КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 3 (31), 2025
ISSN 2663 - 4023
DOI 10.28925/2663-4023.2025.31.1076
УДК 004.056.53:004.415

Соколов Володимир Юрійович
кандидат технічних наук, доцент
доцент кафедри інформаційної та кібернетичної безпеки імені професора Володимира Бурячка
Київський столичний університет імені Бориса Грінченка, м. Київ, Україна
ORCID: 0000-0002-9349-7946
v.sokolov@kubg.edu.ua

Поліковський Богдан Володимирович
магістр кафедри інформаційної та кібернетичної безпеки імені професора Володимира Бурячка
Київський столичний університет імені Бориса Грінченка, м. Київ, Україна
ORCID: 0009-0005-4821-8089
bvpolikovskyi.fitm24m@kubg.edu.ua

Ворохоб Максим Віталійович
PhD in Cybersecurity
доцент кафедри інформаційної та кібернетичної безпеки імені професора Володимира Бурячка
Київський столичний університет імені Бориса Грінченка, м. Київ, Україна
ORCID: 0000-0001-5160-7134
m.vorokhob@kubg.edu.ua

Цируль Олександр Олександрович
аспірант
Інститут телекомунікацій і глобального інформаційного простору НАН України, Київ,Україна
ORCID:0009-0002-5945-5918
tsyruloleksandr@gmail.com

ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ БІБЛІОТЕК САНІТИЗАЦІЇ ДЛЯ XSS-АТАК В ВЕБ-ДОДАТКАХ

Анотація. Міжсайтові скриптові атаки (Cross-Site Scripting, XSS) залишаються однією з найбільш поширених і критичних уразливостей сучасних веб-додатків, оскільки дозволяють зловмисникам виконувати довільний шкідливий код у браузері користувача, порушуючи конфіденційність, цілісність і доступність даних. Одним із ключових підходів до протидії XSS є використання бібліотек санітизації, призначених для очищення або безпечного перетворення користувацького вводу перед його обробкою та відображенням. У статті проведено комплексне експериментальне дослідження ефективності популярних бібліотек санітизації HTML у контексті захисту веб-додатків від XSS-атак. Запропоновано та використано спеціалізований датасет зі 100 унікальних XSS-векторів, який охоплює як класичні сценарії атак (script-теги, обробники подій), так і сучасні та менш очевидні техніки, зокрема CSS-ін'єкції, SVG-вектори, DOM clobbering, encoded payloads, а також зловживання сучасними браузерними API. Для проведення експериментів розроблено автоматизований тестовий стенд на базі Node.js з використанням інструментів браузерної емуляції, що дозволило відтворити реалістичні умови виконання шкідливого коду. Порівняльний аналіз бібліотек DOMPurify, js-xss, sanitize-html та OWASP Java HTML Sanitizer здійснювався у дефолтних конфігураціях за показниками рівня блокування XSS-векторів, продуктивності та споживання пам'яті, а також із застосуванням багатокритеріального оцінювання з урахуванням безпеки, підтримки та практичної придатності. Отримані результати засвідчили, що жодна з досліджуваних бібліотек не забезпечує повного захисту «з коробки», а спільною проблемою для всіх рішень є вразливість до DOM clobbering і кодованих векторів атак. Сформульовано практичні рекомендації щодо конфігурації санітизаційних бібліотек та їх використання в межах стратегії багаторівневого захисту веб-додатків.

Ключові слова: XSS-атаки, міжсайтовий скриптинг, веб-безпека, санітизація даних, Content Security Policy, валідація вхідних даних, DOM-based XSS, кібербезпека, захист веб-додатків.

