НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

До захисту допущено
Завідувач кафедри

_____ **Дмитро ЛАНДЕ**
(підпис)

«_____»_____2024 р.

Дипломна робота

на здобуття ступеня бакалавра
за освітньо-професійною програмою
«Системи, технології та математичні методи кібербезпеки»
спеціальності 125 «Кібербезпека»

на тему: Метод захисту веб-застосунків заснованих на технології React

Виконав: здобувач вищої освіти **IV** курсу, групи ФБ - 01
(шифр групи)

Свірщук Ярослав Юрійович
(прізвище, ім'я, по батькові) (підпис)

Керівник к.ек.н, доцент, Ткач Володимир Миколайович
(посада, науковий ступінь, вчене звання, прізвище, ім'я, по батькові) (підпис)

Рецензент асистент кафедри ІБ НН ФТІ, Котов Денис Олегович
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище, ім'я, по батькові) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Здобувач вищої освіти _____
(підпис)

Київ – 2024 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

імені ІГОРЯ СІКОРСЬКОГО»

НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Кафедра інформаційної безпеки

Рівень вищої освіти – перший (бакалаврський)
Спеціальність – 125 «Кібербезпека»
Освітньо-професійна програма «Системи, технології та математичні методи кібербезпеки»

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ Дмитро ЛАНДЕ
(підпис)

«___»_____2024 р.

ЗАВДАННЯ
на дипломну роботу здобувачу вищої освіти

Свірщук Ярослав Юрійович
(прізвище, ім'я, по батькові)

1. Тема роботи: Метод захисту веб-застосунків заснованих на технології React

керівник роботи Ткач Володимир Миколайович, к.ек.н, доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від « 31» травня 2024 р. № 2251-с

2. Термін подання здобувачем вищої освіти роботи 7 червня 2024 р.

3. Вихідні дані до роботи: статті та документації пов'язані з методами захисту веб-застосунків.

4. Зміст роботи: огляд існуючих вразливостей для клієнтської та серверної частини веб-застосунків, існуючих методів захисту від оглянутих вразливостей, створення статичного аналізатору веб-застосунків, створених на React, аналіз його роботи та порівняння з існуючими аналізаторами.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): презентація

6. Дата видачі завдання: 01.11.2023

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів дипломної роботи	Примітка
1	Отримання завдання	01.11.2023	Виконано
2	Огляд та опрацювання інформаційних джерел	05.01.2024 – 10.02.2024	Виконано
3	Аналіз вразливостей та загроз веб-застосунків	11.02.2024 – 20.02.2024	Виконано
4	Аналіз методів захисту веб-застосунків	21.02.2024 – 29.02.2024	Виконано
5	Дослідження питання методів захисту веб-застосунків заснованих на технології React	01.03.2024 – 11.03.2024	Виконано
6	Створення статичного аналізатора безпеки веб-застосунків заснованих на технології React	15.04.2024 – 05.05.2024	Виконано
7	Дослідження можливостей вдосконалення статичного аналізатора безпеки веб-застосунків заснованих на технології React	06.05.2024 – 26.05.2024	Виконано
8	Проведення порівняння результатів роботи створеного статичного аналізатора безпеки веб-застосунків та існуючого	27.05.2024 – 01.06.2024	Виконано

Здобувач вищої освіти _____ Ярослав Свірщук
(підпис) (**Власне ім'я, ПРІЗВИЩЕ**)

Керівник роботи _____ Володимир Ткач
(підпис) (**Власне ім'я, ПРІЗВИЩЕ**)

РЕФЕРАТ

Обсяг дипломної роботи **«70»** сторінок, **«30»** ілюстрацій, **«3»** таблиці, **«1»** додаток і **«19»** джерел літератури. В роботі проаналізовано основні види вразливостей веб-застосунків заснованих на технології React, розроблено власний метод захисту та проведена оцінка його функціонування.

Об’єкт дослідження: веб-застосунки засновані на фреймворк React.

Предмет дослідження: методи захисту веб-застосунків заснованих на фреймворку React.

Мета дослідження: аналіз методів захисту веб-застосунків заснованих на фреймворку React.

Методи дослідження: аналіз літературних джерел, створення програмного застосунку.

Результати роботи можуть використовуватися для створення методів захисту веб-застосунків заснованих на React.

Ключові слова: React, вразливість, методи захисту, статичний аналізатор, XSS, CSRF.

ABSTRACT

The volume of the thesis is 70 pages, 30 illustrations, 3 tables, 1 appendices, and 19 references. The work analyzes the main types of vulnerabilities of web applications based on React technology, develops its own method of protection and evaluates its functioning.

Object of study: web applications based on the React framework.

Subject of research: methods of protecting web applications based on the React framework.

Purpose of the study: to analyze the methods of protecting web applications based on the React framework.

Research methods: analysis of literary sources, creation of a software application.

The results of the work can be used to create methods for protecting web applications based on React.

Keywords: React, vulnerability, protection methods, static analyzer, XSS, CSRF.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

HTML - HyperText Markup Language (мова розмітки гіпертексту)

CSS - Cascading Style Sheets (Каскадні таблиці стилів)

JS – JavaScript

DOM - Document Object Model (Об'єктна модель документа)

Скрипт (сценарій) - алгоритм дій, описаних за допомогою скриптової мови програмування (JavaScript, PHP, Perl, Python і ін.) для автоматичного виконання завдань

JSX – JavaScript XML, розмітка, що поєднує JavaScript та HTML теги

Props (Пропси) - вхідні дані React-компонента

URL - Uniform Resource Locator (єдиний вказівник на ресурс, уніфікований локатор ресурсів)

SQL - Structured Query Language (мова структурованих запитів)

Cookie (cookies) - невеликий фрагмент даних, відправлений веб-сервером, що зберігається на комп'ютері клієнта

XSS – Cross Site Scripting (міжсайтовий скриптинг, міжсайтове виконання сценаріїв)

CSRF – Cross-site request forgery (Міжсайтова підробка запиту)

IDOR - Insecure direct object reference (Незахищене пряме посилання на об’єкт)

RCE - Remote Code Execution (Віддалене виконання коду)

DoS – Denile of Service (Атака на відмову в обслуговуванні)

DDoS – Distributed Denial of Service (Розподілена атака на відмову в обслуговуванні)

SAST - Static application security testing (Статичний аналізатор безпеки застосунку)

DAST - Dynamic application security testing (Динамічний аналізатор безпеки застосунку)

ВСТУП

Кількість веб-застосунків збільшується з кожним роком завдяки їх зручності та доступності. Зі збільшенням кількості збільшилася також й складність розробки, через що на заміну чистому JavaScript прийшли різні фреймворки, такі як Angular, Vue.js, React. Однак зі збільшенням популярності цих технологій, зросла й увага зловмисників до них. Тому захист від можливих загроз є дуже важливим етапом під час розробки, впровадження та підтримки веб-застосунку для будь-якої організації, яка використовує його для забезпечення своєї функціональності.

Згідно з дослідженнями [1], React є найпопулярнішим фреймворком для створення клієнтської частини веб-застосунків та використовується для розробки у 40.58% опитуваних. Оскільки цей фреймворк є найпопулярнішим серед розробників, він був обраний для огляду наявних методів захисту, які є вбудованими у React або є окремими технологіями для захисту веб-застосунків, а також для створення нового методу захисту.

Одними з найпопулярніших видів загроз веб-застосунків є Cross-Site Scripting (XSS) та Cross Site Request Forgery (CSRF). Згідно з дослідженнями [2], атаки з використанням XSS займають 53% від кількості всіх атак на веб-застосунки у 2023 році, а атаки з використанням CSRF – близько 20%. Таким чином, разом ці атаки складають майже три четверті від усіх атак на веб-застосунки та є найпоширенішими атаками у минулому році.

1 ОГЛЯД ФРЕЙМВОРКУ REACT ТА АТАК НА ВЕБ-ЗАСТОСУНКИ

1.1 Особливості фреймворку React

Фреймворк — це структура або набір інструментів та бібліотек, що надають готові рішення для побудови програмного забезпечення. Веб-фреймворк — це фреймворк, призначений для підтримки розробки веб-застосунків, включаючи веб-сервіси, веб-ресурси та веб-інтерфейси API. Ці фреймворки значно спрощують розробку нових веб-застосунків, надаючи готові інструменти та рішення для виконання типових завдань, що виникають при створенні застосунків.

React - це популярний фреймворк JavaScript з відкритим кодом, який використовується для створення інтерфейсів користувача у односторінкових веб-застосунках, де інтерфейс користувача має бути динамічним та інтерактивним. Він був розроблений компанією Facebook і підтримується ними разом із спільнотою окремих розробників і компаній. React відомий своїм декларативним підходом до створення компонентів інтерфейсу користувача, що робить процес розробки ефективнішим і легшим в управлінні.

Застосунки, створені за допомогою React, побудовані з компонентів, які є незалежними частинами інтерфейсу користувача, які можна багаторазово використовувати в різних частинах програми, зменшуючи загальну кількість програмного коду та полегшуючи створення застосунку. Кожен компонент може мати власний стан і пропси, і їх можна компонувати для створення складних користувацьких інтерфейсів. Стан компоненту – це представлення набору даних, які впливають на рендеринг веб-затосунку. Зміна стану компоненту запускає його повторний рендеринг, щоб синхронізувати користувацький інтерфейс з даними програми. Пропси компоненту – це набір даних, що передається від батьківського компоненту до дочірнього та не впливає на рендеринг дочірнього компоненту. Пропси є незмінюваними та використовуються для обміну даними, такими як числа, рядки, масиви,

об’єкти, функції та інші, між різними компонентами. Важливо зазначити, що пропси не можуть передаватися напряму від дочірнього компоненту до батьківського, чи між сусідніми компонентами. З одного боку, такий підхід може створювати незручності для передачі даних між компонентами, однак з іншого боку, при наявності одностороннього потоку даних «зверху вниз», зберігається простота відслідковування потоків даних у веб-застосунку. Якщо ж така модель передачі даних не задовольняє потреби застосунку, наприклад, визначений набір даних має бути доступний для багатьох компонентів на різних рівнях ієрархії, то для вирішення цієї проблеми можуть бути використані бібліотеки для керування станом веб-застосунку. Найпопулярніші бібліотеки, що використовуються для управління станом – Redux, MobX, Recoil, Jotai, Valtio тощо.

