МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра кібербезпеки

КАРПЕЦ Дмитро Олександрович

Алгоритми протидії вразливостям Cross-Site Scripting у веб додатках / Algorithms for Combating Cross-Site Scripting Vulnerabilities in Web Applications

спеціальність: 125 – Кібербезпека та захист інформації
освітньо-професійна програма – Кібербезпека

Кваліфікаційна робота

Виконав студент групи КБм-21
Д.О.Карпец

Науковий керівник
к.т.н., доцент С.В.Івасьєв

Кваліфікаційну роботу
допущено до захисту:

«____» _____ ___2025 р.

Завідувач кафедри
_____ В.В.Яцків

ТЕРНОПІЛЬ - 2025

Факультет комп'ютерних інформаційних технологій
Кафедра кібербезпеки
Освітній ступінь «магістр»
спеціальність: 125 - Кібербезпека та захист інформації
освітньо-професійна програма –Кібербезпека

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ В.В.Яцків
«____» _____ 2024 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
КАРПЕЦА ДМИТРА ОЛЕКСАНДРОВИЧА
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи:
Алгоритми протидії вразливостям Cross-Site Scripting у веб додатках / Algorithms for Combating Cross-Site Scripting Vulnerabilities in Web Applications
керівник роботи д.т.н., доцент С.В. Івасьєв
затверджені наказом по університету від 20 грудня 2024 року № 938
2. Строк подання студентом закінченої випускної кваліфікаційної роботи 5 грудня 2025 року.
3. Вихідні дані до кваліфікаційної роботи: завдання на випускну кваліфікаційну роботу студента, наукові статті, технічна література.
4. Основні питання, які потрібно розробити:
- дослідити алгоритми та підходи виявлення XSS-вразливостей;
- розглянути техніки обходу XSS-фільтрів та їх вплив на безпеку;
- визначити ефективні методи захисту та профілактики XSS-атак;
- створення тестового веб-застосунку, що містить контрольовані XSS-вразливості;
- проведення експлуатації моделей загроз, пов'язаних із XSS;
- розробку та впровадження функцій і алгоритмів запобігання XSS-вразливостям у веб-застосунку.

5. Перелік графічного матеріалу у роботі:
- Схема роботи веб-застосунку.
- Кібер актори та мотивація.
- Популярність категорій вразливостей згідно OWASP Top 10.
- Популярність вразливостей CWE Top 25.
- Успішна XSS атака у контексті HTML тіла.
- Схема алгоритму протидії XSS вразливостям через валідацію даних.
- Схема алгоритму протидії XSS вразливостям через очищення даних.

6. Консультанти розділів кваліфікаційної роботи

	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 20 грудня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строки виконання етапів кваліфікаційної роботи	Примітка
1	Теоретичний аналіз атак на веб застосунки	12.2024 р. – 03.2025 р.	
2	Аналіз методів виявлення та запобігання xss вразливостям у веб застосунках	03.2025 р. – 06.2025 р.	
3	Проектування та розробка	06.2025 р. – 11.2025 р.	

Студент _____ Дмитро КАРПЕЦ
(підпис)

Керівник роботи _____ к.т.н., доцент Степан ІВАСЬЄВ
(підпис)

АНОТАЦІЯ

Карпец Д.О. Алгоритми протидії вразливостям Cross-Site Scripting у веб додатках. Західноукраїнський національний університет, Тернопіль, 2025.

Магістерська кваліфікаційна робота за спеціальністю 125 – «Кібербезпека та захист інформації», освітньо-професійною програмою «Кібербезпека».

Метою роботи є розробка алгоритмів протидії Cross-site Scripting (XSS) вразливостям у веб додатках. В ході роботи було досліджено XSS веб вразливості, їх види, причини виникнення, ризики та наслідки пов'язані з ними. Комплексно досліджено методи виявлення XSS вразливостей, їх поведінка відносно контекстного розташування, досліджено принципи створення шкідливих навантажень для тестування веб застосунку на наявність XSS, а також обходу фільтрів запобігання та інших механізмів безпеки. Проаналізовано реалізації обробки даних у веб застосунках, досліджено методи коректної валідації та виводу даних. Розроблено та впроваджено функції запобігання XSS вразливостей на основі досліджених методів.

Розроблено алгоритм валідації, алгоритм очищення та алгоритм кодування даних для запобігання XSS вразливостей у веб застосунках. Розроблено тестовий веб застосунок вразливий до XSS, на якому протестовано роботу алгоритмів шляхом передавання шкідливих навантажень у вхідні точки передачі даних. Алгоритми продемонстрували позитивні результати щодо протидії XSS.

Ключові слова: Cross-site Scripting; XSS; Міжсайтовий скриптинг; XSS Mitigation; Input Validation; Improper Neutralization; Injection;

ABSTRACT

Karpets D.O. Algorithms for Combating Cross-Site Scripting Vulnerabilities in Web Applications. West Ukrainian National University, Ternopil, 2025.

Master’s qualification thesis in specialty 125 – "Cybersecurity and Information Protection," educational-professional program "Cybersecurity."

The purpose of this work is to develop algorithms for countering Cross-Site Scripting (XSS) vulnerabilities in web applications. During the research, XSS web vulnerabilities were investigated, including their types, causes, and associated risks and consequences. A comprehensive study was conducted on XSS vulnerability detection methods, their behavior in relation to contextual placement, principles of creating malicious payloads for testing web applications for XSS presence, as well as bypassing prevention filters and other security mechanisms. Data processing implementations in web applications were analyzed, and methods for proper data validation and output were examined. XSS vulnerability prevention functions were developed and implemented based on the researched methods.

A validation algorithm, a sanitization algorithm, and a data encoding algorithm were developed to prevent XSS vulnerabilities in web applications. A test web application vulnerable to XSS was developed, on which the algorithms were tested by transmitting malicious payloads to data input points. The algorithms demonstrated positive results in countering XSS.

Keywords: Cross-Site Scripting; XSS; XSS Mitigation; Input Validation; Improper Neutralization; Injection.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

CSRF – Cross-Site Request Forgery, підроблення міжсайтового запиту;

DOM – Document Object Model, об’єктна модель документа;

HEX – Hexadecimal, шістнадцятковий формат;

HTML – HyperText Markup Language, мова розмітки гіпертексту;

HTTP – Hypertext Transfer Protocol, протокол передавання гіпертексту;

PHP – Hypertext Preprocessor, препроцесор гіпертексту;

URI – Uniform Resource Identifier, уніфікований ідентифікатор ресурсу;

URL – Uniform Resource Locator, уніфікований локатор ресурсу;

WAF – Web Application Firewall, мережевий екран веб застосунку;

XSS – Cross-site Scripting, міжсайтовий скриптинг.

ВСТУП

Актуальність роботи. Сучасне цифрове середовище неможливо уявити без веб застосунків, вони використовуються для онлайн покупок, фінансових операцій, комунікації, зберігання персональних матеріалів і багатьох інших повсякденних потреб. Кожен із цих процесів передбачає передачу та обробку чутливих даних, які вимагають високого рівня захисту, що буде гарантувати конфіденційність, цілісність та доступність інформації.

Разом із розвитком веб технологій активно зростає й кількість кіберзагроз. Зловмисники використовують слабкості у веб системах для отримання несанкціонованого доступу, викрадають або пошкоджують дані та часто мають на меті отримання фінансової вигоди. Тому питання безпеки веб застосунків потребує особливої уваги.

Серед великого переліку веб вразливостей, однією з найбільш поширених та небезпечних протягом останніх двох десятиліть залишається Cross-Site Scripting (XSS). За даними провідних рейтингових списків, попри добре відомі принципи захисту, XSS стабільно входить до переліку найбільш розповсюджених загроз. Таким чином, дослідження алгоритмів протидії XSS вразливостям залишається актуальною задачею й сьогодні.

Мета роботи полягає в дослідженні XSS вразливостей, методів захисту та розробці алгоритмів протидії. Практичному застосуванню розроблених алгоритмів та оцінці їх ефективності. Огляду популярних веб вразливостей та місцю XSS вразливостей серед них. Дослідженні методів виявлення XSS вразливостей, аналізу корінних причин виникнення, яким чином вони впливають на загальну безпеку веб застосунків, ризики та наслідки пов'язані з ними. Дослідженні й аналізу алгоритмів атак, методів маніпулювання вхідними даними, методів обходу захисних фільтрів, розробці тестового застосунку та реалізації функцій захисту на ньому.

Досягнення визначеної мети передбачає вирішення таких завдань:

- дослідити алгоритми та підходи виявлення XSS-вразливостей;

- розглянути техніки обходу XSS-фільтрів та їх вплив на безпеку;
- визначити ефективні методи захисту та профілактики XSS-атак;
- створення тестового веб-застосунку, що містить контрольовані XSS-вразливості;
- проведення експлуатації моделей загроз, пов’язаних із XSS;
- розробку та впровадження функцій і алгоритмів запобігання XSS-вразливостям у веб-застосунку.

Об’єкт дослідження – процес забезпечення безпеки веб-додатків від атак типу Cross-Site Scripting.

Предмет дослідження – алгоритми, методи та технічні підходи виявлення, запобігання та нейтралізації XSS-вразливостей у веб-додатках.

Наукова новизна одержаних результатів: запропоновано модифікації алгоритмів запобігання Cross-Site Scripting вразливостям, що покращують процеси обробки вхідних даних та збільшують безпеку веб застосунку. Розроблено алгоритми та функції запобігання XSS вразливостей, які можуть бути використаними у веб застосунках під час обробки даних.

Практична цінність: розроблено тестовий веб застосунок на якому відпрацьовано роботу алгоритмів та підтверджено їх ефективність.

Публікації та апробація кваліфікаційної роботи.

1. Карпец Д.О., Наслідки Cross Site Scripting атак у веб додатках. Збірник матеріалів науково-практичної конференції молодих вчених, аспірантів та студентів «Кібербезпека та комп’ютерно-інтегровані технології», Тернопіль, 2024. 31-32 с.

2. Карпец Д.О., Алгоритм протидії Cross-Site Scripting атакам на веб додатки. Збірник текстів, наукових матеріалів доповідей та тез учасників XIV міжнародної науково-технічної конференції «ITSec: Безпека інформаційних технологій»., Тернопіль, 2025. 95-96 с.

1 ТЕОРЕТИЧНИЙ АНАЛІЗ АТАК НА ВЕБ ЗАСТОСУНКИ

1.1 Огляд вразливостей веб застосунків

Веб застосунок – це програмне забезпечення клієнт-серверної моделі, яке забезпечує обслуговування клієнтів через комп'ютерну мережу, використовуючи перелік стандартизованих веб технологій. Веб додаток можна розділити на дві основні складові, клієнтську та серверну частину, де клієнтська у парі з функціоналом веб браузера виконує взаємодію з користувачем через інтерфейс, представляючи дані у бажаному вигляді, а серверна забезпечує віддалену серверну логіку веб застосунка, в основі якої лежить отримання, обробка та відправлення даних [4].

Клієнтські та серверні частини використовують безліч технологій, проте основним каркасом для клієнтської виступає Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), JavaScript, а для серверної перелік мов програмування (JavaScript, Python, Java, PHP та інші), середовища виконання, бази даних, веб сервера (рисунок 1.1) [5].

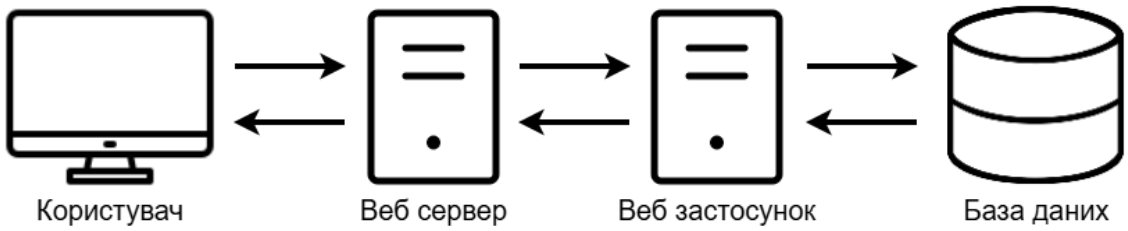

Рисунок 1.1 – Схема роботи веб-застосунку

Дані у веб застосунках є найціннішим і найвразливішим ресурсом, а операції над даними – одне з головних завдань веб серверів. Транспортування даних виконує протокол передачі гіпертексту Hypertext Transfer Protocol (HTTP) та його більш захищена версія Hypertext Transfer Protocol Secure (HTTPS), саме з їх допомогою відбувається комунікація користувачів з веб застосунками. Окрема надважлива задача це збереження даних, яка зазвичай досягається за допомогою

використання реляційних Structured Query Language (SQL) або не реляційних баз даних [2, 7].

Там де є дані, там мають бути впроваджені механізми їх захисту. Тріада, яка складається з конфіденційності, цілісності та доступності є основою моделі (рисунок 1.2), яка має використовуватись під час планування, розробки, оцінки, впровадження та підтримання систем.

Конфіденційність – принцип, мета якого гарантувати доступ до даних виключно уповноваженій особі, особам чи групі, та не допустити несанкціонований доступ до інформації чи її вільне розголошення [3,5].

Цілісність – принцип, мета якого гарантувати узгодження, точність, достовірність даних та не допустити знешкодження, пошкодження чи будь які інші зміни над даними [3,5].

Доступність – принцип, мета якого гарантувати своєчасний доступ до даних, системи, інших ресурсів, в момент коли користувач цього буде потребувати [3,5].

Рисунок 1.2 – Тріада CIA

Говорячи про безпеку веб застосунків, визначення вразливості слід розуміти, як недолік, слабкість, або помилка яка була допущена в дизайні логики, налаштувань, під час проектування або прямо під час роботи веб застосунка та його інших елементів, які взаємодіють з ним. Наявність вразливостей це можливість для зловмисника скористатися ними задля отримання конфіденційних даних, пошкодження цих даних або зробити їх не доступними.

Вразливості можуть з'являтися під час будь якого етапу життєвого циклу програмного забезпечення (ЖЦПЗ), це може бути неуважність під час реалізації функцій кодування, налаштувань конфігурацій, погано продуману валідацію вхідних даних, недостатньо перевірені залежності, оновлення залежностей на іншу версію, тощо [8, 13].

Атака на веб додаток – це навмисна дія або серія дій зловмисника, спрямованих на використання відомих йому вразливостей веб застосунку, або пов'язаних з ним залежностей [7].

Зловмисник, який використовуючи вразливість веб застосунку виконав успішну атаку, може здійснити витік даних, несанкціонований доступ, підвищення привілеїв, вшивання бекдору, виконання шкідливого коду, відмову в обслуговуванні чи повну компрометацію системи [7, 8].

Хоча зазвичай мета зловмисників основана на грошовому збагачені, нерідко вона може суттєво відрізнятися. Існує пряма залежність від типу зловмисника та його навиків (рисунок 1.3).

Advanced Persistent Threats (APT) – це високо організоване та мотивоване угропування, як правило спонсоруються країною походження самого угрупування. Їм під силу виконувати атаки найскладніших категорій, як от Supply Chain атаки, здійснювати вразливості нульового дня. Окремою особливістю є часте застосування націленого фішінгу [10, 17].

Кібер терористи – група лиць яка діє щоб підірвати та, або дестабілізувати певне становище, наприклад політичні вибори.

Хактивісти – зловмисники мотивовані через політичні, соціальні чи ідеологічні погляди, часто фокусуються на здійснені DDoS атак, витоку даних та публічному повідомленні своїх мотивів.

Скрипт діти – зловмисники, які не володіють поглибленими знаннями та які, як правило використовують готові інструменти та експлойти. Своєю жертвою вибирають слабо захищені цілі.

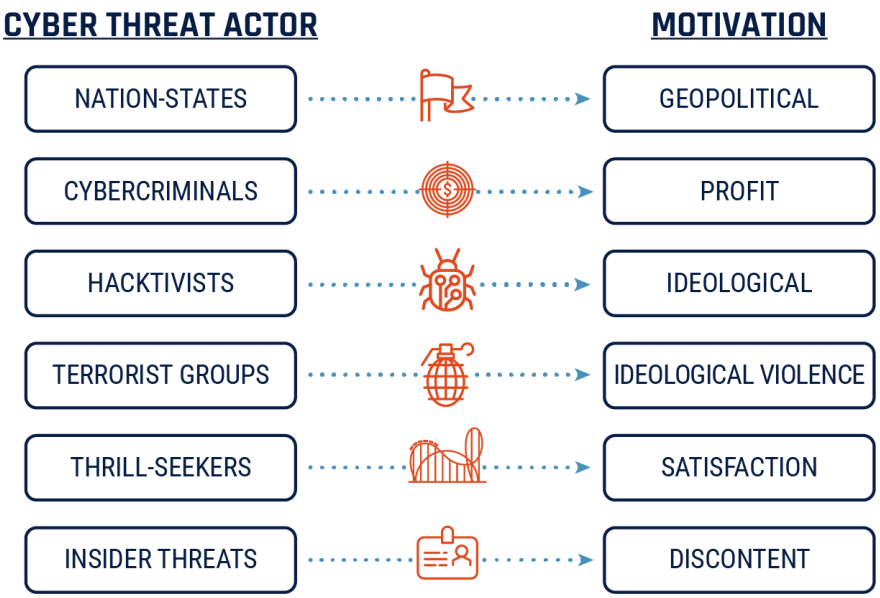

Рисунок 1.3 – Кібер актори та мотивація

Common Vulnerabilities and Exposures (CVE) – це система каталогу, яка відстежує, описує та оновлює вразливості. Кожна вразливість отримує унікальний ідентифікатор, детальний опис про наслідки та шляхи відтворення, а також оцінку критичності [14, 15].

CVE система, як величезна база про вразливості у веб застосунках та не тільки, є незамінним помічником для фахівців з кібербезпеки, дослідників та розробників для поглибленого вивчення вразливостей.

CVE ідентифікатор має формат у вигляді CVE-YYYY-NNNN, де Y позначає рік присвоєння, а N – номер порядку (наприклад, CVE-2025-1641).

Common Vulnerability Scoring System (CVSS) – це система оцінювання вразливостей, яка має на меті оцінити критичність вразливості шляхом оцінювання багатьох факторів, які впливають на складність її реалізації та наслідки до яких вона може привести. Кожна окрема вразливість може бути оцінена від 0 до 10, де 10 найвищий рівень критичності. Така оцінка допомагає ефективно зрозуміти про потенційні ризики та визначити пріоритетність заходів з усунення [21, 22].

Common Weakness Enumeration (CWE) – це стандартизована система класифікації загального переліку слабкостей в програмному та апаратному

забезпечені, підтримується корпорацією MITRE. CWE виступає загальним фреймворком для ідентифікування, каталогізації та опису недоліків чи вразливих місць в програмних системах які можуть призвести до порушення принципів CIA тріади.

Слабкість – це стан програмного чи апаратного забезпечення, який під час певних обставин може бути причиною реалізації вразливостей.