ВСТУП

XSS є однією з найпоширеніших вразливостей веб-додатків, яка дозволяє зловмисникам вводити шкідливий код (наприклад, JavaScript) у веб-сторінки, що переглядаються іншими користувачами. Бібліотеки санітизації допомагають запобігти таким атакам шляхом очищення або екранування небажаного вмісту в користувацьких даних. Це дослідження базується на аналізі наукових робіт, рекомендацій OWASP та практичних порівнянь бібліотек, таких як DOMPurify, Bleach, OWASP Java HTML Sanitizer та HTMLPurifier. Ми розглянемо їх ефективність, сильні та слабкі сторони, а також рекомендації для використання в різних контекстах (HTML, JavaScript, CSS тощо).

Систематичний аналіз фреймворків веб-додатків показує, що багато з них надають обмежену підтримку санітизації XSS, часто ігноруючи контекстно-чутливі сценарії, що призводить до вразливостей.

Постановка проблеми. Об’єктом дослідження є процес забезпечення захисту веб-додатків від XSS. Предметом дослідження є методи та засоби виявлення, запобігання та нейтралізації XSS-атак у сучасних веб-додатках, зокрема бібліотеки санітизації HTML та політики безпеки контенту. 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:

- 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;
- 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.

Аналіз останніх досліджень і публікацій. Емпіричне дослідження очищення в основних веб-фреймворках виявило, що багато захистів XSS, що надаються фреймворками, не працюють у реальних випадках використання, а деякі функції безпеки створюють хибне відчуття безпеки, коли розробники припускають, що вони охоплюють більше контекстів, ніж вони охоплюють насправді. Очищення XSS тісно пов’язане з поведінкою парсингу браузера; ледь помітні крайні випадки HTML/JavaScript часто обходять наївні кодери/фільтри [1]. Було показано, що вбудований у браузер автоматизований санітайзер на стороні клієнта виявляє та запобігає XSS на практиці, але попередні бібліотеки, такі як плагіни на основі OWASP ESAPI, санітайзер Jsoup та Haskell-xss-sanitizer, мали помітні обмеження, такі як неповне покриття типів XSS, залежність від крихких білих списків та втрата пробілів/форматування [2].

Систематичні дослідження підкреслюють процедури санітайзера та безпечні API як одну з трьох основних практик безпечного кодування для зменшення XSS, але зазначають важливі обмеження в існуючих бібліотеках [3]:

- ранні інструменти, такі як WebSSARI, автоматично вставляли санітайзери з фіксованої бібліотеки, але давали високий рівень хибнопозитивних результатів (≈26,9%) та залежать від попередньо визначеного набору санітайзерів;
- DOMPurify, широко використовуваний санітайзер HTML, ефективно видаляє небезпечні конструкції та навіть розшифровує/застосовує політики до зашифрованих корисних навантажень, але його можна обійти певними класами атак (наприклад, CR-XSS) і його необхідно вручну інструментувати в кожній точці введення, що схильне до помилок у великих кодових базах;
- доменно-орієнтована мова може формально описувати поведінку санітайзера та статично перевіряти такі властивості, як правильність, комутативність та ідемпотентність, допомагаючи створювати доказово коректні санітайзери, але це здебільшого дослідницький інструментарій, а не загальноприйнята бібліотечна практика [4].

Систематичний огляд показує, що не існує єдиного рішення, яке ефективно пом'якшує всі атаки XSS. Захисні засоби фрагментовані (фільтри, статичний/динамічний аналіз, інструменти браузера, машинне навчання тощо) і часто зосереджені на відображених XSS, залишаючи прогалини для варіантів на основі DOM та більш просунутих [3]. Інше велике опитування виявляє «брак механізмів відновлення вразливостей» та підкреслює упередженість до атак, орієнтованих на JavaScript, підкреслюючи, що багато підходів до санітарної обробки є неповними [4].

МЕТОДИКА ДОСЛІДЖЕННЯ

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.