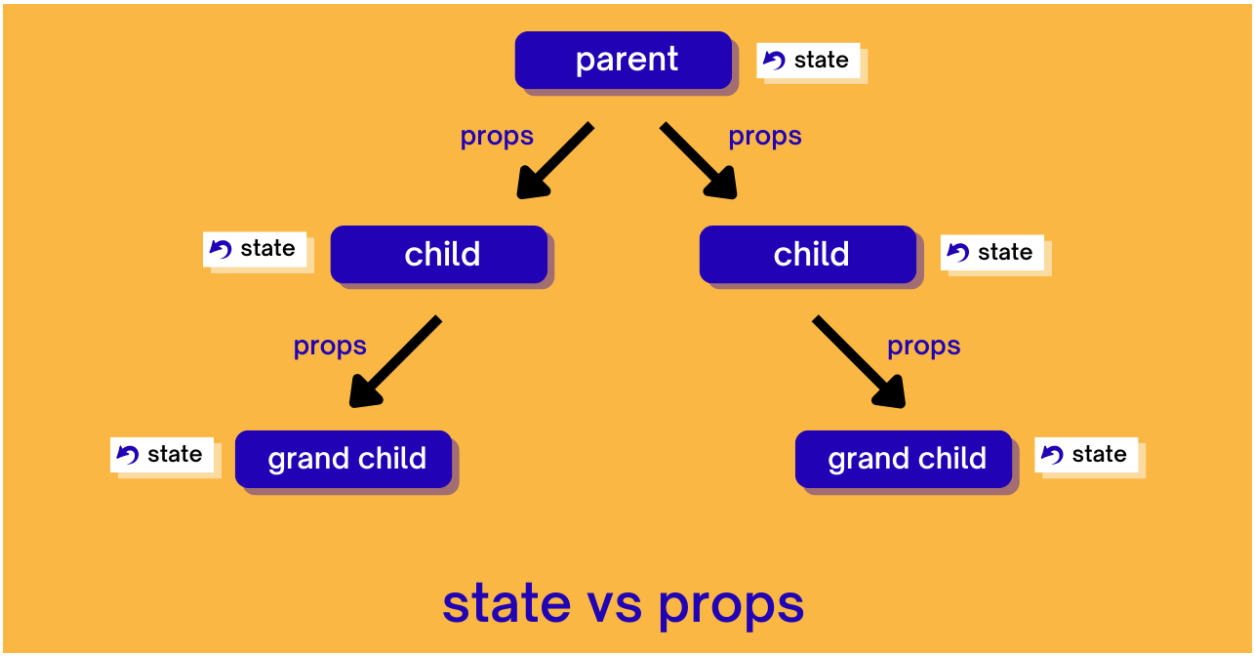


Рисунок 1.1 - Передача даних між компонентами

React використовує віртуальну об’єктну модель документу - DOM для оптимізації продуктивності веб-застосунку. Коли стан об’єкта змінюється, React спочатку оновлює віртуальну DOM, яка є полегшеною копією фактичної DOM. Потім він порівнює віртуальну DOM із справжньою DOM і оновлює лише ті частини моделі, які змінилися, а не повторно рендерить всю сторінку.

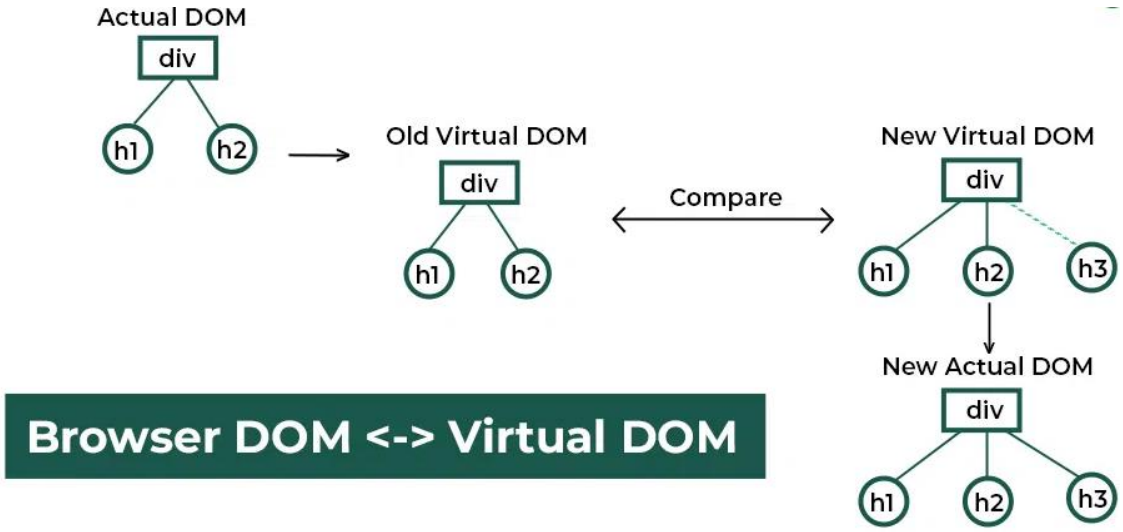


Рисунок 1.2 - Відслідковування змін за допомогою віртуальної DOM та зміна справжньої [6]

React використовує JSX, синтаксичне розширення, яке дозволяє писати HTML-подібний код у JavaScript. Це робить код більш читабельним і легшим для написання завдяки поєднанню макета інтерфейсу користувача та логіки застосунку.

1.2 Види вразливостей та атак на веб-застосунки

Вразливість у контексті веб-розробки — це слабка сторона або недолік у дизайні, реалізації або конфігурації веб-сайту, якою можуть скористатися зловмисники, щоб отримати неавторизований доступ, порушити роботу застосунку, викрасти дані або виконати інші зловмисні дії.

Кібератака на веб-застосунок — це зловмисна спроба використати вразливості для отримання несанкціонованого доступу, викрадення даних, порушення роботи застосунку або виконання інших шкідливих дій. Веб-застосунки є особливо привабливими цілями для зловмисників, оскільки вони часто обробляють конфіденційні дані та надають критично важливі послуги.

Впровадження захисту для веб-застосунків є важливим етапом під час розробки та підтримки застосунку, через можливі наслідки від успішної реалізації кібератаки:

- Порушення безпеки конфіденційних даних - вразливості можна використати для доступу та викрадення конфіденційної інформації, такої як особисті дані, фінансові записи та конфіденційна інформація. Захист від атак допомагає запобігти витоку даних, що може мати серйозні наслідки як для окремих осіб, так і для організацій.
- Втрата репутації та довіри клієнтів - користувачі очікують, що веб-застосунки будуть безпечними. Порушення безпеки можуть підірвати довіру та змусити користувачів уникати використання веб-застосунку. Крім того, інцидент із безпекою може завдати значної шкоди репутації власникам застосунку, що призведе до втрати користувачів та можливих збитків.
- Відповідальність за зберігання користувацьких даних та юридичні зобов’язання – власники веб-застосунку можуть бути притягнуті до юридичної відповідальності у випадку, якщо конфіденційні дані користувачів були викрадені зловмисниками чи потрапили у відкритий доступ через недбалість їх використання та зберігання або невиконання вимог чинного законодавства.
- Безперервність роботи - вразливості можна використати для порушення нормальної роботи веб-застосунку, що призведе до простою та припинення обслуговування користувачів.
- Фінансові втрати - інциденти безпеки можуть призвести до прямих фінансових втрат через витрати на реагування на інцидент та на відновлення після нього.

Існує багато різних специфічних видів кібератак на веб-застосунки, однак ми розглянемо загальні категорії цих атак. Усі атаки будуть описані в контексті веб-застосунків.

1.2.1 Міжсайтовий сценарій

Міжсайтовий сценарій (Cross-Site Scripting, XSS) - це тип атаки, коли зловмисник може вставити у довірений веб-застосунок шкідливі сценарії, які потім виконує браузер жертви. Ця атака використовує використовує таку вразливість, як неналежне або відсутнє оброблення даних, введених користувачем. Це може призвести до різних шкідливих дій, таких як викрадення даних користувачів, файлів cookie, токенів сеансу чи іншої конфіденційної інформації, перенаправлення користувачів на шкідливі сайти.

Існує декілька типів XSS:

- Активний (Stored) – шкідливий сценарій, що вводиться зловмисником, назавжди зберігається на сервері, наприклад, у полі для коментарів на сайті блогу або в дописі на форумі. Коли користувач отримує збережені зловмисником дані, сценарій доставляється як частина довіреної веб-сторінки та виконується браузером користувача. Прикладом може слугувати веб-блог, де користувачі можуть розміщувати свої дописи. Якщо дезінфекція введених користувачами даних відсутня, зловмисник може розмістити допис, що буде містити шкідливий сценарій. Скрипт буде збережений на сервері та виконуватись кожного разу, коли користувач буде переглядати допис зловмисника.
- Відображений (Reflected) - це вид атаки, у якій дані, що вводяться зловмисником у запит до серверу, передаються на сервер, однак не зберігаються на ньому, а результат запиту повертається та виводиться клієнту без належної перевірки та обробки даних. Прикладом може слугувати пошукова стрічка веб-застосунку, яка перед результатами пошуку виводить рядок «Результати пошуку за запитом: *запит користувача*». Якщо дані, що вводяться клієнтом для пошуку, не

перевіряються на наявність шкідливого коду та не оброблюються належним чином, то зловмисник може вставити будь-який скрипт у пошукову стрічку та отримати результат його виконання при виводі рядку «Результати пошуку за запитом: *шкідливий скрипт зловмисника*». Потім злочинець надсилає жертві URL посилання на довірений сайт з шкідливим сценарієм у пошуковій стрічці, наприклад «https://vulnerable-website.com/search?name=<script>alert(“XSS attack”)</script>» , який виконується при переході жертвою за посиланням.

- На основі DOM – ця атака можлива, якщо у програмі веб-застосунку наявний код, який вносить дані з ненадійного джерела напряму в DOM. Це дозволяє виконати шкідливий скрипт, за допомогою якого можна, наприклад, перехопити дані користувача. Для прикладу розглянемо веб-застосунок, який вносить дані з URL запиту напряму у модель DOM. У такому випадку зловмисник може додати шкідливий сценарій в URL запит, веб-застосунок внесе цей сценарій у об’єктну модель документу і браузер користувача виконає цей сценарій, оскільки він буде думати, що це код довіреного застосунку.