База даних CVE тісно пов'язана з CWE, де остання описує слабкості як абстрактні речі, які мають тенденцію систематичної появи у багатьох програмних системах, ці слабкості не є власне вразливостями, а фундаментальними умовами в яких можуть виникнути вже конкретні вразливості. Для прикладу, запис CWE може загально описувати неналежну валідацію вхідних даних, в той час як CVE запис документує перший конкретний випадок такої вразливості, з конкретним програмним об'єктом, з конкретною версією цього об'єкта та іншими умовами.

Кожен CWE запис складається з унікального ідентифікатор (для прикладу CWE-79), детального опису слабкості, потенційних наслідків, прикладів вразливого коду, переліку дій для усунення слабкості, взаємозв'язки з іншими CWE записами.

Open Worldwide Application Security Project (OWASP) – неприбуткова організація, яка функціонує з метою покращення безпеки програмного забезпечення. OWASP включає перелік проектів з відкритим кодом, документацію та стандарти, які підтримуються спільнотою. Організація налічує більше 250 відділень по всьому світу [5].

OWASP Top 10 – це документ, який вже є стандартом що описує безпеку веб додатків. Він зосереджений на детальному представлені природи найбільш небезпечних вразливостей що властиві веб додаткам та кореляції ризиків пов'язаних з ними. Документ широко відомий серед веб розробників та спеціалістів кібербезпеки.

Порівнюючи рейтингові списки OWASP Top 10 від 2017, 2021 та 2025 років, що до категорій веб вразливостей зображених на рисунку 1.4 можна

підкреслити, що певні категорії які у 2017 році були менш поширеними, зараз посідають перші місця. Проте також помітно що більшість з них довготривало не відхиляються від своєї позиції на значну кількість сходинок впродовж років. як правило огляд та порівняння застосунків спираючись на список 25-ти найпопулярніших слабкостей 2024 року зі рейтингу CWE Top 25 [19, 23].

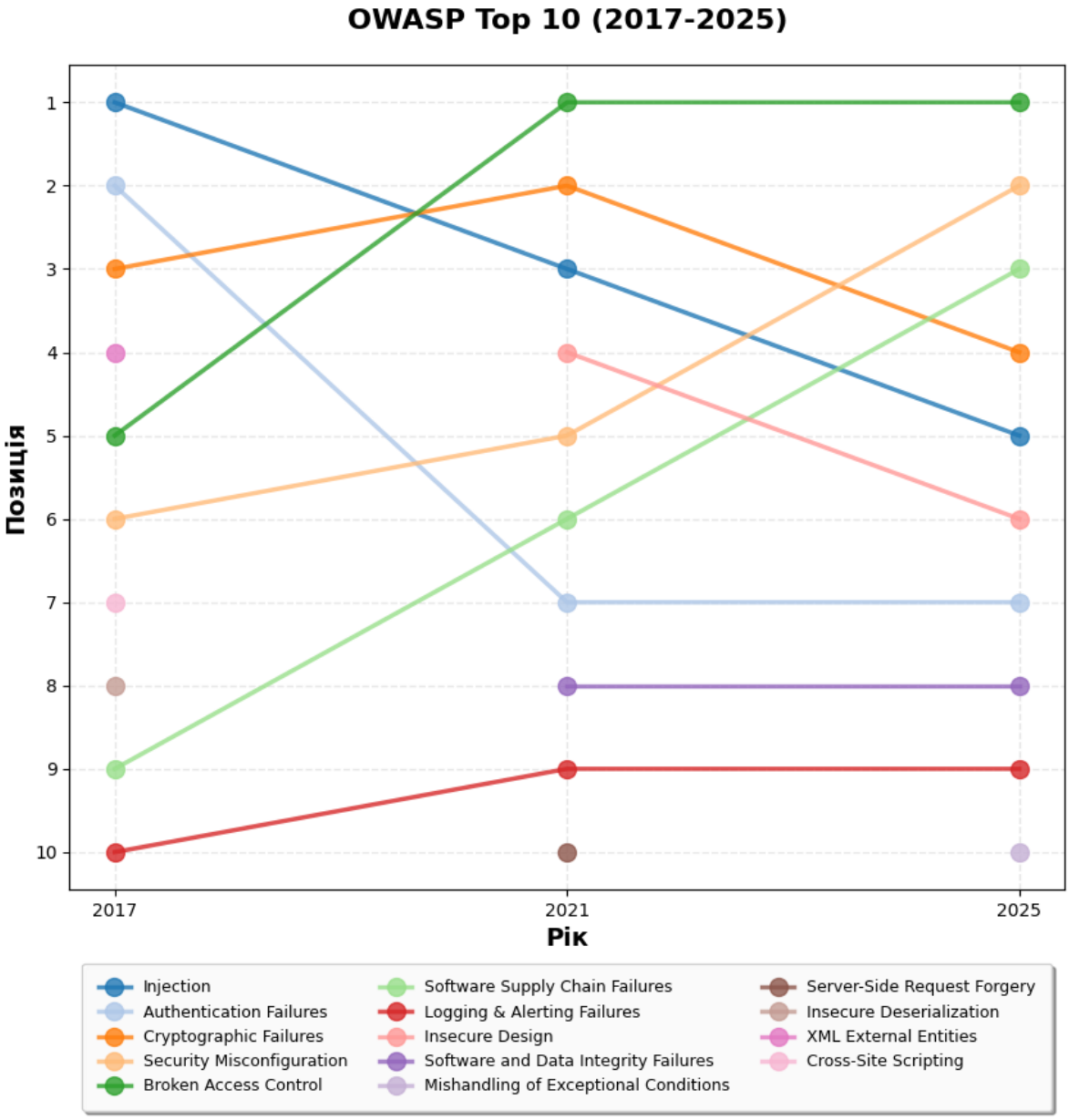

Рисунок 1.4 – Популярність категорій вразливостей згідно OWASP Top 10 від 2017-25 років

CWE Top 25 Most Dangerous Software Weaknesses – рейтинговий список найбільш небезпечних слабкостей у програмному забезпечені на основі бази CWE, яка включає більше 30 тисяч поширених вразливостей.

Згідно рейтингових списків CWE Top 25, від 2020-24 років, на рисунку 1.5 представлено історію змін позицій до таких окремих слабкостей та вразливостей, як Improper Input Validation, Cross-site Scripting та SQL Injection, що всі разом відносяться до категорії Injection.

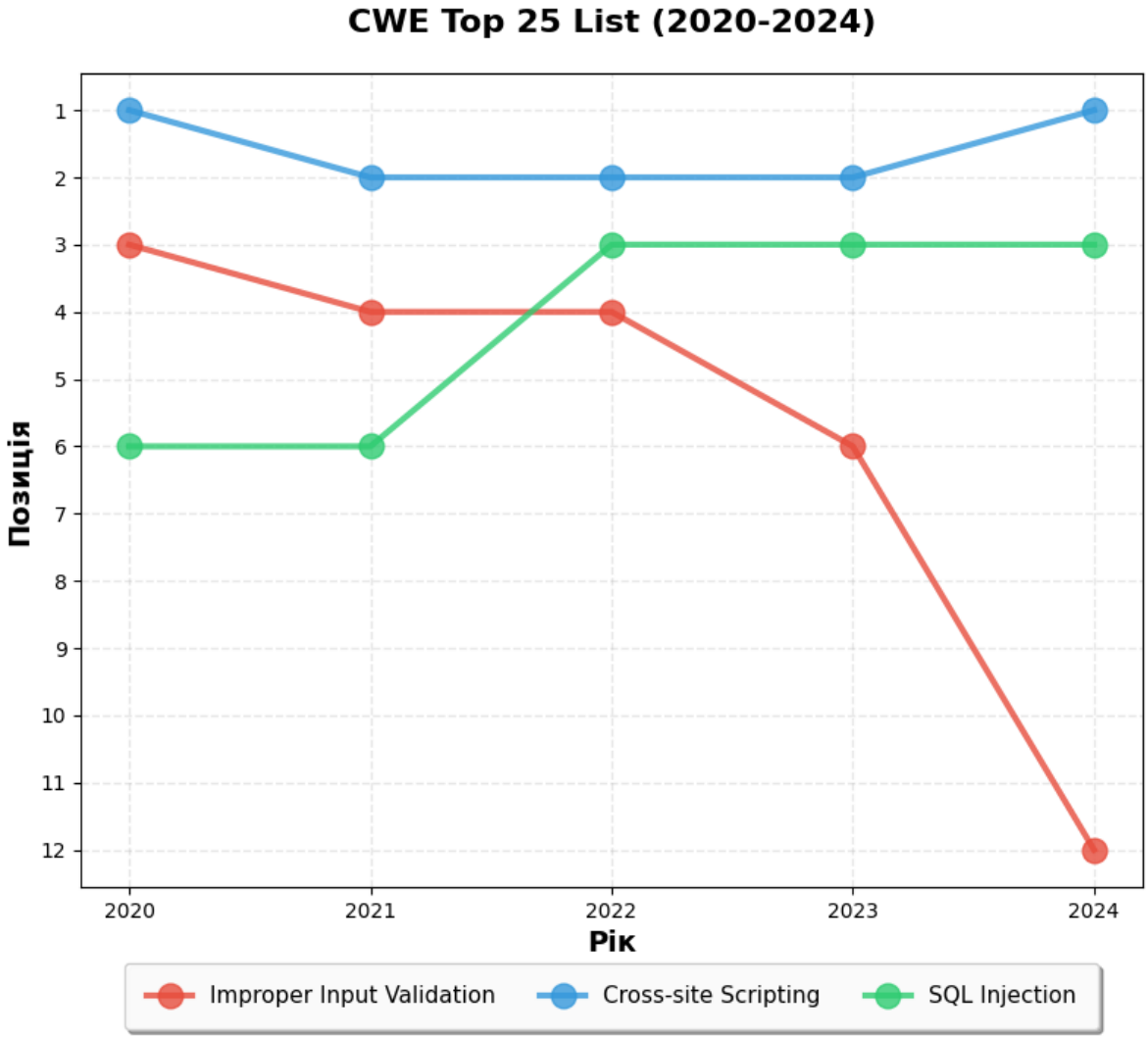

Рисунок 1.5 – Популярність вразливостей CWE Top 25, за 2020-24 роки

Cross-site Scripting (XSS) – веб вразливість, яка згідно з класифікацією CWE, належить до CWE-79 та надає зловмиснику можливість впровадити на стороні клієнта виконання довільного скрипта. Під скриптом, зазвичай

розуміють саме JavaScript скрипт, через свою над популярність та інтегрованість в сучасні браузери по замовчуванню.

Ідея полягає в доставці жертві шкідливого скрипта, результати виконання якого можуть мати серйозні наслідки, оскільки JavaScript має глобальний доступ до веб сторінок у браузері, а семе до Document Object Model (DOM) системи в цілому. Для прикладу, якщо довільний скрипт зловмисника отримає доступ до cookie файлів, як результат він може заволодіти конфіденційними даними, створити шкідливий контент чи посилання, виконати операції від імені жертви, як от заповнення веб-форм [14].

Код нище, бере значення з URL параметра city із HTTP GET запиту та виводить його в HTML.

```
$city = $_GET['city'];
echo '<p> Are you from ' . $city . '? </p>';
```

Оскільки не здійснюється жодна обробка даних, які передаються, параметром city можна маніпулювати, наприклад, передати виконання скрипт блоку.

```
http://blank.com/guest.php?city=<script>print()</script>
```

Як результат, після передачі такого шкідливого навантаження, браузер виведе вікно друку сторінки.

SQL Injection (SQLi) – вразливість в веб застосунках, яка надає зловмиснику можливість маніпулювати Structured Query Language (SQL) запитами, що виконуються Database Management System (DBMS) системою. Через таку маніпуляцію, зловмисник може обійти процес аутентифікації, заполучити конфіденційні дані з бази даних, порушити їх оригінальність, видалити їх, записати довільні нові, чи загалом здійснити привілейовані дії над базою даних [6].

Код нище виконує SQL запит, де отримує дані, які відповідають імені користувача, який пройшов автентифікацію.

```
string mainCity = ctx.getMainCityRequest();
string query = "SELECT * FROM items WHERE city= '" + mainCity + "'
AND itemname = '" + itemName.Text + "'";
sda = new SqlDataAdapter(query, conn);
DataTable dt = new DataTable();
sda.Fill(dt);
```

Таким чином, вигляд фінального запиту буде буде мати вигляд:

*SELECT * FROM items WHERE city= <mainCity> AND itemname = <itemName>;*

Оскільки запит складається з обєднання через контантинацію та містить друге значення itemName з даними, які користувач вводить у відповідне поле, то такий запит буде працювати до тих пір, поки користувач не введе в дане поле одинарні лапки.

Якщо зловмисник, для itemName передасть name' OR 'a'='a', тоді, SQL запит змінить свій вигляд на:

*SELECT * FROM items WHERE city= 'ternopil' AND itemname = 'name' OR 'a'='a';*

Додавання OR 'a'='a заставляє умову WHERE завжди повертати true, через таку особливість, запит можна прирівняти до еквівалентного вигляду:

*SELECT * FROM items;*

Це дозволяє зловмиснику обійти умову де city має містити значення тільки конкретного міста, та як результат поверне дані абсолютно всіх без визначень.

Cross-Site Request Forgery (CSRF) – вразливість у веб застосунках, яка надає зловмиснику можливість виконати несанкціоновані запити від імені авторизованих користувачів, без їх відома [22].

Припустимо що у веб застосунку, яким користується жертва реалізовано функціонал оновлення чутливих даних профіля. HTML код наведено нище.

```
<form action="/path/user.php" method="post">
<input type="text" name="fname"/>
<input type="text" name="lname"/>
<br/>
<input type="text" name="email"/>
<input type="submit" name="submit" value="Change"/>
</form>
```

У user.php реалізовано функціонал захисту, а саме перевіряється чи відправлення форми здійснюється користувачем у якого коректна сесія.

```
session_start();
if (!session_is_registered("uname")) {
echo "The session is invalid!";
[...] // Перенаправлення на сторінку входу
exit;
}
update_profile();
function update_profile {
putNewToDB($_SESSION['uname'], $_POST['email']);
[...]
echo "Профіль оновлено";
```

```
}
```

Такий захист, не зможе запобігти CSRF атаці, оскільки підроблені запити будуть виконуватися довіреним браузером користувача.

Для атаки, зловмисник на власному веб сервері розміщує нейтральну сторінку з HTML конструкцією, яка складається з спеціальних тегів чи включає певні атрибути, які виконують автоматичне звернення при певних умовах.

```
<script>
function doAttack () {
form.email = "attackeremail@blank.com";
form.submit();
}
</script>
<body onload="javascript:doAttack();">
<form action="http://popularsite.com/path/user.php" id="form"
method="post">
<input type="hidden" name="fname" value="Blank">
<input type="hidden" name="lname" value="Blank">
<br/>
<input type="hidden" name="email">
</form>
```

Оскільки веб сторінка містить приховані поля, а функція doAttack() виконується як тільки користувач завантажить сторінку, то зловмисник залишає жертву в абсолютній сліпоті про операції, що відбулись. Таким чином, люба особа яка відкриє дану сторінку та має активну сесію на цільовому веб застосунку, стає жертвою CSRF атаки [25].

1.2 Сканування веб застосунків на вразливості

Забезпечення безпеки веб застосунка є безперервними процесом, який розпочинається на етапі проектування та розробки, до повного його провадження та під час підтримки. Перевірку на вразливості розділяються на дві основні категорії, ручну, таку де перевірка відбувається в ручному режимі, фахівцем або групою фахівців з відповідними знаннями та на автоматичну, де перевірка відбувається шляхом сканування спеціалізованим автоматичним сканером.

Автоматичний сканер тестує веб застосунок шляхом відправлення спеціальних тестових вхідних даних, аналізуючи відповіді застосунку на ці дані та знаходження певних закономірностей, які притамані відомим вразливостям. Головною перевагою автоматичного сканування є скорість, а отже час затрачений на нього є в рази меншим в порівнянні з ручним. Автоматичні сканери часто здібні знайти такі популярні вразливості як XSS, SQLi, CSRF, IDOR, а також критичні недоліки в конфігурації систем чи в реалізації інших процесів, наприклад реалізований функціонал сесій, передбачувані значення в генерації токенів, тощо. Проте недоліком є брак розуміння контексту в якому сканер виконує тестування, що означає неможливість знайти зв'язки між різними частинами веб застосунку, які залежать один від одного [22, 24].

Слід розуміти, що автоматичне сканування не дивлячись на велику кількість переваг, не може гарантувати безпеку веб застосунку. Оскільки перевірка на вразливості є процесом, а не продуктом у вигляді сканера, таке сканування слід розглядати як один з кроків на шляху до безпечного веб застосунку, тому як автоматичне сканування часто є недосконалим та навіть неефектим, якщо говорити про його поглиблене оцінювання [23].

Ручне тестування усуває головний недолік автоматичного сканування, оскільки особа яка його здійснює має розуміння контексту коду та його взаємозв'язків, така перевага також дозволяє знаходити недоліки бізнес-логіки, на що сканери не спроможні. Ручне тестування може виконувати фахівець по

безпеці, тестувальник на проникнення чи розробник веб застосунку. Основна частина процесу полягає в поверхневому перегляді вихідного коду веб застосунку на наявність практик безпечного кодування та оцінці їх реалізацій, прикладами одних з таких практик є валідація вхідних даних чи безпечна обробка SQL запитів. Таким чином, не тільки фахівець з безпеки, а й розробник має володіти відповідними знанням популярних вразливостей та знати як запобігти їх появі, адже саме під час написання коду гарантується швидке виправлення вразливостей без затрат на додатковий час чи фінанси [21, 25, 27].

Гібридне сканування – комбінація автоматичних та ручних методів сканування, є найбільш ефективним та точним оскільки об'єднує переваги двух попередніх підходів, скорість автоматичного сканування та точність ручного. Оскільки такий підхід є найбільш ефективним, це значить що кількість хибно позитивних та хибно негативних результатів є найнижчою [28].

При використані автоматичних сканерів вразливостей, хибно позитивні та хибно негативні результати є регулярним явищем. Автоматичні інструменти можуть не правильно класифікувати поведінку на той чи інший тест, таким чином безпечний результат позначити вразливістю (хибно негативний) або негативний результат тесту позначити як безпечний результат (хибно позитивний). Зважаючи на це, єдиним вірним рішенням буде додаткова ручна перевірка над отриманими результати автоматичного сканера, що підтвердить чи спростує їх достовірність. Якщо таку додаткову перевірку не проводити і довіряти результатам сканеру, хибне відчуття безпеки може призвести до критичних наслідків [30].

Широко відомі автоматичні сканери вразливостей діляться на абсолютно безкоштовні з відкритим кодом та платні комерційні з закритим кодом. Їх головна мета автоматизувати часто повторювані дії.

Burp Suite – платний комплексний сканер веб застосунків, для автоматичного та ручного сканування. Позиціонує себе як інструмент номер один в світі в сфері веб тестувань на проникнення. Містить у собі багато додаткових функцій, такі як перехоплення пакетів та їх редагування через проксі,

вбудоване декодування та кодування, генерування звіту та можливість встановлення додаткових розширень розроблених спільнотою.