Експериментальне дослідження 2021 року [5] систематично протестувало 12 динамічних XSS-фільтрів з відкритим кодом, вставивши кожен фільтр на виході веб-застосунку та запустивши автоматизований набір XSS-тестів як зі шкідливими, так і з безпечними корисними навантаженнями:

- жоден з фільтрів не був надійним «з коробки», але з налаштуваннями за замовчуванням лише деякі фільтри задовільно обробляли шкідливі корисні навантаження, і ті часто порушували безпечні вхідні дані (надмірна фільтрація);
- фільтри, які покладаються переважно на чорні/білі списки «небезпечних/безпечних елементів», часто ігнорували перевірку значень, що призводило до позначення безпечних значень та прослизання деяких шкідливих шаблонів;
- після переналаштування кожен фільтр мав принаймні одну конфігурацію, яка успішно очищала всі відомі корисні навантаження XSS, що використовувалися в експерименті, що вказує на те, що бібліотеки можуть бути ефективними, але потребують ретельного налаштування та тестування.

РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

Категорії датасетів.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.

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.

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.

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.

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.

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 (табл. 1). 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 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 одному 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.

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.

Проведення експерименту. Представлені результати отримані в ході експериментального тестування чотирьох бібліотек санітизації HTML з використанням їх дефолтних конфігурацій без додаткових налаштувань або оптимізацій (табл. 2). 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 2

Ефективність бібліотек санітизації

Бібліотека	XSS блоко-вано, %	Використа-но пам'яті, байт	Середній час, мс	Пейлоади, які не були блоковані
Без санітизації	0	502	0,028	—
DOMPurify	96	90 103	1,107	data attribute, css injection, encoded, dom clobbering
js-xss	95	3 271	0,204	svg, deprecated, applet tag, encoded, dom clobbering
sanitize-html	98	7 501	0,179	encoded, dom clobbering
OWASP Java HTML Sanitizer	98	89 365	0,356	encoded, dom clobbering

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:

$$n_i = \frac{x_i - x_{min}}{10(x_{max} - x_{min})}, \quad (1)$$

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

$$n_i = \frac{x_{max} - x_i}{10 \cdot (x_{max} - x_{min})}, \quad (2)$$

де n_i – нормалізоване значення показника для *i*-ї бібліотеки, x_i – абсолютне значення показника, x_{min} і x_{max} – мінімальне і максимальне значення показника серед усіх досліджуваних бібліотек.

Детальний аналіз нормалізованих показників.

Можна виділити кілька критеріїв:

1. 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. 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. 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. 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.

Інтегральна оцінка S_i для кожної бібліотеки розраховується за формулою зваженої суми

$$S_i = \sum (w_j \cdot n_{ij}), \quad (3)$$

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), показано в табл. 3:

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 (7,5 кБ).

2. OWASP Java HTML Sanitizer (7,28) забезпечує високий рівень безпеки (98% блокування) та добру підтримку проєкту, однак демонструє найгірші показники використання пам'яті (89 кБ) 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 (90 кБ), що суттєво знижує її інтегральну оцінку.

Таблиця 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

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.

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

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.

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.

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.

ОБГОВОРЕННЯ ТА РЕКОМЕНДАЦІЇ ДО ВПРОВАДЖЕННЯ

Дослідження захисту від XSS показують, що бібліотеки санітарної обробки є важливими, але далеко не безвідмовними, а їхня ефективність залежить від контексту, конфігурації та правильного використання.

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.

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.

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.

В оцінках та опитуваннях виявляється кілька повторюваних проблем:

