 р. – [Б. м.], 2018. – Режим доступу: https://doi.org/10.1109/icseng.2018.8638176

26. Security Tools Benchmarking [Електронний ресурс] // Security Tools Benchmarking. – Режим доступу: https://sectooladdict.blogspot.com/

27. GitHub - s0md3v/XSStrike: Most advanced XSS scanner. [Електронний ресурс] // GitHub. – Режим доступу: https://github.com/s0md3v/XSStrike

28. Класифікація: точність, повнота, влучність і пов'язані метрики | Machine Learning | Google for Developers [Електронний ресурс] // Google for Developers. – Режим доступу: https://developers.google.com/machine-learning/crash-course/classification/accuracy-precision-recall?hl=uk

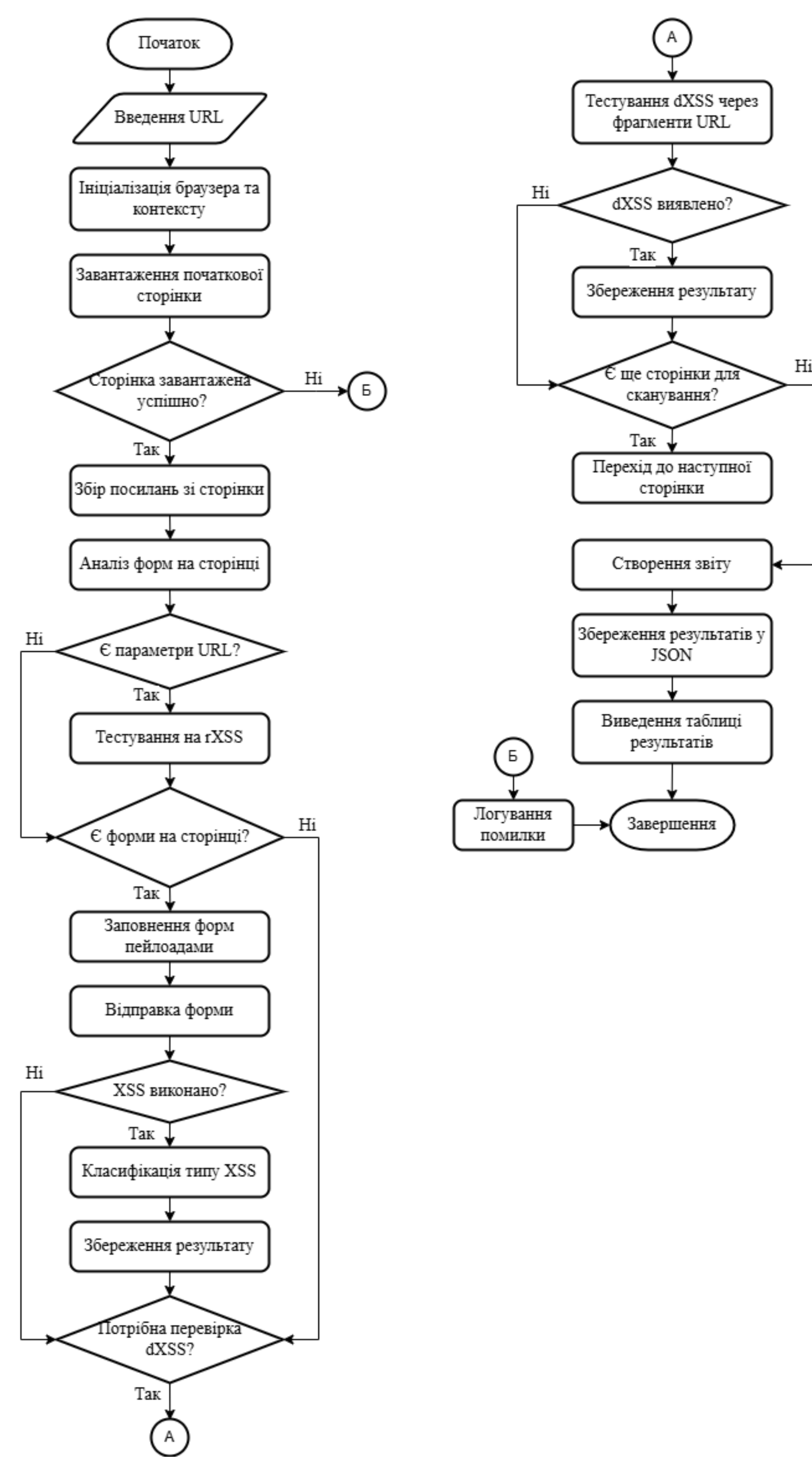
29. OWASP Benchmark [Електронний ресурс] // OWASP Foundation, the Open Source Foundation for Application Security. – Режим доступу: https://owasp.org/www-project-benchmark/