OWASP ZAP (Zed Attack Proxy) – безкоштовний автоматичний сканер з відкритим кодом. Підтримує проксі для перехоплення трафіку. Відомий своїм модулем додаткових розширень.

Nessus – платний сканер переважно зосереджений на мережевих але також й веб вразливостях, використовує свою базу відбитків популярних вразливостей, яка регулярно оновлюється. Доступна обмежена, проте безплатна версія. Має високу швидкість сканування та підтримує додаткові розширення.

Nikto – безкоштовний сканер з відкритим кодом, значно програє в розширеному сканувані в порівнянні з іншими сканерами та часто має хибні результати. Як праввило, застосовується для базового помірного сканування веб серверів.

Acunetix – комерційний сканер веб застосунків, який використовує переваги прогресу сучасного світу, інтегрувавши машине навчання для аналізу поведінки з цілю зменшення хибно позитивних результатів та підвищення точності в цілому.

WPScan – найпопулярніший сканер вразливостей для Content Management System (CMS) WordPress. Визначає вразливі версії WordPress, теми, плагіни. Сканер має підтримку команди розробників самої WordPress.

1.3 Поширені вразливості та ризики пов’язані з XSS

XSS атака є процесом, головна мета якого – впровадити скрипт у веб застосунок. Не зважаючи який браузер буде використовувати користувач, скрипт зловмисника має виконуватись як тільки конкретна очікувана дія буде виконана, переважно достатньо переходу за певним посиланням. Зловмисний скрипт

набуває вигляду такого, наче він є частино коду самого веб застосунку, якому довіряє користувач [6, 10].

XSS атаки відрізняються від багатьох інших тим, що атака здійснюється на користувача, а не на сервер застосунку чи інші його залежності. Навіть у випадку коли шкідливий XSS скрипт зберігається на сервері веб застосунку, головою метою залишається змусити користувача до таких дій, щоб під час користування браузером відбулося виконання шкідливого скрипту саме на стороні користувача.

Reflected XSS – є найбільш популярним типом XSS, що реалізується шляхом передачі веб серверу шкідливого навантаження, через використання доступних точок передачі даних. Сервер негайно обробляє отриманий HTTP запит та повертає свою HTTP відповідь у якій міститься передане шкідливе навантаження.

Stored XSS – найнебезпечніший тип XSS, що реалізується шляхом передачі веб серверу шкідливого навантаження, яке сервер зберігає у себе в інфраструктурі, зазвичай в базі даних та незмінно розміщує його в частині веб застосунку. Для прикладу розміщення може бути в частині форуму, розділі коментарів, в профілях користувачів. Як тільки жертва відкриває сторінку з розміщеним шкідливим навантаженням, виконується зловмисний скрипт. Таким чином, даний тип не вимагає прямої взаємодії з жертвою, а виконання скрипту виконується кожен раз при відвідуванні інфікованої сторінки застосунку.

DOM XSS – найрідкісніший тип XSS, що реалізується шляхом передачі локальному приймачу (sink) шкідливого навантаження, який той в свою черго опрацьовує його небезпечним способом. Цей тип жодним чином не взаємодіє з веб сервером, а отже атака відбувається виключно на стороні клієнта. Через свою природу, DOM XSS має тенденцію зустрічатись все рідше і рідше.

Для покращення продуктивності, зручності користувачів і зниження навантаження на сервер, браузери дають можливість веб застосунку зберігати певні дані локально у браузері на стороні користувачів. XSS вразливості мають властивість цією можливістю скористатись. Проте, з ціллю безпеки один веб

застосунок не має можливості оперувати збереженими даними інших застосунків. Оскільки успішне впровадження зловмисного скрипта робить його абсолютно довіреним зі сторони конкретного веб застосунку – це дозволяє скрипту звертатись до його локально збережених даних, часто це сеансові токени або cookie файли [17, 27].

Кінцева шкода яку може завдати успішно реалізована XSS атака має широку варіативність наслідків та їх тяжкість. Для прикладу це можуть бути незначні дратівливі події під час користування браузером, як от постійно спливаюче вікно, або набагато більш суттєві, як повна компрометація облікових записів користувача. Часто розробники веб застосунків та інші фахівці галузі інформаційних технологій не сприймають XSS, як щось серйозне, що може спричинити масштабні негативні наслідки. Саме тому серйозність цих наслідків залежить від рівня досвідченості зловмисника, де використовуючи можливості на максимум, можна завдати вагомої шкоди [23].

Початковий і найбільш очевидний вплив XSS атак відчувають жертви цих атак, а саме користувачі веб застосунків. Через те що зловмисник отримує можливість діяти в контексті сесії конкретної жертви, а значить володіє ідентичними правами доступу до сторінки, що й сам користувач, це призводить до низки небезпечних наслідків.

Викрадення сесії призводить до того що зловмисник повністю обходить процес аутентифікації та отримує несанкціонований доступ до облікового запису жертви. Часто, цього можна досягти через викрадення cookie файлів у яких міститься відповідний запис з ідентифікатором сесії. Токен сесії зловмисник отримує через надсилання на свій веб сервер [6].

Викрадення облікових даних на відміну від викрадення сесії дозволяє зловмиснику не тільки здійснити аутентифікацію в обліковий запис жертви на конкретному веб застосунку, а й використати їх у спробах здійснити вхід в облікові записи на інших ресурсах чи спробу відновлення доступу, оскільки володіє паролем жертви.

Маніпуляція вмістом веб сторінки, яку бачить жертва, дозволяє спонукати жертву до виконання небажаних дій, як от через створену підроблену форму оплати – ввести свої банківські дані, або піддатися спрямованій дезінформації.

CSRF подібна атака може бути реалізована шляхом створення шкідливого навантаження у якому міститься URL адреса на запит певних дій в межах вразливого до XSS, веб застосунку. Браузер намагаючись завантажити ресурс, автоматично виконує запланований запит [31, 32].

Витончені шкідливі навантаження можуть сканувати IP адреси локальної мережі, сканувати порти, записувати нажимання клавіш, викрадати CSRF токени, викрадати будь які інші конкретні елементи даних на сторінці. Ці креативні та складні методи використання XSS вразливостей демонструють всю глибину та гнучкість цієї загрози [29, 30].

Наслідки XSS атак виходять далеко за межі шкоди окремому користувачеві. Нематеріальні наслідки XSS атак можуть бути навіть більш руйнівними, ніж прямі фінансові збитки. Для прикладу, коли фішингові атаки реалізовані через XSS, відбуваються з довіреного домену компанії, це підриває довіру актуальних клієнтів та простих відвідувачів. Відновлення репутації після такого інциденту є тривалим і дорогим процесом, а втрата довіри може призвести до довгострокових збитків для бренду [19].

Не рідко XSS слугує початковим вектором для більш складних, багатоетапних атак.

2 АНАЛІЗ МЕТОДІВ ВИЯВЛЕННЯ ТА ЗАПОБІГАННЯ XSS ВРАЗЛИВОСТЯМ У ВЕБ ЗАСТОСУНКАХ

2.1 Методи виявлення XSS вразливостей

Щоб виявити Stored та Reflected XSS вразливості, у всі можливі точки входу веб застосунку передають довільні унікальний набір даних. Мета – виявити місця розташування переданих даних в отриманих HTTP відповідях від веб сервера. Після виявлення таких місць, дані, які передаються у вхідні точки, модифікують таким чином, що до них додається корисне навантаження, яке має виконати відповідний JavaScript код.

Щоб виявити DOM XSS вразливості, вперш за все акцентують увагу на маніпулюванні URL параметрами, в які так само передаються унікальні вхідні дані, аналізують їх появу у DOM структурі HTML сторінки та намагаються їх модифікувати таким чином, щоб знайти можливість виконати довільний JavaScript код. Аналіз вхідних точок у тестуванні на DOM XSS вразливості, окрім URL параметрів є дуже складним та часозатратним, оскільки аналіз JavaScript приймачів немає подібних альтернатив як у випадку з HTML.

Варто зазначити, що під час перших кроків до визначення XSS вразливостей, в вхідні точки спочатку передають не корисні навантаження, а випадковий вміст символів

Фаззінг – це техніка автоматичного тестування програмного забезпечення, метою якого є виявлення не запланованої аномальної поведінки чи вразливостей через передачу в точки входу випадкових або спеціально підібраних даних.

Для ефективного маніпулювання корисним навантаженням з ціллю виявлення XSS вразливостей, необхідно розуміти в якому контексті це навантаження буде відображатися.

XSS контекст – зона в структурі HTML сторінки в якій розташовується передане корисне навантаження після.

Під час виявлення XSS вразливостей у тілі HTML, передане корисне навантаження розташовується прямо у головних структурних елементах, як от <html>, <body> чи в загальному поміж тегами. Тому, незалежно в якому контексті буде розміщуватись корисне навантаження, в першу чергу необхідно дослідити чи є можливість його створити таким, щоб вийти в конекст тіла HTML. Перехід у вищий контекст за рівнем дає можливість створювати нові HTML елементи використовуючи відповідні символи < та >. Якщо ці критично необхідні символи піддалися процесу кодування чи очищення тоді XSS вразливості не існує. Якщо символи були повернуті у оригінальному вигляді тоді є висока ймовірність існування XSS вразливості.

У сприятливих умовах для контексту тіла HTML, корисні навантаження можуть мати наступний вигляд:

<script>print()</script>
<img src=x onerror=print() >
* *
<iframe onload=print()></iframe>

Корисне навантаження може опинитись в середині атрибуту HTML елемента, така зона розміщення є окремим XSS контекстом. У цьому випадку, ставиться задача вийти за межі атрибуту та елементу. При успішному виходу за межі елементу, стає доступним можливість в використанні загально відомих корисних навантажень щой для контексту тіла HTML. Проте, якщо за межі елементу вийти можливостей немає, але все ж вдається вийти за межі атрибуту, тоді необхідно спробувати створити нові атрибути, які дають можливість виконати JavaScript код.

" autofocus onfocus=print() x="
" onfocus=print() id=t tabindex=0 style=display:block>#t

Інколи якщо немає можливості вийти за межі атрибуту, реалізація XSS все одно залишається можливою. Це напряму залежить від того в якому саме атрибуті розміщується корисне навантаження, оскільки не у всіх є можливість до виконання JavaScript коду. Так наприклад використовуючи атрибут onclick, код виконується при нажиманні на елемент, а в середині атрибуту href можна передати протокол javascript.

White-Box тестування передбачає повний доступ до вихідного коду додатка. Тестувальник перевіряє логіку обробки даних, знаходячи потенційно небезпечні конструкції та відсутність належної обробки даних. Тестування білої скриньки є найефективнішим методом для виявлення всіх потенційних вразливостей. Таке тестування дозволяє зрозуміти першопричину проблеми, проте вимагає значних часових витрат та глибоких знань мов програмування та безпеки. Тестування білої скриньки більш ефективне через відкриті зання про контекст коду. Таким чином, для аналізу коду на XSS вразливості слід перевірити всі змінні в які можуть бути передані дані користувача та проаналізувати реалізовані методи обробки даних загалом.

Black-Box тестування проводиться без доступу до вихідного коду. Тестувальник взаємодіє з додатком як звичайний користувач, намагаючись знайти вразливості через точки вводу. Імітує реальні сценарії атак. Не вимагає знання мов програмування або архітектури додатка. Може пропустити вразливості, що знаходяться в неочевидних або рідко використовуваних частинах коду.

Gray-Box тестування є комбінованим підхідом, що поєднує елементи чорної та білої скриньки. Тестувальник має обмежені знання про внутрішню структуру додатка, наприклад, доступ до документації, обліковий запис або частковий доступ до коду. Таким чином, у контексті тестування на XSS вразливості, тестувальнику може бути відомо, як веб додаток обробляє, приймає та відправляє дані, проте невідомо щодо реалізації інших функцій. Тестування чорної скриньки є найбільш популярним, оскільки організації не часто бажають

демонструвати свій код відкрито, а також зацікавлені у реальній імітації зловмисника. Таке тестування можна розділити на три фази.

Перша фаза має на меті визначити всі місця через які можна передати дані та зрозуміти, яким чином їх можна передати. Включно з прихованими полями, вибраними значеннями по замовчуванню у веб формах та іншими не очевидними формами. Зазвичай для такого ефективного тестування, використовують або вбудовані у браузери інструменти HTML редагування або спеціальні веб проксі застосунки, які дають можливість відредагувати або переглянути HTTP запити та відповіді.

Друга фаза має на меті у всі знайдені місця через які можна передати дані, передати корисне навантаження, яке не має бути перевантаженим, радше наоборот, достатньо короткого безпечного вмісту, щоб визвати реакцію браузера про можливу вразливість.

Третя фаза має на меті аналіз отриманих результатів з попередньої, необхідно визначити чи отримані результати мають дійсно вагомий вплив на безпеку веб застосунку. Такий аналіз, включає в себе перегляд да оцінку отриманої HTML веб сторінки та пошук попередньо знайдених місць передачі даних. При поверхневому огляді таких місць, необхідно впевнений чи є хоч якісь спеціальні символи, які були не належним чином закодовані, замінені чи відфільтровані. Контекст вихідних даних дуже важливий при фінальному висновку, оскільки існують зони такі зони розташування, в яких спеціальні символи зовсім не підвищують рівень вразливості. В кінцевому результаті всі символи перелічені в таблиці 2.1 мають на виході бути закодованими.

Таблиця 2.1 – Символи для кодування

Опис	**Символ**
Більше	>
Менше	<
Амперсант	&
Одинарна лапка	'
Подвійна лапка	"

Якщо контекст розміщення це JavaScript код, або HTML атрибут який може його виконати, варто пересвідчитись чи спеціальні символи наведені у таблиці 2.2 мають відповідне кодування або чи були задіяні інші операції обробки над ними.

Таблиця 2.2 – Операції обробки для заміни

Опис	**Символ**
Новий рядок	\n
Повернення каретки	\r
Одинарна лапка	'
Подвійна лапка	"
Зворотній слеш	\
Unicode значення	\uXXXX

Розгляньмо, приклад коли приймається параметр city, який впливає на відображення певного міста для доставки.

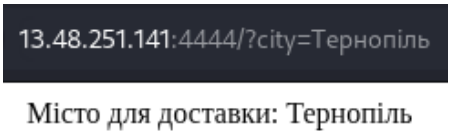

Рисунок 2.1 – Приклад отримання параметру city

Параметр city є точкою передачі даних, а значить обов'язково потрібно проаналізувати вивід та спробувати заставити браузер виконати довільний код.

Рисунок 2.2 – Заміна параметру виконуваним кодом

Якщо у веб застосунку не реалізовано належну обробку вхідних даних, браузер відповість спливаючим вікном, викликане за допомогою переданого навантаження.

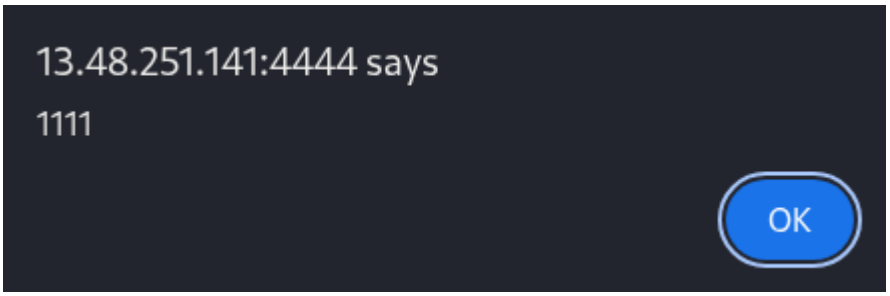

Рисунок 2.3 – Результат виконання підміненого коду

У тестуванні на Stored XSS варто памятати, що цей тип потребує два HTTP запити, на відміну від одного у Reflected XSS, та те що жертва не повинна відкривати специфічно сформоване посилання, достатньо відвідати сторінку вразливу до Stored XSS.

Таким чином, Stored XSS атака часто складається з чотирьох фаз, зловмисник передає шкідливе корисне навантаження додатку, жертва автентифікується у веб додатку, відвідує вразливу частину додатку, шкідливий код виконується на стороні жертви.

Отже, головними цілями є ідентифікація точок входу, які отримуючи дані, зберігають їх та які будуть доступні для перегляду на стороні клієнта. А також дослідження які саме дані ці точки можуть прийняти та чи застосовується будь яке кодування на виході.

Перша фаза тестування Stored XSS – знаходження відповідних точок передачі даних, які будуть зберігатись на серверній частині веб застосунку та будуть доступні для перегляду зі сторони клієнтської частини. У таблиці 2.3 перелічено популярні приклади таких точок.

Таблиця 2.3 – Місця збереження вхідних даних у веб застосунках

Функціонал додатку	Опис
Сторінка профілю	Можливість редагувати дані свого профілю, як от імя, фамілію, телефон, опис, аватар, фон та інше.
Форум	Застосунок у якому можна створювати теми спілкування та робити в них дописи
Коментарі	Секція у якій можна залишати коментарі
Завантаження файлів	Можливість завантажити файли не зважаючи на їх формат
Налаштування	Можливість застосовувати будь які налаштування
Форми звернення	Можливість заповнити форму зі зверненням, наприклад у підтримку застосунку
Кошик товарів	Можливість в інтернет магазинах маніпулювати кошиком товарів

Особливу увагу потрібно приділити точкам передачі даних, які повязані з обліковими записами привілейованих осіб, як от адміністратори, модератори, редактори. Отримання їх облікових даних чи сесійних токенів у випадку Stored XSS мають значно вищу критичність.

Як приволо, дані що можна передати та які доступні для перегляду – розміщуються між HTML тегами, проте нерідко можна зустріти їх розміщення у контексті JavaScript коду.

Приклад Stored XSS у редагуванні особистих даних в профілі користувача зображено на рисунку 2.4 та рисунку 2.5.

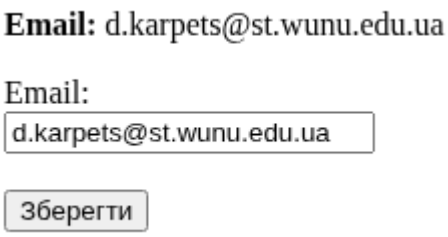

Рисунок 2.4 – Редагування email в профілі користувача

Рисунок 2.5 демонструє Stored XSS, де корисне навантаження автоматично виконується при оновлені сторінки та відкриває спливаюче вікно. У випадку якщо корисне навантаження повернуло закодований або очищений вміст, тоді необхідно застосувати методи обходу фільтрів та. Для наглядного прикладу, застосунок може замінювати <script> теги на пустий вміст у вигляді пробілу чи символу NULL, це однозначно свідчить що у застосунку реалізовані засоби фільтрації.

