Вінницький національний технічний університет

Факультет менеджменту та інформаційної безпеки

Кафедра менеджменту та безпеки інформаційних систем

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка програмного модуля для захисту веб-додатків від XSS-атак з використанням адаптивної фільтрації та ескейпінгу вхідних даних на серверній стороні з автоматичним оновленням правил безпеки»

Виконав: студент 4 курсу, гр.1КІТС-21б
спеціальності 125– Кібербезпека
Освітня програма – Кібербезпека
інформаційних технологій та систем
(шифр і назва напряму підготовки, спеціальності)

Вітковський А. В.
(прізвище та ініціали)

Керівник: д.ф. (PhD), доц., доцент каф. МБІС
Салієва О. В.
(прізвище та ініціали)
«____» _____ 2025 р.

Рецензент: к.т.н., доц., доцент каф. ОТ
Томчук М.А.
(прізвище та ініціали)
«____» _____ 2025 р.

Допущено до захисту
Голова секції УБ кафедри МБІС
д.т.н., проф. Яремчук Ю.Є. _____
"_____" _____2025р.

Вінниця ВНТУ – 2025

Вінницький національний технічний університет
Факультет менеджменту та інформаційної безпеки
Кафедра менеджменту та безпеки інформаційних систем

Рівень вищої освіти І-й (бакалаврський)
Галузь знань 12 – Інформаційні технології
Спеціальність 125 – Кібербезпека
Освітньо-професійна програма - Кібербезпека інформаційних технологій та систем

ЗАТВЕРДЖУЮ
Голова секції УБ, кафедра МБІС

д.т.н., проф. Яремчук Ю.Є.
" 20 " березня 2025 р.

З А В Д А Н Н Я
на бакалаврську кваліфікаційну роботу студенту
Вітковському Андрію Валентиновичу
(прізвище, ім`я, по-батькові)

1. Тема роботи Розробка програмного модуля для захисту веб-додатків від XSS-атак з використанням адаптивної фільтрації та ескейпінгу вхідних даних на серверній стороні з автоматичним оновленням правил безпеки.

Керівник роботи д.ф. (PhD), доц. кафедри МБІС Салієва Ольга Володимирівна
(прізвище, ім`я, по-батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від " 20 " березня 2025року № 97
2. Термін подання студентом роботи 2 червня 2025 року
3. Вихідні дані до роботи є технічне завдання, вимоги до захисту веб-додатків згідно з OWASP, перелік типових XSS-атак, шаблони фільтрації, структура вхідних та вихідних даних, а також вимоги до реалізації REST API. Застосовуються сучасні засоби фільтрації, ескейпінгу, база даних MySQL та Node.js як серверне середовище.
4. Зміст текстової частини: Текстова частина містить вступ, теоретичний огляд XSS-атак, аналіз існуючих рішень, постановку задач, розробку структури модуля та бази даних, опис алгоритму захисту, реалізацію, тестування і висновки. Додатки містять код, приклади, ілюстрації та технічну документацію.
5. Перелік ілюстративного матеріалу (з точним зазначенням обов`язкових креслень)
Ілюстративний матеріал включає ER-діаграму, блок-схему алгоритму, скриншоти з Postman, фрагменти таблиць бази даних та порівняльні таблиці. Обов'язковими є схема роботи модуля захисту та структура бази даних.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Розділ I	доцент каф. МБІС Салієва О. В.		
Розділ II	доцент каф. МБІС Салієва О. В.		
Розділ III	доцент каф. МБІС Салієва О. В.		

7.Дата видачі завдання 20 березня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№	Назва та зміст етапу	Термін виконання		Примітка
		початок	закінчення	
1	Визначення напрямку бакалаврської роботи, формулювання теми	20.03.2025	23.03.2025	Виконано
2	Аналіз предметної області обраної теми	24.03.2025	13.04.2025	Виконано
3	Розробка алгоритму роботи	14.04.2025	27.04.2025	Виконано
4	Написання бакалаврської роботи на основі розробленої теми	28.04.2025	25.05.2025	Виконано
5	Передзахист бакалаврської кваліфікаційної роботи	26.05.2025	27.05.2025	Виконано
6	Виправлення, уточнення, корегування бакалаврської дипломної роботи	28.05.2025	02.06.2025	Виконано
7	Захист бакалаврської кваліфікаційної роботи	09.06.2025	10.06.2025	Виконано

Студент (підпис) Віниковський Н.В. (прізвище та ініціали)

Керівник роботи (підпис) Ольга Салієва (прізвище та ініціали)

АНОТАЦІЯ

Бакалаврська кваліфікаційна робота присвячена розробці програмного модуля для захисту веб-додатків від Cross-Site Scripting (XSS) атак, які є найбільш поширеними та небезпечними для сучасних веб-ресурсів. Метою роботи є створення адаптивного модуля безпеки, який забезпечує надійний захист шляхом фільтрації та перевірки вхідних даних на стороні сервера. Запропонований підхід включає автоматичне оновлення правил безпеки та дозволяє налаштовувати параметри безпеки відповідно до конкретних системних вимог, забезпечуючи ефективний та гнучкий захист від мінливих загроз.

У роботі проаналізовано типи XSS-атак (Stored, Reflected та DOM-based XSS), розглянуто їх характеристики та способи використання. Огляд існуючих рішень для захисту від XSS-атак дозволяє обґрунтувати вибір оптимальних методів, придатних для реалізації в розробленому модулі. Також проаналізовано технології серверної фільтрації, скринінгу та адаптивного оновлення правил, які є ключовими для динамічного захисту сучасних веб-додатків.

На основі проведеного дослідження запропоновано методологію розробки модуля, що використовує статичну фільтрацію та скринінг, а також автоматичне оновлення правил безпеки. Структура модуля включає компоненти для обробки вхідних даних, перевірки XSS вразливостей та гнучкого налаштування параметрів безпеки. Це дозволяє адаптувати модуль до різних типів даних, досягаючи оптимального балансу між безпекою та продуктивністю.

Ключові слова: XSS-атаки, веб-додатки, фільтрація, ескейпінг, адаптивний захист, серверна перевірка, контекстна обробка, DOM-аналіз, шаблони загроз, Node.js, MySQL, автоматичне оновлення правил, інформаційна безпека, REST API.

ABSTRACT

The bachelor's qualification work is dedicated to the development of a software module for protecting web applications from Cross-Site Scripting (XSS) attacks, which are among the most common and dangerous for modern web resources. The purpose of the work is to create an adaptive security module that ensures reliable protection through filtering and validating incoming data on the server side. The proposed approach includes automatic updating of security rules and allows configuring security parameters according to specific system requirements, providing effective and flexible protection against evolving threats.

The work analyzes types of XSS attacks (Stored, Reflected, and DOM-based XSS), considers their characteristics and methods of exploitation. A review of existing solutions for protecting against XSS attacks justifies the selection of optimal methods suitable for implementation in the developed module. The study also analyzes technologies of server-side filtering, screening, and adaptive rule updating, which are essential for dynamic protection of modern web applications.

Based on the conducted research, a methodology for developing the module is proposed, which uses static filtering and screening, as well as automatic updating of security rules. The structure of the module includes components for processing input data, checking for XSS vulnerabilities, and flexible configuration of security parameters. This allows the module to be adapted to different types of data, achieving an optimal balance between security and performance.

Keywords: XSS attacks, web applications, filtering, escaping, adaptive protection, server-side validation, contextual processing, DOM analysis, threat patterns, Node.js, MySQL, automatic rule updating, information security, REST API.

ЗМІСТ

ВСТУП

Актуальність роботи. Безпека веб-додатків є ключовою проблемою в сучасних інформаційних технологіях, оскільки кількість атак, спрямованих на компрометацію даних користувачів і порушення роботи веб-систем, продовжує зростати з кожним роком. Однією з найбільш поширених і небезпечних вразливостей є міжсайтовий скриптинг (XSS), який дозволяє зловмисникам впроваджувати шкідливий код у веб-сторінки, отримуючи несанкціонований доступ до даних користувача. Це створює значні загрози для конфіденційності та цілісності даних, потенційно ставить під загрозу стабільність системи. Тому розробка надійного програмного модуля для захисту веб-додатків від XSS-атак є актуальною і потребує ґрунтовних досліджень.

Крім того, особливого значення набуває створення адаптивних рішень. Ці рішення повинні не тільки захищати від відомих загроз, але й автоматично підлаштовуватися під нові вразливості, які можуть виникнути через динамічну природу веб-додатків. Впровадження адаптивної фільтрації та скринінгу вхідних даних на стороні сервера підвищує ефективність протидії XSS-атакам за рахунок аналізу контексту вхідних даних та налаштування захисту на основі поточних ризиків.

Метою роботи є розробка програмного модуля, який захищає веб-додатки від XSS-атак шляхом статичної фільтрації та скринінгу вхідних даних на стороні сервера, з автоматичним оновленням правил безпеки. Для досягнення цієї мети були поставлені наступні **завдання:**

– проаналізувати типи XSS-атак, способи їх використання та визначити найбільш ефективні заходи протидії;

– дослідити та порівняти існуючі рішення для протидії XSS-атакам, оцінити їх переваги та недоліки;

– розробити методологію адаптивної фільтрації та скринінгу вхідних даних для зменшення ризику XSS-атак;

– створити гнучку архітектуру модуля безпеки, що дозволяє конфігурувати параметри безпеки;

– реалізувати та протестувати модуль для оцінювання його ефективності та продуктивності.

Об'єктом дослідження є веб-додатки та їх вразливості до XSS-атак.

Предметом дослідження є методи адаптивної фільтрації, скринінгу вхідних даних та автоматичного оновлення правил для захисту веб-додатків.

Новизна роботи. В даному дослідженні представлено адаптивний підхід до фільтрації та скринінгу, який враховує різні типи вхідних даних, забезпечуючи захист від різних форм XSS-атак. Крім того, вдосконалено метод оновлення правил безпеки, що дозволяє швидко адаптуватися до нових загроз без втручання користувача.

Практична цінність роботи. Розроблений програмний модуль може бути інтегрований у веб-додатки, пропонуючи надійний захист від XSS-атак та зменшуючи потребу в постійному оновленні системи завдяки адаптивному управлінню правилами безпеки.

1 АНАЛІЗ ТЕОРЕТИЧНИХ АСПЕКТІВ ЗАХИСТУ ВІД XSS-АТАК

У цьому розділі буде проаналізовано основні типи XSS (Cross-Site Scripting) атак, включаючи Stored, Reflected та DOM-атаки, висвітлено їхні особливості та способи використання. Буде описано основні стратегії захисту від XSS-атак та проведено оцінювання сучасних підходів сучасні підходи до фільтрації та перевірки вхідних даних. Крім того, буде обґрунтовано вибір технологій для розробки захисного програмного модуля.

1.1 Класифікація XSS-атак

Перш за все, міжсайтовий скриптинг (XSS) являє собою категорію кібератак, в яких зловмисники впроваджують шкідливий JavaScript код у веб-сторінки, що відображаються іншим користувачам. Ця вразливість виникає, коли веб-додаток не обробляє або не фільтрує належним чином вхідні дані, що уможливлює виконання небажаних сценаріїв на стороні клієнта. XSS-атаки особливо згубні через те, що їх важко виявити, а також через потенційно катастрофічні наслідки, які вони можуть спричинити – від крадіжки особистих даних до повного контролю над обліковим записом користувача. Існує три основні категорії XSS-атак, кожна з яких має свої особливості та методи використання: Stored XSS, Reflected XSS та XSS на основі DOM (рис. 1.1) [1].

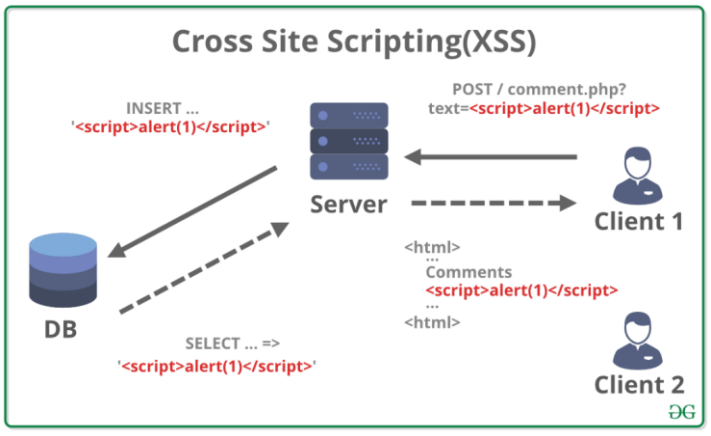


Рисунок 1.1 – XSS атака [1]

Різні форми міжсайтових скриптових атак (XSS) можна розділити на дві основні категорії: збережені XSS (також відомі як персистентні XSS). Це один з

найнебезпечніших типів XSS-атак, оскільки шкідливий код зберігається на сервері додатків (наприклад, у базі даних). При взаємодії з контентом, в якому зберігається шкідливий код (наприклад, при читанні коментарів або перегляді профілю), браузер автоматично виконує код. Збережений XSS дозволяє зловмиснику не лише викрасти персональні дані, але й отримати контроль над сеансами користувачів або навіть поширювати шкідливе програмне забезпечення. Ця форма атаки особливо небезпечна в контексті соціальних мереж і форумів, де значна кількість користувачів має доступ до спільного контенту.

Розглянемо відмінності між різноманітними XSS-атаками (табл. 1.1).

Таблиця 1.1 – Відмінності між різними типами XSS-атак

Тип XSS-атаки	Місце впровадження	Механізм атаки	Приклад використання	Рівень загрози
Stored (персистентна)	Сервер (база даних, журнали)	Шкідливий код зберігається на сервері (наприклад, у базі даних) і виконується щоразу, коли сторінка завантажується іншим користувачем.	Користувач залишає коментар з шкідливим <script>, який викрадає cookie тих, хто переглядає сторінку з коментарем.	Високий
Reflected (відображена)	Веб-сервер, але шкідливий код не зберігається	Зловмисник вставляє шкідливий код у запит (URL або форму), і сервер «відображає» цей код у відповіді. Виконується в браузері користувача.	Зловмисник надсилає посилання з URL: http://example.com?search=<script>alert("XSS")</script>, і скрипт виконується в браузері жертви.	Середній
DOM-based	Обробка на стороні клієнта (браузер, DOM-дерево сторінки)	Шкідливий код впроваджується в об'єктну модель документа (DOM) і виконується без взаємодії із сервером, використовуючи JavaScript або інші клієнтські технології.	URL: http://example.com/#<script>alert('XSS')</script>. Браузер обробляє фрагмент сторінки і виконує скрипт у DOM.	Високий

Відображений XSS – це тип атаки, який ґрунтується на відображенні шкідливого коду в URL-запитах. У більшості випадків зловмисник створює URL-адресу зі шкідливим кодом, який потім передається жертві. Після відкриття URL-адреси в браузері жертви виконується шкідливий скрипт. Відображений XSS є поширеною тактикою фішингових атак, коли зловмисники обманом змушують користувачів натискати на посилання, створюючи враження, що воно є легітимним. Основною метою таких атак є отримання файлів cookie, токенів автентифікації або перенаправлення користувачів на шкідливі веб-сайти (рис. 1.2) [2].

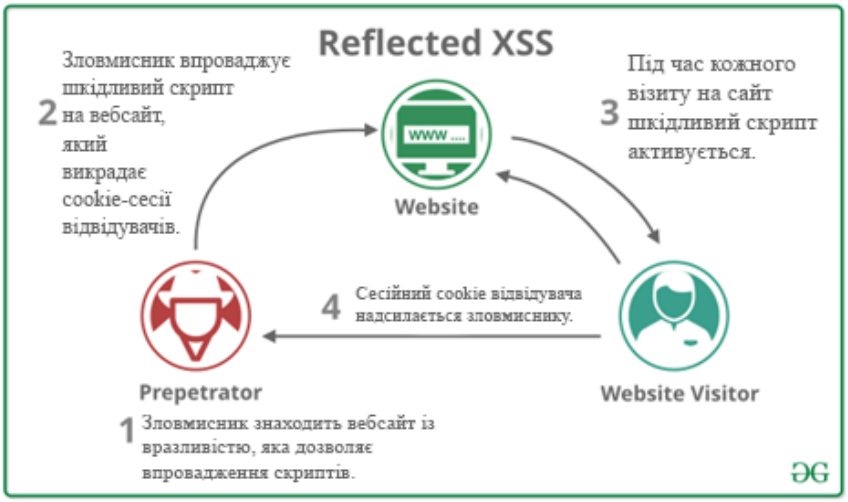


Рисунок 1.2 – Відображена XSS [1]

Ще одна категорія міжсайтових сценаріїв – це XSS на основі DOM, яка виникає, коли вразливість існує виключно на стороні клієнта, тобто при обробці об'єктної моделі документа (DOM) в браузері користувача. У цьому випадку сервер взагалі не взаємодіє зі шкідливим кодом, а атака здійснюється через маніпуляції з DOM, часто за допомогою JavaScript. Зловмисник змінює структуру або вміст сторінки, наприклад, передаючи шкідливі параметри в URL-адреси, які обробляються браузером як частина сторінки. Складність виявлення таких атак полягає в тому, що шкідливий код не залишає жодних слідів на сервері, а впливає безпосередньо на браузер жертви [3].

Потенціал міжсайтових скриптових атак (XSS) становить значний ризик для інтернет-користувачів. Одним з особливо небезпечних аспектів таких атак є перехоплення сеансів користувача, що може призвести до крадіжки конфіденційної

інформації або виконання несанкціонованих транзакцій. Зловмисник може скористатися вразливістю в системі, щоб отримати несанкціонований доступ до облікових записів користувачів, перехопивши файли cookie. Це особливо актуально у випадку з конфіденційними акаунтами, такими як електронна пошта та платіжні рахунки (рис.1.3).