1.2.2 Атака міжсайтової підробки запитів

Атака міжсайтової підробки запитів (Cross-site request forgery, CSRF) — це тип атаки, коли зловмисник обманом спонукає користувача виконувати дії у веб-застосунку, де він пройшов автентифікацію. Це може статися без відома користувача, який використовує свій автентифікований сеанс для виконання дій від імені жертви. Атаки CSRF використовують довіру веб-програми до браузера користувача, а не використовують вразливість у самій програмі. Для того, щоб виконати CSRF атаку, веб-застосунок, на який проводиться атака, має підлягати 2 пунктам: По-перше, застосунок має використовувати лише токен (кукі) сесії для автентифікації та валідації користувача. По-друге, для виконання будь-яких дій у застосунку не потрібна додаткова валідація,

наприклад, введення поточного паролю для його зміни на новий. Розглянемо простий приклад атаки типу CSRF. Якщо користувач авторизований в веб-застосунку банку, і для подальших дій у застосунку не потрібна додаткова валідація, зловмисник може надіслати користувачу свій сайт з шкідливим CSRF скриптом. Цей скрипт містить запит до серверу банку зі здійсненням якоїсь дії, наприклад переказ грошей. При переході на сайт, браузер користувача виконає запит до застосунку банку і автоматично підставить токен (кукі) сесії користувача. Сервер отримає запит з правильним кукі і підтвердить виконання запиту. Таким чином, зловмисник може вкрасти гроші жертви, перевівши їх на свій рахунок (достатньо в тілі запиту вказати карту зловмисника).

1.2.3 Ін’єкція SQL

Ін’єкція SQL – це атака, яка дозволяє зловмиснику виконувати SQL-запити до бази даних веб-застосунку. Ця атака може бути виконана, якщо дані, які вводяться користувачем, одразу передаються в SQL-запит без їх перевірки та обробки. Завдяки цій атаці зловмисник може отримати, змінити або видалити інформацію, до якої він не повинен був мати доступ, з бази даних застосунку. Існує декілька видів SQL ін’єкцій:

- Атака на основі помилок (Error-based SQL injection) - використовує повідомлення про помилки бази даних для збору інформації про структуру бази даних і використання вразливостей. Наприклад, зловмисник вводить «' OR 1=1;--» у поле вводу імені користувача у формі входу. Якщо застосунок не обробляє цей ввід належним чином, вона може створити повідомлення про помилку з розкриттям деталей бази даних.
- Атака на основі оператора UNION (UNION-based SQL injection) - об’єднує результати вихідного запиту з результатами зловмисного запиту за допомогою оператора UNION. Наприклад, зловмисник

вводить «' UNION SELECT username, password FROM users;--» у поле пошуку. Якщо застосунок додає ці дані до запиту SQL без перевірки, вона може надати зловмиснику імена користувачів і паролі з таблиці користувачів.

- Атака всліпу на основі логічного виразу (Boolean-based Blind SQL injection) - зловмисник надсилає запити, які повертають різні відповіді залежно від того, чи є певна умова істинною чи хибною, виводячи інформацію про базу даних. Наприклад, зловмисник вводить «' OR SLEEP(5);--» у поле пошуку. Якщо застосунку потрібно більше часу, щоб відповісти, це вказує на те, що введена умова виконується, тобто TRUE.
- Атака всліпу на основі часу (Time-based Blind SQL) - зловмисник надсилає запити, які спричиняють затримку відповіді бази даних, якщо певна умова відповідає дійсності, що дозволяє йому робити висновок про інформацію на основі часу відповіді.Наприклад, зловмисник вводить «' OR IF(SUBSTRING(database(),1,1)='a',SLEEP(5),0);--» у поле пошуку. Якщо застосунок відповідає через кількість часу, вказаного у запиті зловмисника, то це вказує на те, що першим символом імені бази даних є літера «a».
- Атака другого порядку (Second-Order SQL Injection) - у цьому сценарії введене корисне навантаження SQL безпосередньо не виконує шкідливий запит, але воно зберігається програмою для подальшого використання. Коли збережені дані пізніше використовуються в запиті, виконується введений шкідливий запит SQL. Наприклад, зловмисник у веб-застосунку з відгуками пише у поле тексту «'); DROP TABLE users; --». Застосунок зберігає цей запит у базі даних, однак не виконує його одразу. Оскільки обробка введених зловмисником даних відсутня, то коли адміністратор ресурсу перегляне відгук зловмисника, шкідливий запит виконається від імені адміністратора та видалить таблицю з користувачами застосунку.

1.2.4 Незахищене пряме посилання на об’єкт

Незахищене пряме посилання на об’єкт (Insecure Direct Object Reference, IDOR) - це тип атаки, що використовує вразливості, які виникають, коли програма надає прямий доступ до об’єктів, таких як записи бази даних, файли чи URL-адреси, на основі введених користувачем даних без належної перевірки цих даних. Ця атака може дозволити зловмисникам отримати доступ або маніпулювати даними, до яких вони не мають доступу. Наприклад, існує сайт, який для отримання інформації про обліковий запис користувача з бази даних серверу використовує наступний URL «https://vulnerable-website.com/customer_info?customer_id=132355». У цьому прикладі, номер користувача напряму вказується у посиланні, тому, за відсутності додаткових методів захисту, наприклад автентифікації користувача, зловмисник може підмінити номер у посиланні та отримати інформацію про будь-якого користувача.

1.2.5 Порушена автентифікація

Порушена автентифікація (Broken authentication) - це тип атаки, яка базується на вразливості, через яку зловмисник може зламати механізми автентифікації веб-програми, дозволяючи йому отримати несанкціонований доступ до конфіденційних даних застосунку або облікових записів інших користувачів. Ця вразливість зазвичай виникає через погану реалізацію або конфігурацію функцій автентифікації та керування сеансами. Види вразливостей, завдяки яким можливо реалізувати дану атаку:

- Слабка політика паролів - відсутність дотримання правил надійних паролів, таких як мінімальна довжина, вимоги до складності або термін дії, полегшує зловмисникам вгадування або підбір паролів користувачів.

- Ненадійні методи обміну даними користувача - використання небезпечних методів обміну даними користувача, таких як передача облікових даних незашифрованим текстом через HTTP, або слабкі алгоритми шифрування, які можуть бути перехоплені та легко розшифровані зловмисниками.
- Проблеми з управлінням сеансом користувача - неналежне керування сеансами, як-от відсутність використання безпечних cookie, відстуність завершення сеансів після визначеного періоду бездіяльності користувача може призвести до викрадення сеансу або атак із фіксацією сеансу.
- Передбачувані ідентифікатори сеансу - використання легких та передбачуваних ідентифікаторів сеансу, які можуть бути вгадані або знайдені методом перебору зловмисниками, що дозволяє їм видавати свої сеанси за сеанси довірених користувачів.
- Недостатній контроль доступу – не ведеться належний контроль доступу, як-от відсутність автентифікації для доступу до конфіденційних даних або ресурсів, що дозволяє неавторизованим користувачам отримувати доступ до привілейованих областей застосунку.

1.2.6 Атака захвату кліку

Атака захвату кліку (Clickjacking) - це тип атаки, коли зловмисник обманом змушує користувача натиснути щось, що відрізняється від того, що користувач бачить, таким чином виконуючи дії, про які він навіть не підозрює. Зазвичай це досягається шляхом накладення або приховування шкідливих елементів інтерфейсу користувача поверх довірених веб-сторінок. Коли користувачі взаємодіють з візуально видимими елементами, вони насправді взаємодіють із прихованими елементами, що призводить до неочікуваних дій. Наприклад, зловмисник створює веб-сторінку з кнопкою «Ви виграли приз!

Натисніть, щоб забрати». Поверх цієї сторінки він накладує невидиму сторінку з власною публікацією із соціальної мережі та з кнопкою «Подобається». Оскільки ця сторінка соціальної мережі повністю невидима, користувач думає, що натискає на кнопку з отриманням призу, однак насправді він натискає на кнопку «Подобається», яка знаходиться у тому ж місці поверх видимої кнопки.

1.2.7 Віддалене виконання коду

Віддалене виконання коду - (Remote Code Execution, RCE) — це тип атаки, яка дозволяє зловмиснику виконувати довільний код на сервері веб-застосунку. Ця атака може бути реалізована, коли програма приймає та обробляє введені користувачем дані таким чином, що дозволяє зловмиснику вводити та виконувати зловмисний код на сервері, на якому працює застосунок. Для прикладу розглянемо застосунок, який дозволяє завантажувати користувачам файли з довільною назвою на сервер, що працює на операційній системі Linux, після чого сервер зчитує дані з файлу за допомогою команди «cat». Якщо немає ніякої обробки назви файлу, зловмисник може завантажити файл з назвою «; rm -rf /». Сервер, намагаючись прочитати цей файл, виконає набір команд «cat ; rm -rf /», що може призвести до видалення всієї файлової системи веб-застосунку.

1.2.8 Атака на відмову в обслуговуванні та розподілена атака на відмову в обслуговуванні

Атака на відмову в обслуговуванні (Denial of Service, DoS) - це тип атаки, спрямований на те, щоб переповнити застосунок великим потоком запитів або споживаючи його ресурси для порушення чи припинення його функціонування. Це може перешкодити звичайним користувачам отримати доступ до застосунку, спричинивши перебої у роботі. Традиційні DoS-атаки

походять з одного джерела, що полегшує їх виявлення та усунення порівняно з розподіленими атаками на відмову в обслуговуванні (DDoS), які надходять із кількох джерел.

Розподілена атака на відмову в обслуговуванні (Distributed Denial of Service, DDoS) – це підвид DoS атаки, який також має за мету погіршення чи припинення роботи застосунку, однак DDos використовує декілька скомпрометованих систем, часто розподілених по всьому світу, для переповнення застосунку великою кількістю трафіку або запитів.

1.3 Вразливості веб-застосунків, створених за допомогою React

У попередньому пункті були перечислені різні види атак, які спрямовані як на клієнтську частину веб-застосунку, так і на серверну. Оскільки React – фреймворк для створення клієнтської частини застосунку, то лише частина з цих атак може бути виконана через вразливості клієнтської частини.