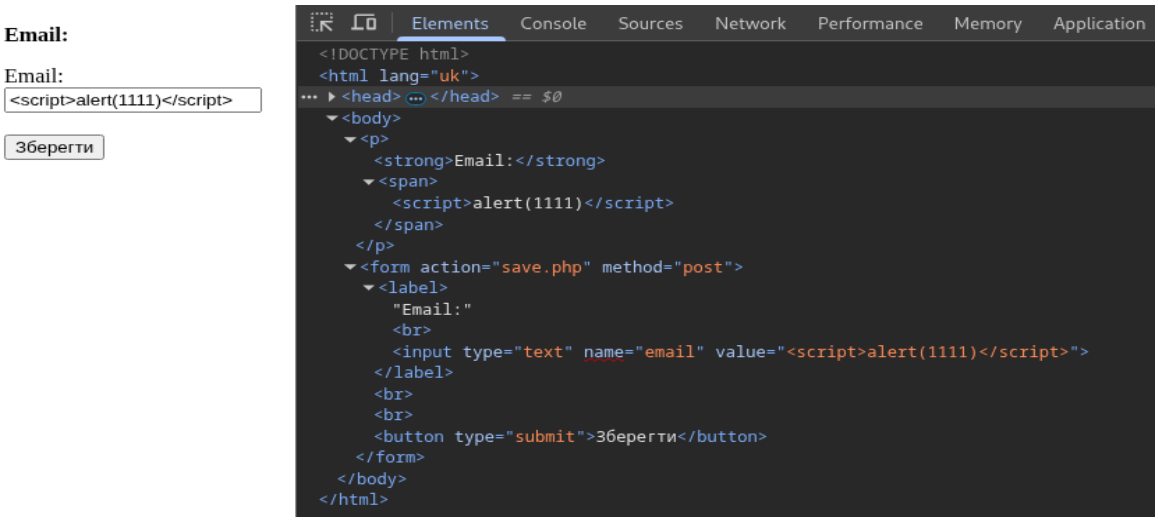

Рисунок 2.5 – Stored XSS при редагуванні email користувача

Необхідно бути впевненим, що застосунок приймає дані дійсно у такому вигляді, як у текстовому полі веб форми. Для цього зазвичай достатньо відключити використання JavaScript на сторінці щоб будь які локальні скрипти не модифікували HTTP запит, або відредагувати його власноруч через веб проксі

інструменти. Інколи також додатково необхідно протестувати інєкцію як через HTTP GET так й через HTTP POST запити.

Завантаження файлів. Якщо веб застосунок дозволяє завантажувати файли, дуже важливо дослідити які саме формати файлів доступні до завантаження, особливий акцент вартно приділити файлам формату HTML. Формати такі як HTML, дозволяють передати шкідливе навантаження прямо в середині вмісту файлу. Окремо, необхідно дослідити чи можливо замінити MIME тип файлу та як веб застосунок на це відреагує.

Файли інших форматів також можуть бути використані у маніпуляції з MIME типами, наприклад завантаживши файл формату JPG та замінивши тіло HTTP запиту на шкідливе навантаження, а у заголовку Content-Type змінити тип на text/html. У випадку непродуманої обробки зі сторони застосунка це призведе до Stored XSS, так як завантажений файл буде розглядатися браузером як файл HTML.

POST /upload.php HTTP/1.1
Content-Disposition: form-data; name="ternopil"; filename="C:\ternopil.jpg"
Content-Type: text/html
<script>print()</script>

При виявлені Stored XSS під час тестування сірої скриньки, процес та поетапність дій залишаються незмінними, проте використовуючи переваги відкритості системи, а саме, передавши тестові навантаження у точки передачі даних, проаналізувати відповіді на їх постійне розміщення у застосунку та особливу увагу приділити як саме дані зберігаються зі сторони серверної частини та що з ними відбувається під час HTTP запиту.

А у випадку тестування білої скриньки, дослідити використання попередньо визначених змінних, які використовуються для збереження даних отриманих через HTTP запити. Такі найпопулярніші змінні відносно до мов програмування наведені у таблиці 2.4.

Таблиця 2.4 – Найпопулярніші змінні відносно до мов програмування

PHP	ASP	JSP
$_GET	Request.QueryString	doGet, doPost
$_POST	Request.Form	request.getParameter
$_REQUEST	Server.CreateObject	
$_FILES		

Під час виявлення DOM XSS, варто мати на увазі що даний тип вразливостей зазвичай виникає при поганій практиці кодування. Оскільки немає комунікації з веб сервером, реалізована валідація та обробка вхідних даних стає неефективною проти цього типу XSS.

Наведено приклад коду, який розміщує посилання з адресного рядку прямо в HTML документ.

```
<script>
document.write("Посилання на вміст: " + document.location.href + ".");
</script>
```

Оскільки символ # розпізнається браузером як фрагмент, при формуванні HTTP запиту, браузер пропускає все що стоїть після # символу. Через це, до адресного рядка можна додати #<script>print()</script> що заставить браузер відкрити вікно друкування сторінки.

Головними цілями при виявлені DOM XSS є виявлення усіх приймачів та створення корисного навантаження, окремо для кожного типу приймача.

Передані користувачем дані поділяються на ті що повернулися від сервера у такому вигляді що роблять XSS не можливим, та на ті що були отримані від JavaScript об'єктів на стороні клієнта.

Наведено приклади, які часто можна побачити, якщо відповідь була отримана напряму від сервера

var city = "Екрановані дані отримані від сервера";
var choose = rndFunction("Екрановані дані отримані від сервера");

Та приклади, які часто можна побачити, якщо взаємодії з сервером не було.

var city = window.location;
var result = rndFunction(window.referrer);

2.2 Методи обходу XSS фільтрів

Метод через кодування даних полягає у маскуванні шкідливого коду шляхом представлення його в іншому форматі, який фільтр не розпізнає як небезпечний, але кінцева інтерпретація браузер буде в необхідному для зловмисника вигляді. Ця техніка безпосередньо експлуатує головний недолік фільтрації на основі чорного списку та сигнатурного аналізу, через неможливість передбачити всі варіанти представлення шкідливого навантаження. У таблиці 2.5 наведені кілька прикладів HTML кодування.

Таблиця 2.5 – Корисне навантаження в залежності від типу кодування

Тип кодування	Корисне навантаження
HTML сутності	'-alert(1)-'
HTML hex без нулів	&#x27-alert(1)-&#x27
HTML hex з нулями	&#x00027-alert(1)-&#x00027
HTML dec без нулів	&#39-alert(1)-&#39
HTML dec з нулями	&#00039-alert(1)-&#00039

Маніпуляція HTML-контекстом. У випадку якщо фільтри зосереджені лише на блокуванні очевидних HTML елементів, як от тег <script> та ігнорують безліч інших контекстів HTML, де може виконуватися JavaScript, у формуванні шкідливого навантаження слід використовувати несподівані або менш поширені HTML теги, атрибути та події для виконання скриптів. Обробників подій, такі як onerror, onload та onmouseover, можуть бути використані в тегах <img>, <body>, <iframe> або навіть <video> для запуску JavaScript. Фільтр, що блокує <script>, може пропустити таке шкідливе навантаження. Деякі теги, хоча й не призначені для виконання коду, можуть бути використані для виконання запиту на зовнішній ресурс, що може бути частиною атаки, наприклад, для передачі викрадених даних.

<img src=t onerror=print() >
<img src=t onerror=print()
<img src=t onerror=print() <
<img src=t onerror=print() //

Різні браузери можуть по-різному інтерпретувати невалідний або нестандартний HTML та JavaScript. Ці розбіжності можна використовувати для обходу фільтрів, налаштованих на поведінку одного конкретного браузера.

Використання javascript: протоколу, цей псевдо протокол дозволяє виконати JavaScript безпосередньо з атрибута href посилання або іншого URI-обробника.

HTTP Parameter Pollution (HPP) – Сенс атаки полягає у надсиланні однакового HTTP-параметра кілька разів. Ця техніка експлуатує не слабкості браузерного парсера, а неузгодженість в обробці HTTP-запитів різними серверними технологіями, що призводить до обходу логіки валідації на рівні застосунку.

Таблиця 2.6 – Реакція на HPP атаки в залежності від технології

Технологія	Поведінка	Результат
ASP.NET, IIS	Конкатенація значень через кому	first, second
PHP, Apache	Використання останнього переданого значення	second
JSP, Tomcat	Використання першого переданого значення	first

Ця розбіжність створює можливість для обходу фільтра, а саме зловмисник може надіслати один параметр із безпечним значенням, а другий – зі шкідливим (`?param=safe&param=<script>...`), розраховуючи, що WAF перевірить перше значення, а серверний застосунок (наприклад, на PHP) виконає друге.

Екранування символу екранування. Якщо розробник реалізував власний фільтр, який екранує лише подвійні лапки (замінюючи " на \"), зловмисник може обійти його. Передавши шкідливе навантаження, що містить послідовність \", він може нейтралізувати екранування, оскільки фільтр перетворить \" на \\" що дозволить впровадити шкідливий код.

Успішний обхід фільтра часто вимагає комбінування кількох технік, адаптованих до конкретної цілі.

<iMg/src=t onerror=print`` <

Методи обходу можна класифікувати за тим, яку саме слабкість вони експлуатують. Зловмисники рідко використовують прості шкідливі навантаження, такі як <script>alert(1)</script>, оскільки вони легко виявляються навіть найпростішими фільтрами.

Ефективність методів обходу XSS фільтрів безпосередньо залежить від логічних прогалин у механізмах фільтрації та захисту загалом. Завдання зловмисника знайти такий формат шкідливого навантаження, який система захисту не розпізнає як загрозу, а браузер кінцевого користувача дозволить виконати його як довільний код.

Популярний підхід полягає у блокуванні відомих шкідливих патернів, таких як <script>, onerror, javascript:. Основний недолік, як зазначається у стандарті CWE-79, полягає в тому, що покладання виключно на пошук шкідливих даних є хибною стратегією. Практично неможливо передбачити всі можливі вектори атак та їхні обфусковані варіації. Зловмисники постійно вигадують нові способи кодування та представлення даних, що робить будь-який "чорний список" неповним за визначенням.

Якщо у зоні чистого HTML, критичні спеціальні < > символи повертаються в оригінальному вигляді, але помічено що атрибути, теги чи події піддаються видаленню на етапі обробки чи валідації, слід дослідити всі можливість використання аналогів. Сумарна кількість тегів та їх атрибутів є дуже великою, більше того постійно з'являються нові. Тому у випадку якщо веб додаток використовує методи чорного чи біло списку, ймовірність підібрати такий атрибут, тег чи подію, що пройде обробку чи валідацію – є достатньо високою.

HTML дозволяє створювати свої власні HTML теги. Для цього необхідно поміж символів < та > вставити виключно літерами довільний текстовий вміст та створити його закриваючий тег. У випадку, коли жоден тег не вдалось підібрати, доцільно створити власний тег застосувати подію onfocus та якірне посилання на цей елемент. Отже, у такому випадку корисне навантаження передане через URL адресу, може мати наступний вигляд:

/?s=<ternopil+id=city+onfocus=print()+tabindex=1>#city

Якщо було виявлено, що веб додаток використовує фільтр саме за чорним списком не дозволених елементів, слід дослідити перелік способів обходу такого фільтру. Якщо веб додаток блокує <script> елемент, мін може не заблокувати <sCrIpT> елемент.

Подвійна передача елементу. Ефективно якщо фільтр вилучає елементи тільки за одну ітерацію чи перший знайдений елемент.

<script><script>

<scrip<script>t>

<scripscriptt>

Згідно синтаксису мови JavaScript, дозволено використання косих лапок `` замість круглих дужок ().

onerror=print``

Символ пробілу може бути замінений на альтернативні символи відступу.

Таблиця 2.7 – Приклад альтернативних символів відступу

Опис	Символ
Слеш	/
URL-кодування слешу	%2F
Form Feed (FF)	%0C
Табуляція	%09
Новий рядок	%0A
Повернення каретки	%0D

Якщо у веб застосунку реалізовано слабкий алгоритм очищення на базі чітко визначеного вмісту, такий механізм безпеки легко обійти через маніпуляцію з додатковими не стандартними атрибутами.

<script t>

<script t="00000">

<script ~>

<script/t>

Обхід в середині атрибутів подій через використання HTML сутностей. HTML сутності в середині значень атрибутів декодуються на ходу, що дозволяє отримати необхідні символи, якщо їх не екрановані варіанти піддаються обробці.

Посилання

Посилання

Посилання

Посилання

2.3 Методи запобігання XSS вразливостям

Для запобігання XSS вразливостям, необхідно дотримуватися основного принципу, за яким стоїть нульова довіра до вхідних даних. Усі дані, що надходять до додатку, необхідно розглядати як неперевірені та потенційно небезпечні. Методи запобігання XSS вразливостей є широковідомі між розробниками веб додатків, проте розмір та складність самих веб додатків часто є різною. Надійний захист вимагає підходу у використані декількох методів одночасно, оскільки використання лише одного єдиного методу неминуче залишає слабкі місця, якими може скористатися зловмисник.

Валідація вхідних даних забезпечує що тільки дані відповідного формату, можуть бути прийняті та передані для подальших дій над ними. Процес валідації часто є першим етапом для запобігання XSS та інших вразливостей, які мають за основу маніпулювання вхідними даними. Сама по собі валідація не є достатнім захистом від XSS, оскільки залишає за собою можливості обходу, тому завжди має працювати в комплексі з іншими методами запобігання.

При валідації слід завжди надавати перевагу підходу на основі білих списків над чорними, оскільки чорний список є нескінченним переліком, що

постійно змінюється разом з розвитком HTML, CSS, JavaScript та браузерами загалом, постійно з'являються нові HTML елементи, атрибути, тощо. На противагу, білий список має скінченний чітко визначений перелік, що дозволяє строго обмежити набір символів та отримати необхідний формат вхідних даних.

Таблиця 2.8 – Порівняння білого та чорного списків

Білий список	Чорний список
a, b, c, d...y, z	‘, “, <, >, /, \, =
1, 2, 3, 4…9, 0	<script>, <[тест]>
-, +, /, *	document.cookie, on[текст], javascript:

Не лише дані, які були введені безпосередньо користувачем потрібно валідувати, а також інші що передаються разом з HTTP запитом. До такого переліку даних входять приховані поля, файли cookie, заголовки, параметри та сама URL адреса.

Таким чином, валідація обмежує формат даних які можуть бути передані користувачем, наприклад дозволити лише цифри, лише текст, заборонити використання спеціальних символів, обмежити розмір тексту кількістю символів, відповідність тексту до певного шаблону, тощо.

Кодування – це процес заміни потенційно небезпечних символів їхніми відповідними безпечними еквівалентами, які браузер буде інтерпретувати як дані, а не як виконуваний код. Кодування є найнадійнішим та найважливішим методом для запобігання XSS вразливостей. Навіть якщо зловмиснику вдалось обійти процес валідації, саме належне кодування захищає від того, щоб передане шкідливе навантаження поверталося без змін.

Процес кодування важливо виконувати залежно від контексту розміщення даних, які будуть відображатися на сторінці. В контексті HTML тіла між тегами, символи < та > необхідно кодувати як < та > через HTML сутності, щоб браузер не інтерпретував їх як нові HTML теги. В середині значень HTML

атрибутів, окрім < та >, необхідно кодувати лапки ", ' як " або ', щоб зловмисник не міг вийти за межі атрибута і додати нові.

У середині де розміщується JavaScript код, як от <script> елементи, щоб запобігти передчасному завершенню рядка або впровадженню нового коду, необхідно використовувати Unicode екранування. У випадку з JavaScript доцільним рішенням буде використання спеціалізованих бібліотек, в іншому випадку кожен символ має бути закодований через \xHH формат.

Для URL контексту, наприклад формування значення в параметрах, потрібно відповідне URL-кодування. Змінні які вставляються в URL посилання займають позицію параметра або фрагмента, та часто передаються веб серверу або використовуються на стороні користувача. До таких змін потрібно застосовувати кодування в форматі %HH, де H – значення в Hex форматі. Одними з популярних HTML тегів де використовується вставка URL посилання в відповідні їм атрибути href та src, є <a> та <img>.

Таблиця 2.9 – Кодування символів з чорного списку

Символ	HTML	URL	JavaScript
<	&lt;	%3C	\u003C
>	&gt;	%3E	\u003E
"	&quot;	%22	\u0022
'	&#39;	%27	\u0027
&	&amp;	%26	\u0026
/	&#47;	%2F	\u002F

Якщо є необхідність збереження вхідних даних сервером, вони піддаються виключно валідації оминаючи кодування, оскільки має бути збережена їх оригінальність. А отже, кодування застосовується безпосередньо перед виводом. Воно є останньою операцією перед тим, як дані вставляються у HTTP відповідь.

Приймачі (Sinks) – це місця, де розміщуються змінні, які передають або вставляють дані. Більшість приймачів є безпечними, оскільки інтерпретують дані як звичайний текст, тим самим не дозволяють його виконання як коду.

```
<body>
  <div id="first">
    <img src="x" onerror="print()">
  </div>
  <div id="second"><img src=x onerror=print()></div>
  <script>
            const first = document.getElementById('first');
            const second = document.getElementById('second');

            first.innerHTML = "<img src=x onerror=print()>";
            second.textContent = "<img src=x onerror=print()>";
  </script>
</body>
```

Рисунок 2.6 – Приклад небезпечного приймача

Web Application Firewall (WAF) – це інструмент безпеки, міжмережевий екран для веб застосунків, який аналізує HTTP трафік і може виявляти та блокувати поширені атаки, включаючи XSS, на основі сигнатур. WAF може бути ефективний як додатковий рівень захисту, що доповнює основні методи захисту, проте він не є основнии способом запобігання вразливостям.

Content Security Policy (CSP) – це вбудований функціонал у веб браузерах, що забезпечує захист від переліку веб атак, включаючи XSS, через контроль завантаження ресурсів та обмеження їх використання зі сторонніх джерел. Для прикладу обмеження завантаження скриптів, зображень, інтегрування одних сторінок в інші. CSP може запобігти виконанню шкідливих скриптів, навіть якщо XSS вразливість була успішно реалізована.

Content-Security-Policy – це HTTP заголовок який застосовують під час HTTP відповідей, з певними значеннями, які визначають політику обмежень. Політика складається з переліку параметрів та їх значень, декілька параметрів розділяються через крапку з комою.

Веб фреймворк – програмний фреймворк, який використовується під час створення веб додатку, головна мета якого полягає в тому щоб за рахунок готових шаблонів та автоматизації прискорити розробку додатку.

Сучасні веб фреймворки вважаються безпечними для їх використання під час розробки, оскільки дозволяють розробнику не заглиблюватись в певні аспекти безпеки, а застосовувати готові рішення. Для запобігання XSS, веб фреймворки можуть використовувати авто кодування даних та ряд готових шаблонів безпеки. Хоча сучасні веб застосунки, які використовують веб фреймворки набагато менш вразливі до XSS, розробник повинен розуміти, як саме той чи інший веб фреймворк протидіє їм. У випадку якщо розробник по своїм причинам не використовує кодування від фрейморку, строго рекомендується використовувати готові спеціалізовані бібліотеки кодування.