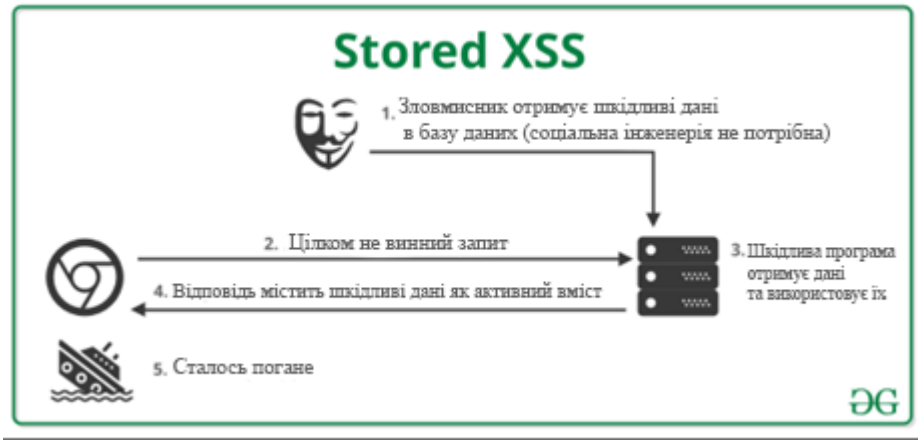


Рисунок 1.3 – Збережений тип атаки [1]

Використання фішингу та методів соціальної інженерії є ще одним напрямком атак. Атаки на міжсайтовий скриптинг (XSS) часто використовуються для створення оманливих інтерфейсів з метою обманом змусити користувачів розкрити свої облікові дані або іншу конфіденційну інформацію. Наприклад, зловмисник може представити форму для входу в систему, яка виглядає автентичною, але насправді передає введені дані до власної бази даних [4].

В результаті користувач перенаправляється на шкідливий веб-сайт. За допомогою XSS-атаки зловмисник може автоматично перенаправити користувача на фішинговий або шкідливий веб-сайт, що може призвести до зараження системи користувача шкідливим програмним забезпеченням або крадіжки даних.

Маніпуляції з інтерфейсами – ще один спосіб експлуатації. Зловмисники можуть використовувати XSS для того, щоб змінити візуальне представлення та робочі характеристики веб-сторінки. Наприклад, в дизайн можуть бути включені нові елементи інтерфейсу, що змушує користувача взаємодіяти з ними, а отже, передавати свої дані зловмисникам.

Крадіжка конфіденційної інформації та кейлоггінг: зловмисники можуть впровадити в систему скрипти для реєстрації натискання клавіш, щоб відстежувати введення паролів, повідомлень та іншої конфіденційної інформації.

Використання привілейованого доступу: міжсайтовий скриптинг (XSS) може бути використаний для отримання доступу до функцій веб-додатків, до яких зловмисник зазвичай не має доступу. Це дозволяє зловмиснику розширити свої привілеї в системі, що може призвести до отримання повного контролю над додатком або сервісом.

Запобігання міжсайтовим скриптовим атакам (XSS) вимагає багатогранної стратегії.

Впровадження процедур фільтрації та перевірки даних має важливе значення для запобігання XSS-атакам. Необхідно, щоб усі вхідні дані проходили фільтрацію та екранування, щоб запобігти виконанню шкідливого коду. Особливо важливо відфільтровувати спеціальні символи, в тому числі менш поширені, такі як <, >, &, які використовуються в HTML [5].

Рекомендується впровадити політику безпеки вмісту (CSP), що підлягає виконанню. Політика безпеки вмісту (CSP) обмежує джерела, з яких можуть виконуватися скрипти, дозволяючи використовувати тільки ті джерела, які були явно дозволені для JavaScript, CSS і медіаконтенту. Це значно зменшує ймовірність виконання шкідливих скриптів на сторінці.

Рекомендується використовувати сучасні бібліотеки та фреймворки, які часто містять вбудовані механізми безпеки. Сучасні фреймворки, такі як React або Angular, мають вбудовані засоби захисту від XSS-атак. Наприклад, вони автоматично кодують спеціальні символи в HTML, тим самим знижуючи ризик XSS [6].

Вкрай важливо впровадити процес регулярного оновлення правил і модулів безпеки. У сучасному цифровому середовищі зловмисники постійно шукають нові методи обходу заходів безпеки. Тому важливо використовувати адаптивні системи безпеки, які здатні автоматично оновлювати правила безпеки у відповідь на нові загрози.

Отже, атаки на міжсайтовий скриптинг (XSS) становлять значну і постійну небезпеку для сучасних веб-додатків, що вимагає багатогранного і постійно вдосконалюваного апарату безпеки. Реалізація ефективної стратегії захисту залежить від постійного моніторингу, використання сучасних технологій безпеки та дотримання найкращих практик кібербезпеки.

1.2 Методи та технології захисту від XSS-атак

Методи та технології захисту від XSS-атак є важливою частиною забезпечення безпеки веб-додатків. Також варто зазначити як сучасні фреймворки допомагають запобігти XSS (табл.1.2):

– React автоматично екранує всі значення, що вставляються в JSX, перетворюючи спеціальні символи (<, >, &) у безпечний формат. Наприклад, значення, вставлене через {someInput}, автоматично очищається від потенційно небезпечного HTML.

– Angular реалізує Strict Contextual Escaping (SCE), щоб захистити додаток від небезпечних виразів. Angular автоматично очищає всі дані, що вставляються в HTML, але для динамічного HTML (через [innerHTML]) рекомендується використовувати DomSanitizer.

Таблиця 1.2 – Порівняння методів захисту від XSS

Метод захисту	Stored XSS	Reflected XSS	DOM-based XSS	Особливості та зауваження
Фільтрація вхідних даних	Висока	Висока	Низька	Працює добре для обробки даних на сервері. Неефективна проти DOM-based XSS, оскільки вплив здійснюється на клієнті.
Екранування символів HTML	Середня	Середня	Низька	Запобігає виконанню шкідливих скриптів, але може бути обійдено, якщо атрибути обробляються без кодування.

Продовження табл.1.2.

Метод захисту	Stored XSS	Reflected XSS	DOM-based XSS	Особливості та зауваження
Політика безпеки вмісту (CSP)	Висока	Висока	Висока	Дуже ефективний метод, але потребує налаштування. Захищає від шкідливого контенту, вбудованого у веб-сторінки.
Бібліотеки для санітизації	Висока	Висока	Висока	Інструменти, як DOMPurify, працюють на клієнті, забезпечуючи очищення HTML перед обробкою.
Фреймворки (React, Angular)	Висока (за замовчуванням)	Висока (за замовчуванням)	Середня	React/Angular автоматично екранують HTML, але потрібен додатковий захист для впровадження небезпечного JavaScript.

Фільтрація та перевірка вхідних даних:

– одним з найважливіших методів захисту є перевірка і фільтрація вхідних даних на сервері. Будь-які дані, введені користувачами або з ненадійних джерел, повинні бути очищені до того, як вони потраплять в HTML-код;

– екранування символів HTML: перетворення спеціальних символів (наприклад, <, >, &, ', ') в їхній HTML-код, щоб запобігти виконанню шкідливого коду в браузері;

– фільтрація: видалення або зміна потенційно небезпечних тегів і атрибутів (наприклад, <script>, onload, onclick).

Ось основні підходи та інструменти, що використовуються для запобігання таким атакам (рис.1.4):

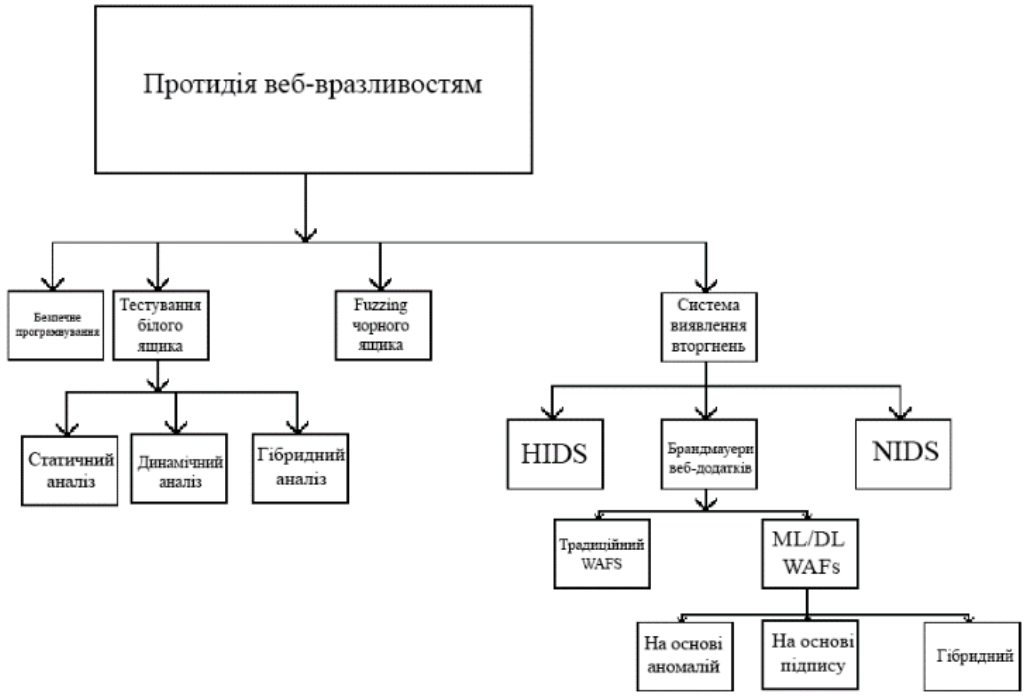


Рисунок 1.4 – Заходи боротьби з веб-уразливостями [7]

Політика безпеки вмісту (CSP):

– CSP це механізм безпеки, який допомагає виявляти і нейтралізувати певні типи атак, такі як XSS, шляхом обмеження джерел контенту, який може бути завантажений на веб-сторінку;

– CSP дозволяє вказати, які ресурси можуть бути завантажені та запущені на веб-сторінці, обмежуючи можливість виконання шкідливого коду. Наприклад, щоб запобігти виконанню скриптів з ненадійних доменів або вбудованих скриптів, які є поширеними цілями для XSS.

Екранування вхідних даних на сервері:

– для забезпечення безпеки на сервері важливо використовувати методи екранування даних, щоб запобігти небажаному виконанню скриптів;

– екранування JSON: Якщо дані передаються між сервером і клієнтом через JSON, важливо правильно обробляти і екранувати дані;

– екранування HTML: Використовуйте бібліотеки або фреймворки для автоматичного екранування вхідних даних, щоб запобігти їх інтерпретації як частини HTML-коду.

Фільтри XSS на рівні веб-браузерів, ряд веб-браузерів включають заходи захисту від XSS, які блокують будь-які скрипти, що вважаються підозрілими. Наприклад, механізми в Chrome, Firefox та інших браузерах призначені для автоматичного спрацьовування в разі спроби виконання скриптів з небезпечного джерела.

Автоматичне оновлення правил безпеки, створення системи, яка автоматично оновлює правила безпеки, дозволяє адаптувати захисні заходи до нових вразливостей. Такий підхід дозволяє зробити наступне. Рекомендується оновлювати фільтри або перевірки відповідно до появи нових типів атак. Можна підвищити здатність системи до адаптації у відповідь на зміни у веб-додатку або зміну тактики зловмисників.

Виявлення атак міжсайтового скриптингу (XSS), здійснюється за допомогою використання сигнатурних методологій, які передбачають порівняння вхідних даних з базою даних відомих шкідливих шаблонів. Підхід базується на зберіганні шаблонів шкідливих скриптів, які можуть бути виявлені під час обробки запитів.

Обмеження прав доступу (найменші привілеї), обмеження доступу до різних частин веб-додатку може допомогти мінімізувати шкоду від XSS-атак. Тому важливо, щоб користувачі та процеси мали лише ті права доступу, які їм потрібні для виконання своїх завдань.

Використання бібліотек та фреймворків:

– з метою захисту є життєздатним підходом. Розгортання спеціальних бібліотек, таких як OWASP Java HTML Sanitizer або DOMPurify, дозволяє забезпечити надійний захист від XSS-атак на рівні сервера або клієнта [7];

– DOMPurify це бібліотека, яка виконує функцію санітарної обробки небезпечного HTML перед його вставкою в об'єктну модель документа (DOM) (рис1.5) [1].

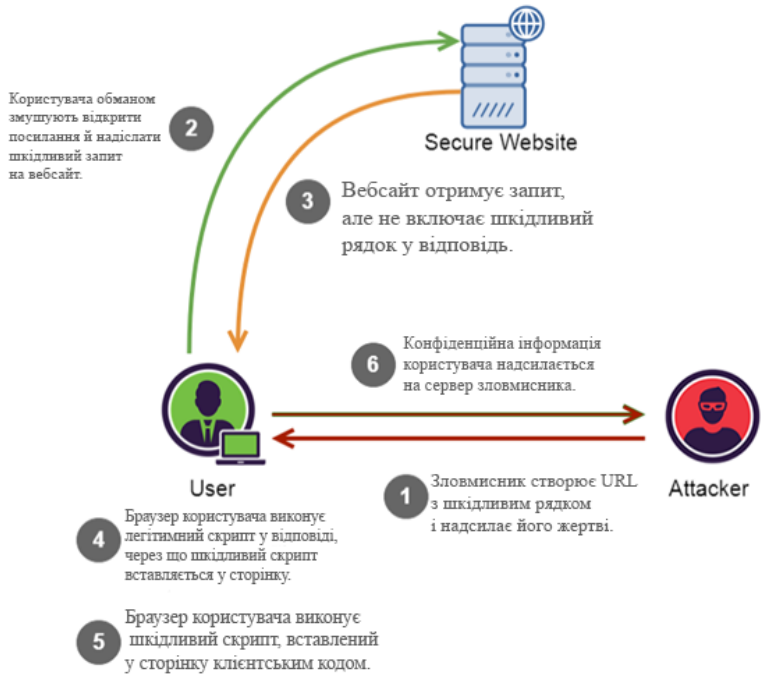


Рисунок 1.5 – XSS на основі DOM [9]

Важливо реалізувати валідацію даних як на стороні сервера, так і на стороні клієнта. Хоча перевірка даних на сервері є важливим кроком, не менш важливою є перевірка даних на стороні клієнта. Така система дозволяє виявляти помилки та небезпечні дані ще на етапі взаємодії користувача з веб-сторінкою. Використання регулярних виразів для перевірки формату даних дозволяє запобігти ненавмисному впровадженню шкідливих скриптів.

Використання файлів cookie та сесій у веб-розробці:

– для захисту від уразливостей міжсайтового скриптингу (XSS) необхідно застосовувати відповідні налаштування конфігурації для файлів cookie;

– використання атрибутів HttpOnly і Secure для файлів cookie може допомогти запобігти їх використанню через JavaScript;

– атрибут SameSite для файлів cookie дозволяє обмежити доступ до файлів cookie виключно у визначених контекстах, тим самим забезпечуючи захист від певних типів XSS-атак.

Регулярний аудит і тестування [7]:

– для забезпечення безпеки веб-додатків можна використовувати інструменти тестування вразливостей на постійній основі;

– інструменти OWASP ZAP (Zed Attack Proxy) і Burp Suite часто використовуються для сканування веб-додатків на наявність XSS та інших вразливостей. Рекомендується регулярно проводити тести на проникнення з метою виявлення нових вразливостей [7].

Захист веб-додатків від атак на міжсайтовий скриптинг (XSS) вимагає багаторівневого підходу, що охоплює як статичні методи фільтрації та скринінгу, так і більш динамічні методи, такі як автоматичне оновлення правил безпеки та використання політик безпеки контенту (CSP). Комплексний підхід забезпечує високий рівень захисту веб-додатків від цих поширених і небезпечних атак [4].

1.3 Порівняльний аналіз існуючих рішень

Для захисту веб-додатків від XSS-атак наразі існує ціла низка добре зарекомендованих себе рішень, які суттєво відрізняються за методами реалізації, можливостями налаштування, рівнем безпеки та зручністю використання.

Основними категоріями рішень є:

– системи на основі списків дозволених або заборонених елементів, у деяких системах безпеки для обмеження можливості впровадження шкідливого коду використовується список дозволених або заборонених HTML-елементів і атрибутів. Такі системи дозволяють фільтрувати або видаляти небезпечний вміст на основі заздалегідь визначеного списку елементів або атрибутів, які вважаються небажаними. Основна перевага такого підходу полягає в тому, що він легко адаптується і може бути швидко інтегрований в систему. Однак такі рішення мають обмежену гнучкість і можуть виявитися неефективними проти нових або еволюційних форм XSS-атак, оскільки атака з використанням дозволених елементів все одно може бути успішною [10];

– автоматизовані системи екранування вхідних даних, деякі рішення застосовують екранування вхідних даних перед їх відображенням у веб-додатку. Такі системи автоматично перетворюють спеціальні символи в безпечний формат, що знижує ймовірність виконання небажаного коду. Екранування є ефективним

методом для простих форм захисту, однак його може бути недостатньо для складних випадків, таких як динамічна генерація JavaScript;

– інструменти аналізу та валідації коду, інструменти статичного аналізу коду, такі як SonarQube або OWASP ZAP, полегшують виявлення потенційних вразливостей на етапі розробки. Це досягається шляхом сканування коду на наявність потенційно шкідливих конструкцій, які можуть призвести до міжсайтових скриптових атак (XSS). Такі інструменти допомагають розробникам виявляти вразливості ще до розгортання коду. Основним недоліком статичного аналізу є те, що він не завжди здатен виявити складні вразливості, а також вимагає додаткових ресурсів для інтеграції та підтримки в середовищі розробки [11].