- контекстно-нечутлива санітарна обробка для використання одного універсального санітарного засобу для HTML, атрибутів, URL-адрес та контекстів JavaScript дозволяє обійти ці помилки, оскільки кодування/екранування відрізняються залежно від контексту [1, 4];
- часткова інструментація, у якій розробники часто не застосовують санітарні засоби у всіх чутливих точках введення або помилково кодують вхідні дані замість вихідних, залишаючи прогалини, які можна використовувати [4];
- компроміс надмірної та недостатньої фільтрації: Бібліотеки або пропускають складні/заплутані корисні навантаження (недостатня фільтрація), або пошкоджують безпечний контент, агресивно видаляючи розмітку чи символи (надмірна фільтрація) [4, 5];
- конфігурація та оновлення бібліотек: Конфігурації за замовчуванням можуть бути слабкими, і бібліотеки повинні розвиватися з новими корисними навантаженнями та поведінкою браузера. Експерименти показують суттєве покращення лише після цілеспрямованого налаштування [5];
- статичні/автоматичні інструменти відновлення, такі як saferXSS, можуть знаходити вразливі приймачі та автоматично вставляти правильне вихідне кодування, видаляючи всі реальні вразливості XSS у тестових програмах, але є специфічними для мови, а не загальними бібліотеками [6];
- детектори машинного та глибокого навчання (наприклад, XSS-SAFE, CNN–LSTM та інші системи на основі машинного навчання) можуть виявляти відомі та обфусковані корисні навантаження XSS з дуже високою точністю та низьким рівнем хибнопозитивних результатів, але служать шарами виявлення, а не заміною для правильної санітарної обробки [7–10].

Загалом, бібліотеки санітарної обробки можуть бути високоефективними, коли;

- враховують контекст, ретельно протестовані на основі комплексних корпусів корисних даних та правильно налаштовані [4, 5];
- розробники точно розуміють, які контексти захищає певний кодер/очисник, та інструментують його в кожній відповідній точці виводу [1, 4].

Однак дослідження постійно показують, що одних лише бібліотек недостатньо для повного захисту від XSS і часто зазнають невдачі в реальному використанні через неправильну конфігурацію, неповне покриття та еволюціонуючу природу методів XSS [1, 3–5]. Вони найкраще працюють як частина багаторівневого захисту, який також включає безпечне шаблонування, CSP, автоматичне екранування на рівні фреймворку,

інструменти статичного аналізу/виправлення та (за бажанням) виявлення на основі машинного навчання.

ВИСНОВКИ ТА ПЕРСПЕКТИВИ ПОДАЛЬШИХ ДОСЛІДЖЕНЬ

В роботі проведено комплексне експериментальне дослідження ефективності бібліотек санітизації HTML для захисту від XSS-атак, що включало розробку репрезентативного датасету із 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.

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.

Подальші дослідження мають зосереджуватися на розширенні та ускладненні датасету XSS-векторів (зокрема мутаційних і framework-specific), вдосконаленні методології оцінювання безпеки, продуктивності й порівняльного аналізу санітизаційних бібліотек у поєднанні з defense-in-depth механізмами. Перспективними є автоматизація безперервного тестування (CI/CD, генерація нових векторів, ML-підходи), аналіз впливу нових браузерних стандартів та кросплатформне порівняння серверної й клієнтської санітизації в сучасних веб-архітектурах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Weinberger, J., Saxena, P., Akhawe, D., Finifter, M., Shin, R. & D. Song (2011). An Empirical Analysis of XSS Sanitization in Web Application Frameworks. Technical Report No. UCB/EECS-2011-11. Electrical Engineering and Computer Sciences University of California at Berkeley.
2. K. Patil, D., & R. Patil, K. (2015). Client-side Automated Sanitizer for Cross-Site Scripting Vulnerabilities. *Int. J. Comput. Appl., 121(20),* 1–8. https://doi.org/10.5120/21653-5063
3. Hydara, I., Sultan, A., Zulzalil, H., & Admodisastro, N. (2015). Current State of Research on Cross-Site Scripting (XSS) – A Systematic Literature Review. *Inf. Softw. Technol., 58,* 170–186. https://doi.org/10.1016/j.infsof.2014.07.010