Не все залежить він впроваджених методів для запобігання вразливостям, оскільки багато веб застосунків використовують багато сторонніх залежностей. Саме в них не рідко можна зустріти певні вразливості, про які розробники залежностей не знають, а якщо знають то мають витратити час на дослідження та випуск нової версії. Регулярні оновлення бібліотек, модулів та різних залежностей загалом у парі з аудитом безпеки, це підніх який дозволяє виявити вразливості на ранніх етапах небезпеки.

3 ПРОЕКТУВАННЯ ТА РОЗРОБКА

3.1 Проектування тестового веб застосунку з XSS вразливостями

Щоб наблизити умови до реальних, тестовий стенд веб застосунку був розгорнутий за допомогою сервісу AWS Lightsail, на окремому віртуальному сервері, на базі ОС Ubuntu 24.04 LTS. Такий вибір зумовлений легкістю та швидкістю розгортання інфраструктури для невеликих проектів, доступністю необхідних ресурсів та відсутністю надмірних фінансових витрат. В свою чергу, ОС Ubuntu є одним з найпопулярніших Linux дистрибутивів, що підтримується більшістю веб серверів, а її LTS версія забезпечує довгострокові оновлення, що важливо для контрольованого тестового середовища.

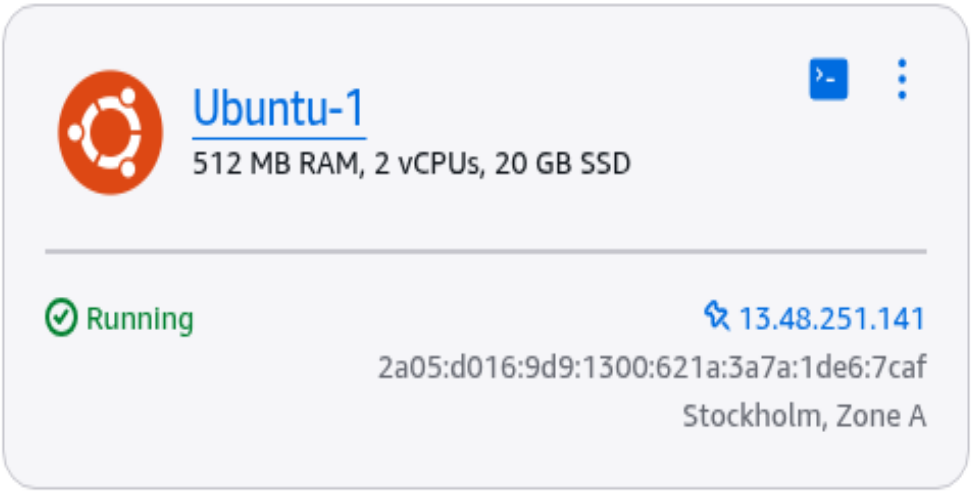

Рисунок 3.1 – Віртуальний сервер на AWS Lightsail

Оскільки задачею є наглядна демонстрація алгоритмів у роботі, підхід до вибору технологій був зосереджений на легкості та швидкості. Використання зайвих технологій було зведено до мінімуму.

Веб сервером було обрано вбудований сервер PHP, який призначений для ефективної розробки, тестування та демонстрації веб застосунків. Він потребує мінімальні системні вимоги та не вимагає додаткового програмного

забезпечення крім самого PHP інтерпретатора. Для його запуску достатньо виконати лише одну команду.

Мовою програмування для серверної частини стенду обрано PHP, так як вона широко застосовується для створення сучасних веб застосунків та не потребує встановлення додаткових бібліотек для базових веб функцій, оскільки нативно опрацьовує HTTP запити, сесії, cookie файли. Її легко вбудовувати в будь які типи контекстів та має одну з найповніших документацій серед інших.

Структура файлів зображено на рисунку 3.2.

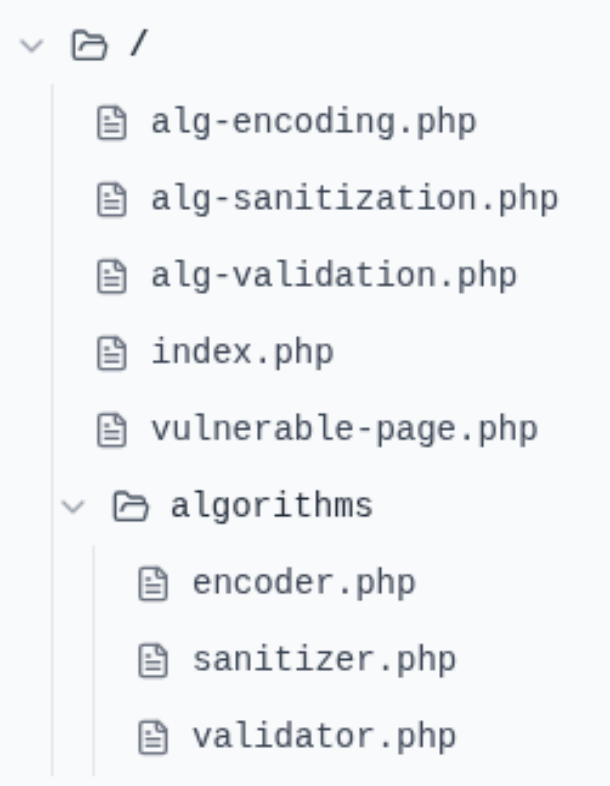

Рисунок 3.2 – Структура файлів тестового веб застосунку

Запити URI обробляються з поточного робочого каталогу /, в якому було запущено вбудований веб сервер PHP, Якщо в запиті URI не вказано файл, то з каталогу повертається index.php або index.html. У таблиці 3.1 наведено короткий опис до кожного окремого файлу.

Таблиця 3.1 – Файли тестового застосунку

Файл	Опис
index.php	Містить посилання на тестові сторінки
vulnerable-page.php	Тестова сторінка без задіяних алгоритмів
alg-validation.php	Тестова сторінка алгоритму валідації
alg-sanitization.php	Тестова сторінка алгоритму очищення
alg-encoding.php	Тестова сторінка алгоритму кодування
validator.php	Алгоритм валідації
sanitizer.php	Алгоритм очищення
encoder.php	Алгоритм кодування

На головній веб сторінці index.php розміщуються декілька посилань, які ведуть на окремі сторінки тестування, на яких вже реалізовані відповідні алгоритми протидії, а також на окрему тестову сторінку, яка є абсолютно вразливою до XSS вразливостей.

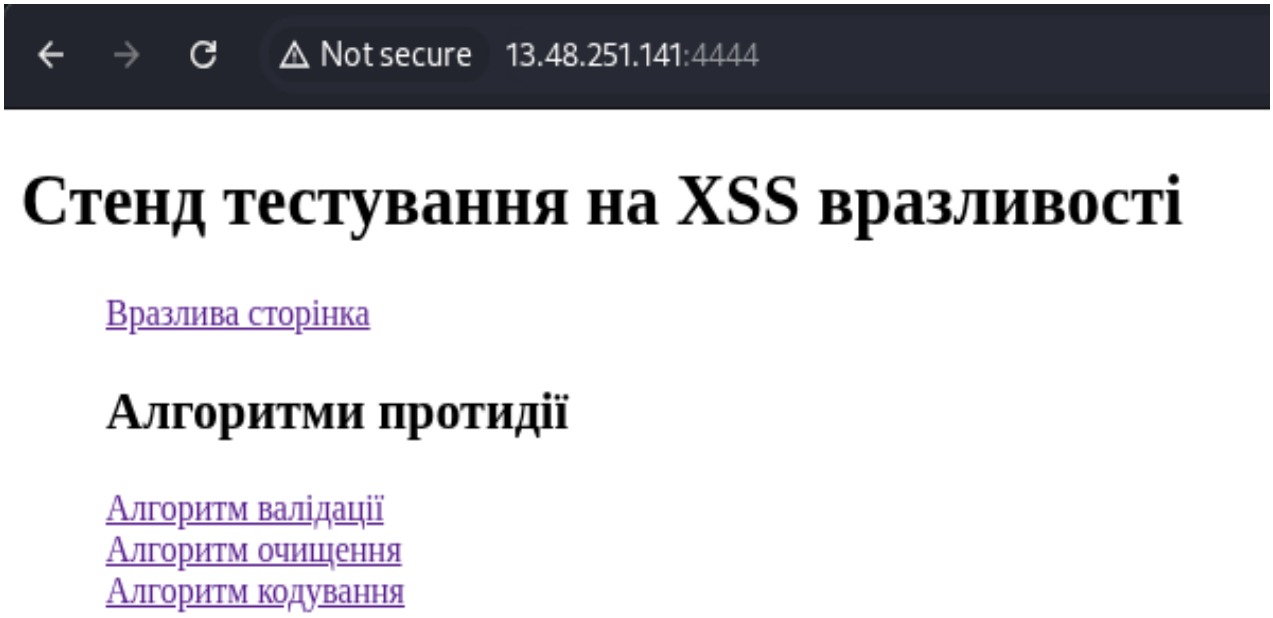

Рисунок 3.3 – Головна сторінка тестового застосунку

3.2 Експлуатація моделей загроз з XSS вразливостями

Вразлива сторінка для тестування (vulnerable-page.php), містить у собі веб форму з полем для введення тексту, в яке передається шкідливе навантаження. Після нажимання кнопки для відправки форми, браузер ініціює HTTP запит до vulnerable-page.php за методом GET, як результат до URL додається параметр input з відповідним значенням, яке було введено в текстове поле.

На цій сторінці та на всіх інших тестових сторінках, методом відправки веб форму було обрано метод GET, а не POST. Такий вибір зумовлений тим, що шкідливе навантаження відразу видно в адресному рядку, можливістю його в ньому редагувати та наглядному спостереженню.

```
<?php
    $input = $_GET['input'] ?? '';
?>

<form action="vulnerable-page.php" method="GET">
<label>Введення: <input type="text" name="input" value=""></label>
<button type="submit">Надіслати</button>
</form>
```

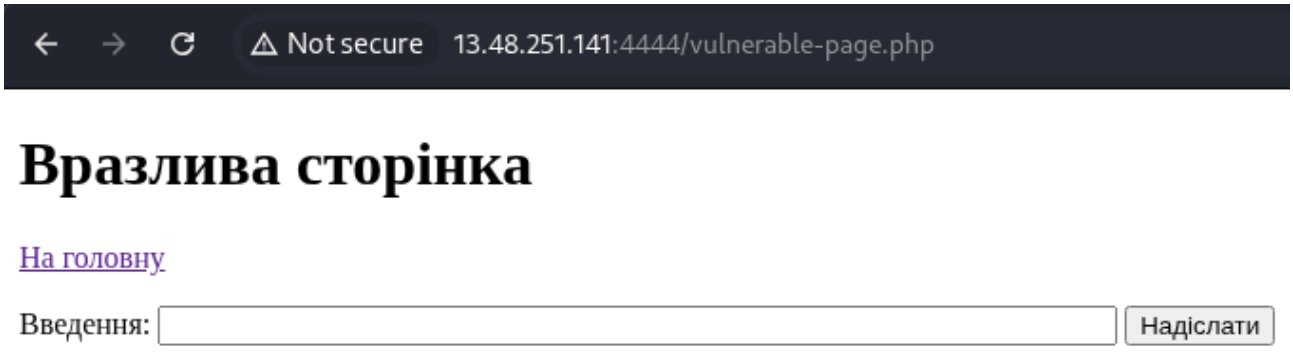

Рисунок 3.4 – Вразлива тестова сторінка

Якщо в URL адресі міститься не пустий параметр input, тоді формуються додаткові HTML елементи, які собою представляють чотири різних контекста, які мають вразливість до XSS.

```
<?php if (!empty($input)): ?>
...
<?php endif; ?>
```

Перший контекст це Тіло HTML, в якому шкідливе навантаження розміщується між тегами HTML, елементу p.

```
<h2>Контекст 1: Тіло HTML</h2>
<div>
<p><?php echo $input; ?></p>
</div>
```

Другий контекст це Атрибут HTML, в якому шкідливе навантаження розміщується на місці значення атрибута title. При наведені мишею на текст у елементі div, атрибут показує системне віконце з підказкою.

```
<h2>Контекст 2: Атрибут HTML</h2>
<div>
<div title="<?php echo $input; ?>">Наведіть на цей текст</div>
</div>
```

Третій контекст це JavaScript, в якому шкідливе навантаження розміщується як значення змінної userInput. Для наглядності, вигляд актуального значення змінної виводиться відразу на сторінку.

```
<h2>Контекст 3: JavaScript</h2>
<div>
<p id="js-output"></p>
<script>
var userInput = "<?php echo $input; ?>";
document.getElementById('js-output').textContent = "Вміст в JavaScript
змінній: " + userInput;
</script>
</div>
```

Четвертий контекст це URL параметр, в якому шкідливе навантаження розміщується на місці значення параметра data, який в свою чергу розміщується в атрибуті href. Даний контекст також відноситься й до контексту атрибута, тому найбільш актуальний при дії алгоритму кодування.

```
<h2>Контекст 4: URL Параметр</h2>
<div>
<a href="page.php?data=<?php echo $input; ?>">Посилання</a>
</div>
```

На рисунку 3.5 в текстове поле передано безпечні дані у вигляді звичайного тексту та наглядно зображено вміст кожного контексту під час перегляду DOM структури.

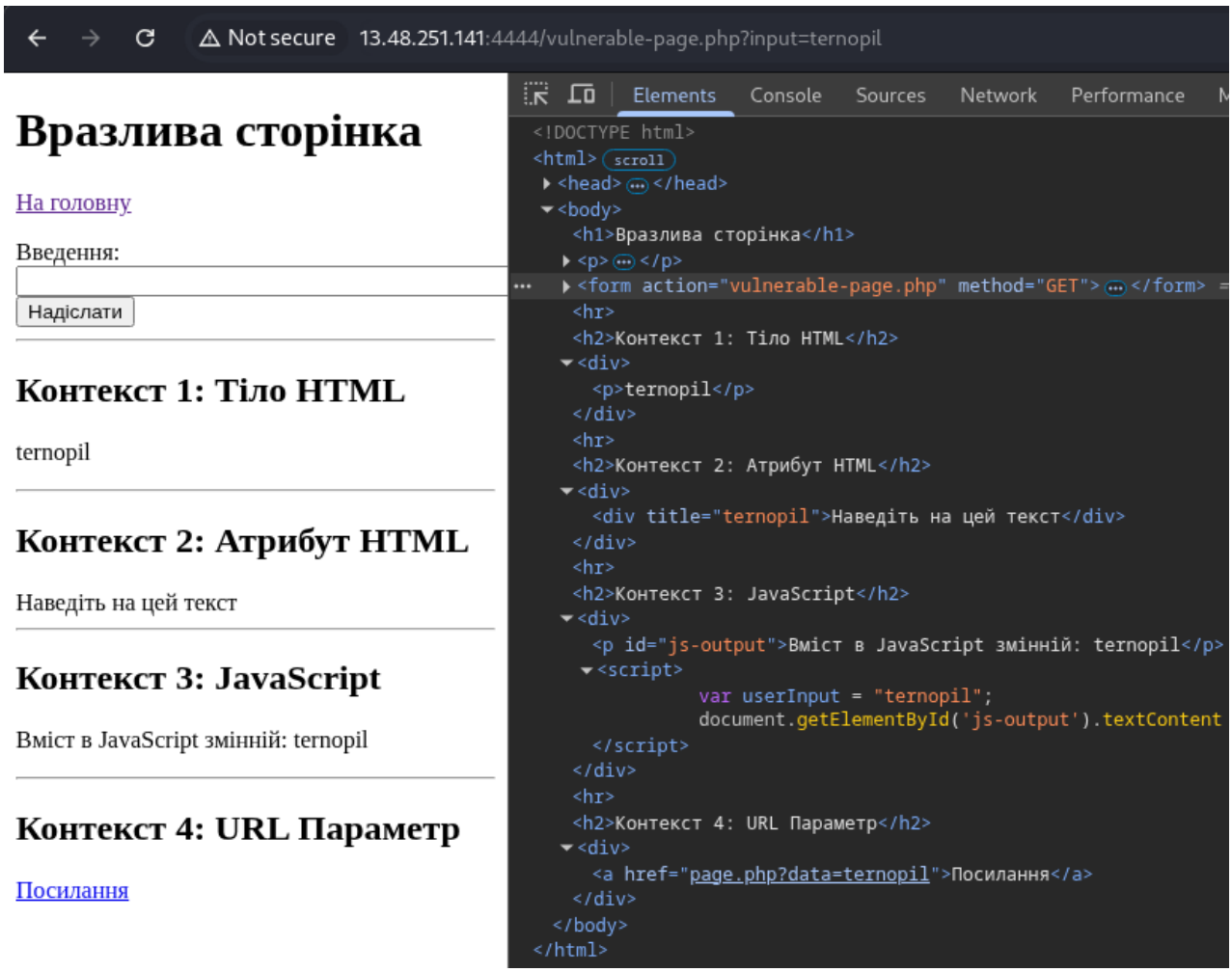

Рисунок 3.5 – Результат передачі безпечних даних

Після передачі шкідливого навантаження у вигляді "; alert(1111); // яке націлене на вихід з контексту JavaScript, відбувається виклик спливаючого вікна, що свідчить про успішно проексплуатовану XSS вразливість. Також, на рисунку 3.6 можна побачити, що через подвійну лапку відбувається вихід за межі атрибутів у контекстах 1 та 4, а решта шкідливого навантаження стають атрибутами елементів.