30. Кримінальний кодекс України [Електронний ресурс] : Кодекс України від 05.04.2001 № 2341-III : станом на 9 трав. 2025 р. – Режим доступу: https://zakon.rada.gov.ua/laws/show/2341-14#Text

ДОДАТКИ

Додаток А

Загальна схема роботи програмного модуля



Додаток Б

Алгоритми з оптимізацією сканування

```
async def _crawl_website(self, playwright: Playwright, browser: Browser, context) -> None:
    with Progress() as progress:
        crawl_task = progress.add_task("[green]Сканування сторінок...", total=self.max_urls_to_scan)
        scan_task = progress.add_task("[yellow]Пошук XSS-вразливостей...", total=None)
        urls_to_scan = [self.target_url]
        scanned_count = 0
        optimal_batch_size = min(self.concurrency * 3, 30)
        while urls_to_scan and scanned_count < self.max_urls_to_scan:
            batch_size = min(optimal_batch_size, len(urls_to_scan),
                             self.max_urls_to_scan - scanned_count)
            batch_urls = urls_to_scan[:batch_size]
            urls_to_scan = urls_to_scan[batch_size:]
            unique_batch_urls = []
            for url in batch_urls:
                if url not in self.visited_urls:
                    self.visited_urls.add(url)
                    unique_batch_urls.append(url)
                    scanned_count += 1
                    progress.update(crawl_task, completed=scanned_count)
            if not unique_batch_urls:
                continue
            semaphore = asyncio.Semaphore(self.concurrency)
```

```
        async def process_with_semaphore(url):
            async with semaphore:
                return await self._process_url(context, url, urls_to_scan, progress,
scan_task)
        tasks = [process_with_semaphore(url) for url in unique_batch_urls]
        results = await asyncio.gather(*tasks, return_exceptions=True)
        error_count = sum(1 for result in results if isinstance(result, Exception))
        if error_count > len(results) // 2:
            logger.warning(f"Високий відсоток помилок в батчі:
{error_count}/{len(results)}")

    async def _should_check_dom_xss(self, page: Page, url: str) -> bool:
        try:
            cache_key = f"dom_check_{hash(url)}"
            if hasattr(self, '_dom_check_cache') and cache_key in self._dom_check_cache:
                return self._dom_check_cache[cache_key]
            if not hasattr(self, '_dom_check_cache'):
                self._dom_check_cache = {}
            has_dom_characteristics = await page.evaluate("""
            () => {
                try {
                    const inlineScripts = Array.from(document.scripts)
                        .filter(script => !script.src)
                        .map(script => script.textContent || script.innerHTML || '')
                        .join('');
                    if (!inlineScripts.trim()) {
                        return false;
                    }
                    const criticalPatterns = [
                        /location\.(hash|search|href)/i,
```

```
            /window\.location/i,
            /document\.URL/i,
            /innerHTML|outerHTML|document\.write|eval\s*\(/i,
            /router|route|hashchange/i
          ];
          return criticalPatterns.some(pattern => pattern.test(inlineScripts));
        } catch (e) {
          return false;
        }
      }
      """)
      self._dom_check_cache[cache_key] = has_dom_characteristics
      return has_dom_characteristics
    except Exception as e:
      logger.debug(f"Помилка перевірки DOM характеристик: {str(e)}")
      return False
```