4. Hannousse, A., Yahiouche, S., & Nait-Hamoud, M. (2024). Twenty-Two Years since Revealing Cross-Site Scripting Attacks: A Systematic Mapping and a Comprehensive Survey. *Comput. Sci. Rev., 52*, 100634. https://doi.org/10.1016/j.cosrev.2024.100634
5. Talib, N., & Doh, K. (2021). Assessment of Dynamic Open-Source Cross-Site Scripting Filters for Web Application. *KSII Trans. Internet Inf. Syst., 15*, 3750–3770. https://doi.org/10.3837/tiis.2021.10.015
6. Shar, L. K., & Tan, H. B. K. (2012). Automated Removal of Cross Site Scripting Vulnerabilities in Web Applications. *Inf. Softw. Technol., 54(5)*, 467–478. https://doi.org/10.1016/j.infsof.2011.12.006
7. Gupta, S., & Gupta, B. B. (2015). XSS-SAFE: A Server-Side Approach to Detect and Mitigate Cross-Site Scripting (XSS) Attacks in JavaScript Code. *Arab. J. Sci. Eng., 41(3)*, 897–920. https://doi.org/10.1007/s13369-015-1891-7
8. Tadhani, J. R., Vekariya, V., Sorathiya, V., Alshathri, S., & El-Shafai, W. (2024). Securing Web Applications against XSS and SQLi Attacks using a Novel Deep Learning Approach. *Sci. Rep., 14(1)*. https://doi.org/10.1038/s41598-023-48845-4
9. Ibrahim Khalaf, O., Sokiyna, M., Alotaibi, Y., Alsufyani, A., & Alghamdi, S. (2021). Web Attack Detection Using the Input Validation Method: DPDA Theory. *Comp. Material. Continua, 68(3)*, 3167–3184. https://doi.org/10.32604/cmc.2021.016099
10. Oshoiribhor, E., & John-Otumu, A. (2025). XSS-Net: An Intelligent Machine Learning Model for Detecting Cross-Site Scripting (XSS) Attack in Web Application. *Machin. Learn. Res., 10(1)*, 14–24. https://doi.org/10.11648/j.mlr.20251001.12

КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 3 (31), 2025
ISSN 2663 - 4023

Volodymyr Y. Sokolov
Ph.D., associate professor
associate professor of the Department of Information and Cyber Security
named after Professor Volodymyr Buriachok
Borys Grinchenko Kyiv Metropolitan University, Kyiv, Ukraine
ORCID: 0000-0002-9349-7946
v.sokolov@kubg.edu.ua

Bogdan V. Polikovskyi
master of the Department of Information and Cyber Security
named after Professor Volodymyr Buriachok
Borys Grinchenko Kyiv Metropolitan University, Kyiv, Ukraine
ORCID: 0009-0005-4821-8089
bvpolikovskyi.fitm24m@kubg.edu.ua

Maksym V. Vorokhob
Ph.D. in Cybersecurity
associate professor of the Department of Information and Cyber Security
named after Professor Volodymyr Buriachok
Borys Grinchenko Kyiv Metropolitan University, Kyiv, Ukraine
ORCID: 0000-0001-5160-7134
m.vorokhob@kubg.edu.ua

Oleksandr O. Tsyrul
postgraduate
Institute of Telecommunications and Global
Information Space of the NASof Ukraine, Kyiv, Ukraine
ORCID:0009-0002-5945-5918
tsyruloleksandr@gmail.com

RESEARCH ON THE EFFECTIVENESS OF SANITIZATION LIBRARIES FOR XSS ATTACKS IN WEB APPLICATIONS

Abstract. Cross-Site Scripting (XSS) attacks remain one of the most prevalent and critical vulnerabilities in modern web applications, as they allow attackers to execute arbitrary malicious code in the user's browser, compromising confidentiality, integrity, and availability of data. One of the key approaches to mitigating XSS is the use of sanitization libraries designed to clean or safely transform user input before it is processed and rendered. This article presents a comprehensive experimental study of the effectiveness of popular HTML sanitization libraries in the context of protecting web applications against XSS attacks. A specialized dataset of 100 unique XSS vectors is proposed and utilized, covering both classical attack scenarios (script tags, event handlers) and modern, less obvious techniques, including CSS injections, SVG-based vectors, DOM clobbering, encoded payloads, and abuse of contemporary browser APIs. To conduct the experiments, an automated testing framework based on Node.js and browser emulation tools was developed, enabling realistic reproduction of malicious code execution conditions. A comparative analysis of DOMPurify, js-xss, sanitize-html, and OWASP Java HTML Sanitizer was performed using their default configurations and evaluated according to XSS blocking rate, performance, and memory consumption, as well as through a multi-criteria assessment considering security, maintainability, and practical applicability. The experimental results demonstrate that none of the analyzed libraries provides complete out-of-the-box protection, while a common weakness across all solutions is vulnerability to DOM clobbering and encoded attack vectors. Based on the findings, practical recommendations are formulated regarding the configuration and deployment of sanitization libraries as part of a defense-in-depth strategy for modern web applications.