Платформи веб-безпеки, які інтегрують захист від XSS, використовують комбінований підхід, що включає динамічний аналіз трафіку та блокування підозрілих запитів. Прикладами таких платформ є Cloudflare, Akamai Kona Site Defender та Amazon Web Application Firewall (WAF). Такі рішення забезпечують високий рівень захисту, фільтруючи атаки в режимі реального часу. Основною перевагою цих систем є комплексний підхід до захисту від цілого ряду потенційних загроз. Однак їх значна вартість і необхідність постійного моніторингу та налаштування є суттєвими недоліками (рис.1.6).

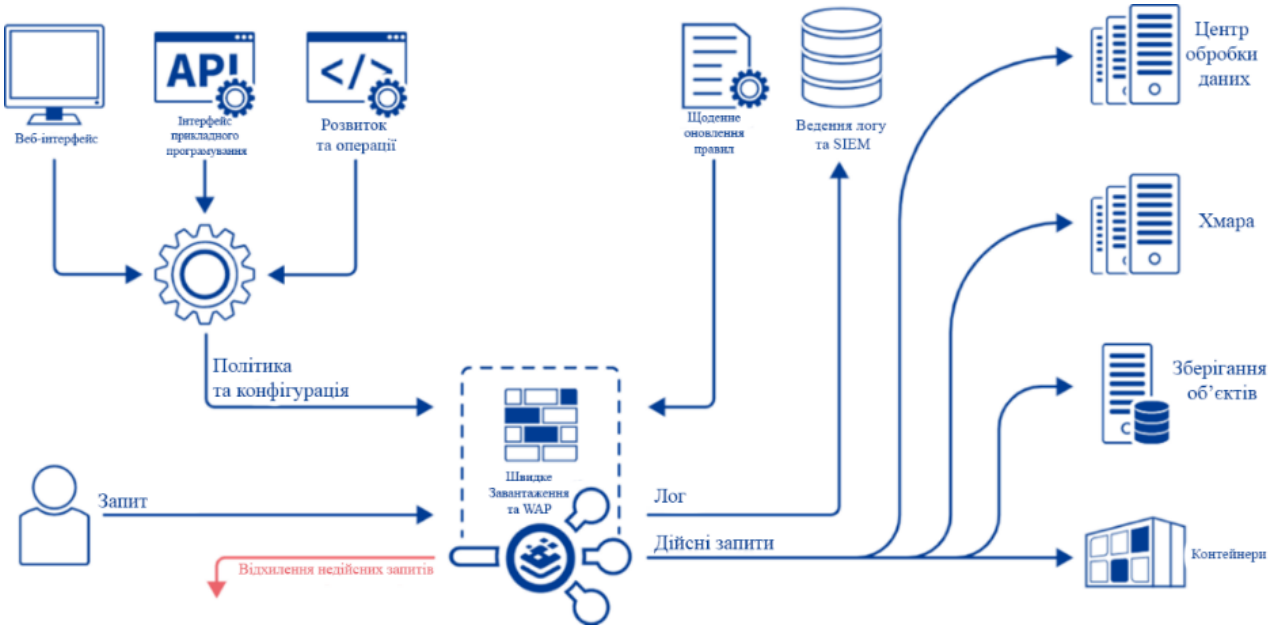


Рисунок 1.6 – Amazon Web Application Firewall [16]

Включення захисту від міжсайтового скриптингу (XSS) у веб-фреймворки та бібліотеки стало стандартною функцією в сучасній веб-розробці. Такі популярні фреймворки, як React, Angular та Vue.js, включають запобігання XSS через екранування даних як стандартний захід безпеки. Автоматична обробка вхідних даних шляхом перетворення потенційно небезпечних символів значно знижує ризик XSS-атак. Такий підхід пропонує зручність на рівні розробки, однак він не позбавлений обмежень. Хоча фреймворки захищають певні дані, вони не захищають від усіх потенційних форм XSS, особливо при взаємодії із зовнішнім кодом або сторонніми бібліотеками [3].

Порівняльний аналіз різних методів захисту від XSS показує, що кожен з них має свої сильні та слабкі сторони. Наприклад, системи, засновані на списках дозволених елементів, ефективно регулюють вміст, але демонструють обмежену адаптивність. Використання екранування є ефективним методом захисту простих HTML-структур, але не забезпечує всебічного захисту від виконання динамічного коду. Інструменти статичного аналізу сприяють ранньому виявленню вразливостей, однак їх ефективність обмежується особливостями досліджуваного коду.

Інтеграція механізмів безпеки за замовчуванням у фреймворки є вигідною для розробників, оскільки спрощує процес реалізації заходів безпеки. Однак такої інтеграції може бути недостатньо для досягнення комплексного захисту на рівні сервера. Платформи безпеки, такі як WAF, забезпечують багаторівневий захист у режимі реального часу, але вимагають значних фінансових інвестицій [11].

Вибір методу захисту від XSS залежить від конкретних специфікацій веб-додатку, наявних фінансових ресурсів, наявності ресурсів підтримки та бажаного рівня безпеки. Для досягнення оптимальної ефективності рекомендується використовувати комбінований підхід, що включає статичний аналіз коду, витіснення даних, адаптивну фільтрацію та використання WAF-платформ, що забезпечить багаторівневий захист від XSS-атак.

1.4 Аналіз вимог до безпеки веб-додатків

Аналіз вимог до безпеки веб-додатків є ключовим етапом у забезпеченні захисту даних користувачів та відмовостійкості додатків. Сучасні веб-додатки вразливі до безлічі кіберзагроз, включаючи атаки на міжсайтовий скриптинг (XSS), SQL-ін'єкції, атаки на відмову в обслуговуванні (DoS), атаки на підробку міжсайтових запитів (CSRF) та інші. Звідси випливає, що необхідні заходи безпеки для таких додатків повинні охоплювати як технічні, так і організаційні аспекти [13].

Розглянемо фундаментальні вимоги до безпеки веб-додатків.

Аутентифікація та авторизація:

– процес автентифікації слугує для перевірки особи користувача. Необхідно, щоб безпечні додатки підтримували багатофакторну автентифікацію (2FA, 3FA) і використовували надійні паролі;

– авторизація гарантує, що користувачі отримують доступ виключно до тих ресурсів і функцій, які відповідають їхній ролі та обов'язкам [14].

Необхідно підтримувати конфіденційність і цілісність даних:

– передача даних захищена від перехоплення або несанкціонованого втручання завдяки використанню шифрування HTTPS (TLS);

– використання шифрування для зберігання конфіденційних даних, включаючи персональні дані та платіжну інформацію, є фундаментальною вимогою для захисту конфіденційності.

Вхідні дані від користувачів:

– для захисту від атак міжсайтового скриптингу (XSS), SQL-ін'єкцій та інших форм кіберзагроз необхідно, щоб вхідні дані користувачів проходили сувору перевірку і фільтрацію на рівні сервера;

– також необхідно впровадити екранування вхідних даних і політику безпеки контенту (Content Security Policy, CSP) для обмеження типів дозволеного контенту.

Управління сеансами є фундаментальним аспектом будь-якої комп'ютерної системи:

– використання захищених файлів cookie з атрибутами Secure та HttpOnly зменшує ризик перехоплення сеансу, захищаючи сеанси користувачів;

– використання технології JSON Web Token (JWT) у поєднанні з періодичним оновленням токенів для цілей автентифікації є надійним підходом до управління сеансами (рис.1.7) [15].

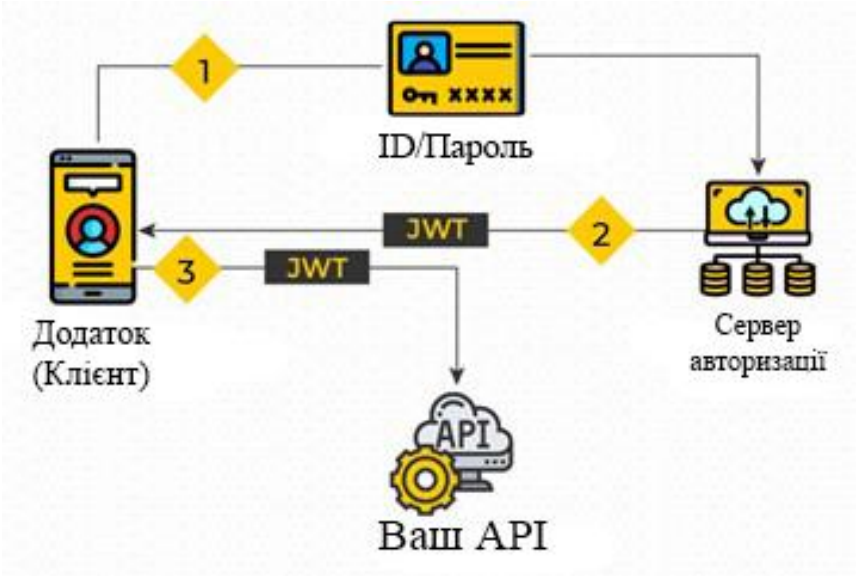


Рисунок 1.7 – JSON Web Token [17]

Запобігання міжсайтовим підробкам запитів (CSRF):

– атаки на підробку міжсайтових запитів (CSRF) обманом змушують користувачів виконувати небажані дії в додатках, в яких вони аутентифіковані. Необхідність реалізації CSRF-захисту досягається шляхом додавання до запитів CSRF-токенів;

– обмеження доступу до інтерфейсу прикладного програмування (API) є важливим заходом безпеки;

– для зменшення ризику несанкціонованого доступу до API рекомендується використовувати обмеження IP-адрес, токени доступу та ролі.

Інтеграція CSRF-захисту. CSRF-токени (Cross-Site Request Forgery). Кожна форма або запит має містити унікальний токен, який перевіряється сервером під час виконання запиту. Тобто спершу сервер генерує CSRF-токен і додає його до HTML-форм, потім клієнт надсилає цей токен разом із запитом, а вже під кінець сервер перевіряє токен перед виконанням дії.

Приклад у Node.js (з використанням бібліотеки csurf):

```
const csrf = require('csurf');
const express = require('express');
const app = express();
```

```
app.use(csrf({ cookie: true }));

app.get('/form', (req, res) => {
  res.render('form', { csrfToken: req.csrfToken() });
});

app.post('/submit', (req, res) => {
  // Запит успішно пройшов перевірку CSRF
  res.send('Form submitted!');
});
```

Атрибути HttpOnly і SameSite для файлів cookie:

HttpOnly. Цей атрибут запобігає доступу до файлів cookie через JavaScript, знижуючи ризик їх викрадення за допомогою XSS.

```
Set-Cookie: sessionId=abc123; HttpOnly
```

У цьому випадку JavaScript не може отримати доступ до файлу cookie sessionId.

SameSite. Цей атрибут обмежує надсилання файлів cookie для запитів із сторонніх сайтів, що захищає від CSRF-атак.

Strict. Файли cookie передаються лише при запитах із того ж домену.

Lax. Дозволяє файли cookie для GET-запитів (наприклад, переходів за посиланнями).

```
Set-Cookie: sessionId=abc123; SameSite=Strict
```

Виявлення та запобігання вторгненням:

– інтеграція інструментів моніторингу та аналітики дозволяє швидко виявляти підозрілу активність та оперативнцо реагувати на загрози;

– рекомендується розгортання систем виявлення вторгнень (IDS) і систем запобігання вторгненням (IPS).

Реєстрація та моніторинг даних є важливим аспектом будь-якої системи безпеки, тому слів віднести до них такі правила:

– вимогою безпеки є реєстрація подій у журналах, які дозволяють відстежувати спроби злому та несанкціонованого доступу;

– впровадження контролю доступу та обмеження прав на перегляд даних журналів запобігає будь-яким маніпуляціям з цими записами.

Існує низка загальновизнаних стандартів, таких як «Десять найкращих стандартів безпеки відкритих веб-додатків» (Open Web Application Security Project, OWASP), які описують найпоширеніші загрози для веб-додатків і способи їхнього запобігання. Крім того, ISO/IEC 27001 та NIST Cybersecurity Framework надають рекомендації щодо оптимальних практик забезпечення безпеки на рівні організації [12].

Всебічний аналіз вимог безпеки дозволяє розробляти веб-додатки, які є стійкими до поширених загроз, захищають конфіденційність даних і конфіденційність користувачів. Ретельне виконання цих вимог гарантує надійний захист як персональних, так і корпоративних додатків, тим самим підвищуючи їх надійність і якість обслуговування.

1.5 Висновки до розділу 1 та постановка задач

У першому розділі представлено всебічний аналіз небезпек, притаманних XSS-атакам, які є поширеною вразливістю сучасних веб-додатків. Було розглянуто характеристики трьох основних категорій XSS-атак: збережених, відображених та DOM-атак. Крім того, ретельному аналізу піддаються механізми їх реалізації та вплив на безпеку користувачів. Далі в розділі описано різні методи протидії XSS-атакам, включаючи використання технологій фільтрації та перевірки вхідних даних, екранування символів, а також сучасні підходи, такі як впровадження політики безпеки контенту (Content Security Policy, CSP) для обмеження джерел контенту.

Було проведено порівняльний аналіз існуючих рішень з метою виявлення їхніх переваг та недоліків. Цей аналіз став основою для вибору оптимальних засобів безпеки, здатних забезпечити надійний захист веб-додатків від XSS-атак. Крім того, дослідження вимог до безпеки веб-додатків передбачало вивчення основних принципів захисту даних, управління сесіями користувачів та обмеження доступу до потенційно шкідливого контенту. Комплексний захист від XSS-атак рекомендується досягати шляхом реалізації багаторівневих заходів, які включають як статичний аналіз коду, так і динамічне оновлення правил захисту.

В результаті аналізу були сформульовані вимоги до функціональності модуля безпеки, який слугуватиме для зниження ризику несанкціонованого доступу до даних користувачів та забезпечення динамічної адаптації до нових загроз. Ці висновки лягли в основу наступних етапів розробки, спрямованих на створення модуля, здатного автономно адаптуватися до змін у середовищі безпеки та задовольняти вимоги сучасних веб-додатків.

Опираючись на результати проведеного аналізу та виходячи з мети і актуальності обраної теми, були поставлені наступні задачі для подальшої роботи:

– проаналізувати типи XSS-атак, способи їх використання та визначити найбільш ефективні заходи протидії;

– дослідити та порівняти існуючі рішення для протидії XSS-атакам, оцінити їх переваги та недоліки;

– розробити методологію адаптивної фільтрації та скринінгу вхідних даних для зменшення ризику XSS-атак;

– створити гнучку архітектуру модуля безпеки, що дозволяє конфігурувати параметри безпеки;

– реалізувати та протестувати модуль для оцінювання його ефективності та продуктивності.

2 УДОСКОНАЛЕННЯ СИСТЕМИ ЗАХИСТУ ВІД XSS-АТАК

Сучасні веб-додатки стають мішенню нових кібератак, зокрема XSS. Для захисту від загроз необхідно впроваджувати надійні стратегії безпеки. Існуючі підходи, такі як статична фільтрація або списки дозволених елементів, часто обмежені у своїй здатності протидіяти динамічним загрозам через їхню недостатню гнучкість. Потрібні вдосконалення, щоб забезпечити своєчасне виявлення нових вразливостей і ефективний захист навіть у разі складних або цілеспрямованих атак.

2.1 Обґрунтування необхідності вдосконалення

Незважаючи на постійний розвиток засобів безпеки, атаки на міжсайтовий скриптинг (XSS) залишаються значною вразливістю сучасних веб-додатків. Ці атаки дозволяють зловмисникам виконувати шкідливі скрипти у веб-браузерах користувачів, викрадати облікові дані, модифікувати веб-контент або навіть використовувати пристрої жертв як платформу для подальших зловмисних дій. Наприклад, у 2022 році XSS-атака на один із популярних сервісів з обміну повідомленнями дозволила зловмисникам отримати доступ до акаунтів користувачів без їх відома, скориставшись недостатньо захищеним механізмом введення тексту.

Зростаюча складність та інтерактивність веб-додатків значно ускладнюють забезпечення їхнього захисту. Традиційні методи, такі як фільтрація вхідних даних або жорстке кодування правил безпеки, демонструють недостатню ефективність у світлі сучасних загроз. Наприклад, статичні фільтри часто не можуть впоратися з багаторівневими атаками, такими як використання різних форматів енкодингу (наприклад, UTF-8 чи Base64). Це підкреслює необхідність вдосконалення підходів до захисту [18].

Розглянемо основні недоліки існуючих рішень:

Явище обходу фільтрів, статичні правила, засновані на чорних списках, часто виявляються неефективними, коли зловмисники застосовують методи маскування або кодування. Наприклад, вразливість може залишитися невиявленою, якщо шкідливий код впроваджується у вигляді закодованого JavaScript.

Недостатня контекстна адаптація є ще одним недоліком існуючих рішень, сучасні веб-додатки використовують різноманітні контексти для обробки даних. Прикладами відповідних технологій є: HTML, CSS, JavaScript та JSON.

Відсутність гнучкості в роботі з кожним із цих контекстів збільшує ризик не виявлення загроз. Наприклад, система може бути налаштована на фільтрацію HTML, але не виявляти шкідливий код у JavaScript.

Проблеми, пов'язані з автоматизацією, значна кількість існуючих рішень потребує ручного налаштування та регулярного оновлення правил безпеки, що призводить до збільшення витрат на обслуговування та підвищеного ризику людських помилок

Інтеграція машинного навчання, незважаючи на значні перспективи машинного навчання у виявленні нових атак, його інтеграції в системи безпеки перешкоджають значні обчислювальні вимоги, пов'язані з ним.