1.3.1 dangerouslySetInnerHTML

dangerouslySetInnerHTML - це спеціальний атрибут HTML тегу у React, який дозволяє вставляти текст з HTML розміткою безпосередньо в компонент. Це схоже на властивість innerHTML у звичайному JavaScript, але з вбудованим в назву попередженням через потенційні ризики безпеці, пов’язані з його використанням. dangerouslySetInnerHTML в основному використовується, коли вам потрібно вставити контент з HTML, який містить HTML-теги безпосередньо в компонент React. Він зазвичай використовується під час роботи з вмістом, який надходить із CMS, візуального редактора або будь-якого іншого джерела з HTML розміткою.

```
<div dangerouslySetInnerHTML={{ __html: <div>Some HTML-content</div> }} />
```

Рисунок 1.3 - Приклад використання атрибуту dangerouslySetInnerHTML

Якщо HTML контент не оброблено належним чином, наприклад в тег напряму вставляється ввід користувача без дезінфекції HTML, використання dangerouslySetInnerHTML може наражати програму на атаки міжсайтового сценарію (XSS). Як приклад можна розглянути простий застосунок, який має елемент «input» для отримання вводу від користувача, та елемент «div» для прямого виводу на сторінку вводу користувача. У такому застосунку користувач може ввести будь-який скрипт у поле вводу, натиснути кнопку Submit - і цей сценарій вбудується у код застосунку та виконається у вікні користувача через відсутність фільтрації та дезінфекції користувацького вводу. Програмний код застосунку зображено на рисунку 2.2

```jsx
import { useState } from "react";

const Example = () => {
  let [msg, setMsg] = useState();
  let [msg1, setMsg1] = useState();

  let handleChange = (e) => {
    setMsg(e.target.value);
  };

  let handleSubmit = () => {
    setMsg1(msg);
  };

  return (
    <div>
      <form onSubmit={handleSubmit}>
        <input
          onChange={handleChange}
          value={msg}
          style={{ width: "400px", marginRight: "20px" }}
        ></input>
        <button type="submit">Submit</button>
      </form>

      <div dangerouslySetInnerHTML={{ __html: msg1 }}></div>
    </div>
  );
};

export default Example;
```

Рисунок 1.4 - Застосунок для введення даних та виводу їх на веб-сторінку

1.3.2 Неналежне зберігання конфіденційної інформації в клієнтській частині програми

Зберігання секретних даних, таких як, наприклад, API токен, логіни та паролі користувачів тощо, у незашифрованому вигляді напряму у коді

програми є вразливістю через те, що будь хто може переглянути код програми та використати ці дані у власних цілях.

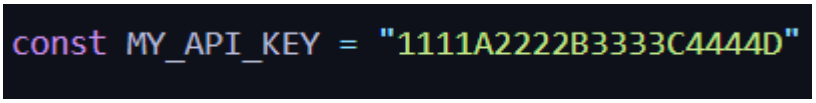


Рисунок 1.5 - Приклад небезпечного зберігання ключа API у клієнтській частині застосунку

1.3.3 Неналежне зберігання конфіденційної інформації у localStorage

localStorage - це функція веб-браузерів, яка дозволяє веб-додаткам зберігати дані локально на пристрої користувача. Хоча вона пропонує зручність для зберігання невеликих обсягів даних, вона також представляє ризик для безпеки застосунку, якщо використовувати її неправильно. Вразливості локального сховища можуть призвести до розкриття конфіденційних даних, атак міжсайтового сценарію (XSS) та інших проблем безпеки. Прикладом може слугувати програма, яка отримує токен автентифікації користувача від сервера та зберігає його напряму без шифрування у localStorage.

```
const getAuthToken = (username, password) => {
  const authToken = fetch(
    `http://insecure-connection.com/token?user=${username}&password=${password}`
  ).then((response) => {
    return response.json();
  });

  return authToken;
};

const saveToken = async (username, password) => {
  const response = await getAuthToken(username, password);
  if (response.token) {
    localStorage.setItem("authToken", response.token);
  } else {
    throw Error("Authentication error!");
  }
};
```

Рисунок 1.6 - Небзпечне зберігання токену автентифікації у localStorage

1.3.4 Використання незахищеного способу комунікації з сервером

Використання незахищеного способу комунікації між клієнтом та сервером є однією з головних проблем витоку конфіденційної інформації. Коли комунікації незахищені, зловмисники можуть перехопити, змінити або підслухати дані, що передаються. Це може призвести до витоку даних, атак типу "людина посередині" (MITM) та інших вразливостей безпеки. Прикладом може слугувати використання HTTP замість захищеного HTTPS.

```
fetch("http://insecure-connection.com/users.json")
  .then((response) => {
    return response.json();
  })
  .then((data) => {
    console.log(data);
  });
```

Рисунок 1.7 - Приклад незахищеного запиту до серверу з використання протоколу HTTP

1.3.5 Відсутність CSRF-токену при запиті до серверу

Майже у кожному веб-застосунку в наш час використовується система користувачів з використанням конфіденційних даних, такі як логін та пароль користувача, електронна пошта, його методи оплати тощо, та з функціями, що взаємодіють з цими даними, наприклад зміна логіну чи паролю, оплата онлайн, переказ грошей з картки та інші. Використання лише одноразової автентифікації як довірчого фактору для здійснення наведених вище операцій є недостатнім методом захисту, оскільки, за допомогою CSRF атаки, зловмисник може виконати таку дію, як, наприклад, переказ грошей з картки користувача, навіть не володіючи даними його облікового запису, якщо користувач авторизований у додатку. Запобігти цій атаці можна за допомогою постійної автетифікації користувача перед виконанням будь-якої його операції, що не завжди є зручним методом зі сторони клієнту, оскільки погіршується використання веб-застосунку через збільшення часу на проведення операцій. Іншим способом захисту є використання CSRF-токену, який працює як додатковий фактор перевірки користувача, не збільшуючи кількість дій зі сторони користувача, що він має виконати для проведення операції. Звісно, для застосунків, де вимоги до безпеки є найвищими, як для того ж онлайн-банку, краще використовувати обидва методи, однак для менш важливих завдань можна обійтися лише токеном.

Оскільки CSRF токен неможливо реалізувати лише у клієнтській частині, то як вразливість може вважатися відсутність поля CSRF токену у запиті до серверу.

```
fetch("/transfer", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify({ amount: 1000, recipient: "user_account" }),
});
```

Рисунок 1.8 - Запит до серверу без використання CSRF-токену

Висновки до розділу 1

У цьому розділі були розглянуті основні принципи фреймворку React, такі як компоненти, стан компоненту та його пропси, віртуальний DOM та розмітка JSX. Оскільки веб-застосунки зараз використовуються майже кожною компанією, увага зловмисників до них також є високою. Були оглянуті основні види атак та вразливостей на веб-застосунки в цілому. Виходячи зі специфікації фреймворку React, а саме створення клієнтської части веб-застосунку, були виділені вразливості, які притаманні саме цьому фреймворку, а саме використання атрибуту dangerouslySetInnerHTML без використання дезінфекції та обробки вмісту, що передається цьому атрибуту, що може призвести до реалізації XSS атаки. Також були оглянуті вразливості небезпечного зберігання конфіденційних даних у програмному коді застосунку та у localStorage, використання ненадійних каналів комунікації та відсутність використання CSRF токену у клієнтській частині. Усі ці вразливості можуть призвести до витоку конфіденційних даних користувачів, що є небажаним для будь-якої компанії, тому у наступному розділі ми розглянемо методи захисту від описаних атак та вразливостей.

2 ОГЛЯД ІСНУЮЧИХ МЕТОДІВ ЗАХИСТУ ФРЕЙМВОРКУ REACT

На даний момент, існує багато різних методів захисту веб-застосунків, у тому числі, створених на технології React, однак вони відрізняються своїм призначенням та способом використання. Тому далі будуть розглянуті декілька категорій методів захисту застосунків.

2.1 Вбудовані методи захисту в технологію React

У цій частині будуть розглянуті методи, які вбудовані в React, а також ті, які для свого впровадження не потребують додаткових сторонніх бібліотек чи застосунків. Ці методи включають: автоматичне екранування даних у JSX, а також використання політики безпеки вмісту, захищених каналів комунікації, токенів CSRF.

2.1.1 Автоматичне екранування даних у JSX

Однією з ключових функцій безпеки в React є автоматичне екранування вмісту, що вставляється в JSX розмітку. Ця функція призначена для запобігання атакам міжсайтових сценаріїв (XSS), поширеній вразливості безпеки у веб-додатках, коли зловмисник вставляє шкідливі сценарії на веб-сторінки, які переглядають інші користувачі. Коли React рендерить JSX, він автоматично екранує будь-які значення, вбудовані в JSX, щоб гарантувати, що вони розглядаються як звичайний текст, а не виконуваний код. Це означає, що будь-які потенційно небезпечні символи в значеннях перетворюються на відповідні сутності HTML. Тобто, такі символи, як «<», «>», «&» і «"», замінюються на «<», «>», «&» та «"» відповідно. Наприклад, у програмі, наведеній на рисунку 2.1, змінна userInput містить HTML-тег скрипт, який може потенційно містити шкідливий код. Однак, оскільки ця змінна вставляється через JSX розмітку, React автоматично екранує усі дані, та

перетворює їх у звичайний текст, тому «<» і «>» заміняться на «<» і «>», тому вміст тегу <script> та будь-якого іншого тегу не виконається взагалі, а буде відображений як звичайний текст.

```
const userInput = "<script>alert('XSS')</script>";
const Component = () => <div>{userInput}</div>;
```

Рисунок 2.1 - Автоматична обробка даних у JSX

2.1.2 Використання політики безпеки вмісту

Політика безпеки вмісту (Content security policy, CSP) - це HTTP-заголовок, який дозволяє веб-розробникам визначати домени, які браузер повинен розглядати як довірені джерела виконуваних сценаріїв та інших ресурсів. Політика CSP доставляється через заголовок HTTP (або тег <meta>) і складається з правил (директив), які визначають дозволені джерела для різних типів вмісту. Потім браузер виконує ці директиви, блокуючи будь-який вміст, який не відповідає вказаним критеріям. Ця функція безпеки допомагає запобігти різним типам атак, у тому числі міжсайтовим сценарієм (XSS), захвату кліку (clickjacking) та іншим атакам із впровадженням шкідливого коду. Це дозволяє контролювати ресурси, які веб-браузеру дозволено завантажувати для певної сторінки.