Додаток В

Алгоритми з контекстним аналізом

```
async def _comprehensive_dom_xss_check(self, page: Page, url: str, progress=None, scan_task: TaskID = None) -> None:
    try:
        if progress and scan_task:
            progress.update(scan_task, description="[yellow]Комплексна перевірка DOM XSS")
        await self._wait_for_spa_load(page)
        page_analysis = await self._analyze_page_for_dom_vulnerabilities(page, url)
        is_spa = await self._detect_spa(page)
        if not page_analysis.get('hash_processors') and not page_analysis.get('url_processors') and not is_spa:
            return
        dom_payloads = self._create_targeted_dom_payloads(page_analysis)
        parsed_url = urlparse(url)
        base_url = f"{parsed_url.scheme}://{parsed_url.netloc}{parsed_url.path}"
        for i, payload in enumerate(dom_payloads):
            if progress and scan_task:
                progress.update(scan_task, description=f"[yellow]DOM XSS тест {i + 1}/{len(dom_payloads)}")
            test_scenarios = []
            if page_analysis.get('hash_processors') or is_spa:
                test_scenarios.extend([
                    f"{base_url}#{payload}",
                    f"{base_url}#/{payload}",
                    f"{base_url}#!/{payload}",
                    f"{base_url}#1{payload}",
```

```
                f"{base_url}#{urllib.parse.quote(payload)}",
                f"{base_url}#/popular/{payload}",
                f"{base_url}#test={payload}",
            ])
        if page_analysis.get('search_processors'):
            test_scenarios.extend([
                f"{base_url}?test={urllib.parse.quote(payload)}",
                f"{base_url}?q={urllib.parse.quote(payload)}"
            ])
        for test_url in test_scenarios:
            try:
                await page.goto(test_url, timeout=self.timeout * 1000,
wait_until="networkidle")
                await self._wait_for_spa_load(page)
                await self._activate_dom_payload_enhanced(page, test_url, payload,
page_analysis)
                if await self._check_dom_execution_strict(page, payload):
                    detection_details = await self._get_detection_details(page)
                    self.results.append(ScanResult(
                        url=url,
                        parameter="DOM_fragment",
                        payload=payload,
                        xss_type="DOM-based XSS",
                        is_vulnerable=True,
                        details=f"DOM XSS через URL фрагмент: {test_url}. Детекція:
{detection_details}"
                    ))
                    return
            except Exception as e:
```

```
                logger.debug(f"Помилка тестування DOM XSS для {test_url}: {str(e)}")
                continue
        except Exception as e:
            logger.error(f"Помилка комплексної перевірки DOM XSS: {str(e)}")

    def _generate_context_aware_payloads(self, xss_type: str, context: str = "form") -> List[str]:
        if xss_type == "stored" or context == "form":
            stored_payloads = [
                f"<img src=x onerror=console.log(\"{self.xss_identifier}\")>",
                f"<svg onload=console.log(\"{self.xss_identifier}\")>",
                f"<details ontoggle=console.log(\"{self.xss_identifier}\") open>",
                f"<input onfocus=console.log(\"{self.xss_identifier}\") autofocus>",
                f"<video><source onerror=console.log(\"{self.xss_identifier}\")>",
                f"<iframe srcdoc=\"<img src=x onerror=console.log('{self.xss_identifier}')>\">",
                f"<object data=\"javascript:console.log('{self.xss_identifier}')\">",
                f"<embed src=\"javascript:console.log('{self.xss_identifier}')\">",
            ]
            return stored_payloads
        elif xss_type == "reflected":
            reflected_payloads = [
                f"<script>console.log(\"{self.xss_identifier}\")</script>",
                f"<img src=x onerror=console.log(\"{self.xss_identifier}\")>",
                f"<svg onload=console.log(\"{self.xss_identifier}\")>",
                f"javascript:console.log(\"{self.xss_identifier}\")",
                f"'><script>console.log(\"{self.xss_identifier}\")</script>",
                f"\"><script>console.log(\"{self.xss_identifier}\")</script>"
            ]
```

```
        return reflected_payloads
    elif xss_type == "dom":
        return self._create_targeted_dom_payloads({})
    return [
        f"<img src=x onerror=console.log(\"{self.xss_identifier}\")>",
        f"<svg onload=console.log(\"{self.xss_identifier}\")>",
        f"<script>console.log(\"{self.xss_identifier}\")</script>"
```