Здатність пристосовуватися до зростання і змін є вирішальним аспектом масштабованості, рішення, засновані на ручному налаштуванні правил, часто виявляються недостатніми для забезпечення необхідної продуктивності у великих додатках з високим навантаженням. Наприклад, при обробці тисяч запитів в секунду вручну перевіряти кожен запит непрактично [19].

Для того, щоб забезпечити надійний захист сучасних веб-додатків від XSS-атак, необхідно впровадити систему, яка має наступні характеристики:

Контекстна адаптація, система повинна визначати контекст вхідних даних (наприклад, HTML, JavaScript, URL тощо) і застосовувати відповідний метод обробки. Наприклад, у випадку HTML система може використовувати екранування символів, тоді як у контексті JavaScript вона може проводити детальний синтаксичний аналіз.

Система повинна бути оснащена механізмом автоматичного оновлення правил безпеки, важливо, щоб система мала можливість динамічно оновлювати правила безпеки у відповідь на появу нових загроз. Це може бути досягнуто шляхом автоматичного завантаження нових шаблонів з бази даних загроз.

Розгортання машинного навчання, виявлення складних шаблонів загроз полегшується завдяки використанню алгоритмів машинного навчання. Наприклад, вхідні дані можуть бути проаналізовані для того, щоб визначити, чи є схожість з відомими атаками.

Система повинна бути здатна працювати на високому рівні продуктивності., використання сучасних алгоритмів обробки даних дозволяє системі забезпечувати захист у реальному часі, навіть за високих навантажень.

Очікується, що вдосконалена система захисту від XSS дасть наступні результати:

– ризик успішних атак буде зменшено завдяки поєднанню адаптивних правил та машинного навчання;

– автоматизація процесу оновлення правил зведе до мінімуму необхідність втручання адміністратора, що полегшить обслуговування системи;

– система буде достатньо гнучкою, щоб адаптуватися до нових типів загроз без необхідності повної реконфігурації;

– запропонований підхід полегшить захист великих систем з високою швидкістю обробки запитів завдяки використанню сучасних алгоритмів [20].

Забезпечення надійного захисту сучасних веб-додатків від атак типу XSS (Cross-Site Scripting) є критично важливим завданням для будь-якої організації, яка працює з веб-технологіями.

У зв'язку з постійним зростанням складності та динамічності веб-додатків, а також збільшенням кількості загроз, стандартні підходи до безпеки вже не можуть гарантувати належний рівень захисту.

Тому запропоновані вдосконалення, зокрема впровадження бібліотек для очищення та валідації даних, таких як DOMPurify, а також використання інструментів на основі штучного інтелекту, наприклад, TensorFlow.js, є надзвичайно важливими.

2.2 Робота з базами даних і обґрунтування вибору моделі

Для реалізації ефективної системи захисту від XSS-атак необхідно зберігати різні типи даних, зокрема вхідні дані користувачів, результати їх обробки, виявлені загрози та правила фільтрації. Саме тому вибір структури бази даних є критично важливим етапом у забезпеченні стабільності, масштабованості та функціональності програмного модуля.

Для зберігання та обробки даних у даній роботі було обрано реляційну СУБД MySQL. Цей вибір зумовлений такими факторами:

– MySQL забезпечує високу продуктивність при обробці великої кількості запитів, що є ключовим для системи, яка аналізує користувацькі дані в реальному часі;

– СУБД підтримує складні транзакції, індексацію, а також забезпечує цілісність зв’язків між таблицями, що критично для підтримки логіки фільтрації загроз;

– MySQL інтегрується з серверною частиною системи, реалізованою на Node.js, через бібліотеку mysql2, що дозволяє легко здійснювати взаємодію між модулем і БД.

У системі захисту реалізовано п’ять основних таблиць, кожна з яких відповідає за окремий аспект роботи з вхідними даними, шаблонами загроз і правилами фільтрації.

Logs таблиця журналу, яка реєструє вхідні дані користувачів, результати очищення та виявлені загрози.

– log_id (PK), унікальний ідентифікатор запису.

– user_input, sanitized_input, необроблені та оброблені дані.

– detected_threats, опис або кількість виявлених загроз.

– timestamp: час фіксації події.

Threat_Patterns зберігає шаблони виявлених XSS-атак.

– pattern_id (PK), ідентифікатор шаблону.

– pattern, регулярний вираз або рядок, що представляє загрозу.

– description, опис загрози.

– added_on, дата додавання шаблону.

Rules таблиця динамічних правил фільтрації, що використовуються при аналізі даних.

– rule_id (PK): унікальний ідентифікатор правила.

– pattern (FK): зовнішній ключ до таблиці Threat_Patterns.

– context: контекст застосування (HTML, JavaScript, JSON тощо).

– added_on, last_updated: дати створення та оновлення правила.

Sanitization_History фіксує, які методи очищення були застосовані, та їхню ефективність.

– history_id (PK), ідентифікатор запису.

– log_id (FK), посилання на відповідний запис у таблиці Logs.

– method_used, застосований метод фільтрації або ескейпінгу.

– effectiveness, ефективність у відсотках або за шкалою.

– timestamp, час застосування методу.

Подано ER-діаграму бази даних, яка демонструє зв’язки між таблицями та відображає логіку їх використання в модулі захисту від XSS (рис.2.1).

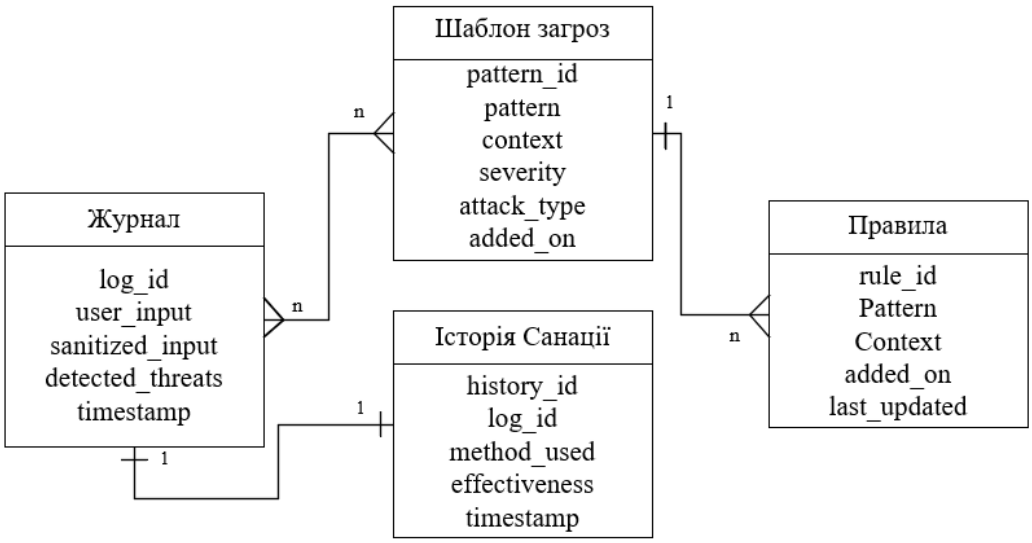


Рисунок 2.1 – ER-діаграма бази даних системи захисту від XSS

Розроблена реляційна структура забезпечує:

– логічну та надійну взаємодію між всіма елементами модуля (користувач

– вхідні дані виявлена загроза – метод захисту);

– можливість розширення (додавання нових правил, шаблонів, логів);

– автоматичне оновлення шаблонів за допомогою подій або сервісів (наприклад, вставка нових записів при виявленні DOM-загроз);

– зберігання результатів перевірок, що дозволяє провести подальший аналіз ефективності захисту.

2.3 Обґрунтування вибору технологій для динамічної фільтрації

Динамічна фільтрація є одним із ключових елементів захисту веб-додатків від атак типу XSS, оскільки дозволяє адаптувати механізми обробки вхідних даних до конкретного контексту їх використання. Це забезпечує більшу точність виявлення загроз та мінімізує кількість хибно позитивних або хибно негативних результатів. Для реалізації динамічної фільтрації у рамках програмного модуля були обрані технології, які поєднують ефективність, гнучкість та інтеграційні можливості. Цей підхід забезпечує другий рівень захисту після первинної фільтрації виконується глибший аналіз HTML-структури.

Серверна платформа Node.js була обрана як основа розробки модуля з огляду на її асинхронну архітектуру, що дозволяє обробляти одночасно велику кількість запитів із мінімальними затримками. Node.js є стабільною та перевіреною технологією, яка широко застосовується для створення захищених і високонавантажених систем. Платформа також підтримує розгалужену екосистему модулів, зокрема бібліотеки для роботи з регулярними виразами, логування подій та взаємодії з базами даних.

Для реалізації аналізу HTML-контенту у модулі використовується бібліотека JSDOM, яка дозволяє створити віртуальне середовище DOM без участі браузера. Це дає змогу перевіряти очищені дані на наявність прихованих загроз у структурі документа, зокрема скриптів, небезпечних атрибутів (таких як onerror, onload) або підозрілих посилань. Бібліотека дозволяє здійснити другий рівень перевірки після застосування регулярної фільтрації [27].

Основою первинного аналізу є набір регулярних виразів, які охоплюють найбільш типові шаблони XSS-атак. До них відносяться конструкції типу <script>, eval(), javascript: та інші варіанти небезпечного коду. Для кожного з таких шаблонів

у модулі задається відповідний контекст, у якому він може зустрічатися. Це дозволяє не лише визначити факт загрози, а й правильно обробити дані згідно з типом, через відповідний метод ескейпінгу (escapeHTML, escapeJS, escapeURL, escapeJSON).

Особливістю реалізації є підтримка автоматичного оновлення шаблонів загроз. У разі, якщо у процесі обробки вхідних даних було виявлено новий фрагмент, що відповідає потенційній загрозі, система:

– автоматично витягує його з вхідного тексту;

– перевіряє наявність у базі шаблонів;

– і, за відсутності збігів, додає його у вигляді нового запису з зазначенням контексту, рівня небезпеки та типу джерела (наприклад, DOM-based).

Таким чином, модуль фільтрації не лише виконує пасивне очищення даних, а й активно оновлює базу знань системи, адаптуючись до нових загроз без втручання адміністратора. Це дозволяє говорити про реалізацію механізмів адаптивного захисту, що є особливо актуальним для динамічних веб-додатків.

Усі вибрані технології сумісні між собою, не потребують складного налаштування, легко інтегруються у серверне середовище та забезпечують баланс між продуктивністю та безпекою.

2.4 Розробка алгоритму захисту: структура та вдосконалення

Розроблений алгоритм захисту базується на багаторівневому підході до перевірки вхідних даних із урахуванням контексту використання, евристичних правил та динамічного оновлення шаблонів. Структура модуля сформована таким чином, щоб забезпечити максимальну адаптивність до нових типів атак і змін середовища, у якому функціонує веб-додаток.

Побудова ефективного захисту від XSS вимагає не лише наявності статичних перевірок, а й впровадження адаптивного, контекстно-залежного механізму фільтрації. У межах даного модулю реалізовано алгоритм, який поєднує регулярну евристичну перевірку, контекстну обробку, DOM-аналіз та механізм автоматичного оновлення бази шаблонів загроз.

На відміну від класичних фільтрів, які працюють виключно за принципом чорного списку, запропоноване рішення орієнтоване на гнучку архітектуру. Всі правила та шаблони загроз зберігаються в базі даних і можуть автоматично оновлюватися у процесі роботи системи.

Загальна логіка алгоритму передбачає наступні етапи. Спочатку система приймає вхідні дані, після чого вони проходять первинне декодування, заміну HTML-сутностей і юнікод-послідовностей на відповідні символи. Це дозволяє зняти перший рівень обфускації, характерний для XSS-атак.

Далі визначається контекст використання даних HTML, JavaScript, JSON або URL. Для цього застосовується спеціалізована функція detectContext(), яка на основі аналізу синтаксису вхідного рядка встановлює відповідний режим обробки. Такий підхід дає змогу адаптувати фільтрацію та ескейпінг до типу вмісту.

Після встановлення контексту дані перевіряються на наявність відомих шаблонів атак, які зберігаються у таблицях rules та threat_patterns. Кожен шаблон представлено у вигляді регулярного виразу, що дозволяє здійснювати гнучке порівняння з вхідним текстом. Наприклад:

```
const patterns = [
  /<script.*?>.*?<\s*\/script>/gi,
  /on\w+\s*=\s*["']?.*?["']?/gi,
  /javascript:/gi,
  /eval\(.*?\)/gi
];
```

У разі збігу застосовується відповідна функція ескейпінгу залежно від контексту, escapeHTML, escapeJS, escapeJSON або escapeURL.

Після завершення регулярної перевірки система додатково виконує аналіз очищених даних за допомогою DOM-моделі, створеної через бібліотеку jsdom. Це дозволяє виявити складніші, замасковані або залишкові конструкції, які могли пройти попередню фільтрацію. Зокрема, виявляються атрибути onload, onerror, а також небезпечні посилання на javascript:.

З роботи модуля побудовано блок-схему, що відображає обробку даних, визначення контексту, фільтрацію та збереження результатів у базіметою візуалізації (рис.2.2).

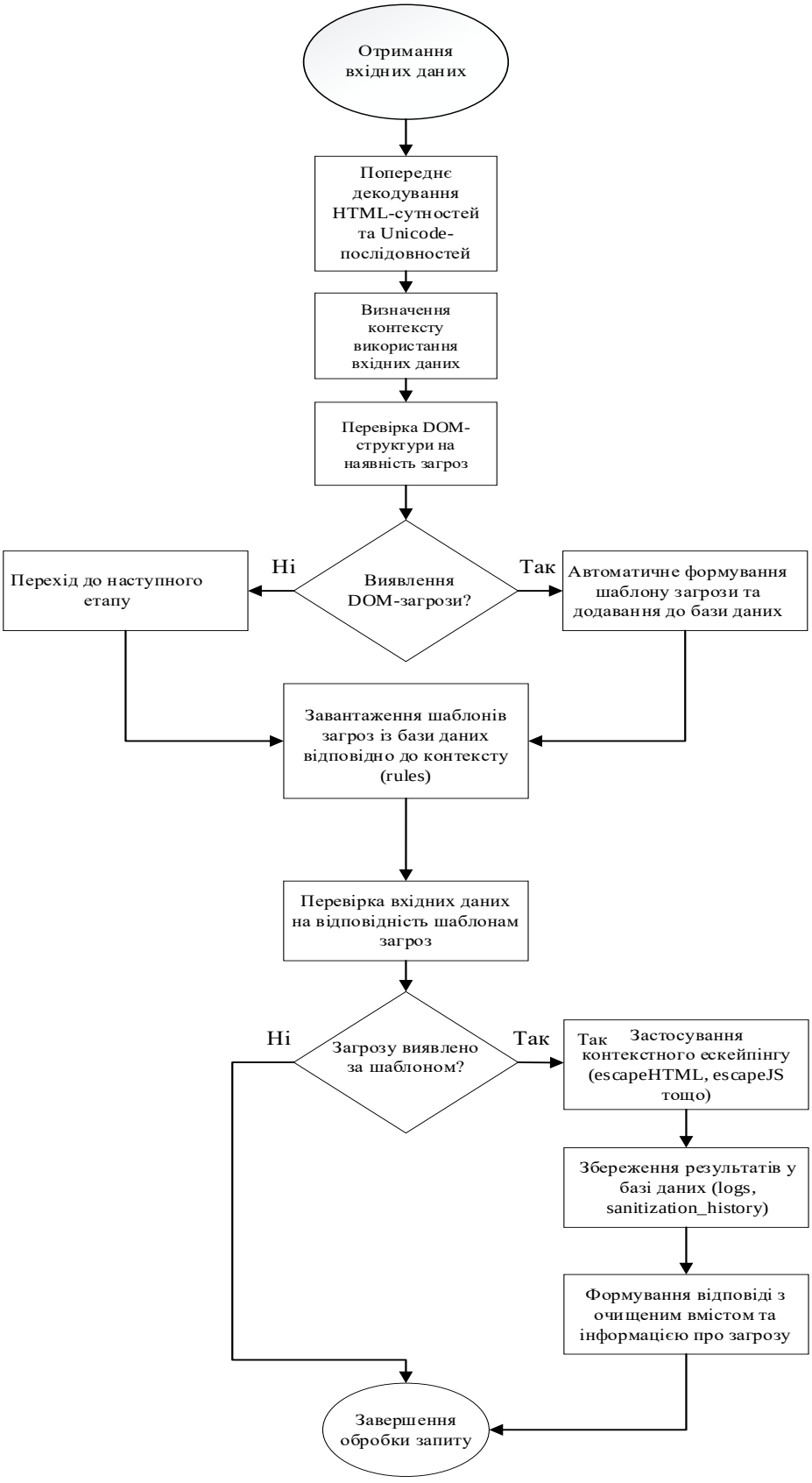


Рисунок 2.2 – Блок-схема алгоритму динамічного захисту від XSS-атак

При виявленні нової підозрілої конструкції модуль автоматично генерує новий шаблон. Після перевірки на унікальність він додається до бази шаблонів разом із контекстом, рівнем загрози та типом джерела:

```
const autoPattern = extractPattern(input);
if (autoPattern && !(await patternExists(autoPattern))) {
  await savePattern(autoPattern, context, 'high', 'dom');
}
```

Цей механізм забезпечує динамічне поповнення бази даних і перетворює систему захисту на самонавчальний модуль.