Keywords: XSS attacks, cross-site scripting, web security, data sanitization, Content Security Policy, input validation, DOM-based XSS, cybersecurity, web application protection.

REFERENCES

1. Weinberger, J., Saxena, P., Akhawe, D., Finifter, M., Shin, R. & D. Song (2011). An Empirical Analysis of XSS Sanitization in Web Application Frameworks. Technical Report No. UCB/EECS-2011-11. Electrical Engineering and Computer Sciences University of California at Berkeley.
2. K. Patil, D., & R. Patil, K. (2015). Client-side Automated Sanitizer for Cross-Site Scripting Vulnerabilities. *Int. J. Comput. Appl., 121(20)*, 1–8. https://doi.org/10.5120/21653-5063
3. Hydara, I., Sultan, A., Zulzalil, H., & Admodisastro, N. (2015). Current State of Research on Cross-Site Scripting (XSS) – A Systematic Literature Review. *Inf. Softw. Technol., 58*, 170–186. https://doi.org/10.1016/j.infsof.2014.07.010
4. Hannousse, A., Yahiouche, S., & Nait-Hamoud, M. (2024). Twenty-Two Years since Revealing Cross-Site Scripting Attacks: A Systematic Mapping and a Comprehensive Survey. *Comput. Sci. Rev., 52*, 100634. https://doi.org/10.1016/j.cosrev.2024.100634
5. Talib, N., & Doh, K. (2021). Assessment of Dynamic Open-Source Cross-Site Scripting Filters for Web Application. *KSII Trans. Internet Inf. Syst., 15*, 3750–3770. https://doi.org/10.3837/tiis.2021.10.015
6. Shar, L. K., & Tan, H. B. K. (2012). Automated Removal of Cross Site Scripting Vulnerabilities in Web Applications. *Inf. Softw. Technol., 54(5)*, 467–478. https://doi.org/10.1016/j.infsof.2011.12.006
7. Gupta, S., & Gupta, B. B. (2015). XSS-SAFE: A Server-Side Approach to Detect and Mitigate Cross-Site Scripting (XSS) Attacks in JavaScript Code. *Arab. J. Sci. Eng., 41(3)*, 897–920. https://doi.org/10.1007/s13369-015-1891-7
8. Tadhani, J. R., Vekariya, V., Sorathiya, V., Alshathri, S., & El-Shafai, W. (2024). Securing Web Applications against XSS and SQLi Attacks using a Novel Deep Learning Approach. *Sci. Rep., 14(1)*. https://doi.org/10.1038/s41598-023-48845-4
9. Ibrahim Khalaf, O., Sokiyna, M., Alotaibi, Y., Alsufyani, A., & Alghamdi, S. (2021). Web Attack Detection Using the Input Validation Method: DPDA Theory. *Comp. Material. Continua, 68(3)*, 3167–3184. https://doi.org/10.32604/cmc.2021.016099
10. Oshoiribhor, E., & John-Otumu, A. (2025). XSS-Net: An Intelligent Machine Learning Model for Detecting Cross-Site Scripting (XSS) Attack in Web Application. *Machin. Learn. Res., 10(1)*, 14–24. https://doi.org/10.11648/j.mlr.20251001.12