Додаток Г

Основні алгоритми виявлення XSS

```
def _get_enhanced_detection_scripts(self) -> List[str]:
    return [f"""
(function() {{
    window.domXssDetected = false;
    window.xssIdentifier = '{self.xss_identifier}';
    window.xssDetectionLog = [];
    function logDetection(method, content) {{
        window.xssDetectionLog.push({{
            method: method,
            content: content,
            timestamp: Date.now()
        }});
        window.domXssDetected = true;
        console.log(`DOM XSS detected via ${{method}}: ${{content}}`);
    }}
    const originalMethods = {{
        innerHTML: Object.getOwnPropertyDescriptor(Element.prototype, 'innerHTML'),
        outerHTML: Object.getOwnPropertyDescriptor(Element.prototype, 'outerHTML'),
        eval: window.eval,
        documentWrite: document.write,
        documentWriteln: document.writeln,
        setAttribute: Element.prototype.setAttribute
    }};
    if (originalMethods.innerHTML && originalMethods.innerHTML.set) {{
```

```
            Object.defineProperty(Element.prototype, 'innerHTML', {{
                set: function(value) {{
                    if (value && typeof value === 'string' &&
value.includes(window.xssIdentifier)) {{
                        logDetection('innerHTML', value);
                    }}
                    return originalMethods.innerHTML.set.call(this, value);
                }},
                get: originalMethods.innerHTML.get
            }});
        }}
        if (originalMethods.outerHTML && originalMethods.outerHTML.set) {{
            Object.defineProperty(Element.prototype, 'outerHTML', {{
                set: function(value) {{
                    if (value && typeof value === 'string' &&
value.includes(window.xssIdentifier)) {{
                        logDetection('outerHTML', value);
                    }}
                    return originalMethods.outerHTML.set.call(this, value);
                }},
                get: originalMethods.outerHTML.get
            }});
        }}
        window.eval = function(code) {{
            if (code && typeof code === 'string' && code.includes(window.xssIdentifier))
{{
                logDetection('eval', code);
            }}
            return originalMethods.eval.call(window, code);
        }};
```

```
        document.write = function(content) {{
            if (content && typeof content === 'string' &&
content.includes(window.xssIdentifier)) {{
                logDetection('document.write', content);
            }}
            return originalMethods.documentWrite.call(document, content);
        }};
        Element.prototype.setAttribute = function(name, value) {{
            if (value && typeof value === 'string' &&
value.includes(window.xssIdentifier)) {{
                logDetection('setAttribute', `${{name}}=${{value}}`);
            }}
            return originalMethods.setAttribute.call(this, name, value);
        }};
        const originalConsoleLog = console.log;
        console.log = function() {{
            const args = Array.from(arguments).join(' ');
            if (args.includes(window.xssIdentifier)) {{
                window.domXssDetected = true;
            }}
            return originalConsoleLog.apply(console, arguments);
        }};
        const originalAlert = window.alert;
        window.alert = function(message) {{
            if (message && message.toString().includes(window.xssIdentifier)) {{
                logDetection('alert', message.toString());
            }}
            return originalAlert.call(window, message);
        }};
        if (typeof MutationObserver !== 'undefined') {{
```

```
            const observer = new MutationObserver((mutations) => {{
                mutations.forEach((mutation) => {{
                    mutation.addedNodes.forEach((node) => {{
                        if (node.nodeType === 1) {{
                            const html = node.outerHTML || node.innerHTML || '';
                            if (html && html.includes(window.xssIdentifier)) {{
                                logDetection('MutationObserver', html);
                            }}
                        }}
                        if (node.nodeType === 3 && node.textContent && 
node.textContent.includes(window.xssIdentifier)) {{
                            logDetection('TextNode', node.textContent);
                        }}
                    }});
                }});
            }});
            observer.observe(document.documentElement, {{childList: true, subtree: true, 
characterData: true}});
        }}
        window.addEventListener('error', function(e) {{
            if (e.message && e.message.includes(window.xssIdentifier)) {{
                logDetection('onerror', e.message);
            }}
        }}, true);

    }})();
    """]

    async def _check_xss_execution(self, page: Page, deep_check: bool = False) -> 
bool:
```

```
try:
    if page.is_closed():
        logger.debug("Сторінка закрита, неможливо перевірити XSS")
        return False
    timeout = 2.0 if not deep_check else 3.0
    result = await asyncio.wait_for(
        page.evaluate(f"""
        () => {{
            try {{
                const scriptDetected = window.domXssDetected === true ||
                    (window.alertCalled &&
window.alertCalled.includes('{self.xss_identifier}'));
                if (scriptDetected) return {{ detected: true, method: 'script' }};
                if (!document || !document.documentElement) {{
                    return {{ detected: false }};
                }}
                const content = document.documentElement.outerHTML;
                if (content.includes('{self.xss_identifier}')) {{
                    const hasXssMarkers = [
                        '<script', 'onerror=', 'onload=', 'javascript:',
                        'alert(', '<img', '<svg'
                    ].some(marker =>
content.toLowerCase().includes(marker.toLowerCase()));
                    if (hasXssMarkers) return {{ detected: true, method: 'dom' }};
                }}
                return {{ detected: false }};
            }} catch(e) {{
                console.log('XSS check error:', e);
                return {{ detected: false }};
            }}
```

```
            }}
            """),
            timeout=timeout
        )
        return result['detected']
    except (asyncio.TimeoutError, Exception) as e:
        logger.debug(f"Помилка при перевірці XSS (можливо контекст знищений): {str(e)}")
        return False
```