Таблиця 2.1 - Перелік директив CSP

Директива	Опис директиви
`default-src`	Директива, яка буде виконуватися за замовчуванням, якщо інші не вказані. Ця директива включає всі дерективи, вказані далі
`script-src`	Визначає дозволені джерела для JavaScript ресурсів
`style-src`	Визначає дозволені джерела для CSS ресурсів
`img-src`	Визначає дозволені джерела для зображень
`connect-src`	Визначає дозволені кінцеві точки для з'єднань типу AJAX, WebSocket, EventSource
`font-src`	Визначає дозволені джерела для шрифтів
`object-src`	Визначає дозволені джерела для тегів `<object>`, `<embed>`, `<applet>`
`media-src`	Визначає дозволені джерела для елементів медіа (аудіо та відео)
`frame-src`	Визначає дозволені джерела для вкладених контекстів перегляду, таких як `<iframe>`
`child-src`	Застаріла директива, на заміну якій прийшли `frame-src` та `worker-src`
`worker-src`	Визначає дозволені джерела для Web Workers and Shared Workers
`frame-ancestors`	Визначає дозволені батьківські фрейми, що можуть вбудовувати поточну сторінку (запобігає атаці захвату кліку)
`form-action`	Визначає дозволені джерела для URI-адрес дій форм
`base-uri`	Обмежує використання URL-адрес в тегі `<base>`
`plugin-types`	Обмежує набір плагінів, які можна вставляти у документ
`sandbox`	Вмикає пісочницю для запитуваного ресурсу
`report-uri`	Визначає URL-адресу, куди браузер надсилає звіти про порушення політики безпеки вмісту
`report-to`	Визначає назву групи звітування, визначену HTTP заголовком Report-To для надсилання звітів про порушення CSP

Кінець таблиці 2.1

`manifest-src`	Визначає дозволені джерела для файлів маніфесту програми
`prefetch-src`	Визначає дозволені джерела для попереднього вибору, завантаження та рендерингу вмісту
`navigate-to`	Обмежує перелік URL-адрес, які може відкривати документ будь-яким способом (тобто посилання у тегі `<a>`, у елементі `window.location` тощо)
`script-src-elem`	Визначає дозволені джерела для елементів <script>
`script-src-attr`	Визначає дозволені джерела для оброблювачів подій JavaScript
`style-src-elem`	Визначає дозволені джерела для CSS елементів `<style>` та `<link>` з використанням CSS
`style-src-attr`	Визначає дозволені джерела для вбудованих стилів CSS

Основні директиви, що використовуються у політиках безпеки вмісту – це «default-src», «script-src», «style-src», «img-src», «connect-src», «font-src», «frame-ancestors». Для прикладу розглянемо політику безпеки вмісту, яка дозволяє веб-застосунку використовувати картинки з власного домену та «trusted-images.com», сценарії з власного та «secure-js.com», стилі з власного та «secure-css.com», а весь інший вміст використовувати тільки з власного домену, а також забороняє вбудовування поточної сторінки у будь яку іншу. Тоді отримаємо наступну політику, показану на рисунку 2.2:

```
<meta
    httpEquiv="Content-Security-Policy"
    content="default-src 'self';
    img-src 'self' 'trusted-images.com';
    script-src 'self' 'secure-js.com';
    style-src 'self' 'secure-css.com';
    frame-ancestors 'none';"
></meta>;
```

Рисунок 2.2 - Приклад політики безпеки вмісту, описаної в тегі <meta>

Політику безпеки вмісту краще налаштовувати на сервері, щоб весь веб-застосунок підпорядковувався визначеним правилам, однак, за відсутності можливості зміни конфігурації серверу, CSP можна налаштувати і в клієнтській частині, в тегі <meta>, однак тоді ця політика буде працювати тільки в документі, в якому вона визначена.

2.1.3 Використання захищених каналів комунікації з севером

Використання HTTPS (HyperText Transfer Protocol Secure) замість HTTP (HyperText Transfer Protocol) також має важливе значення для захисту веб-додатків. HTTPS шифрує дані, якими обмінюються клієнт і сервер, забезпечуючи конфіденційність і цілісність даних. Тому при обміні даними між клієнтом та сервером веб-застосунку, чи клієнтом та API важливо використовувати саме HTTPS.

```
fetch("https://secure-connection.com/users.json")
    .then((response) => {
        return response.json();
    })
    .then((secure_data) => {
        console.log(secure_data);
    });
```

Рисунок 2.3 - Приклад HTTPS запиту до серверу

2.1.4 Використання CSRF токенів

Токен CSRF — це унікальне, випадкове та секретне значення, яке генерується сервером і включається в запити HTTP для підтвердження того, що запит надійшов із законного джерела. Він використовується для запобігання атакам CSRF, гарантуючи, що запит надходить від користувача, який ініціював сеанс. Існує 3 поширених методи імплементації токенів у веб-застосунок – «Synchronizer Token Pattern», «Double-Submit Cookie Pattern» та «SameSite Cookies».

Synchronizer Token Pattern - підхід, в якому токен зберігається на стороні серверу. Сервер генерує токен на початку кожної сесії або з кожним новим запитом, зберігає його у себе та надсилає користувачу. Коли користувач надсилає запит серверу, він має прикріпити свій CSRF токен у спеціальне поле заголовку або за допомогою прихованого поля форми у тіло запиту. Сервер перевіряє, чи збігається токен користувача і той, що зберігається в нього, і якщо токени однакові – запит вважається легітимним. Перевагою цього підходу є те, що токен зберігається в безпечному місці на сервері, однак недоліком є те, що цей метод є доволі важким у реалізації та погано масштабується.

Double-Submit Cookie Pattern – підхід, в якому сервер також генерує токен CSRF на початку сесії користувача, однак не зберігає його у себе, а надсилає одразу клієнту, у якого токен зберігається у вигляді кукі. Коли клієнь бажає надіслати запит серверу, він має прикріпити токен CSRF двома способами, а саме як кукі, які автоматично додаються браузером у заголовки запиту, а також як окремий спеціальний заголовок запиту або за допомогою прихованого поля форми у тіло запиту. Сервер, отримавши запит, порівнює токен в кукі та додатковий токен в заголовку чи тіла запиту. Якщо вони однакові – запит вважається достовірним. Цей метод є простіше в реалізації, але є менш надійним, оскільки якщо зловмисник отримає значення кукі користувача (наприклад, за допомогою XSS атаки) – то може його використати для обходу цього методу.

Same-Site Cookies – це атрибут, що з’явився доволі нещодавно, який контролює коли браузер має автоматично прикріпляти кукі сесії в запит до серверу веб-застосунку. У цього атрибуту може бути три значення – Strict, Lax та None.

Strict – кукі зі з атрибутом «SameSite=Strict» будуть автоматично додаватися в запит до сервера, що був ініційований з того ж сайту, що і встановив кукі клієнта. Всі інші запити не будуть мати кукі користувача у собі.

Lax - кукі зі з атрибутом «SameSite=Lax» будуть надсилатися автоматично з того ж сайту, який встановлював кукі, а також з інших сайтів у разі верхньорівневої навігації (наприклад, перехід на цей сайт за посиланням, що знаходиться на сторонньому сайті). Однак, якщо прийде запит на картинку чи на фрейм, що знаходиться на цьому сайті, з стороннього сайту, то тоді кукі користувача не буде прикріплене в запит.

None – кукі користувача буде прикріплене в запит з будь-якого сайту до сервера застосунку. Однак для використання цього варіанту маж бути встановлений атрибув Secure, який вказує, що кукі буде прикріплене в запит до серверу тільки за умови, що запит здійснюється за допомогою протоколу HTTPS.

Із опису варіантів значень атрибуту Same-Site можна зрозуміти, що варіант «Strict» є найбезпечнішим, однак його потрібно встановлювати тільки на запити, що можуть бути отримані лише з поточного сайту (зміна паролю, надсилання транзакції тощо). Якщо ж легітимний запит може бути надісланий і з іншого сайту (наприклад, посилання на поточний сайт з стороннього), то є сенс ставити значення «Lax», щоб веб-застосунок міг одразу визначити, який користувач перейшов за посиланням та, наприклад, одразу підставити для нього персоналізований контент. «None» не рекомендується використовувати взагалі.

Всі методи імплементації CSRF токенів, описані вище, мають бути впроваджені у серверній частині застосунку та не можуть працювати лише у клієнтській. Тому як метод захисту саме у клієнтській частині можна вважати наявність CSRF токену у запитах до серверу.

```
fetch("/api/submit", {
    method: "POST",
    headers: {
        "Content-Type": "application/json",
        "CSRF-Token": csrfToken,
    },
    body: JSON.stringify({ data: secure_request_body }),
});
```

Рисунок 2.4 - Приклад запиту до серверу з використанням CSRF токену

2.2 Сторонні методи захисту веб-застосунків, створених на технології React

У цій частині будуть розгдянуті методи засобу, що надаються сторонніми бібліотеками, такими як DOMPurify, dotenv та React Helmet

2.2.1 DOMPurify

DOMPurify — це потужна бібліотека з можливістю конфігурації, розроблена для очищення HTML, SVG і MathML для запобігання атакам міжсайтового сценарію (XSS). Це допомагає гарантувати, що будь-який вміст, який відображається у вашій програмі, є безпечним і не містить шкідливого коду. Основний елемент бібліотеки DOMPurify – функція sanitize(), яка приймає на вхід HTML розмітку, прибирає з неї теги або атрибути тегів, які потенційно можуть містити шкідливий вміст, наприклад тег <script>, атрибут onerror, який найчастіше використовується з тегом <img>, атрибут onload, тег <iframe> тощо. Бібліотека підтримується розробниками, постійно оновлюється та має програму винагороди за виявлення помилок в роботі бібліотеки, тому її можна впевнено використовувати, якщо потрібно вставити HTML розмітку напряму в DOM, наприклад за допомогою розглянутого вище атрибуту dangerouslySetInnerHTML.

```
DOMPurify.sanitize("<img src=x onerror=alert(1)//>"); // Вивід команди буде <img src="x">
DOMPurify.sanitize('<script>alert("Xss")</script>'); // Вивід команди буде порожнім
console.log(DOMPurify.sanitize("<img src=x onerror=alert(1)//>"));
console.log(DOMPurify.sanitize('<script>alert("Xss")</script>'));
```

Рисунок 2.5 - Приклад використання бібліотеки DOMPurify

```
<img src="x">                                    Analyzer.jsx:86
                                                 Analyzer.jsx:87
```

Рисунок 2.6 - Результат виконання попереднього скрипту

Також існують й інші бібліотеки, які пропонують такий ж функціонал, наприклад sanitize-html, js-xss, однак вони не так активно підтримуються, тому в них можуть бути наявні не всі типи небезпечних елементів.