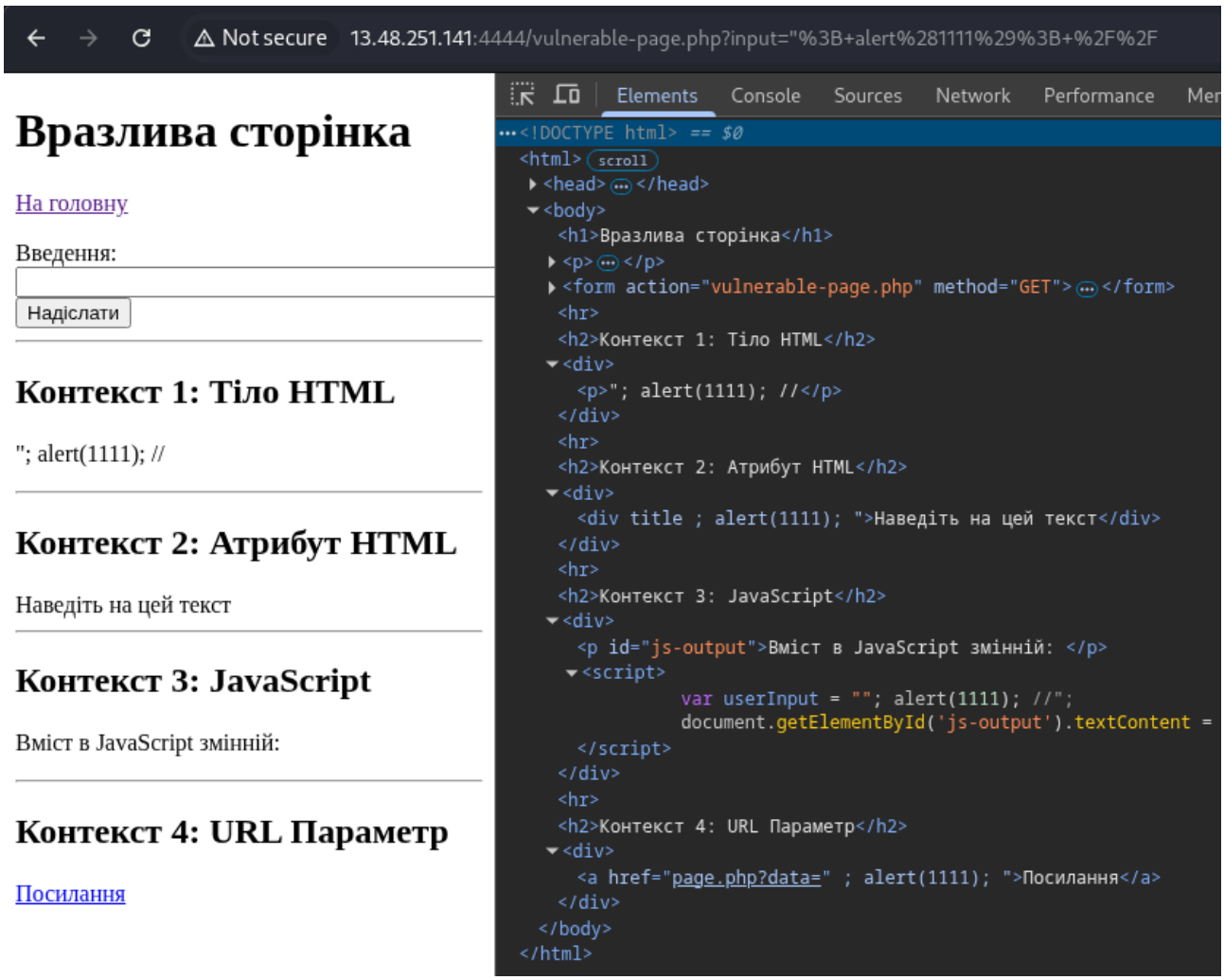

Рисунок 3.6 – Успішна XSS атака через вихід з JavaScript контексту

Для експуатації у тілі HTML, передано шкідливе навантаження у вигляді <img src=x onerror=alert(1111)>, яке створює елемент зображення з невірним шляхом у атрибуті src. Особливість атрибута onerror в тому, що це атрибут подій, який виконується при певних умовах, в даному випадку при виникненні помилки. При відповідній події, він дає вказівку вивести спливаюче вікно. Додатковою особливістю даного шкідливого навантаження в тому, що значення в атрибутах не обгорнуті в подвійні лапки, у такому випадку браузер самостійно їх додасть з метою виправити помилку. Через те що лапки не передаються, решта контекстів залишаються в своєму оригінальному випадку без зламаної структури як у випадку попереднього навантаження.

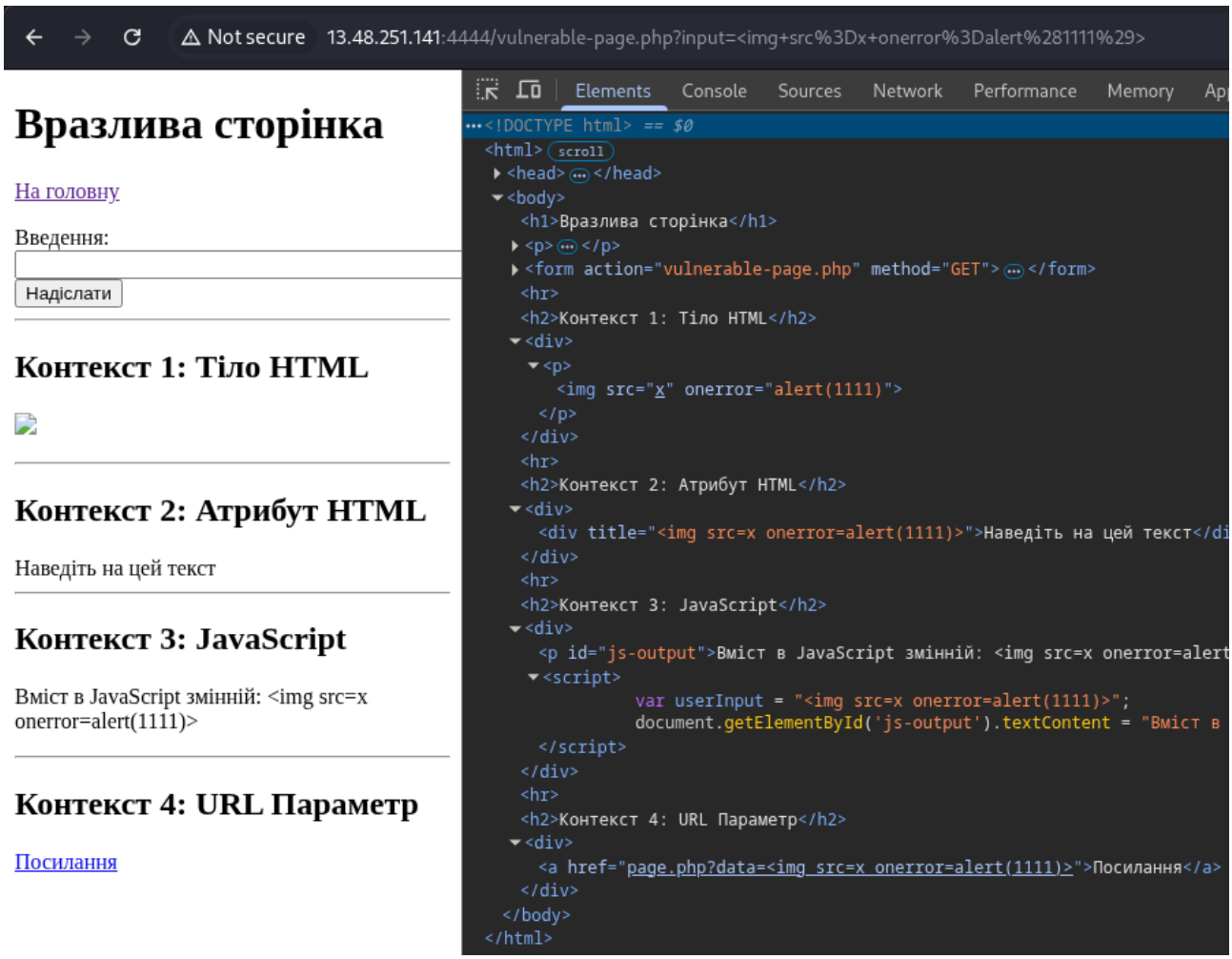

Рисунок 3.7 – Успішна XSS атака у контексті HTML тіла

3.3 Розробка функцій запобігання XSS вразливостей у веб застосунку

Алгоритм валідації ставить за мету перевірити отримані дані від користувача, щоб вони строго відповідали необхідному формату. Для прикладу, щоб в даних не містилися потенційно небезпечні символи для реалізації XSS атаки, та або, щоб вони відповідали певній конкретній довжині.

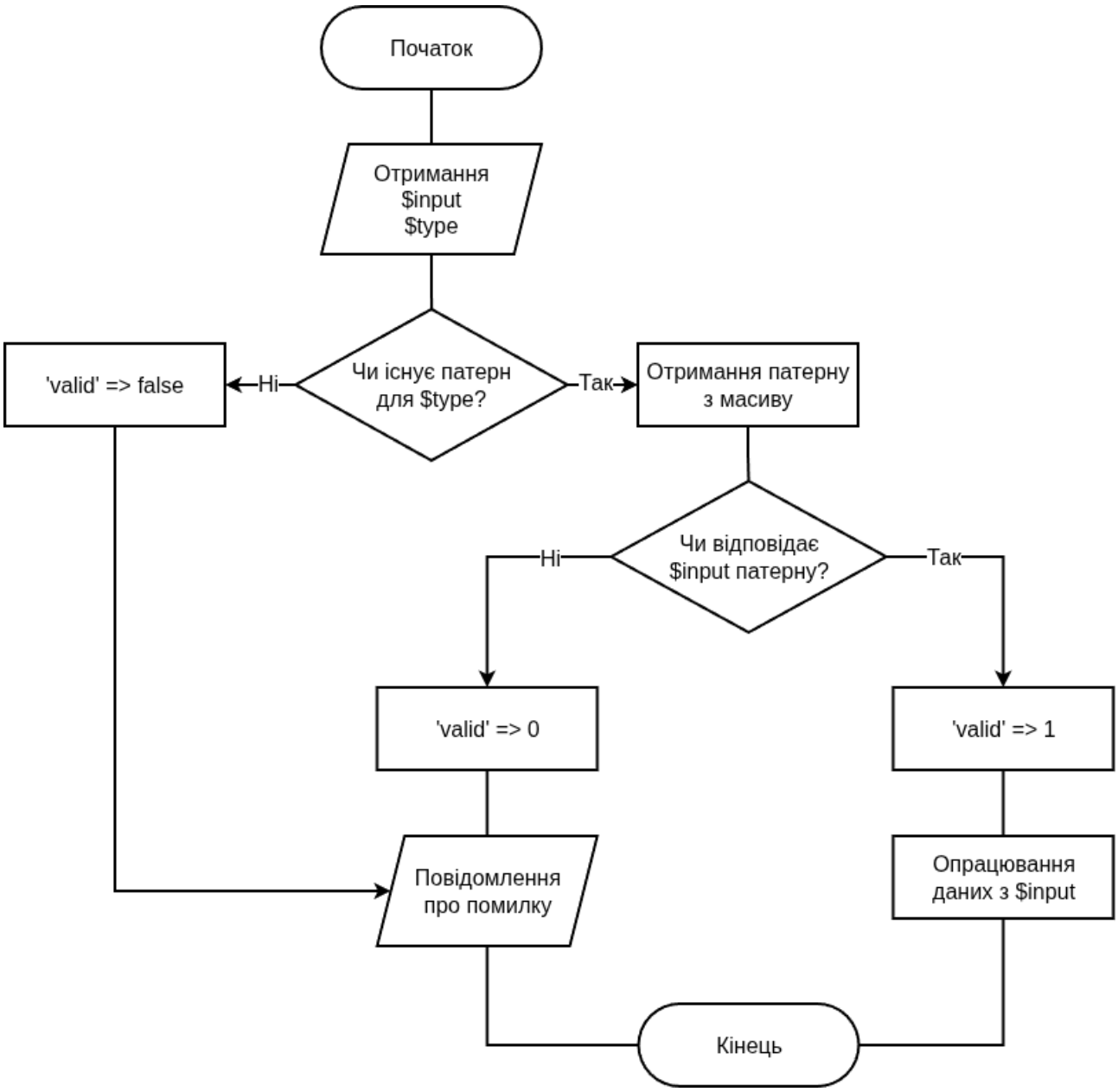

Рисунок 3.8 – Схема алгоритму протидії XSS вразливостям через валідацію даних

Реалізується клас InputValidator, який містить у собі публічну статичну функцію validate, яка в свою чергу очікує отримати $input де містяться передані дані, та $type де вказується конкретний тип, якому відповідає певний формат даних.

У середині функції розміщується масив $patterns, який містить ключі формату даних та значення, як регулярні вирази. Так, наприклад дані формату

name що відповідає нікнейму користувачу можуть містити символи літер, цифри та бути довжиною до 12 символів та не менше 2 символів.

```
<?php
class InputValidator {

    public static function validate($input, $type) {
        $patterns = [
            'name' => '/^[a-zA-Z0-9]{2,12}$/',
            'email' => '/^(?=.{1,30}$)[a-zA-Z0-9.]+@[a-zA-Z0-9.]+\.[a-zA-Z]{2,}$/',
            'text' => '/^[a-zA-Z0-9\s.,!?\-\'\"@#$%^&*()_+=\[\]{}|\\\:;<>\/~`]{1,30}$/'
        ];

        if (!isset($patterns[$type])) {
            return ['valid' => false];
        }

        return ['valid' => preg_match($patterns[$type], $input)];
    }
}
?>
```

Якщо у масиві немає такого патерну даних отриманого у $type тоді повертається false, в іншому випадку дані з $input перевіряються на відповідність $type щодо вказаного регулярного виразу та повертає 1 якщо дані не містять не дозволених символів та чи відповідають довжині, в іншому випадку повертається 0.

Розроблений алгоритм очищення даних, ставить за мету перевірку отриманих даних від користувача на відповідність патерну <[текст]>. У випадку

виявлення подібного патерну виконується очищення даних між небезпечними символами < та >. Якщо такої закономірності у даних не було знайдено, вони повертаються в оригінальному вигляді назад.

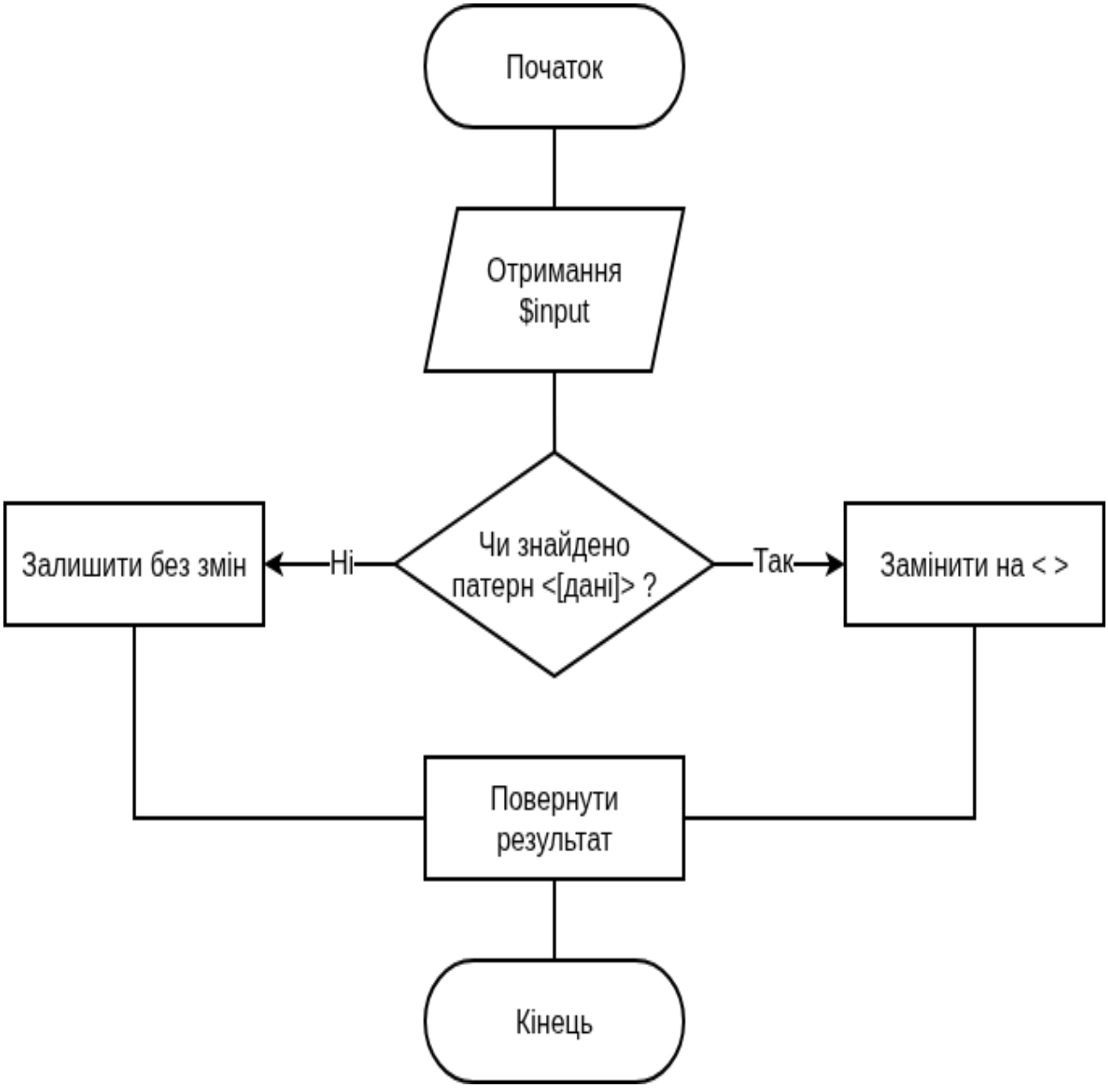

Рисунок 3.9 – Схема алгоритму протидії XSS вразливостям через очищення даних

Реалізується клас InputSanitization з публічною статичною функцією sanitize, яка очікує отримати $input з даними від користувача. Зі сторони

preg_replace над змінною $input виконується пошук регулярного виразу та у випадку співпадіння заміняє дані на < >. В кінці повертається результат.

```
<?php

class InputSanitization {

   public static function sanitize($input) {

      $sanitized = preg_replace('/<(\S[^>]*)>/', '< >', $input);

      return $sanitized;
   }
}
?>
```

Розроблений алгоритм екранування даних, ставить за мету перевірку вхідних даних на наявність небезпечних символів та якщо такі є тоді закодувати. Особливістю даного алгоритму є його контекстно залежність, таким чином, якщо дані будуть розміщуватися в тілі HTML вони будуть закодовані в HTML сутності, якщо в URL адресі вони будуть закодовані в %HEX, якщо в JavaScript тоді відповідним чином екрановані. Також у випадку розміщення даних в контексті атрибута HTML елементу, додатково буде перевірено наявність небезпечних URI протоколів, які будуть вилучені у випадку їх наявності. Більше дельніший принцип алгоритму зображений на схемі рисунку 3.10.