Завершальним етапом є збереження результатів перевірки у таблицях logs і sanitization_history. У цих таблицях фіксуються як початкові й очищені дані, так і знайдені шаблони, застосовані методи фільтрації та ефективність кожного методу. Це дає змогу проводити ретроспективний аналіз роботи модуля, виявляти вразливі місця та покращувати існуючі правила.

Таким чином, вдосконалений алгоритм дозволяє реалізувати не лише базовий захист від XSS-атак, а й побудувати динамічну, адаптивну систему, здатну самостійно оновлювати власну базу знань та зберігати стабільність при збільшенні обсягу оброблюваних даних.

2.5 Висновки до розділу 2

Отже, у межах другого розділу було здійснено комплексне обґрунтування концепції вдосконалення захисту веб-додатків від атак типу XSS шляхом розробки та впровадження динамічного, контекстно-орієнтованого механізму фільтрації. У розділі послідовно висвітлено архітектурні, технологічні та логічні аспекти системи, а також наведено структуру алгоритму, що забезпечує адаптивну обробку вхідних даних та виявлення потенційних загроз у режимі реального часу.

Обґрунтовано вибір платформи Node.js як основи серверної частини модуля завдяки її високій продуктивності, асинхронності та підтримці великої кількості сторонніх бібліотек, що спростили інтеграцію з базою даних і реалізацію механізмів фільтрації. Обрані технології дозволили створити систему, яка здатна не лише виконувати статичну перевірку, а й адаптуватися до нових загроз у процесі роботи.

Особливу увагу приділено структурі бази даних, що слугує центральним компонентом логіки захисту. У розробленій реляційній моделі передбачено таблиці для зберігання шаблонів загроз (threat_patterns), правил фільтрації (rules), логів користувацьких запитів (logs) і результатів обробки (sanitization_history). Така структура дозволяє здійснювати повний цикл перевірки: від фіксації введення до оцінки ефективності застосованих методів захисту.

У межах підрозділу, присвяченого вибору технологій, було продемонстровано доцільність використання бібліотеки jsdom для побудови віртуального DOM та виявлення залишкових загроз, які можуть уникнути звичайної регулярної перевірки. Також обґрунтовано необхідність контекстної фільтрації, що реалізується за допомогою функції detectContext() та ескейпінгу відповідно до типу вхідних даних.

Одним із ключових результатів цього розділу є опис алгоритму, який реалізує багаторівневу перевірку вхідних даних із можливістю адаптивного оновлення бази шаблонів. Завдяки використанню регулярних виразів, контекстної логіки, повторного DOM-аналізу та функції автоматичного збереження нових шаблонів, система здатна не лише виявляти типові загрози, а й ефективно реагувати на нові або обфусковані варіанти атак.

Крім того, розроблений підхід забезпечує розширення захисного функціоналу без втрати цілісності та продуктивності системи. Це досягається завдяки гнучкій структурі бази даних, що дозволяє легко додавати нові шаблони та оновлювати правила фільтрації без переривання роботи модуля.

Таким чином, результати, представлені в розділі 2, підтверджують ефективність обраного підходу до побудови динамічного модуля захисту. Запропонована система характеризується масштабованістю, гнучкістю, самонавчальними властивостями та високим ступенем адаптації до сучасних загроз інформаційної безпеки.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО МОДУЛЯ ЗАХИСТУ

У цьому розділі наведено реалізацію програмного модуля захисту веб-додатків від XSS-атак із використанням адаптивної фільтрації та контекстного ескейпінгу. Описано архітектуру системи, технології, що використовувались під час розробки, структуру основних функцій, а також проведено тестування модуля з метою перевірки його ефективності та відповідності поставленим завданням.

3.1 Архітектура програмного модуля та вибір технологій

Розробка програмного модуля захисту веб-додатків від XSS-атак базується на принципах модульності, масштабованості, адаптивності та орієнтації на контекст вхідних даних. Архітектура модулю побудована таким чином, щоб забезпечити гнучку інтеграцію в існуючі інформаційні системи та можливість автоматичного оновлення механізмів фільтрації відповідно до нових типів загроз.

Модуль реалізовано у вигляді серверного застосунку на платформі Node.js, що забезпечує обробку HTTP-запитів, логіку фільтрації, взаємодію з базою даних і формування відповіді. Клієнтська частина в архітектурі модулем не обмежується, оскільки вся перевірка та фільтрація виконуються на стороні сервера, що гарантує незалежність від середовища браузера [23].

Архітектура системи поділена на такі логічні рівні:

Рівень взаємодії з клієнтом (API) на цьому рівні реалізовано маршрутизацію запитів від користувача. Запити надсилаються на API-інтерфейс, зокрема на маршрут /sanitize, який викликає функцію filterInput(input). Передача даних здійснюється у форматі JSON.

Рівень логіки фільтрації, ядро системи представлено окремим модулем (filter.js), який реалізує основні функції:

– декодування вхідного тексту від HTML-entity та Unicode-символів;

– визначення контексту введення (HTML, JavaScript, JSON, URL);

– перевірку на відповідність шаблонам із бази даних;

– очищення/ескейпінг залежно від контексту;

– глибоку перевірку DOM-структури через jsdom;

– автоматичне збереження нових шаблонів у БД;

– формування результату для відповіді клієнту.

Рівень взаємодії з базою даних, реалізований через зберігання даних реалізовано на базі MySQL. Для взаємодії із СУБД використовується бібліотека mysql2 (через обгортку db.js), яка забезпечує підключення до бази та виконання параметризованих SQL-запитів. У структурі бази даних реалізовано чотири основні таблиці:

– logs, зберігає вхідні та очищені дані;

– threat_patterns, містить шаблони загроз;

– rules, зберігає правила відповідно до контексту;

– sanitization_history. фіксує методи очищення та їхню ефективність.

Рівень системного аналізу (DOM-перевірка) реалізується через бібліотеку JSDOM, яка дозволяє створювати віртуальну DOM-структуру з очищених даних. Це дозволяє виявити приховані загрози на рівні елементів, атрибутів (onerror, onload, href="javascript:"), які складно виявити за допомогою лише регулярних виразів.

Рівень логування та оновлення шаблонів, якщо виявлено загрозу, яка не збігається з шаблонами в базі, система формує новий шаблон (extractPattern) та зберігає його у таблиці threat_patterns. Завдяки цьому реалізується адаптивна поведінка системи, вона навчається на нових прикладах, що дозволяє підтримувати актуальність захисту [24].

Фрагмент реалізації основної функції фільтрації вхідних даних:

```
async function filterInput(input) {
  input = decodeObfuscated(input);
  let cleaned = input;
  let threat = 'Safe';
  let patternMatched = null;

  const context = detectContext(input);
  const rules = await getRules();

  for (const rule of rules) {
    if (rule.context.toLowerCase() !== context.toLowerCase()) continue;
    let regex = new RegExp(rule.pattern, 'gi');
    if (regex.test(cleaned)) {
```

```
    threat = 'XSS detected';
    cleaned = applyEscaping(cleaned, context);
    patternMatched = rule.pattern;
   }
  }

  const dom = new JSDOM(`<body>${cleaned}</body>`);
  const scripts = dom.window.document.querySelectorAll('script, [onerror], [onload], [onclick]');
  if (scripts.length > 0) {
   threat = 'DOM-based';
   patternMatched = '<DOM>';
  }

  return { cleaned, threat, patternMatched };
}
```

У наведеному фрагменті представлено функцію filterInput(input), яка є центральним елементом модуля захисту. Вона реалізує поетапну перевірку введеного тексту. Спочатку відбувається декодування обфускованих символів (наприклад, <script> → <script>). Функція detectContext() визначає контекст використання даних, що дозволяє далі застосувати релевантні правила фільтрації з бази (rules). Для кожного правила перевіряється, чи відповідає вхідний рядок вказаному регулярному виразу. Якщо виявлено збіг, запускається контекстно-залежна функція ескейпінгу (applyEscaping), яка нейтралізує небезпечну конструкцію, після цього виконується DOM-перевірка за допомогою jsdom, яка дозволяє виявити DOM-based загрози, не помітні на попередньому етапі. У підсумку функція повертає об'єкт з очищеним текстом, статусом (Safe або XSS detected) та знайденим шаблоном [24].

Для реалізації системи було обрано такі інструменти:

Node.js, середовище виконання JavaScript на сервері. Завдяки асинхронній обробці подій забезпечує високу продуктивність та здатність обробляти велику кількість одночасних запитів. Підтримка модульності дозволяє виділяти окремі функції захисту у самостійні блоки. Node.js є основою усієї серверної частини модуля. Запуск сервера здійснюється через server.js, де налаштовано маршрути обробки запитів (наприклад, /sanitize). Асинхронність дозволяє обробляти декілька запитів без блокування, що критично важливо для перевірки великої кількості

вхідних даних. Усі функції, пов’язані з обробкою, винесено в окремі модулі (filter.js, patterns.js, rules.js), що забезпечує модульність та зручність розширення.

JSDOM, бібліотека для емуляції DOM у середовищі Node.js. Вона дає змогу створити віртуальну веб-сторінку й виконати її структурний аналіз без реального браузера. Застосовується для виявлення DOM-based атак. Використовується в модулі filter.js після первинного очищення вхідного тексту. Вона дозволяє симулювати браузерне середовище на сервері. Це дає змогу виявити DOM-загрози, які не виявляються через регулярні вирази, наприклад:

```
const dom = new JSDOM(`<body>${cleaned}</body>`);
const scripts = dom.window.document.querySelectorAll('[onload], [onerror], script');
```

Якщо знайдено DOM-загрозу, система автоматично виділяє шаблон і додає його до бази (threat_patterns).

MySQL реляційна СУБД, обрана через свою швидкість, надійність, підтримку складних запитів та широке поширення. Завдяки індексації та чіткій структурі таблиць забезпечується швидкий доступ до шаблонів та логів. У системі створено базу xss_db з таблицями logs, rules, threat_patterns, sanitization_history. Підключення до бази реалізовано через модуль db.js, який ініціалізується через змінні середовища (.env). Всі SQL-запити параметризовані, що забезпечує захист від SQL-ін’єкцій. Зокрема, для збереження шаблону використовується:

```
await db.query("INSERT INTO Threat_Patterns (pattern, context) VALUES (?, ?)", [pattern, context]);
```

Регулярні вирази використовуються для перевірки вхідних даних на наявність відомих XSS-шаблонів. Гнучкість шаблонів дозволяє описати як прості (<script>, onerror=), так і складні обфусковані атаки (eval(...), document.write(...)). У модулі filter.js використовується набір регулярних виразів для пошуку типових XSS-елементів. Вони дозволяють швидко відсікати відомі шаблони до більш глибокого аналізу:

```
/<script.*?>.*?<\s*\/script>/gi
/on\w+\s*=\s*["']?.*?["']?/gi
```

Регулярні вирази також зберігаються в таблиці rules, що дає змогу розширювати набір шаблонів без зміни коду.

JSON (формат даних) усі вхідні та вихідні запити та відповіді передаються у форматі JSON, що забезпечує просту інтеграцію з будь-яким веб-додатком. Уся

взаємодія між клієнтом і сервером реалізована у форматі JSON. На вхід передається об'єкт типу { "input": "<html>..." }, а у відповідь повертається:

```
{
  "cleaned": "...",
  "threat": "XSS detected",
  "patternMatched": "<script>"
}
```

Це дозволяє інтегрувати модуль у будь-яку систему (веб-сайт, мобільний застосунок тощо), незалежно від мов програмування на клієнтському боці.

Такий підхід до використання інструментів забезпечує не лише гнучкість реалізації, а й можливість адаптації системи до нових типів загроз та змін у середовищі застосування. Всі компоненти легко оновлюються, взаємодіють через чіткі інтерфейси, а логіка фільтрації може масштабуватись як вглиб (точність), так і вшир (типи вхідних даних).

3.2 Реалізація фільтрації, ескейпінгу та оновлення правил

У розробленому модулі захисту веб-додатків від XSS-атак реалізовано комплексний механізм фільтрації введених даних, який поєднує евристичну перевірку регулярними виразами, контекстно-залежний ескейпінг, аналіз DOM-структури та динамічне оновлення бази шаблонів. Основна логіка обробки зосереджена у модулі filter.js, де реалізовано функцію filterInput(input).

Першим етапом є попередня обробка введеного рядка: декодування обфускованих символів, таких як HTML-сутності (<script>) та Unicode-послідовності. Для цього використовується допоміжна функція decodeObfuscated(), яка переводить текст у стандартний вигляд перед аналізом.

Після цього визначається контекст використання даних за допомогою функції detectContext(). Залежно від типу (HTML, JavaScript, JSON, URL), система підбирає відповідні шаблони із бази даних (rules) та проводить перевірку за допомогою регулярних виразів. Якщо введений рядок відповідає певному шаблону, застосовується очищення (ескейпінг) з урахуванням контексту [25].

Фрагмент перевірки по шаблонах:

```
for (const rule of rules) {
  if (rule.context.toLowerCase() !== context.toLowerCase()) continue;
```

```
  let regex = new RegExp(rule.pattern, 'gi');
  if (regex.test(cleaned)) {
    threat = 'XSS detected';
    cleaned = applyEscaping(cleaned, context);
    patternMatched = rule.pattern;
  }
}
```

У модулі реалізовано окремі функції для очищення даних відповідно до контексту:

– escapeHTML(), замінює символи <, >, ", ' на відповідні HTML-сутності;

– escapeJS(), екранує лапки, слеші та переводи рядка;

– escapeJSON(), захищає JSON-рядки від помилок парсингу;

– escapeURL(), кодує URL-параметри для безпечної вставки в адресні рядки.

Це дозволяє зберегти валідність даних після очищення та уникнути порушення структури веб-сторінки чи скрипта.

Фрагмент ескейпінгу HTML:

```
function escapeHTML(text) {
  return text
    .replace(/&/g, '&amp;')
    .replace(/</g, '&lt;')
    .replace(/>/g, '&gt;')
    .replace(/"/g, '&quot;')
    .replace(/'/g, '&#039;');
}
```

Після ескейпінгу очищені дані перевіряються за допомогою бібліотеки jsdom. Вона дозволяє створити віртуальну DOM-структуру з очищеного HTML-коду і перевірити її на наявність потенційно небезпечних елементів: <script>, атрибути onload, onerror, а також href="javascript:...".

Якщо виявлено DOM-загрозу, система виконує автоматичне вилучення підозрілого фрагмента (через extractPattern()) і перевіряє, чи вже міститься він у базі шаблонів (threat_patterns). Якщо фрагмент новий — він зберігається разом із контекстом.

Фрагмент оновлення шаблону DOM-загрози:

```
if (scripts.length > 0 || handlers.length > 0 || javascriptLinks.length > 0) {
  const autoPattern = extractPattern(input);
  if (autoPattern && !(await patternExists(autoPattern))) {
    await savePattern(autoPattern, context, 'high', 'dom');
    patternMatched = autoPattern;
```

```
  }
}
```

Окрім DOM-аналізу, модуль підтримує динамічне доповнення бази шаблонів і правил. Нові шаблони, отримані під час обробки користувацьких запитів, зберігаються у таблиці threat_patterns, після чого можуть бути використані для генерації нових правил у таблиці rules.

Механізм оновлення правил може також запускатись окремо як запланована подія (наприклад, через cron), яка синхронізує шаблони із правилами фільтрації. Це дозволяє системі постійно оновлювати свій захисний арсенал без втручання адміністратора.

У межах реалізації передбачено обробку запитів через REST API, що дозволяє інтегрувати модуль до будь-якого веб-додатка. Нижче наведено приклад вхідного запиту до модуля та відповіді, яку формує система:

```
POST /sanitize
{
  "input": "<img src='x' onerror='alert(1)'>"
}
```

Відповідь, сформована модулем після обробки:

```
{
  "cleaned": "<img src='x'>",
  "threat": "XSS detected",
  "patternMatched": "onerror="
}
```

Як видно з відповіді, система виявила загрозу типу DOM-based, виконала очищення елемента та повідомила шаблон, що спрацював. Завдяки уніфікованому формату JSON результат можна без труднощів обробити на клієнтському боці.

У таблиці threat_patterns зберігаються виявлені шаблони XSS-атак, що надалі використовуються для створення правил фільтрації. Формування записів у цій таблиці може здійснюватися як вручну, так і автоматично при виявленні нових загроз у процесі DOM-аналізу (рис.3.1).

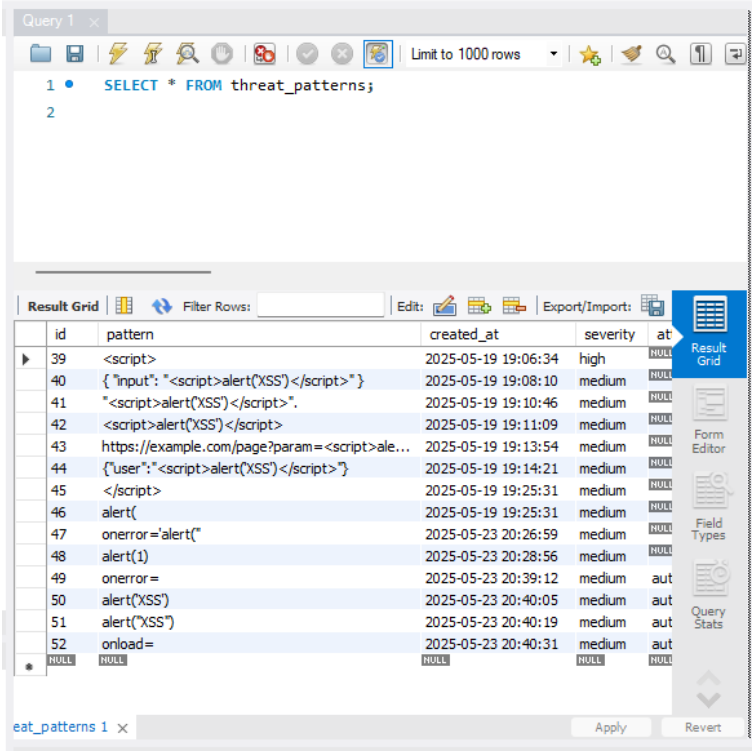


Рисунок 3.1 – Приклад записів таблиці threat_patterns

Окрім збереження шаблонів загроз у таблиці threat_patterns, модуль також використовує таблицю rules, де зберігаються активні правила фільтрації, які застосовуються залежно від контексту. Це дає змогу ефективно розділяти загрози за типами середовища (HTML, JS тощо) та оновлювати набір правил без перезапуску системи (рис.3.2).