2.2.2 dotenv

dotenv — це модуль, який завантажує змінні середовища з файлу .env у process.env. Змінні середовища — це пари ключ-значення, які встановлюються поза програмою та доступні для використання в програмі. Вони часто використовуються для зберігання параметрів конфігурації, таких токени API, рядки підключення до бази даних і ключі автентифікації. Змінні середовища є одним з важливих аспектів розробки клієнтської частини веб-застосунку. Змінні середовища дозволяють зберігати та використовувати параметри конфігурації та конфіденційну інформацію без жорсткого кодування їх безпосередньо в саму програму, що дозволяє використовувати цю інформацію під час роботи застосунку, не розкриваючи її користувачам.

Для того, щоб використовувати змінні середовища, потрібно створити окремий файл «.env», в якому створюється пара ключ-значення, де ключ – це назва змінної, яку ми хочемо створити, а значення – це дані, які ми хочемо розмістити в змінній. На рисунку 2.3 показано приклад такої пари, де назва змінної REACT_APP_MY_API_KEY, а значення 1111A2222B3333C4444D. До речі, для веб-застосунків, створених на React, частина «REACT_APP_» на

початку назви змінної є обов’язковою. Це міра безпеки, яка гарантує, що лише явно дозволені змінні доступні для JavaScript на стороні клієнта.

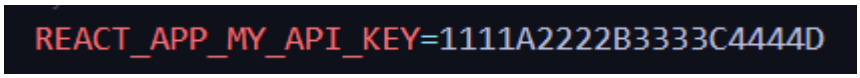


Рисунок 2.7 - Приклад оголошення змінної середовища

2.2.3 React Helmet

React Helmet — це бібліотека, яка використовується для керування змінами в частині <head> документа в веб-застосунку, створеному на React. Це допомагає динамічно встановлювати мета-теги, заголовки та інші елементи в заголовку документа, що в нашому контексті може використовуватись для налаштування політик безпеки вмісту (CSP) для захисту від атак типу XSS. За допомогою цієї бібліотеки тег <head> та теги, що використовуються всередині можливо налаштувати окремо для кожного компонента React. Наприклад, можна налаштувати політику безпеки вмісту, які ми створили у пункті 2.1.2 для компоненту MyCSPComponent.

```
import { Helmet } from "react-helmet";

function MyCSPComponent() {
    return (
        <div>
            <Helmet>
                <title>My Page Title</title>
                <meta
                    httpEquiv="Content-Security-Policy"
                    content="default-src 'self';
    img-src 'self' 'trusted-images.com';
    script-src 'self' 'secure-js.com';
    style-src 'self' 'secure-css.com';
    frame-ancestors 'none';"
                ></meta>
                ;
                <link rel="canonical" href="https://www.example.com/my-page" />
            </Helmet>
            <h1>My Page Content</h1>
        </div>
    );
}
```

Рисунок 2.8 - Приклад налаштування CSP за допомогою бібліотеки React Helmet

2.3 Аналізатори безпеки веб-застосунків

У пунктах 2.1 та 2.2 були розглянуті методи захисту, які захищають веб-застосунки в основному від однієї атаки. Однак також існують і комплексні рішення, які забезпечують захист застосунку в цілому, від різних типів атак.

3.1 Статичні аналізатори безпеки застосунків

Статичний аналізатор безпеки застосунку (Static Application Security Testing, SAST) — це тип перевірки безпеки, який аналізує вихідний код програми, на наявність вразливостей без виконання програмного коду застосунку. Інструменти SAST часто називають інструментами тестування «білого ящика», оскільки вони мають доступ до внутрішньої роботи програми.

Дані аналізатори можуть виявляти вразливості типу XSS, CSRF, SQL injection, Broken Authentication та інші ще на етапі розробки застосунку, що дозволяє виправити велику частину питань безпеки ще до розгортання застосунку.

SAST аналізує всю кодову базу, включаючи всі можливі шляхи виконання, що допомагає виявити вразливості, які можуть бути не виявлені під час виконання. Інструменти SAST можна інтегрувати в конвеєри безперервної інтеграції (CI) і безперервного розгортання (CD), автоматизуючи виявлення вразливостей у кожній збірці. Аналізатор може підтримувати кілька мов програмування та фреймворків, що робить його універсальним для різних типів програм. Також він може створювати детальні звіти та документацію за результатами аналізу, яка є корисною для перевірок і нормативних вимог. До рішень, що відносяться до цього методу захисту є Aikido, Codacy, Snyk та Semgrep.

3.2 Динамічний аналізатор безпеки застосунків

Динамічний аналізатор безпеки застосунків (Dynamic Application Security Testing, DAST) - це тип перевірки безпеки, який аналізує запущену програму для виявлення вразливостей шляхом імітації зовнішніх атак. На відміну від статичного тестування безпеки додатків (SAST), яке перевіряє вихідний код, інструменти DAST аналізують програму під час її роботи, взаємодіючи з веб-застосунком через інтерфейс, надсилаючи різні вхідні дані та спостерігаючи за відповідями. Оскільки DAST не вимагає доступу до вихідного коду, він імітує перспективу зловмисника, тестуючи програму ззовні. Динамічні аналізатори можуть виявляти вразливості типу XSS, CSRF, SQL injection, Broken Authentication та інші під час виконання. DAST, як і SAST, також можна інтегрувати в конвеєри безперервної інтеграції (CI) і безперервного розгортання (CD), підтримує кілька мов програмування та фреймворків, може створювати звіти та документацію за результатами

аналізу. Приклади реалізацій динамічного аналізатора безпеки застосунків - OWASP ZAP, Burp Suite та Acunetix.

Висновки до розділу 2

У цьому розділі були розглянуті різні методи захисту від вразливостей клієнтської частини веб-застосунків створених на технології React, як вбудовані у фреймворк, так і сторонні бібліотеки. Був оглянутий принцип роботи статичних та динамічних аналізаторів, які забезпечують виявлення вразливостей на етапі розробки застосунку та під час його роботи. Найкраще ці аналізатори працюють в парі, однак я обрав створення статичного аналізатора, оскільки етап розробки веб-застосунку передує етапу виконання.

3 СТВОРЕННЯ ВЛАСНОГО МЕТОДУ ЗАХИСТУ ФРЕЙМВОРКУ REACT ТА АНАЛІЗ РЕЗУЛЬТАТІВ ЙОГО РОБОТИ

3.1 Створення статичного аналізатора безпеки веб-застосунків заснованих на технології React з використанням штучного інтелекту

Головна задача статичного аналізатора – це виявлення вразливостей у програмному коді, написаному на React та надання рекомендацій щодо усунення цих вразливостей. Для створення цього аналізатора за основу для розробки був обраний фреймворк React. Також для розгортання та збірки проєкту використовувався інструмент Vite, для додання стилів використовувався CSS а також «react-syntax-highlighter» для стилізації програмного коду, що виводиться на сторінці. Для аналізу коду за допомогою штучного інтелекту була обрана модель «gpt-3.5-turbo» від OpenAI. Результати аналізу ШІ виводяться англійською мовою, оскільки у українського виводу є певні недоліки з перекладом, що погіршує розуміння результатів. Також була використана бібліотека dotenv для зберігання токену OpenAIAPI у змінній середовища.

Інтерфйес аналізатора складається з декількох компонентів – main.jsx, App.jsx, Analyzer.jsx. Компонент main – це кореневий компонент, який вбудовує всі компоненти React у DOM-дерево та відповідає за виведення всіх інших компонентів у вікно браузера. Компонент App – це компонент, який є обгорткою зі стилями CSS для компоненту Analyzer та є збіркою всіх елементів інтерфейсу. Компонент Analyzer – це основний елемент, в якому розташована логічна частина веб-застосунку. Компонент містить такі елементи інтерфейсу, як поле для вводу програмного коду користувача, кнопка «Аналізувати» та поле з виводом рекомендацій. У цьому компоненті спочатку створюється новий об’єкт для з’єднання з OpenAIAPI за допомогою токену. Потім за допомогою асинхронної функції виконується запит до API з

заготовленим текстом-описом того, що має проаналізувати ШІ та повернути у якості результату запиту. Далі отриманий результат виконання запиту виводиться користувачу, а програмний код з виправленими вразливостями виводиться за допомогою окремого компоненту з бібліотеки react-syntax-highlighter <SyntaxHighlighter>.