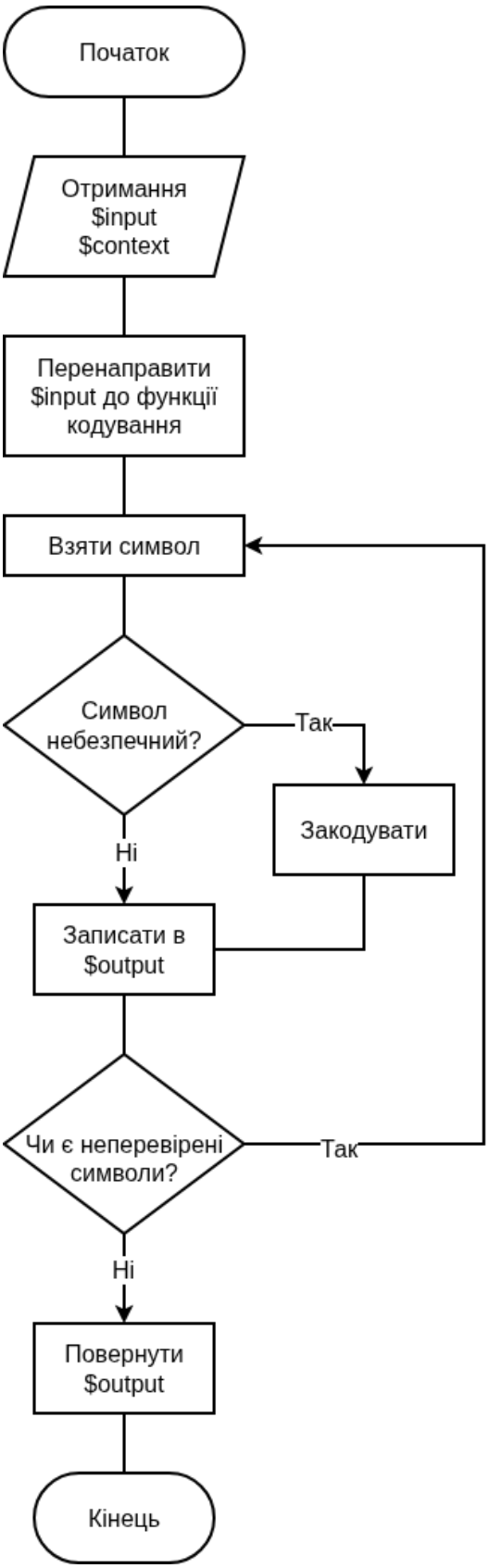

Рисунок 3.10 – Схема алгоритму протидії XSS вразливостям через екранування даних

Реалізовано клас ContextEncoder, який містить функції для кодування. Усі вони наглядно зображені у таблиці 3.2.

Таблиця 3.2 – Функції класу ContextEncoder

Функція	Опис
encodeHTML()	Кодування в HTML сутності
encodeHTMLAttribute()	Кодування в HTML сутності та вилучення протоколів
encodeJavaScript()	Екранування
encodeURL()	Кодування в %HEX
encodeByContext()	Перенаправлення на функцію

Функція encodeHTML() отримує вхідні дані через змінну $input. Створюється основна змінна $output яка буде містити фінальні дані для виводу. Обчислюється довжина вхідних даних, створюється цикл for та виконується до тих пір поки кількість ітерацій менше за довжину вхідних даних.

В циклі створюється змінна $char в якій протягом однієї ітерації зберігається перший вибраний символ з вхідних даних. Над символом за допомогою switch() виконується порівняння над всіма case, які містять небезпечні спеціальні символи, що можуть бути використані під час маніпулювання точками входу з ціллю виконання XSS вразливостей. До такого переліку символів входять <, > &, одинарні та подвійні лапки. Якщо під час ітерації виник зміг між актуальним символом що порівнюється та одним із переліку заборонених, тоді виконується HTML кодування, а саме в змінну $output записується еквівалентне значення символу у вигляді HTML сутності. Оскільки цикл прив'язаний до довжини вхідних даних, вище описані операції відбуваються над усіма символами, один символ одна ітерація. При завершені циклу, відбувається повернення змінною $output з фінальними кодованими

даними, які можна безпечно без ризиків щодо XSS вразливостей у контексті HTML тіла, вставляти в веб сторінку.

```
public static function encodeHTML($input) {
    $output = '';
    $length = strlen($input);
    for ($i = 0; $i < $length; $i++) {
        $char = $input[$i];
        switch ($char) {
            case '<':
                $output .= '&lt;';
                break;
            case '>':
                $output .= '&gt;';
                break;
            case '&':
                $output .= '&amp;';
                break;
            case '"':
                $output .= '&quot;';
                break;
            case "'":
                $output .= '&#x27;';
                break;
            default:
                $output .= $char;
        }
    }
    return $output;
}
```

Для кодування вхідних даних, які будуть розташовуватися в контексті HTML атрибута любого елементу, застосовується функцій encodeHTMLAttribute(). Отримавши дані через змінну $input, першим кроком виконує self::encodeHTML($input) та результат записує у $output.

На основі попереднього опису функції encodeHTML(), в змінні $output містяться закодовані вхідні дані у HTML сутності. Та попри закодовані дані, через розташування на виході у атрибуті HTML елемента, необхідно передбачити застосування таких небезпечних URI протоколів, як от javascript: що дозволяє виконувати довільний JavaScript код при нажимані на відповідний елемент. Тому, функція з масиву $dangerous який містить у собі перелік протоколів ['javascript:', 'data:', 'vbscript:'], бере кожне значення масиву та за допомогою str_ireplace() шукає конкретний небезпечний протокол у вже закодованих вхідних даних у $output. Як тільки один з протоколів був знайдений він видаляється та фінальний вид даних перезаписує змінну $output та повертає як кінечний результат функції

```
public static function encodeHTMLAttribute($input) {
    $output = self::encodeHTML($input);

    $dangerous = ['javascript:', 'data:', 'vbscript:'];
    foreach ($dangerous as $protocol) {
        $output = str_ireplace($protocol, '', $output);
    }

    return $output;
}
```

Для екранування вхідних даних, які будуть розташовуватися в контексті JavaScript коду, застосовується відповідна функція encodeJavaScript(). Вона

аналогічним чином створює такі змінні, як $outup, $length, проходиться по кожному символу через цикл for. Зважаючи на контекст розташування, в перелік небезпечних символів окрім <, >, ^, одинарних та подвійний лапок, також входять \\, \n (новий рядок), \r (повернення каретки), \t (табуляція).

```
public static function encodeJavaScript($input) {
    $output = "";
    $length = strlen($input);

    for ($i = 0; $i < $length; $i++) {
        $char = $input[$i];
        switch ($char) {
            case "":
                $output .= '\\"';
                break;
             [...]
            case '&':
                $output .= '\\x26';
                break;
            default:
                $output .= $char;
        }
    }

    $output .= "";
    return $output;
}
```

Для екранування даних, які будуть розміщуватися в URL адресі, як от параметри чи їх значення, використовується функція encodeURL(). Виконавши початкове створення необхідних змінних та відкриття циклу, ця функція перевіряє кожен окремий символ за дещо іншим принципом.

Під час кожної ітерації, символ проходить через ctype_alnum та перевіряється чи він є одним зі звичайних буквено-цифрових символів або одним з спеціальних символів з наступного переліку -, _, ., ~. Якщо так, актуальний символ розміщується в змінній $output без змін, якщо ні тоді кодується в %HEX формат та передається до всіх інших в $output.

```
public static function encodeURL($input) {
    $output = '';
    $length = strlen($input);

    for ($i = 0; $i < $length; $i++) {
        $char = $input[$i];

        if (ctype_alnum($char) || $char === '-' || $char === '_' || $char === '.' ||
$char === '~') {
            $output .= $char;
        } else {
            $output .= '%' . strtoupper(sprintf('%02X', ord($char)));
        }
    }

    return $output;
}
```

ВИСНОВКИ

Оглянуто популярні веб вразливості у веб застосунках, причини їх виникнення та вплив на безпеку веб застосунку та кінцевих користувачів. Проаналізовано місце XSS вразливостей серед інших, взаємозв'язки, корінні проблеми. Визначено тенденцію та актуальність XSS вразливостей на сьогоднішній день.

Проаналізовано види XSS вразливостей, ризики пов'язані з ними та наслідки. Досліджено методи їх виявлення, маніпулювання вхідними даними та методи обходу впроваджених механізмів захисту, обхід фільтрів.

Спроектовано та створено тестовий веб застосунок вразливий до XSS. Над застосунком здійснено експлуатацію моделей загроз, задіяно різні форми доставки шкідливих навантажень. Проекслуатовано всі основні контексти розміщення даних та проаналізовано наслідки на DOM структуру. Задіяно різні методи щодо створення особливих шкідливих навантажень та продемонстровано їх незвичне застосування під час атак.

Розроблено алгоритми запобігання XSS вразливостей у веб застосунках, а саме алгоритм валідації, алгоритм очищення, алгоритм кодування. Реалізовано відповідні функції обробки вхідних даних. Впроваджено розроблені алгоритми у тестовому веб застосунку, здійснено комплексні атаки на кожен з них, здійснено оцінку їх ефективності. Дія алгоритмів у комплексі демонструє ефективний багаторівневий механізм протидії XSS.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Hannousse A. Twenty-two years since revealing cross-site scripting attacks: A systematic mapping and a comprehensive survey / A. Hannousse, S. Yahiouche, M.Ch. Nait-Hamoud // Computer Science Review. – 2024. – Vol. 52. – P. 1-22.
2. Sharma S. Review on Detection and Prevention Techniques of Scripting Attacks: Gaps, Challenges and Suggestions / S. Sharma, N.S. Yadav // Recent Patents on Engineering. – 2025. – Vol. 19, No. 6. – P. 1–15.
3. Malviya V.K. Development of web browser prototype with embedded classification capability for mitigating Cross-Site Scripting attacks / V.K. Malviya, S. Rai, A. Gupta // Applied Soft Computing. – 2021. – Vol. 102. – P. 1–12.
4. Gandikota P.S.S.K Web Application Security through Comprehensive Vulnerability Assessment / P.S.S.K. Gandikota, D. Valluri, S.B. Mundru, G.K. Yanala, S. Sushaini // Procedia Computer Science. – 2023. – Vol. 230. – P. 168–182.
5. Wen S.-F. A quantitative security evaluation and analysis model for web applications based on OWASP ASVS / S.-F. Wen, B. Katt // Computers & Security. – 2023. – Vol. 135. – P. 1–15.
6. Calzavara S. Measuring Web Session Security at Scale / S. Calzavara, H. Jonker, B. Krumnow, A. Rabitti // Computers & Security. – 2021. – Vol. 111. – P. 1–12.
7. Singh R.P. Chandavarkar B.R. Dynamic Content Security Policy Generation at Client-Side to Mitigate XSS Attacks / R.P. Singh, B.R. Chandavarkar // Proc. of the 2024 International Conference on Computing Communication and Networking Technologies (ICCCNT–2024). – 2024. – P. 1–7.
8. Guan H. A Crawler-Based Vulnerability Detection Method for Cross-Site Scripting Attacks / H. Guan, D. Li, H. Li, M. Zhao // Proc. of the 2022 International Conference on Software Quality, Reliability, and Security Companion (QRS-C 2022). – 2022. – P. 651–655.

9. Alsaedi A. Effective and scalable black-box fuzzing approach for modern web applications / A. Alsaedi, A. Alhuzali, O. Bamasag // Journal of King Saud University – Computer and Information Sciences. – 2022. – Vol. 34, No.10. – P. 10068–10078.
10. Pan H. Few-shot graph classification on cross-site scripting attacks detection / H. Pan, Y. Fang, W. Guo, Y. Xu, C. Wang // Computers & Security. – 2024. – Vol. 140. – P. 1–12.
11. Thajeel I.K. Machine and Deep Learning-based XSS Detection Approaches: A Systematic Literature Review / I.K. Thajeel, K. Samsudin, S.J. Hashim, F. Hashim // Journal of King Saud University – Computer and Information Sciences. – 2023. – Vol. 35, No.7. – P. 1–20.
12. Maurel H. Statically identifying XSS using deep learning / H. Maurel, S. Vidal, T. Rezk // Science of Computer Programming. – 2022. – Vol. 219. – P. 1–15.
13. Wang Q. Black-box adversarial attacks on XSS attack detection model / Q. Wang, H. Yang, G. Wu, K.-K.R. Choo, Z. Zhang, G. Miao, Y. Ren // Computers & Security. – 2022. – Vol. 113. – P. 1–10.
14. Chen L. XSS adversarial example attacks based on deep reinforcement learning / L. Chen, C. Tang, J. He, H. Zhao, X. Lan, T. Li // Computers & Security. – 2022. – Vol. 120. – P. 1–10.
15. Rodríguez G.E. Cross-site scripting (XSS) attacks and mitigation: A survey / G.E. Rodríguez, J.G. Torres, P. Flores, D.E. Benavides // Computer Networks. – 2020. – Vol. 166. – P. 1–25.
16. Bermejo Higuera J. R. Combinatorial Method with Static Analysis for Source Code Security in Web Applications / Bermejo Higuera J. R., Bermejo Higuera J., Sicilia Montalvo J. A., Sureda Riera T., Argyros Ch. I., Magreñán Á. A. // CMES. – 2021. – Vol. 129, No. 2. – P. 541–565.
17. Krishna S. Web Security in the Digital Age / Krishna S., Natarajan R., Flammini F., Alfurhood B. S., Janhavi V., Gupta Sh. K. // International Journal on Semantic Web and Information Systems. – 2025. – Vol. 21, No. 1.

18. Siahaan C. N Study of Cross-Site Request Forgery on Web-Based Application / Siahaan C. N., Rufisanto M., Nolasco R., Achmad S., Siahaan C. R. P. // Procedia Computer Science. – 2023. – Vol. 227. – P. 92–100.
19. Abazi B. Practical analysis on the algorithm of the Cross-Site Scripting Attacks / Abazi B., Hajrizi E. // IWSSIP. – 2022. – P. 1–4.
20. Chen H.-C. Detection and Prevention of Cross-site Scripting Attack with Combined Approaches / Chen H.-C., Nshimiyimana A., Damarjati C., Chang P.-H. // ICEIC. – 2021. – P. 1–4.
21. Fadlalla F. F. Input validation vulnerabilities in web applications: Systematic review, classification, and analysis of the current state-of-the-art 1 / Fadlalla F. F., Elshoush H. T. // IEEE Access. – 2023. – Vol. 11.
22. Khalaf O. I Web Attack Detection Using the Input Validation Method: DPDA Theory / Khalaf O. I., Sokiyna M., Alotaibi Y., Alsufyani A., Alghamdi S. // Computers, Materials & Continua. – 2021. – Vol. 68, No. 3.
23. Fadlil A. Mitigation from SQL Injection Attacks on Web Server using Open Web Application Security Project Framework / Fadlil A., Riadi I., Mu'Min M. A. // International Journal of Engineering. – 2024. – Vol. 37, No. 4. – P. 635–645.
24. Lakshmi B. S. A Proactive Approach for Detecting SQL and XSS Injection Attacks / Lakshmi B. S., Kovvuri D., Bolisetti H. N. V., Chikkala D. S., Karri S., Yadlapalli G. // ICAAIC. – 2024. – P. 1415–1420.
25. Qingyang W. A reflective XSS vulnerability detection method based on fuzzing test / Qingyang W., Yuqiao N., Zhen G., Mingming Y., Shihao X., Yang C. // 2022 International Conference on Algorithms, Data Mining, and Information Technology (ADMIT). – 2022. – P. 25–31.
26. Rajagopal S. Guarding Against Injection Attacks: A Filtering Solution for Web Application Security // ICCC. – 2025. – P. 1–6.
27. Schuckert F., Langweg H., Katt B. Systematic Generation of XSS and SQLi Vulnerabilities in PHP as Test Cases for Static Code Analysis / Schuckert F., Langweg H., Katt B. // ICSTW. – 2022. – P. 261–268.

28. Wang H. Analysis of Modern Web Penetration Techniques and Proactive Defense Mechanisms / Wang H., Meng B., Zhang X., Wang D., Xia H. // ISCTIS. – 2025. – P. 712–716.
29. Riskhan B. Major Vulnerabilities of Web Application in Real World Scenarios and Their Prevention / Riskhan B., Sheikh M. A. U., Hossain M. S., Hussain K., Zainol Z., Jhanjh N. Z. // ICoICC. – 2025. – P. 1–6.
30. Goto H. Design and Implementation of Web Application Penetration Testing with Cross-site Scripting / Goto H., Selvarajah V. // ICMNWC. – 2022. – P. 1–5.
31. Kollepalli R. P. K An Experimental Study on Detecting and Mitigating Vulnerabilities in Web Applications // International Journal of Safety & Security Engineering. – 2024. – Vol. 14, No. 2.
32. Schoenmakers K. The security mindset: characteristics, development, and consequences / Schoenmakers K., Greene D., Stutterheim S., Lin H., Palmer M. J. // Journal of Cybersecurity. – 2023. – Vol. 9, No. 1.

Додаток А.

Копії публікацій

ЗАХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ПОЛІТЕХНІКА»
ГАЛИЦЬКИЙ ФАХОВИЙ КОЛЕДЖ ІМ. В'ЯЧЕСЛАВА ЧОРНОВОЛА

КІБЕРБЕЗПЕКА ТА КОМП'ЮТЕРНО-ІНТЕГРОВАНІ ТЕХНОЛОГІЇ (КБКІТ – 2024)

науково-практична конференція
молодих вчених, аспірантів та студентів

26–28 серпня 2024
Тернопіль

ЗМІСТ

СИСТЕМИ ТА ТЕХНОЛОГІЇ КІБЕРБЕЗПЕКИ

УДК 004.056.5

Дмитро КАРПЕЦ

Західноукраїнський національний університет

НАСЛІДКИ CROSS SITE SCRIPTING АТАК У ВЕБ ДОДАТКАХ

Вступ. Згідно зі статистикою у звіті Incident Response Report 2024 від Palo Alto Network Unit 42, веб додатки є постійною ціллю атак та набувають зростання. Частка розслідувань інцидентів, саме атак на веб додатки займала приблизно 12.7% у 2021, а в останній час досягла 21.2%. Це вказує на те, що компрометація веб додатків залишається значною загрозою для компаній, а використання вразливостей зазвичай пов'язане з отриманням несанкціонованого доступу [1].

Однією з поширених вразливостей безпеки веб додатків є можливість реалізації Cross-Site Scripting атак, в результаті яких зловмисник може отримати несанкціонований доступ через викрадання кукі та сесійних токенів користувачів. Не допущення можливості реалізації таких атак є надзвичайно важливою задачею та досягається під час процесів розробки веб додатків.

Мета: дослідження наслідків Cross-Site Scripting атак у веб додатках**.**

1. Нехтування процесом перевірки безпеки під час життєвого циклу програмного забезпечення

Сьогодні під час розробки програмного забезпечення часто не приділяється увага перевірці на наявність вразливостей та інших помилок, а згадують про це тільки на стадії розгортання. Такий підхід є помилковим та часто призводить до майбутніх надлишкових фінансових витрат [2].

Для запобігання подібного, одним із найефективніших підходів є покращення процесу життєвого циклу програмного забезпечення (ЖЦПЗ), а саме впровадження перевірок безпеки на кожному з його етапів. ЖЦПЗ починається з визначення ПЗ та закінчується його підтримкою після розгортання [3].

На рисунку 1 зображено в загальному вигляді процес ЖЦПЗ та кореляція зростання економічних втрат для виправлення вразливостей відносно проходження етапів розробки ПЗ

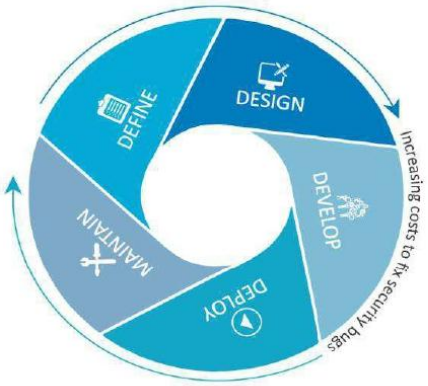

Рисунок 1 - Зростання ціни виправлення вразливостей безпеки відносно етапів життєвого циклу програмного забезпечення

2. Наслідки успішних реалізацій Cross Site Scripting атак у веб додатках

У разі успішного виконання Cross Site Scripting атаки, зловмисник може спричинити різного рівня наслідки. Розгляньмо їх в таблиці 1.