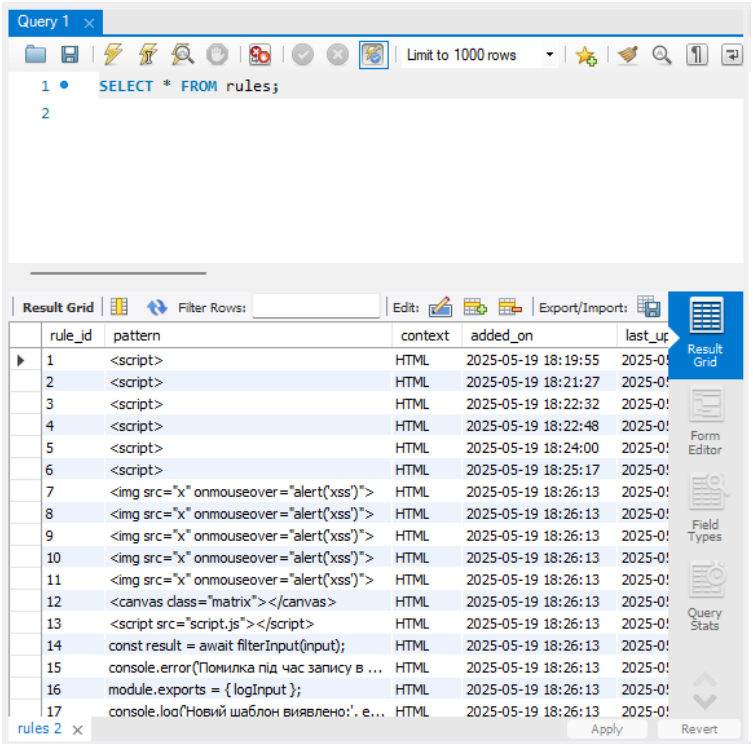


Рисунок 3.2 – Приклад записів таблиці rules

Після обробки запиту результат записується в таблицю logs. Це дозволяє відстежувати історію загроз, перевіряти ефективність шаблонів і здійснювати аудит обробки даних (рис.3.3).

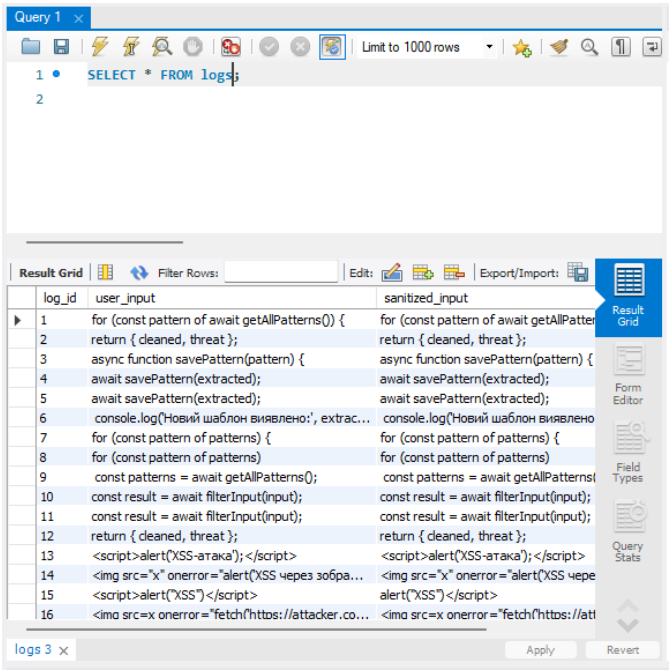


Рисунок 3.3 – Приклад записів таблиці logs

Завдяки логуванню система зберігає як вихідні, так і очищені дані, а також шаблон, який був задіяний під час фільтрації.

Оновлення шаблонів у системі виконується автоматично, коли jsdom виявляє нову DOM-загрозу, і ця конструкція ще не була присутня у базі. Такий шаблон автоматично зберігається у threat_patterns, після чого може бути перенесений у таблицю rules.

Для автоматизації оновлення rules можна використовувати регулярну подію:

```
CREATE EVENT update_rules
ON SCHEDULE EVERY 1 HOUR
DO
  INSERT INTO rules (pattern, context, added_on, last_updated)
  SELECT pattern, context, NOW(), NOW()
  FROM threat_patterns
  WHERE pattern NOT IN (SELECT pattern FROM rules);
```

Такий підхід дозволяє системі самостійно оновлювати набір активних правил, підвищуючи адаптивність і актуальність захисту без залучення адміністратора.

Щоб показати функції ескейпінгу, реалізовані у модулі, які застосовуються залежно від визначеного контексту введення. Такий підхід дозволяє забезпечити цільову нейтралізацію загроз із мінімальною втратою корисних даних (табл.3.2).

Таблиця 3.2 – Реалізовані методи ескейпінгу у програмному модулі.

№	Назва функції	Контекст застосування	Призначення
1	escapeHTML()	HTML	Замінює символи <, >, ', " на відповідні HTML-сутності (<, > тощо), щоб запобігти вставці скриптів у DOM.
2	escapeJS()	JavaScript	Екранує лапки, зворотні слеші, символи переносу рядка, запобігаючи впровадженню коду в JavaScript-блоки.
3	escapeURL()	URL	Кодує спецсимволи у форматі %xx, забезпечуючи безпечну передачу даних у параметрах посилань.
4	escapeJSON()	JSON	Екранує лапки, слеші, переводи рядків, забезпечуючи збереження коректної структури JSON-даних.

Таким чином, реалізований механізм фільтрації охоплює як статичний аналіз на основі шаблонів, так і динамічний DOM-аналіз із автоматичним оновленням бази загроз. Такий підхід дозволяє ефективно нейтралізувати як відомі, так і нові типи XSS-атак, підтримуючи адаптивність і самонавчання системи.

3.3 Результати тестування та аналіз ефективності

Для перевірки працездатності та ефективності реалізованого програмного модуля захисту веб-додатків від XSS-атак було проведено тестування з використанням типових, модифікованих та обфускованих шаблонів атак. Тестування проводилося в умовах, наближених до реальних, із використанням HTTP-запитів через Postman та логуванням результатів у базі даних [26].

Метою тестування було:

– перевірити здатність модуля виявляти як прямі, так і приховані XSS-загрози;

– оцінити якість роботи контекстного ескейпінгу;

– перевірити реакцію системи на обфусковані атаки;

– зафіксувати автоматичне збереження нових шаблонів;

– оцінити стабільність та час обробки запитів.

Було використано ряд XSS-навантажень для перевірки різних механізмів виявлення:

<script>alert(1)</script> – класичний випадок XSS (рис.3.4).

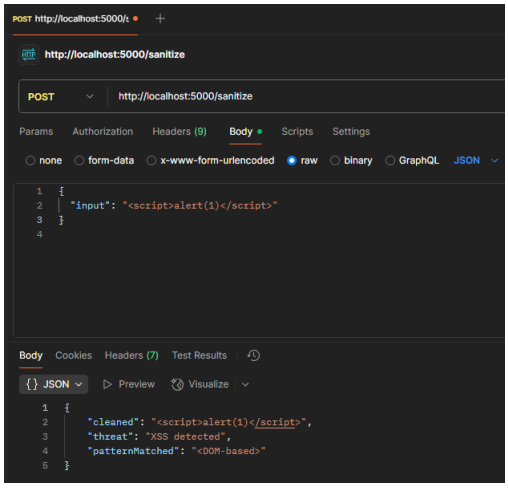


Рисунок 3.4 – Тестування модуля захисту через Postman запит №1

<img src=x onerror=alert(1)> – DOM-based атака (рис.3.5).

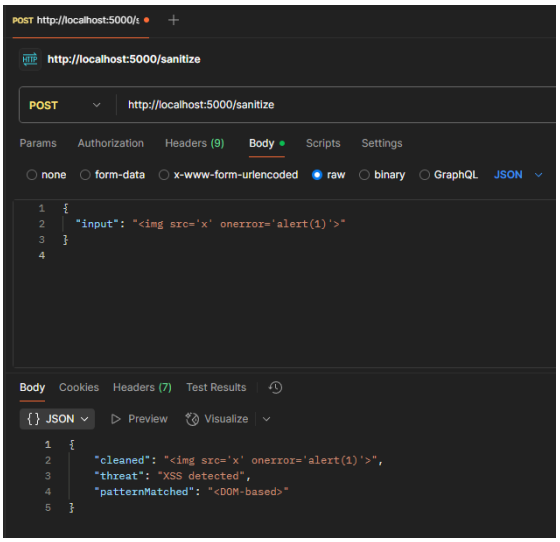


Рисунок 3.5 – Тестування модуля захисту через Postman запит №2

<script>alert(1)</script> – Обфускація, декодування (рис.3.6).

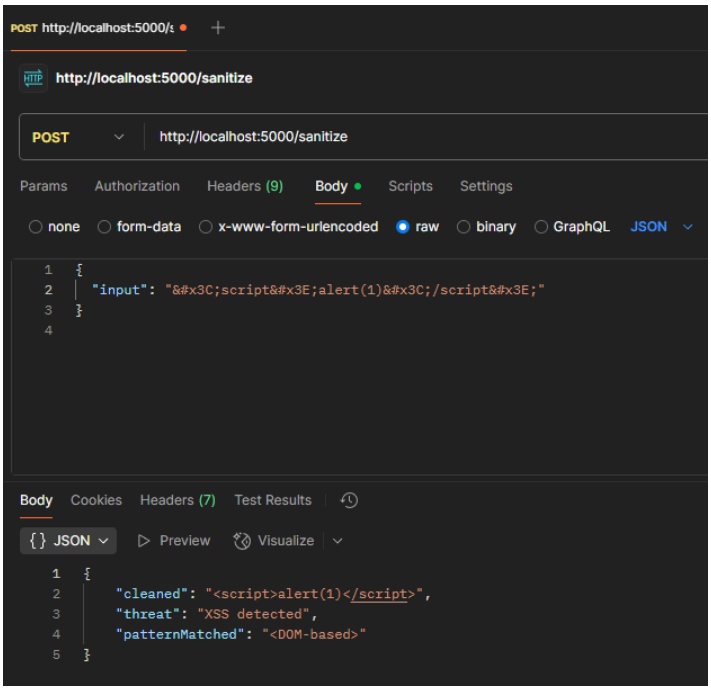


Рисунок 3.6 – Тестування модуля захисту через Postman запит №3 (рис.3.7).

eval(alert('XSS')) – JS-контекст, regex

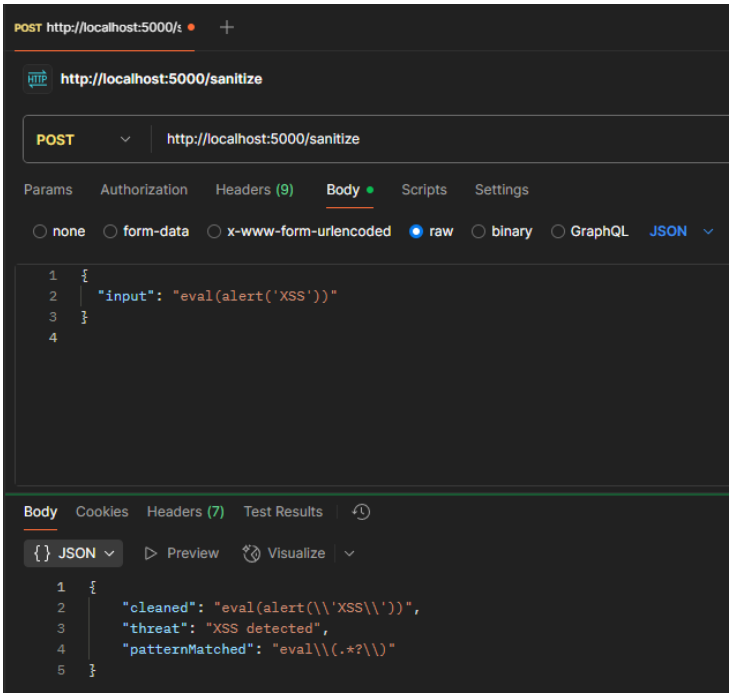


Рисунок 3.6 – Тестування модуля захисту через Postman запит №4

У середньому час обробки запиту склав 42–85 мс залежно від складності введення та кількості правил у базі.

Найбільше часу витрачається на створення DOM-структури через jsdom та перевірку регулярними виразами у великій базі шаблонів. Цей показник є прийнятним для серверної фільтрації в режимі реального часу.

Однією з ключових особливостей реалізованого модуля є підтримка контекстно-залежної фільтрації, яка значно підвищує точність виявлення XSS-загроз і знижує ризик хибнопозитивних результатів.

У класичних підходах фільтрації зазвичай використовується єдиний набір шаблонів незалежно від того, де саме будуть використані вхідні дані (наприклад, у HTML, у JavaScript-блоці або у JSON-рядку). Такий підхід має низьку гнучкість і може призводити до того, що невинні символи екранюються без потреби (порушення структури даних), реальні загрози, замасковані під інший контекст, залишаються не виявленими.

У запропонованому модулі реалізовано механізм визначення контексту вхідних даних за допомогою функції detectContext(input). Система аналізує характерні ознаки у тексті та відносить його до одного з наступних типів:

– html, якщо вхідний текст містить HTML-теги або атрибути;

– js, якщо використано JavaScript-конструкції: eval(...), setTimeout(...), document.write(...);

– json, якщо текст має структуру JSON-об’єкта або масиву;

– url, якщо вхідні дані мають ознаки адресних рядків або протоколів (наприклад, javascript:).

Переваги контекстної фільтрації полягаюсь у точному визначені вразливостей, тобто система не просто виявляє небезпечну конструкцію, а оцінює її залежно від того, де вона буде використана. Ще однією перевагою є збереження структури даних, наприклад, JSON-рядки залишаються валідними після очищення, що критично важливо для API-запитів. Також уникнення надмірного очищення, система не видаляє зайві символи, які не становлять загрози в даному контексті. А ще гнучке оновлення правил, шаблони фільтрації (rules) прив’язані до контекстів, тому розширення набору не впливає на інші сфери використання.

Контекстна фільтрація дозволяє реалізувати більш гнучку, стабільну і розумну систему виявлення загроз, яка адаптується до типу вхідних даних і дозволяє точно визначити метод захисту. Такий підхід забезпечує високу

ефективність фільтрації без шкоди для функціональності або структури переданих даних.

Для оцінки ефективності реалізованого програмного модуля доцільно порівняти його з одним із найпоширеніших рішень у сфері клієнтської фільтрації бібліотекою DOMPurify. Бібліотека є відкритим проєктом, який забезпечує очищення HTML-контенту шляхом видалення небезпечних тегів та атрибутів. Бібліотека орієнтована переважно на клієнтське використання (у браузері), хоча її можливо інтегрувати і на сервер через jsdom (табл.3.3).

Таблиця 3.3 – Порівняння DOMPurify та реалізованого модуля

Критерій	DOMPurify	Реалізований модуль
Рівень роботи	Клієнтський (основний)	Серверний
Типи контекстів	HTML	HTML, JS, JSON, URL
Адаптивне оновлення шаблонів	Відсутнє	Присутнє (через DOM-аналіз)
Збереження шаблонів	Ні	Так (у базі threat_patterns)
Інтеграція з базами даних	Відсутня	Пряма інтеграція з MySQL
Розширення через правила	Часткове (конфігурація)	Повне (таблиця rules)
Контекстно-залежний ескейпінг	Ні	Так (escapeHTML(), escapeJS() тощо)
Самонавчання	Відсутнє	Присутнє

Як видно з таблиці, DOMPurify є ефективним засобом очищення HTML-контенту на клієнтському боці, однак його можливості обмежуються статичною обробкою HTML. Реалізований у цій роботі модуль, на відміну від DOMPurify, функціонує на серверному рівні, підтримує багатоконтекстну обробку, а також здатен адаптуватися до нових загроз за рахунок автоматичного збереження шаблонів і оновлення правил.

Крім того, використання MySQL у якості сховища шаблонів та логів дозволяє здійснювати аналітику, аудит та формування статистики, що є особливо актуальним для корпоративних рішень і складних систем з підвищеними вимогами до безпеки.

3.4 Інструкція користувача та вимоги до середовища

Розроблений програмний модуль захисту від XSS-атак призначений для інтеграції у серверну частину веб-додатків з метою перевірки, фільтрації та нейтралізації потенційно небезпечних вхідних даних. Модуль реалізовано на платформі Node.js та побудовано з урахуванням принципів модульності, масштабованості, кросплатформності та зручності налаштування.

Користувачем модуля в контексті даної системи вважається розробник або адміністратор веб-додатку, який відповідає за інтеграцію захисного механізму у вже існуючу архітектуру веб-додатку або систему управління контентом (CMS).

Інструкція з розгортання та запуску.