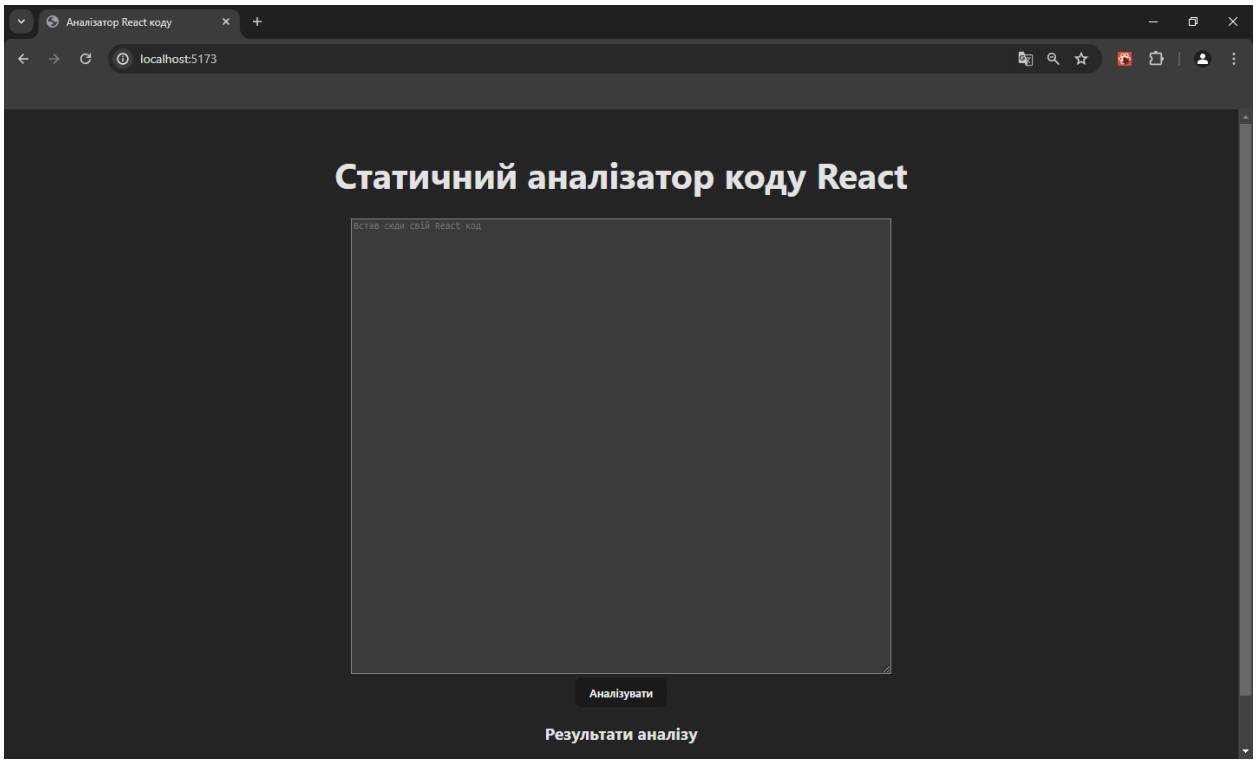


Рисунок 3.1 - Порожня сторінка аналізатора з основними елементами інтерфейсу

При запиті до OpenAIAPI використовується розроблений контекст запиту, у якому описано, на які вразливості потрібно здійснити перевірку та у якому форматі повернути результат запиту.

Рисунок 3.2 – Контекст запиту до OpenAIAPI

3.2 Виявлення вразливостей веб-застосунку розробленим статичним аналізатором

Для перевірки роботи розробленого статичного аналізатора безпеки будуть використані декілька прикладів веб-застосунків з різними видами вразливостей, які були розглянуті у пункті 1.3.

3.2.1 Застосунок, вразливий до XSS

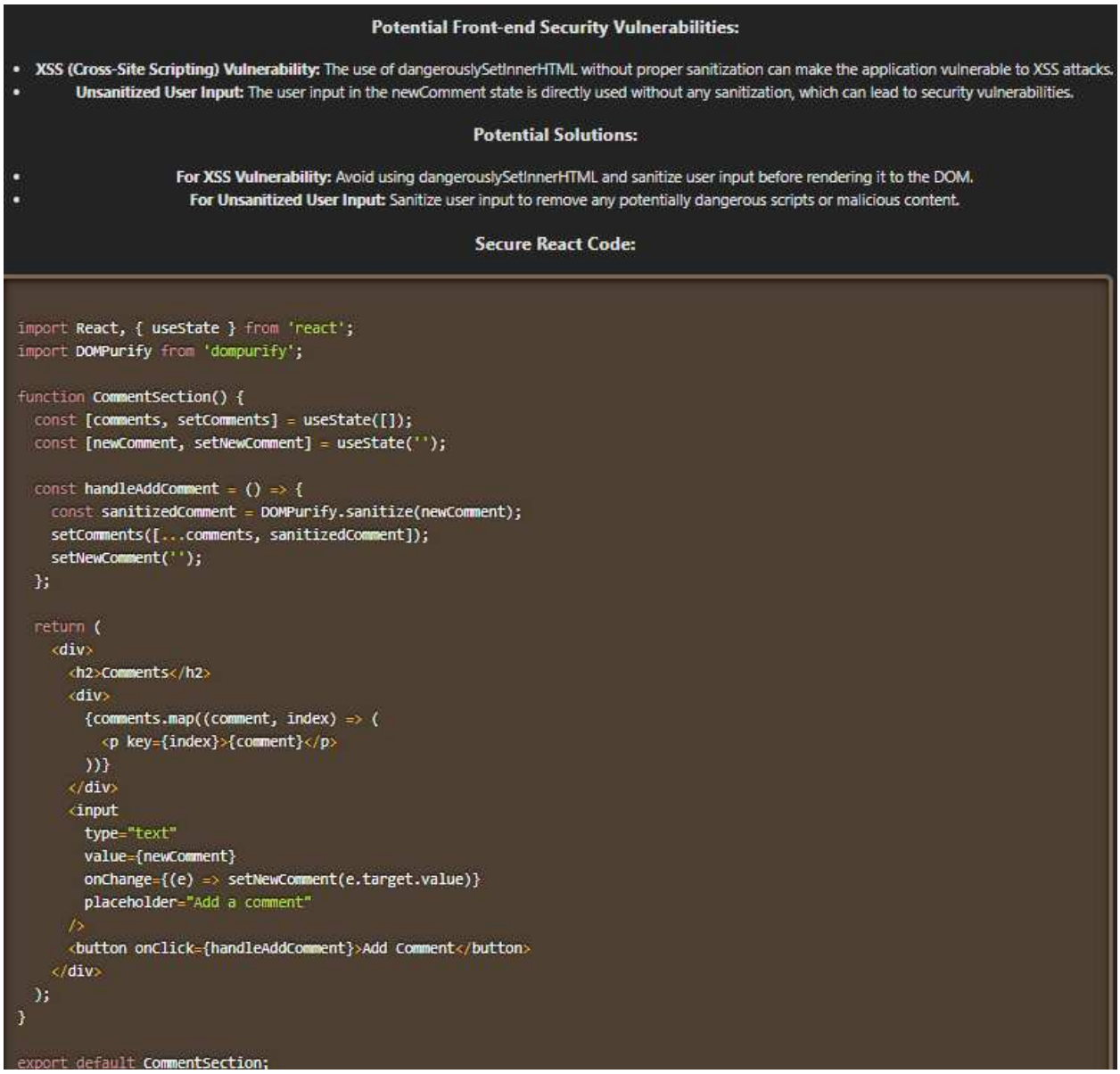
Для перевірки виявлення вразливості XSS був використаний застосунок, який імітує секцію з коментарями, де користувачі можуть писати свої коментарі. Для того, щоб виводити коментарі користувачів використовується небезпечний атрибут dangerouslySetInnerHTML, в який напряму вставляється ввід без обробки та дезінфекції даних. Програмний код застосунку наведений на рисунку 3.2.

```jsx
import React, { useState } from "react";

function CommentSection() {
    const [comments, setComments] = useState([]);
    const [newComment, setNewComment] = useState("");

    const handleAddComment = () => {
        setComments([...comments, newComment]);
        setNewComment("");
    };

    return (
        <div>
            <h2>Comments</h2>
            <div>
                {comments.map((comment, index) => (
                    <p
                        key={index}
                        dangerouslySetInnerHTML={{ __html: comment }}
                    />
                ))}
            </div>
            <input
                type="text"
                value={newComment}
                onChange={(e) => setNewComment(e.target.value)}
                placeholder="Add a comment"
            />
            <button onClick={handleAddComment}>Add Comment</button>
        </div>
    );
}

export default CommentSection;
```

Рисунок 3.3 - Програмний код застосунку, вразливого до XSS

Після аналізу розробленим аналізатором був отриманий наступний результат: була виявлена вразливість XSS через використання dangerouslySetInnerHTML без санітизації вводу, а також була виявлена ще одна, подібна вразливість про відсутність санітизації даних, що вводяться

користувачем. У програмному коді була додана функція з бібліотеки DOMpurify для санітизації користувацькго вводу, а також небезпечний атрибут dangerouslySetInnerHTML був замінений на безпечний вивід за допомогою JSX розмітки.



Potential Front-end Security Vulnerabilities:

- **XSS (Cross-Site Scripting) Vulnerability:** The use of dangerouslySetInnerHTML without proper sanitization can make the application vulnerable to XSS attacks.
- **Unsanitized User Input:** The user input in the newComment state is directly used without any sanitization, which can lead to security vulnerabilities.

Potential Solutions:

- **For XSS Vulnerability:** Avoid using dangerouslySetInnerHTML and sanitize user input before rendering it to the DOM.
- **For Unsanitized User Input:** Sanitize user input to remove any potentially dangerous scripts or malicious content.

Secure React Code:

```
import React, { useState } from 'react';
import DOMPurify from 'dompurify';

function CommentSection() {
  const [comments, setComments] = useState([]);
  const [newComment, setNewComment] = useState('');

  const handleAddComment = () => {
    const sanitizedComment = DOMPurify.sanitize(newComment);
    setComments([...comments, sanitizedComment]);
    setNewComment('');
  };

  return (
    <div>
      <h2>Comments</h2>
      <div>
        {comments.map((comment, index) => (
          <p key={index}>{comment}</p>
        ))}
      </div>
      <input
        type="text"
        value={newComment}
        onChange={(e) => setNewComment(e.target.value)}
        placeholder="Add a comment"
      />
      <button onClick={handleAddComment}>Add Comment</button>
    </div>
  );
}

export default CommentSection;
```

Рисунок 3.4 - Результат виконання аналізу коду, зображеного на рисунку 3.3

3.2.2 Застосунок, що зберігає конфіденційну інформацію у програмному коді та використовує незахищені канали комунікації

Для перевірки на вразливість щодо незахищеного збереження інформації в клієнтській частині програмного коду, а також використання незахищених каналів комунікації був використаний один приклад веб-застосунку, який поєднує обидві вразливості. По-перше, у програмному коді відкрито визначений ключ API, що дає можливість зловмисникам виявити та використовувати цей ключ для власних потреб, а також був використаний протокол HTTP замість більш безпечного HTTPS.

```
import React, { useState, useEffect } from "react";

function WeatherWidget() {
    const [weather, setWeather] = useState(null);
    const [city, setCity] = useState("San Francisco");

    const apiKey = "1234567890abcdef";

    useEffect(() => {
        if (apiKey) {
            fetch(
                `http://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${apiKey}`
            )
                .then((response) => response.json())
                .then((data) => setWeather(data));
        } else {
            console.error("API key is missing");
        }
    }, [city, apiKey]);

    return (
        <div>
            <h2>Weather in {city}</h2>
            {weather && (
                <div>
                    <p>Temperature: {weather.main.temp} °C</p>
                    <p>Condition: {weather.weather[0].description}</p>
                </div>
            )}
        </div>
    );
}

export default WeatherWidget;
```

Рисунок 3.5 - Програмний код застосунку з вразливостями небезпечного зберігання даних та незахищеного каналу комунікації

Результат аналізу застосунку є наступний – моделлю було виявлено 2 вразливості, які були зазначені у описі застосунку. Були надані 2 рекомендації – про використання змінних середовища або серверної частини застосунку для безпечного зберігання ключів API, а також використання захищеного HTTPS протоколу замість HTTP для запиту, виконаного до API сервісу, щоб запобігти атаці типу «людина посередині». У програмному коді звичайний текст ключа був замінений на використання змінних середовища, а також запит був змінений з «http://api.openweathermap.org/…» на «https://api.openweathermap.org/…».