Додаток Д

Результати тестування швидкодії інструменту до оптимізації

Виявлені XSS-вразливості

URL	Параметр	Тип XSS	Вектор	Деталі
https://xss-g…	query	Reflected XSS	<script>alert…	Поле форми query вразливе до Reflected XSS

Всього виявлено вразливостей: 1
Reflected XSS: 1
Stored XSS: 0
DOM-based XSS: 0
Час виконання: 11.44 секунд

Рисунок Д.1 – Результат сканування 1 рівня XSS Game (11,44 с)

Виявлені XSS-вразливості

URL	Параметр	Тип XSS	Вектор	Деталі
https://xss-ga…	content	Stored XSS	<script>alert(…	Поле форми content вразливе до Stored XSS

Всього виявлено вразливостей: 1
Reflected XSS: 0
Stored XSS: 1
DOM-based XSS: 0
Час виконання: 15.17 секунд

Рисунок Д.2 – Результат сканування 2 рівня XSS Game (15,17 с)

Виявлені XSS-вразливості

URL	Параметр	Тип XSS	Вектор	Деталі
https://xss-…	DOM_fragment	DOM-based XSS	<img src=x onerror=cons…	DOM XSS через URL фрагмент: https://xss… src=x onerror=con… Детекція: innerHTML: %3Cimg%20sr…

Всього виявлено вразливостей: 1
Reflected XSS: 0
Stored XSS: 0
DOM-based XSS: 1
Час виконання: 17.06 секунд

Рисунок Д.3 – Результат сканування 2 рівня XSS Game (17,06 с)

URL	Параметр	Тип XSS	Вектор	Деталі
http://testh…	username	Stored XSS	<script>aler…	Поле форми username вразливе до Stored XSS
http://testh…	DOM_fragment	DOM-based XSS	<img src=x onerror=cons…	DOM XSS через URL фрагмент: http://test… src=x onerror=con… Детекція: document.wr… <iframe name="ads_a… src="http:/…

Всього виявлено вразливостей: 2
Reflected XSS: 0
Stored XSS: 1
DOM-based XSS: 1
Час виконання: 440.76 секунд

Рисунок Д.4 – Результат сканування testhtml5.vulnweb.com (440,76 с)