1. Завантажити репозиторій з програмним модулем у локальний каталог (через git або архів ZIP).

2. Встановити актуальну версію Node.js (рекомендовано ≥ v18.0.0). Завантажити можна з офіційного сайту https://nodejs.org.

3. Переконатися, що встановлено систему керування базами даних MySQL (≥ v5.7) і створено базу даних з назвою xss_db.

Установлення залежностей, у терміналі, в корені проєкту, виконати:

```
npm install
```

Ця команда автоматично завантажить усі залежності, зокрема: express, jsdom, mysql2, dotenv, body-parser.

Налаштування змінних середовища.

Створити у кореневій директорії проєкту файл .env з наступним вмістом:

```
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=your_password
DB_NAME=xss_db
PORT=5000
```

Цей файл дозволяє безпосередньо налаштовувати параметри підключення до бази даних, не змінюючи код.

Після налаштування всіх параметрів виконується запуск.

```
node demo/server.js
```

У консолі з’явиться повідомлення:

```
Demo server running on port 5000
```

Це означає, що модуль готовий до прийому запитів.

Перевірка функціональності через Postman.

Відкрити програму Postman, та створити новий POST-запит на адресу http://localhost:5000/sanitize.

У вкладці Headers додати Content-Type: application/json.

У вкладці Body, raw, JSON ввести:

```
{
  "input": "<img src='x' onerror='alert(1)'>"
}
```

Відправити та прогнозуємий результат буде:

```
{
  "cleaned": "<img src='x'>",
  "threat": "XSS detected",
  "patternMatched": "onerror="
}
```

Модуль самостійно виявить загрозу, визначить контекст і застосує відповідний метод очищення.

Програмні компоненти включають Node.js середовище виконання, npm менеджер пакетів, MySQL Server СУБД для збереження шаблонів, логів та правил, Postman / CURL для перевірки API, Visual Studio Code / будь-який редактор коду для модифікацій.

Поради з інтеграції у реальний веб-додаток:

– модуль легко розширюється завдяки поділу на окремі файли: filter.js, patterns.js, rules.js, db.js;

– можна викликати API /sanitize перед збереженням коментаря, форми, або будь-якого текстового введення користувача;

– підтримується робота з будь-яким фронтендом (React, Angular, Vue тощо), який вміє надсилати JSON-запити;

– при масштабуванні систему можна розгорнути на окремому мікросервісі.

У підсумку, реалізований модуль є простим в налаштуванні, зручним у використанні, гнучким у розгортанні та ефективним у захисті від XSS-атак, що дозволяє застосовувати його як у локальних середовищах, так і у хмарних або продакшн-системах.

3.5 Висновки до розділу 3

У третьому розділі дипломної роботи було здійснено повноцінну реалізацію програмного модуля для захисту веб-додатків від XSS-атак з використанням динамічної фільтрації, контекстного ескейпінгу та автоматичного оновлення правил. Реалізація виконана на базі середовища Node.js із використанням актуальних бібліотек, таких як express, jsdom, mysql2, що забезпечило ефективну роботу та зручність інтеграції.

Модуль реалізовано як незалежний серверний застосунок, який приймає вхідні дані через HTTP-запити, визначає контекст використання (HTML, JavaScript, JSON, URL), виконує пошук шаблонів загроз у базі, здійснює відповідне очищення та, за необхідності, формує нові шаблони в процесі обробки. Усі результати логуються та зберігаються у базі даних для подальшого аналізу.

Крім основного функціоналу, система підтримує контекстно-залежну обробку даних, що дозволяє зберігати їхню структуру після очищення і зменшує кількість помилкових спрацьовувань. Завдяки цьому модуль не лише ефективно захищає веб-додаток, але й легко масштабується для використання у більших проєктах, включаючи корпоративні або хмарні рішення.

Таким чином, реалізований модуль повністю відповідає поставленим функціональним вимогам та забезпечує надійний захист від міжсайтових скриптових атак за рахунок гнучкої архітектури, динамічного аналізу та підтримки самонавчання на основі реального вхідного трафіку.

ВИСНОВКИ

У процесі виконання бакалаврської кваліфікаційної роботи було розроблено, реалізовано та протестовано програмний модуль для захисту веб-додатків від атак типу XSS (міжсайтове скриптування) з використанням адаптивної фільтрації та контекстного ескейпінгу вхідних даних на серверній стороні. Актуальність обраної теми підтверджується зростаючими обсягами динамічного контенту у веб-системах та високою поширеністю XSS як одного з найнебезпечніших типів атак згідно з OWASP.

У теоретичній частині роботи було проведено класифікацію XSS-атак, розглянуто існуючі методи та засоби їх виявлення і блокування, а також виконано порівняльний аналіз наявних рішень. На основі цього аналізу сформульовано вимоги до власного модуля, який має підтримувати автоматичне оновлення шаблонів, контекстну обробку вхідних даних, адаптивну фільтрацію та логування результатів.

У другому розділі обґрунтовано вибір бази даних, побудовано реляційну модель, що включає таблиці logs, rules, threat_patterns, sanitization_history, а також розроблено покроковий алгоритм роботи системи. Особливу увагу було приділено питанням масштабованості, гнучкості та швидкодії, з урахуванням майбутнього розширення функціоналу.

У практичній частині реалізовано серверний застосунок на платформі Node.js. Програмний модуль включає систему фільтрації (filter.js), яка поєднує евристичний аналіз, регулярні вирази, контекстне очищення (escapeHTML, escapeJS, escapeJSON, escapeURL) та глибоку перевірку через бібліотеку jsdom. У разі виявлення нових загроз, які не відповідають жодному з шаблонів у базі, система автоматично формує нові шаблони та додає їх у таблицю threat_patterns. Це забезпечує адаптивну поведінку модуля і поступове самонавчання.

Тестування модуля підтвердило його ефективність у виявленні як стандартних, так і обфускованих XSS-атак. Усі тести було проведено за допомогою REST API-запитів через середовище Postman. Було зафіксовано коректне

визначення контексту, застосування релевантних правил, додавання нових шаблонів та збереження логів у базі даних.

У порівнянні з класичними підходами, такими як DOMPurify, реалізований модуль забезпечує ширшу функціональність: роботу з різними контекстами, збереження історії очищення, можливість розширення шаблонів і адаптивну фільтрацію. Це робить запропоноване рішення більш придатним для інтеграції у сучасні серверні системи, зокрема у веб-додатки з високими вимогами до безпеки.

У результаті виконання кваліфікаційної роботи було досягнуто поставлену мету, реалізовано всі задачі дослідження та отримано практичний результат, придатний до використання у реальних інформаційних системах. Отримані результати мають як теоретичне, так і прикладне значення, а створена система може бути основою для подальших розробок у сфері веб-безпеки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What is Cross Site Scripting (XSS). Geeksforgeeks. URL: https://www.geeksforgeeks.org/what-is-cross-site-scripting-xss/ (дата звернення: 16.11.2024).

2. Cross-site scripting (XSS). MDN web docs. URL: https://developer.mozilla.org/en-US/docs/Glossary/Cross-site_scripting#see_also (дата звернення: 16.11.2024).

3. Cross-site Scripting (XSS). Acunetix. URL: https://www.acunetix.com/websitesecurity/cross-site-scripting/#:~:text=Cross-site%20Scripting%20(XSS)%20is%20a%20client-side,web%20page%20or%20web%20application. (дата звернення: 16.11.2024).

4. Cross-site scripting. PortSwigger. URL: https://portswigger.net/web-security/cross-site-scripting

5. Manuel L. Richard K. Raghu K. Dimitris S. Combinatorially XSSing Web Application Firewalls. NIST. URL: https://csrc.nist.gov/pubs/conference/2021/05/28/combinatorially-xssing-web-application-firewalls/final (дата звернення: 16.11.2024).

6. Secure by Design Alert: Eliminating Cross-Site Scripting Vulnerabilities. CISA. URL: https://www.cisa.gov/resources-tools/resources/secure-design-alert-eliminating-cross-site-scripting-vulnerabilities (дата звернення: 16.11.2024).

7. Kirsten S. Cross Site Scripting. OWASP. URL: https://owasp.org/www-community/attacks/xss/. (дата звернення: 16.11.2024).

8. Cuthbert D. Manico J. Lang E. OWASP Application Security Verification Standard (ASVS). OWASP. URL: https://owasp.org/www-project-application-security-verification-standard/ (дата звернення: 17.11.2024).

9. Yassine M. A systematic review and taxonomy of web applications threats. ResearchGate. URL: https://www.researchgate.net/figure/The-process-followed-by-an-attacker-to-produce-an-impact_fig14_348065719 (дата звернення: 17.11.2024).

10. Content Security Policy (CSP). Mdn web docs. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP (дата звернення: 17.11.2024).

11. Learn Web Security with Google. Chrome for Developers. URL: https://www.youtube.com/watch?v=tgEIo7ZSkbQ (дата звернення: 18.11.2024).

12. Achieve Robust Information Security with ISO 27001:2022. Isms.online. URL: https://www.isms.online/iso-27001/ (дата звернення: 18.11.2024).

13. NIST CSF - стандарт, про який варто знати. Its.specialist. URL: https://my-itspecialist.com/what-is-the-nist-csf-standartrt (дата звернення: 18.11.2024).

14. Hardt D. The OAuth 2.0 Authorization Framework. Microsoft. URL: https://datatracker.ietf.org/doc/html/rfc6749 (дата звернення: 18.11.2024).

15. Introduction to JSON Web Tokens. JWT. URL: https://jwt.io/introduction (дата звернення: 18.11.2024).

16. McMullin M. WAF – Web Application Firewall 101. Kemptechnologies. URL: https://kemptechnologies.com/blog/waf-web-application-firewall-101 (дата звернення: 19.11.2024).

17. JSON Web Token (JWT): An Overview. sdrc.co.in. URL: https://sdrc.co.in/?p=3213 (дата звернення: 19.11.2024).

18. Retrofitting spatial safety to hundreds of millions of lines of C++. Google. URL: https://security.googleblog.com (дата звернення: 19.11.2024).

19. Hacking APIs: Breaking Web Application Programming Interfaces. . Stuttard, 2022. (дата звернення: 19.11.2024).

20. Web Security. W3. URL: https://www.w3.org/mission/security/ (дата звернення: 20.11.2024).

21. Що таке MySQL. FreeHost.ua. URL: https://freehost.com.ua/ukr/faq/wiki/chto-takoe-mysql/ (дата звернення: 20.11.2024).

22. База даних MySQL. Promoter URL: https://promoter.net.ua/articles/baza-danix-mysql.html (дата звернення: 20.11.2024).

23. Доліновський Р. М. Методи захисту веб-застосунку від XSS і CSRF вразливостей / Methods of protecting a web application from XSS and CSRF

vulnerabilities. Західноукраїнський національний університет. 2023. (дата звернення: 25.04.2025).

24. Cross-Site Scripting Attacks: Classification, Attack, and Countermeasures. B. Б. Гупта. Taylor & Francis. 2020. (дата звернення: 26.04.2025).

25. O'Reily. XSS Attacks:Cross Site Scripting Exploits and Defense / O'Reily // Syngress. – 2011, 439 с. (дата звернення: 27.04.2025).

26. Gupta, B. B., Chaudhary, P. Cross-Site Scripting Attacks: Classification, Attack, and Countermeasures / B. B. Gupta, P. Chaudhary // CRC Press. – 2020, 170 с. (дата звернення: 27.04.2025).

27. JSDOM. A JavaScript implementation of the DOM. URL: https://github.com/jsdom/jsdom (дата звернення: 28.04.2025).

ДОДАТКИ

Додаток А. Технічне завдання

Вінницький національний технічний університет
Факультет менеджменту та інформаційної безпеки
Кафедра менеджменту та безпеки інформаційних систем

ЗАТВЕРДЖУЮ
Голова секції "Управління інформаційною безпекою" кафедри МБІС
д.т.н., професор
_____ Юрій ЯРЕМЧУК
" 20 " березня 2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ

до бакалаврської кваліфікаційної роботи на тему:
Розробка програмного модуля для захисту веб-додатків від XSS-атак з використанням адаптивної фільтрації та ескейпінгу вхідних даних на серверній стороні з автоматичним оновленням правил безпеки.

08-72.БКР.006.00.000.ТЗ

Керівник бакалаврської кваліфікаційної роботи
д.ф. (PhD), доц.
Салієва О. В.
_____
" _____ "

Вінниця – 2025 р.

1. Найменування та область застосування

Програмний модуль захисту веб-додатків від XSS-атак.
Область застосування: захист серверних веб-систем, REST API та інтерфейсів взаємодії користувачів із веб-застосунками від атак типу Cross-Site Scripting (XSS), що виникають через незахищене оброблення введених даних.

2.Підстава для розробки

Розробка виконується на основі наказу ректора ВНТУ № 97 від 20.03.2025 р.

3. Мета та призначення розробки

3.1 Мета розробки: створення захисного серверного модуля, який здійснює динамічну фільтрацію вхідних даних, визначає контекст використання (HTML, JavaScript, JSON, URL), застосовує відповідні механізми ескейпінгу, виконує DOM-аналіз, а також автоматично формує і додає нові шаблони загроз до бази правил безпеки.

3.2 Призначення: Забезпечення гнучкого та адаптивного захисту веб-застосунків від XSS-атак без втручання у бізнес-логіку веб-додатків, шляхом попереднього очищення введених даних на серверному рівні.

4. Джерела розробки

1. OWASP. Cross-Site Scripting (XSS). URL: https://owasp.org/www-community/attacks/xss/

2. MDN Web Docs. XSS Prevention Cheat Sheet. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Attacks/XSS

3. Node.js. About Node.js. URL: https://nodejs.org/

4. Express.js. Node.js web application framework. URL: https://expressjs.com/

5. JSDOM. A JavaScript implementation of the DOM. URL: https://github.com/jsdom/jsdom

6. MySQL. Data Definition and Security. URL: https://dev.mysql.com/doc/

5. Вимоги до програми

5.1 Вимоги до функціональних характеристик:

5.1.1 Модуль повинен здійснювати прийом POST-запитів через REST API і проводити аналіз вхідних текстових даних;

5.1.2 Система повинна визначати контекст (html, js, json, url) та застосовувати відповідний метод ескейпінгу;

5.1. У випадку виявлення загрози за допомогою регулярних виразів або DOM-аналізу, модуль повинен видалити або екранувати небезпечні конструкції;

5.1.4 Модуль повинен додавати нові шаблони до таблиці threat_patterns, якщо вхідний рядок не відповідає існуючим правилам;

5.1.5 Усі перевірені запити повинні логуватися в базу даних з фіксацією вхідних та очищених даних, шаблону, типу загрози, часу.

5.1.6 Програмний модуль має бути реалізований як middleware в Express.js і працювати незалежно від прикладної логіки.

5.2 Вимоги до надійності:

5.2.1 Модуль повинен працювати стабільно при навантаженні до 1000 запитів на хвилину;

5.2.2 У разі помилок доступу до бази або недоступності шаблонів, система повинна повертати коректне повідомлення та працювати у безпечному режимі;

5.2.3 Дані журналу подій мають зберігатися в базі MySQL з підтримкою розширення структури логів.

5.3 Вимоги до складу і параметрів технічних засобів:

– процесор – не нижче Pentium 1500 МГц або аналогічний

– оперативна пам’ять – не менше 512 МБ;

– вільне місце на диску – від 200 МБ;

– середовище функціонування – операційна система з підтримкою Node.js (Windows/Linux);

– середовище виконання – Node.js v18 або вище, MySQL 5.7 або вище (локально або у хмарі);

– вимоги до техніки безпеки – згідно з чинними нормами користування комп’ютерною технікою та програмним забезпеченням.

64

6. Вимоги до програмної документації

6.1 Обов'язкова поетапна інструкція для майбутніх користувачів, наведена пункті 3.4

7. Вимоги до технічного захисту інформації

7.1 Забезпечення безпеки модулю: захист API-інтерфейсу, контроль прав доступу до шаблонів.

7.2 Неможливість зміни шаблонів або правил стороннім користувачем без авторизації.

8. Техніко-економічні показники

8.1 Цінність результатів використання даного проекту повинна перевищувати витрати на його реалізацію.

8.2 Має бути реалізований таким чином, щоб підходити для використання широкого загалу.

8.3 Потенційне застосування в реальних інформаційних системах без додаткових витрат.

9. Стадії та етапи розробки

№	Назва та зміст етапу	Термін виконання		Примітка
		початок	закінчення	
1	Визначення напрямку бакалаврської роботи, формулювання теми	20.03.2025	23.03.2025	Виконано
2	Аналіз предметної області обраної теми	24.03.2025	13.04.2025	Виконано
3	Розробка алгоритму роботи	14.04.2025	27.04.2025	Виконано
4	Написання бакалаврської роботи на основі розробленої теми	28.04.2025	25.05.2025	Виконано
5	Передзахист бакалаврської кваліфікаційної роботи	26.05.2025	27.05.2025	Виконано
6	Виправлення, уточнення, корегування бакалаврської дипломної роботи	28.05.2025	02.06.2025	Виконано
7	Захист бакалаврської кваліфікаційної роботи	09.06.2025	10.06.2025	Виконано

10. Порядок контролю та прийому

10.1 До приймання бакалаврської кваліфікаційної роботи надається:

– ПЗ до бакалаврської кваліфікаційної роботи;

- демонстрація результату бакалаврської кваліфікаційної роботи;
- презентація;
- відзив керівника роботи;
- відзив рецензента

Технічне завдання до виконання прийняв _____ Вітковський А.В.