Таблиця 1 - Наслідки Cross Site Scripting атак у веб додатках

Дії	Наслідки
Викрадення кукі файлів	Доступ до особистої інформації користувача. Від інформації про взаємодію із сайтом, такої як мова, історія про відвідувані сторінки, адреси, імена - до електронних адрес, логінів, паролів.
Викрадення сесійного токена	У разі якщо сесійний токен зберігається у кукі файлах, його захоплення призводить до повної компрометації облікового запису користувача.
Перенаправлення користувача на іншу сторінку чи сайт	Залежно від плану зловмисника, наслідки можуть бути дуже різними, від фішингу до зараження шкідливим ПЗ
Модифікація відображеного контенту	Ціль такої модифікації залежить від призначення сайту на якому сидить користувач. Одна із популярних це зниження позитивного відношення до компанії, сервісу, тощо.
Встановлення шкідливого ПЗ	Залежно від типу шкідливого ПЗ, наслідки можуть бути менш чи більш значними, проте сам факт можливості його встановлення не піддається оцінці серйозності проблеми

Залежно від дій зазначених в таблиці 1, кінцеві наслідки деяких з них можуть бути від не серйозних до надзвичайно серйозних, це залежить від типу, навичок та мети зловмисника. В той час коли скрипт виконає шкідливі дії на стороні жертви, фаховий спеціаліст може здійснити компрометацію облікового запису жертви.

Висновок. Досліджено на порівняно наслідки Cross Site Scripting атак у веб додатках. Даний тип атак може спричинити серйозні наслідки як зі сторони користувача веб додатка, так і зі сторони його власника. Головним фактором для їх запобігання є процеси перевірки безпеки на кожному етапі розробки веб додатків.

Популярність веб додатків та взаємодія між ними є невід'ємною частиною сучасної людини. Це значить що особисті дані користувачів мають бути конфіденційними та захищеними від компрометації.

Перелік використаних джерел.

1. Unit 42 Palo Alto Networks - Incident Response Report 2024. [Електронний ресурс]. - Режим доступу: https://unit42.paloaltonetworks.com/unit42-incident-response-report-2024-threat-guide/

2. OWASP - Cross Site Scripting (XSS). [Електронний ресурс]. - Режим доступу: https://owasp.org/www-community/attacks/xss/

4. CGISecurity - The Cross-Site Scripting (XSS) FAQ. [Електронний ресурс]. - Режим доступу: https://www.cgisecurity.com/xss-faq.html

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
ЗАХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
UNIVERSITY OF THE NATIONAL EDUCATION COMMISSION, POLAND
TECHNICAL UNIVERSITY IN PRAGUE, CZECH REPUBLIC
Наукова школа "Кібербезпека"
Навчально-науковий інститут Кібербезпеки та захисту інформації ДУІКТ
Кафедра кібербезпеки ЗУНУ
ГО «АСОЦІАЦІЯ СПЕЦІАЛІСТІВ КІБЕРБЕЗПЕКИ»
ГО «АВТОМАТИЗАЦІЯ І КІБЕРБЕЗПЕКА»

ITSec-2025

Безпека інформаційних технологій

МАТЕРІАЛИ

XIV Міжнародної науково-технічної конференції

22-24 травня 2025
м. Тернопіль (Україна)

УДК [003.26+004+519.816]:004.056:65(063)

ITSec: Безпека інформаційних технологій: матеріали XIV Міжнар. наук.-техн. конф., м. Тернопіль, 22-24 трав. 2025 р. Тернопіль-Київ: ЗУНУ-ДУІКТ, 2025. 243с.

Збірник містить тексти наукових матеріалів доповідей та тез учасників XIV міжнародної науково-технічної конференції «ITSec: Безпека інформаційних технологій». Основною метою конференції є ознайомлення з сучасними досягненнями та висвітлення результатів наукових досліджень з усіх аспектів кібербезпеки та захисту інформації.

Призначено вченим, інженерам, аспірантам наукових спеціальностей 05.13.21 – Системи захисту інформації, 21.05.01 – Інформаційна безпека держави, здобувачам вищої освіти спеціальності 125 – Кібербезпека та захист інформації, а також всім зацікавленим.

Алгоритм протидії Cross-Site Scripting атакам на веб додатки

УДК 621.395.7 (043.2)

Дмитро Карпец
Західноукраїнський національний університет,
d.karpets@st.wunu.edu.ua

Cross-Site Scripting (XSS) вразливості, які завжди були широко розповсюджені у веб додатках, залишаються бути такими ж актуальними на сьогоднішній день. Цьому свідчать рейтингові списки популярних веб вразливостей та вразливостей безпеки загалом. Так, MITRE зі своїм щорічним рейтингом "CWE Top 25" віддає XSS атакам першу сходинку [1], а Open Worldwide Application Security Project (OWASP) з рейтинговим списком "OWASP Top Ten", відносячи XSS вразливості до категорії ін'єкцій, розміщують її на третій позиції [2].

Метою роботи є покращення алгоритму протидії Cross-Site Scripting атакам на веб додатки.

Реалізація XSS атак можлива, коли користувач передає довільні дані, а веб додаток не виконуючи належної валідації та обробки цих даних, безпосередньо виводить їх на своїх веб сторінках [3] (Рис.1).

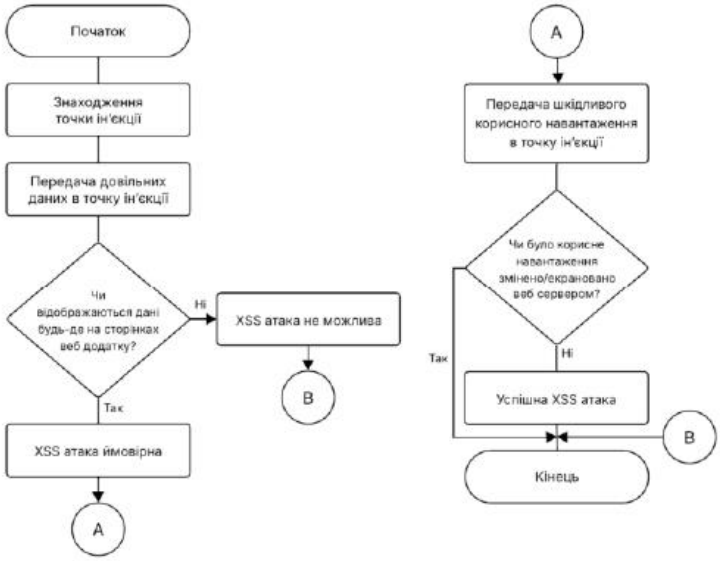

Рис.1. Схема алгоритму XSS атаки

В результаті досліджень запропоновано алгоритм протидії XSS атакам (Рис.2).

У зв'язку з тим, що популярність XSS вразливостей не спадає, створення нових та покращення існуючих алгоритмів протидії XSS атакам є актуальним завданням. На сьогоднішній день, на фоні масштабного прогресу пов'язаного з штучним інтелектом (ШІ), пропонується залучати ШІ до розроблення автоматизованих засобів перевірки коду на наявність XSS вразливостей.

Рис.2. Схема алгоритму протидії XSS атаці

Було розглянуто актуальність XSS вразливостей у веб додатках, алгоритм атаки, алгоритм протидії та запропоновано його покращення.

1. CWE Top 25 Most Dangerous Software Weaknesses 2024. URL: https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html (дата звернення: 01.04.2025).
2. OWASP Top Ten Web Application Security Risks. URL: https://owasp.org/www-project-top-ten/ (дата звернення: 01.04.2025).
3. CWE-79: Improper Neutralization of Input During Web Page Generation. URL: https://cwe.mitre.org/data/definitions/79.html (дата звернення: 01.04.2025).

Протидії кіберзагрозам на основі штучного інтелекту

УДК 004.056.5:004.08
Євген Кіхтенко[1], Олексій Шкурченко[2]
Державний університет інформаційно-комунікаційних технологій,
[1] kykhtenko@stud.duikt.edu.ua, [2] shkurchenko@stud.duikt.edu.ua

Завдяки значному розвитку інформаційних і комунікаційних технологій з'являються та швидко змінюються нові загрози кібербезпеці. Кіберзлочинці впроваджують нові методи, які роблять їхні атаки швидшими та масштабнішими. Таким чином, існує попит на більш адаптивні та компактні

Додаток Б.

Код тестового застосунку

index.php

```
<!DOCTYPE html>
<html>
<body>
   <h1>Стенд тестування на XSS вразливості</h1>

   <ul>
      <a href="vulnerable-page.php">Вразлива сторінка</a>
      <h2>Алгоритми протидії</h2>
      <a href="alg-validation.php">Алгоритм валідації</a><br>
      <a href="alg-sanitization.php">Алгоритм очищення</a><br>
      <a href="alg-encoding.php">Алгоритм кодування</a><br>
   </ul>
</body>
</html>
```

vulnerable-page.php

```
<?php
$input = $_GET['input'] ?? '';
?>
<!DOCTYPE html>
<html>
<head>
   <title>Вразлива сторінка</title>
</head>
<body>
   <h1>Вразлива сторінка</h1>
   <p><a href="index.php">На головну</a></p>

   <form action="vulnerable-page.php" method="GET">
      <label>Введення: <input type="text" name="input" value="" size="60"></label>
      <button type="submit">Надіслати</button>
   </form>

   <hr>

   <?php if (!empty($input)): ?>

   <h2>Контекст 1: Тіло HTML</h2>
   <div>
      <p><?php echo $input; ?></p>
   </div>

   <hr>

   <h2>Контекст 2: Атрибут HTML</h2>
   <div>
```

```
        <div title="<?php echo $input; ?>">Наведіть на цей текст</div>
    </div>

    <hr>

    <h2>Контекст 3: JavaScript</h2>
    <div>
        <p id="js-output"></p>
        <script>
        var userInput = "<?php echo $input; ?>";
        document.getElementById('js-output').textContent = "Вміст в JavaScript змінній: " +
userInput;
        </script>
    </div>

    <hr>

    <h2>Контекст 4: URL Параметр</h2>
    <div>
        <a href="page.php?data=<?php echo $input; ?>">Посилання</a>
    </div>

    <?php endif; ?>
</body>
</html>
```

alg-validation.php

```
<?php
require_once 'algorithms/validator.php';

$nickname = 'dkarpets';
$email = 'd.karpets@st.wunu.edu.ua';
$text = 'Penetration Tester';

$error = '';
if (isset($_GET['submit'])) {
    $inputNickname = $_GET['nickname'] ?? '';
    $inputEmail = $_GET['email'] ?? '';
    $inputText = $_GET['text'] ?? '';

    if (!InputValidator::validate($inputNickname, 'name')['valid']) {
        $error = 'Помилка, не допустимий символ в полі Нікнейм';
    } elseif (!InputValidator::validate($inputEmail, 'email')['valid']) {
        $error = 'Помилка, не допустимий символ в полі Email';
    } elseif (!InputValidator::validate($inputText, 'text')['valid']) {
        $error = 'Помилка, не допустимий символ в полі Текст';
    } else {
        $nickname = $inputNickname;
        $email = $inputEmail;
        $text = $inputText;
    }
}
```

```
?>
<!DOCTYPE html>
<html>
<head>
   <title>Алгоритм валідації</title>
</head>
<body>
   <h1>Алгоритм валідації</h1>
   <p><a href="index.php">На головну</a></p>

   <form action="alg-validation.php" method="GET">
      <p>
         <label>Нікнейм: <input type="text" name="nickname" value="<?php echo $nickname;
?>"></label><br>
         <span style="font-size:13px">a-Z, 0-9 (до 12 символів)</span>
      </p>
      <p>
         <label>Email: <input type="text" name="email" value="<?php echo $email; ?>"
size="30"></label><br>
         <span style="font-size:13px">a-Z, 0-9, . та @ (до 30 символів)</span>
      </p>
      <p>
         <label>Короткий опис: <input type="text" name="text" value="<?php echo $text; ?>"
size="40"></label><br>
         <span style="font-size:13px">a-Z, 0-9, спеціальні символи (до 30 символів)</span>
      </p>
      <button type="submit" name="submit" value="1">Застосувати</button>
   </form>

   <?php if ($error): ?>
   <hr>
   <p style="color: red;"><?php echo $error; ?></p>
   <?php endif; ?>
</body>
</html>

alg-sanitization.php
<?php
require_once 'algorithms/sanitizer.php';

$rawInput = $_GET['input'] ?? '';
$input = InputSanitization::sanitize($rawInput);
?>
<!DOCTYPE html>
<html>
<head>
   <title>Алгоритм очищення</title>
</head>
<body>
   <h1>Алгоритм очищення</h1>
   <p><a href="index.php">На головну</a></p>
```

```
    <form action="alg-sanitization.php" method="GET">
        <label>Введення: <input type="text" name="input" value="" size="60"></label>
        <button type="submit">Надіслати</button>
    </form>

    <hr>

    <?php if (!empty($rawInput)): ?>

    <h2>Контекст 1: Тіло HTML</h2>
    <div>
        <p><?php echo $input; ?></p>
    </div>

    <hr>

    <h2>Контекст 2: Атрибут HTML</h2>
    <div>
        <div title="<?php echo $input; ?>">Наведіть на цей текст</div>
    </div>

    <hr>

    <h2>Контекст 3: JavaScript</h2>
    <div>
        <p id="js-output"></p>
        <script>
        var userInput = "<?php echo $input; ?>";
        document.getElementById('js-output').textContent = "Вміст в JavaScript змінній: " +
userInput;
        </script>
    </div>

    <hr>

    <h2>Контекст 4: URL Параметр</h2>
    <div>
        <a href="page.php?data=<?php echo $input; ?>">Посилання</a>
    </div>

    <?php endif; ?>
</body>
</html>

alg-encoding.php
<?php
require_once 'algorithms/encoder.php';

$rawInput = $_GET['input'] ?? '';

$htmlEncoded = ContextEncoder::encodeByContext($rawInput, 'html');
$attrEncoded = ContextEncoder::encodeByContext($rawInput, 'attribute');
```

```
$jsEncoded = ContextEncoder::encodeByContext($rawInput, 'javascript');
$urlEncoded = ContextEncoder::encodeByContext($rawInput, 'url');
?>
<!DOCTYPE html>
<html>
<head>
   <title>Алгоритм кодування</title>
</head>
<body>
   <h1>Алгоритм кодування</h1>
   <p><a href="index.php">На головну</a></p>

   <form action="alg-encoding.php" method="GET">
      <label>Введення: <input type="text" name="input" value="" size="60"></label>
      <button type="submit">Надіслати</button>
   </form>

   <hr>

   <?php if (!empty($rawInput)): ?>

   <h2>Контекст 1: Тіло HTML</h2>
   <div>
      <p><?php echo $htmlEncoded; ?></p>
   </div>

   <hr>

   <h2>Контекст 2: Атрибут HTML</h2>
   <div>
      <div title="<?php echo $attrEncoded; ?>">Наведіть на цей текст</div>
   </div>

   <hr>

   <h2>Контекст 3: JavaScript</h2>
   <div>
      <p id="js-output"></p>
      <script>
      var userInput = <?php echo $jsEncoded; ?>;
      document.getElementById('js-output').textContent = "Вміст в JavaScript змінній: " +
userInput;
      </script>
   </div>

   <hr>

   <h2>Контекст 4: URL Параметр</h2>
   <div>
      <a href="page.php?data=<?php echo $urlEncoded; ?>">Посилання</a>
   </div>
```

```
    <?php endif; ?>
</body>
</html>
```

validator.php

```
<?php

class InputValidator {

    public static function validate($input, $type) {
        $patterns = [
            'name' => '/^[a-zA-Z0-9]{2,12}$/',
            'email' => '/^(?=.{1,30}$)[a-zA-Z0-9.]+@[a-zA-Z0-9.]+\.[a-zA-Z]{2,}$/',
            'text' => '/^[a-zA-Z0-9\s.,!?\-\'\"@#$%^&*()_+=\[\]{}|\\\\:;<>\/~`]{1,30}$/'
        ];

        if (!isset($patterns[$type])) {
            return ['valid' => false];
        }

        return ['valid' => preg_match($patterns[$type], $input)];
    }
}
?>
```

sanitizer.php

```
<?php

class InputSanitization {

    public static function sanitize($input) {

        $sanitized = preg_replace('/<(\S[^>]*)>/', '< >', $input);

        return $sanitized;
    }
}
?>
```

encoder.php

```
<?php

class ContextEncoder {

    public static function encodeHTML($input) {
        $output = '';
        $length = strlen($input);

        for ($i = 0; $i < $length; $i++) {
            $char = $input[$i];

            switch ($char) {
```

```
            case '<':
                $output .= '&lt;';
                break;
            case '>':
                $output .= '&gt;';
                break;
            case '&':
                $output .= '&amp;';
                break;
            case '"':
                $output .= '&quot;';
                break;
            case "'":
                $output .= '&#x27;';
                break;
            default:
                $output .= $char;
        }
    }

    return $output;
}

public static function encodeHTMLAttribute($input) {
    $output = self::encodeHTML($input);

    $dangerous = ['javascript:', 'data:', 'vbscript:'];
    foreach ($dangerous as $protocol) {
        $output = str_ireplace($protocol, '', $output);
    }

    return $output;
}

public static function encodeJavaScript($input) {
    $output = '';
    $length = strlen($input);

    for ($i = 0; $i < $length; $i++) {
        $char = $input[$i];

        switch ($char) {
            case '"':
                $output .= '\\"';
                break;
            case "'":
                $output .= "\\'";
                break;
            case '\\':
                $output .= '\\\\';
                break;
            case "\n":
```

```
                $output .= '\\n';
                break;
            case "\r":
                $output .= '\\r';
                break;
            case "\t":
                $output .= '\\t';
                break;
            case '<':
                $output .= '\\x3C';
                break;
            case '>':
                $output .= '\\x3E';
                break;
            case '&':
                $output .= '\\x26';
                break;
            default:
                $output .= $char;
        }
    }

    $output .= "";
    return $output;
}

public static function encodeURL($input) {
    $output = '';
    $length = strlen($input);

    for ($i = 0; $i < $length; $i++) {
        $char = $input[$i];

        if (ctype_alnum($char) || $char === '-' || $char === '_' || $char === '.' || $char === '~') {
            $output .= $char;
        } else {
            $output .= '%' . strtoupper(sprintf('%02X', ord($char)));
        }
    }

    return $output;
}

public static function encodeByContext($input, $context) {
    switch ($context) {
        case 'html':
            return self::encodeHTML($input);
        case 'attribute':
            return self::encodeHTMLAttribute($input);
        case 'javascript':
            return self::encodeJavaScript($input);
        case 'url':
```

```
            return self::encodeURL($input);
        default:
            return self::encodeHTML($input);
    }
  }
}
?>
```