Додаток Б. Лістинг серверної частини програми

server.js

```
const express = require('express');
const app = express();
const path = require('path');
const dotenv = require('dotenv');
const sanitizeRoute = require('./routes/sanitize');

dotenv.config({ path: path.resolve(__dirname, '../.env') });

app.use(express.json());
app.use('/sanitize', sanitizeRoute);

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
  console.log(`Demo server running on port ${PORT}`);
});
```

routes/sanitize.js

```
const express = require('express');
const router = express.Router();
const { filterInput } = require('../module/filter');

router.post('/', async (req, res) => {
  try {
    const input = req.body.input;
    const result = await filterInput(input);
    res.json(result);
  } catch (error) {
    res.status(500).json({ error: 'Internal server error' });
  }
});

module.exports = router;
```

module/filter.js

```
const { getRules } = require('./rules');
const { savePattern, patternExists } = require('./patterns');
const { JSDOM } = require('jsdom');
const { detectContext } = require('./contextFilter');

async function filterInput(input) {
  input = decodeObfuscated(input);
  let cleaned = input;
  let threat = 'Safe';
  let patternMatched = null;

  const context = detectContext(input);
  const rules = await getRules();

  for (const rule of rules) {
```

```
    if (rule.context.toLowerCase() !== context.toLowerCase()) continue;
    let regex = new RegExp(rule.pattern, 'gi');
    if (regex.test(cleaned)) {
      threat = 'XSS detected';
      cleaned = applyEscaping(cleaned, context);
      patternMatched = rule.pattern;
    }
  }

  try {
    const dom = new JSDOM(`<body>${cleaned}</body>`);
    const scripts = dom.window.document.querySelectorAll('script, [onerror], [onload], [onclick]');
    if (scripts.length > 0) {
      threat = 'XSS detected';
      patternMatched = '<DOM-based>';

      const autoPattern = extractPattern(input);
      if (autoPattern && !(await patternExists(autoPattern))) {
        await savePattern(autoPattern, context, 'medium', 'dom');
        patternMatched = autoPattern;
      }
    }
  } catch (err) {
    console.error('JSDOM parsing error:', err);
  }

  return { cleaned, threat, patternMatched };
}
module/db.js
let pool;
function init(config) {
  const mysql = require('mysql2/promise');
  pool = mysql.createPool(config);
}

async function query(sql, params) {
  if (!pool) throw new Error('DB not initialized. Call init(config) first.');
  const [rows] = await pool.execute(sql, params);
  return rows;
}

module.exports = { init, query };
module/patterns.js
require('dotenv').config();
const db = require('./db');

db.init({
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
```

```
  database: process.env.DB_NAME
});

async function patternExists(pattern) {
  const rows = await db.query('SELECT id FROM Threat_Patterns WHERE pattern = ?', [pattern]);
  return rows.length > 0;
}

async function savePattern(pattern, context, severity = 'medium', type = 'regex') {
  try {
    await db.query(
      'INSERT INTO Threat_Patterns (pattern, context, severity, attack_type, added_on) VALUES (?, ?, ?, ?,
NOW())',
      [pattern, context, severity, type]
    );
  } catch (err) {
    console.error(' Помилка при збереженні шаблону:', err);
  }
}

module.exports = { patternExists, savePattern };
```

Додаток В. Лістинг програмного модулю захисту

filterInput()

```
async function filterInput(input) {
  input = decodeObfuscated(input);
  let cleaned = input;
  let threat = 'Safe';
  let patternMatched = null;

  const context = detectContext(input);
  const rules = await getRules();

  for (const rule of rules) {
    if (rule.context.toLowerCase() !== context.toLowerCase()) continue;

    let regex;
    try {
      regex = new RegExp(rule.pattern, 'gi');
    } catch (err) {
      console.error(' Invalid RegExp:', rule.pattern);
      continue;
    }

    if (regex.test(cleaned)) {
      threat = 'XSS detected';
      cleaned = applyEscaping(cleaned, context);
      patternMatched = rule.pattern;
    }
  }

  try {
    const dom = new JSDOM(`<body>${cleaned}</body>`);
    const scripts = dom.window.document.querySelectorAll('script, [onerror], [onload], [onclick],
[href]');
    if (scripts.length > 0) {
      threat = 'XSS detected';
      patternMatched = '<DOM-based>';
      const autoPattern = extractPattern(input);
      if (autoPattern && !(await patternExists(autoPattern))) {
        await savePattern(autoPattern, context, 'medium', 'dom');
        patternMatched = autoPattern;
      }
    }
  } catch (err) {
    console.error('JSDOM parsing error:', err);
  }

  return { cleaned, threat, patternMatched };
}
```

contextFilter.js

```
function detectContext(input) {
  if (/^\s*{.*}\s*$/.test(input)) return 'json';
  if (/javascript:/i.test(input)) return 'url';
  if (/eval|setTimeout|document\.write|Function/.test(input)) return 'js';
  return 'html';
}
function decodeObfuscated(text) {
  if (!text) return '';
  try {
    const textarea = new JSDOM().window.document.createElement('textarea');
    textarea.innerHTML = text;
    const decoded = textarea.textContent;
    return decoded.replace(/\u([0-9a-fA-F]{4})/g, (_, code) => String.fromCharCode(parseInt(code, 16)));
  } catch (e) {
    return text;
  }
}
function applyEscaping(text, context) {
  switch (context) {
    case 'html':
      return escapeHTML(text);
    case 'js':
      return escapeJS(text);
    case 'json':
      return escapeJSON(text);
    case 'url':
      return escapeURL(text);
    default:
      return text;
  }
}
function escapeHTML(text) {
  return text
    .replace(/&/g, '&amp;')
    .replace(/</g, '&lt;')
    .replace(/>/g, '&gt;')
    .replace(/"/g, '&quot;')
    .replace(/'/g, '&#039;');
}

function escapeJS(text) {
  return text
    .replace(/\\/g, '\\\\')
    .replace(/'/g, '\\\'')
    .replace(/"/g, '\\"')
    .replace(/\/g, '\\/')
    .replace(/\n/g, '\\n')
    .replace(/\r/g, '\\r');
}
```

```
function escapeJSON(text) {
  return text
    .replace(/\\/g, '\\\\')
    .replace(/"/g, '\\"');
}

function escapeURL(text) {
  return encodeURIComponent(text);
}
function extractPattern(input) {
  const regexList = [
    /on\w+\s*=\s*["'].*?["']/i,
    /href\s*=\s*"javascript:[^"]*"/i,
    /eval\((.*?)\)/i,
    /alert\((.*?)\)/i
  ];
  for (const rex of regexList) {
    const match = input.match(rex);
    if (match) return match[0];
  }
  return null;
}
```

Додаток Г. Ілюстративний матеріал

Розробка програмного модуля для захисту веб-додатків від XSS-атак з використанням адаптивної фільтрації та ескейпінгу вхідних даних на серверній стороні з автоматичним оновленням правил безпеки

Виконав студент групи 1КІТС-21Б Вітковський А.В.

Науковий керівник д.ф. (PhD), доц., доцент каф. МБІС Салієва О. В.

Мета роботи

Метою роботи є розробка програмного модуля, який забезпечує захист веб-додатків від XSS-атак шляхом застосування адаптивної фільтрації, контекстного ескейпінгу вхідних даних на серверній стороні та автоматичного оновлення правил безпеки. Реалізований модуль має бути гнучким, масштабованим, здатним до самонавчання і легко інтегруватися у сучасні інформаційні системи без втручання в бізнес-логіку додатків.

Переваги розробленого модуля

Контекстна фільтрація: обробка вхідних даних з урахуванням типу контенту (HTML, JS, JSON, URL) підвищує точність виявлення загроз.

Адаптивність: автоматичне оновлення шаблонів загроз без потреби втручання адміністратора.

Самонавчання: система здатна розпізнавати нові небезпечні конструкції та поповнювати базу знань.

Серверна обробка: фільтрація відбувається на сервері – незалежно від клієнтського оточення.

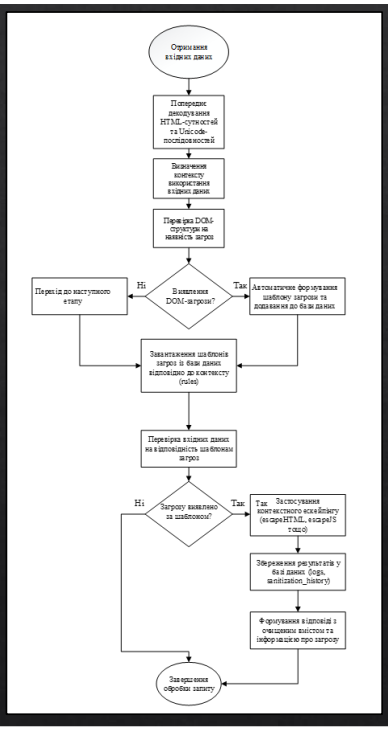
Інтеграція через REST API: модуль легко вбудовується в будь-який веб-додаток.

Збереження логів: усі події зберігаються в базі даних для подальшого аудиту й аналізу.

Масштабованість: рішення придатне як для малих проєктів, так і для корпоративних систем.

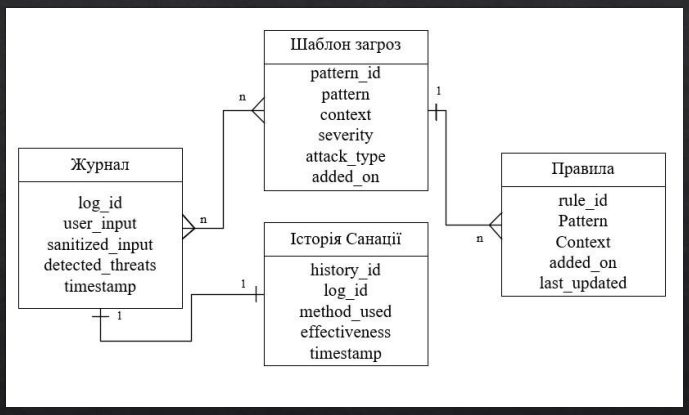
Блок-схема алгоритму динамічного захисту від XSS-атак

1. Система отримує вхідні дані користувача.
2. Виконується декодування HTML-сутностей і обфускованих символів.
3. Визначається контекст (HTML, JS, URL або JSON).
4. Дані аналізуються через JSDOM для виявлення DOM-орієнтованих загроз.
5. Якщо виявлено нову загрозу — формується шаблон і зберігається в базу.
6. З бази правил витягуються шаблони, що відповідають поточному контексту.
7. Перевіряється відповідність вхідного рядка шаблонам:
   - якщо загрозу знайдено — застосовується відповідний метод ескейпінгу;
   - якщо не знайдено — дані залишаються без змін.
8. Результати обробки записуються в базу (логи та історія фільтрації).
9. Формується відповідь із очищеними або позначеними даними.



ER-діаграма бази даних системи захисту від XSS

Журнал (Logs): зберігає всі вхідні дані, результат фільтрації, виявлені загрози та час обробки.
Шаблони загроз (Threat_Patterns): містить зразки XSS-атак із зазначенням контексту, типу та рівня небезпеки.
Правила (Rules): зберігає шаблони, що використовуються для фільтрації, з прив'язкою до контексту та часу оновлення.
Історія санації (Sanitization_History): фіксує, які методи очищення були застосовані, їхню ефективність та час виконання.

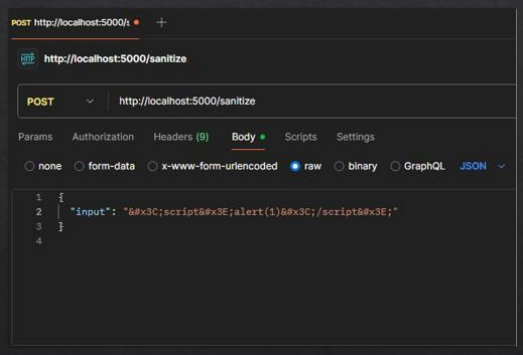


Використані технології

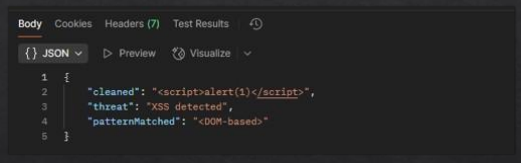
- **Node.js** серверна платформа для обробки HTTP-запитів та реалізації логіки модуля.
- **Express.js** створення REST API для взаємодії з модулем.
- **MySQL** збереження журналів, шаблонів загроз, правил і історії санації.
- **JSDOM** аналіз структури HTML-контенту для виявлення DOM-based атак.
- **Регулярні вирази (RegExp)** виявлення небезпечних шаблонів у тексті.
- **Postman** тестування API-запитів до модуля.
- **JavaScript** основна мова програмування для реалізації всієї логіки модуля.

Процедура надсилання запиту через Postman

Запит

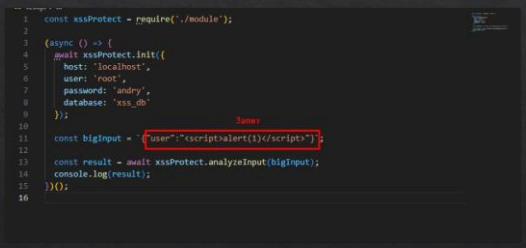


Відповідь

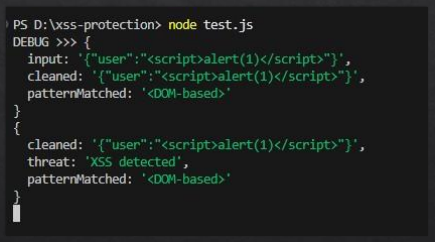


Процедура надсилання запиту через файл test.js

Запит

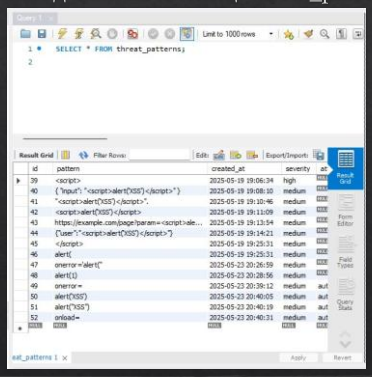


Відповідь

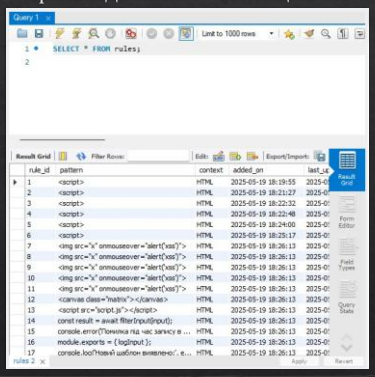


Приклади записів в базу даних MySQL

Приклад записів таблиці threat_patterns

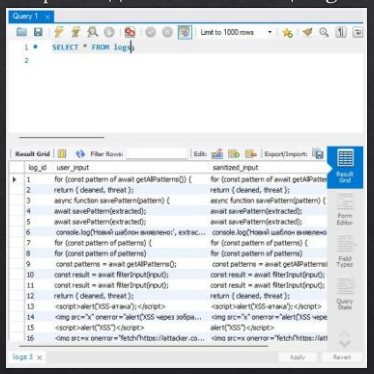


Приклад записів таблиці rules

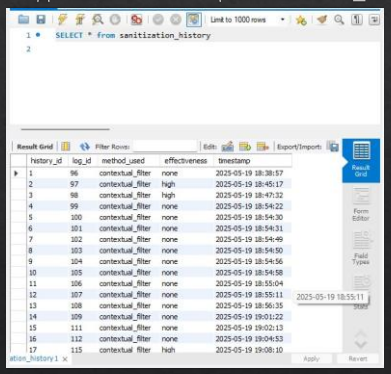


Приклади записів в базу даних MySQL

Приклад записів таблиці logs



Приклад записів таблиці sanitization_history



Висновки

- Під час розробки модуля були досягнуті такі цілі:
- Розроблено захисний модуль, здатний виявляти XSS-атаки з урахуванням контексту введених даних.
- Реалізовано механізм адаптивної фільтрації з автоматичним оновленням шаблонів загроз.
- Побудовано ефективну структуру бази даних для зберігання логів, правил і шаблонів.
- Забезпечено серверну обробку та REST API для інтеграції з іншими системами.
- Проведено тестування, що підтвердило надійність, швидкодію та практичну придатність рішення.

Дякую за увагу!

Додаток Ґ. Протокол перевірки на антиплагіат

ПРОТОКОЛ ПЕРЕВІРКИ НАВЧАЛЬНОЇ (КВАЛІФІКАЦІЙНОЇ) РОБОТИ

Назва роботи:

Розробка програмного модуля для захисту веб-додатків від XSS-атак з використанням адаптивної фільтрації та ескейпінгу вхідних даних на серверній стороні з автоматичним оновленням правил безпеки

Тип роботи: бакалаврська кваліфікаційна робота

Підрозділ Кафедра менеджменту на безпеки інформаційних систем
Факультет менеджменту та інформаційної безпеки
Гр. 1КІТС-21б

Керівник доцент Салієва О.В. _____

Показники звіту подібності

Strike Plagiarism	
Оригінальність	99,31 %
Загальна схожість	0,69 %

Аналіз звіту подібності (відмітити потрібне)

- ☐ **Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.**
- ☐ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.
- ☐ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Заявляю, що ознайомлений (-на) з повним звітом подібності, який був згенерований Системою щодо роботи (додається)

Автор _____ Вітковський А.В.
(підпис) (прізвище, ініціали)

Опис прийнятого рішення

☐ **Допустити до захисту**

_____

_____

_____

_____

Особа, відповідальна за перевірку _____ Коваль Н.П.
(підпис) (прізвище, ініціали)

Експерт _____ _____
(за потреби) (підпис) (прізвище, ініціали, посада)