Potential Security Vulnerabilities:

- The API key is exposed in the front-end code, making it vulnerable to theft. It is not recommended to store sensitive information like API keys in front-end code.

- The request to the weather API is sent over HTTP. It is insecure as it does not encrypt the data transferred between the client and the server.

Advice on how to fix the vulnerabilities:

- Avoid storing sensitive information like API keys directly in the front-end code. Instead, you can set up a server that acts as a proxy, securely communicating with external APIs using the API key.

- Use HTTPS instead of HTTP for requests to ensure data encryption and secure communication between the client and the server.

```
import React, { useState, useEffect } from "react";

function WeatherWidget() {
    const [weather, setWeather] = useState(null);
    const [city, setCity] = useState("San Francisco");

    const apiKey = process.env.REACT_APP_WEATHER_API_KEY;

    useEffect(() => {
        if (apiKey) {
            fetch(
                `https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${apiKey}`
            )
                .then((response) => response.json())
                .then((data) => setWeather(data));
        } else {
            console.error("API key is missing");
        }
    }, [city, apiKey]);

    return (
        <div>
            <h2>Weather in {city}</h2>
            {weather && (
                <div>
                    <p>Temperature: {weather.main.temp} °C</p>
                    <p>Condition: {weather.weather[0].description}</p>
                </div>
            )}
        </div>
```

Рисунок 3.6 - Результат виконання аналізу коду, зображеного на рисунку 3.5

3.2.3 Застосунок, вразливий до CSRF

Для перевірки на вразливість до CSRF атаки було використано компонент застосунку, який дозволяє користувачу змінити свій пароль без вводу поточного паролю та без будь-якої додаткової автентифікації. У запиті до серверу на зміну паролю не використовується CSRF токен, що дозволяє зловмиснику атаку міжсайтової підробки запитів. Також в цьому компоненті знову є вразливість використання HTTP у запиті до серверу.

```
import React, { useState } from "react";

function ChangePassword() {
    const [newPassword, setNewPassword] = useState("");

    const handleChangePassword = () => {
        fetch("http://example.com/api/change-password", {
            method: "POST",
            headers: {
                "Content-Type": "application/json",
            },
            body: JSON.stringify({
                newPassword: newPassword,
            }),
        })
            .then((response) => response.json())
            .then((data) => {
                if (data.success) {
                    alert("Password changed successfully!");
                } else {
                    alert("Error changing password.");
                }
            })
            .catch((error) => {
                console.error("Error:", error);
                alert("Error changing password.");
            });
    };

    return (
        <div>
            <div>
                <label>
                    New Password:
                    <input
                        type="password"
                        value={newPassword}
                        onChange={(e) => setNewPassword(e.target.value)}
                    />
                </label>
            </div>
            <button onClick={handleChangePassword}>Change Password</button>
        </div>
    );
}
```

Рисунок 3.7 - Програмний код застосунку з вразливістю до CSRF атаки

Аналіз коду компоненту показав, що не використовується HTTPS, а також не використовується токен CSRF. Рекомендації щодо виправлення цих проблем наступні – використання HTTPS, впровадження валідації CSRF токену на серверній частині застосунку та використання токену рід час запитів до серверу у клієнтській частині.

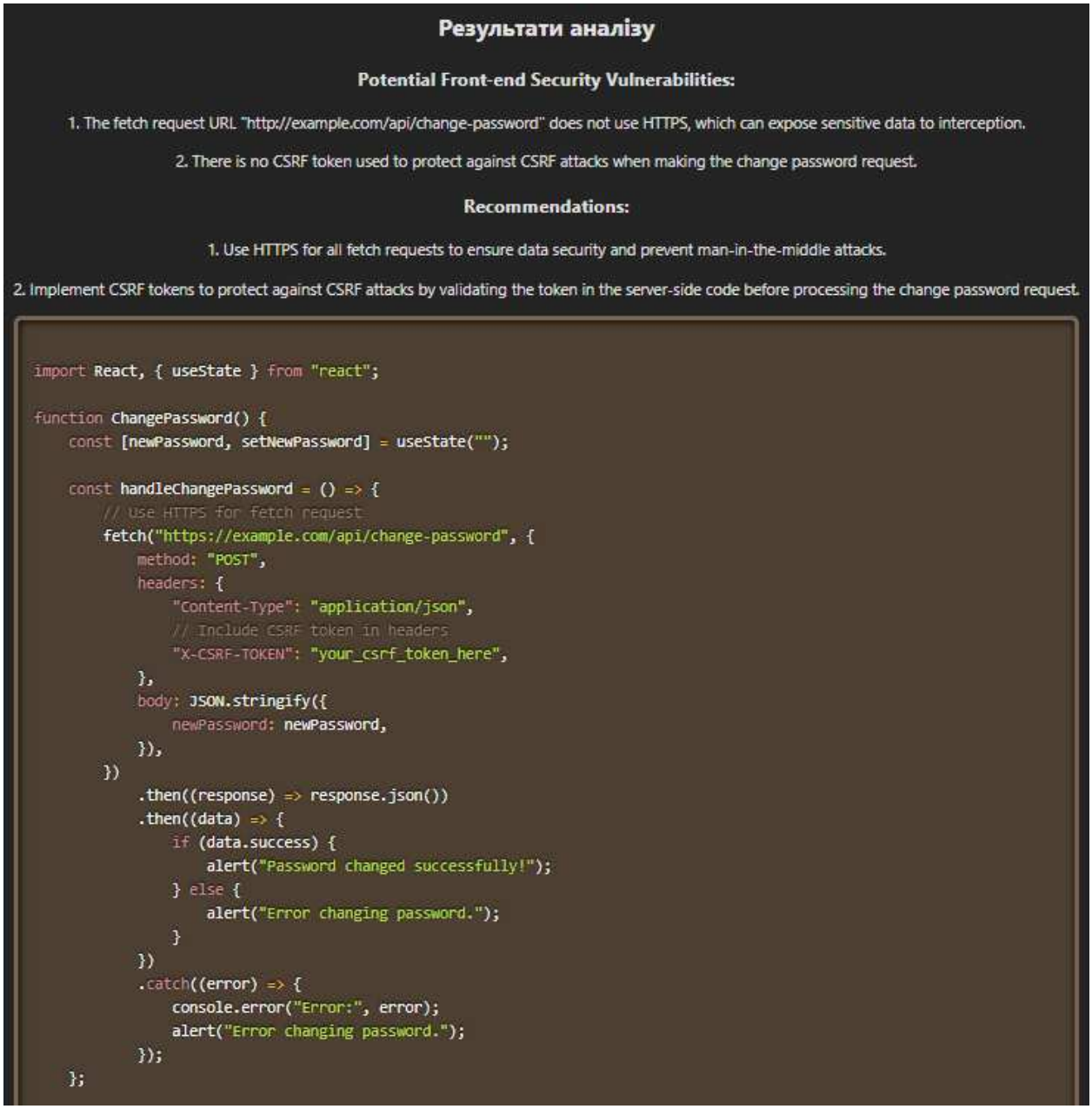


Рисунок 3.8 - Результат виконання аналізу коду, зображеного на рисунку 3.7

3.2.4 Застосунок, що використовує localStorage для зберігання конфіденційних даних

Для тестування на вразливість небезпечного зберігання конфіденційних даних у localStorage була створена програма, яка на вхід отримує логін і пароль користувача, та зберігає їх без шифрування у localStorage. Якщо зловмисник проведе успішну XSS атаку, то він зможе отримати конфіденційну інформацію користувача, яка зберігається без визначеного терміну видалення у localStorage.

```jsx
import React, { useState } from "react";

function Login() {
    const [username, setUsername] = useState("");
    const [password, setPassword] = useState("");

    const handleLogin = () => {
        localStorage.setItem("username", username);
        localStorage.setItem("password", password);
    };

    return (
        <div>
            <h2>Login</h2>
            <div>
                <label>
                    Username:
                    <input
                        type="text"
                        value={username}
                        onChange={(e) => setUsername(e.target.value)}
                    />
                </label>
            </div>
            <div>
                <label>
                    Password:
                    <input
                        type="password"
                        value={password}
                        onChange={(e) => setPassword(e.target.value)}
                    />
                </label>
            </div>
            <button onClick={handleLogin}>Login</button>
        </div>
    );
}

export default Login;
```

Рисунок 3.9 - Програмний код застосунку з небезпечним зберіганням конфіденційних даних користувача

Аналіз програмного коду застосунку показав, що окрім вразливості небезпечного зберігання даних у localStorage, також додатково була виявлена

вразливість відсутності валідації логіну та паролю, введеними користувачем, на відповідність якимось мінімальним критеріям безпеки. Були надані рекомендації щодо зберігання конфіденційної інформації у зашифрованому вигляді у sessionStorage або в кукі, а також про імплементацію мінімальних вимог до логіну і паролю користувача в клієнтській частині. У програмному коді було замінено запис даних в localStorage на запис в sessionStorage, однак не було описане впровадження шифрування.

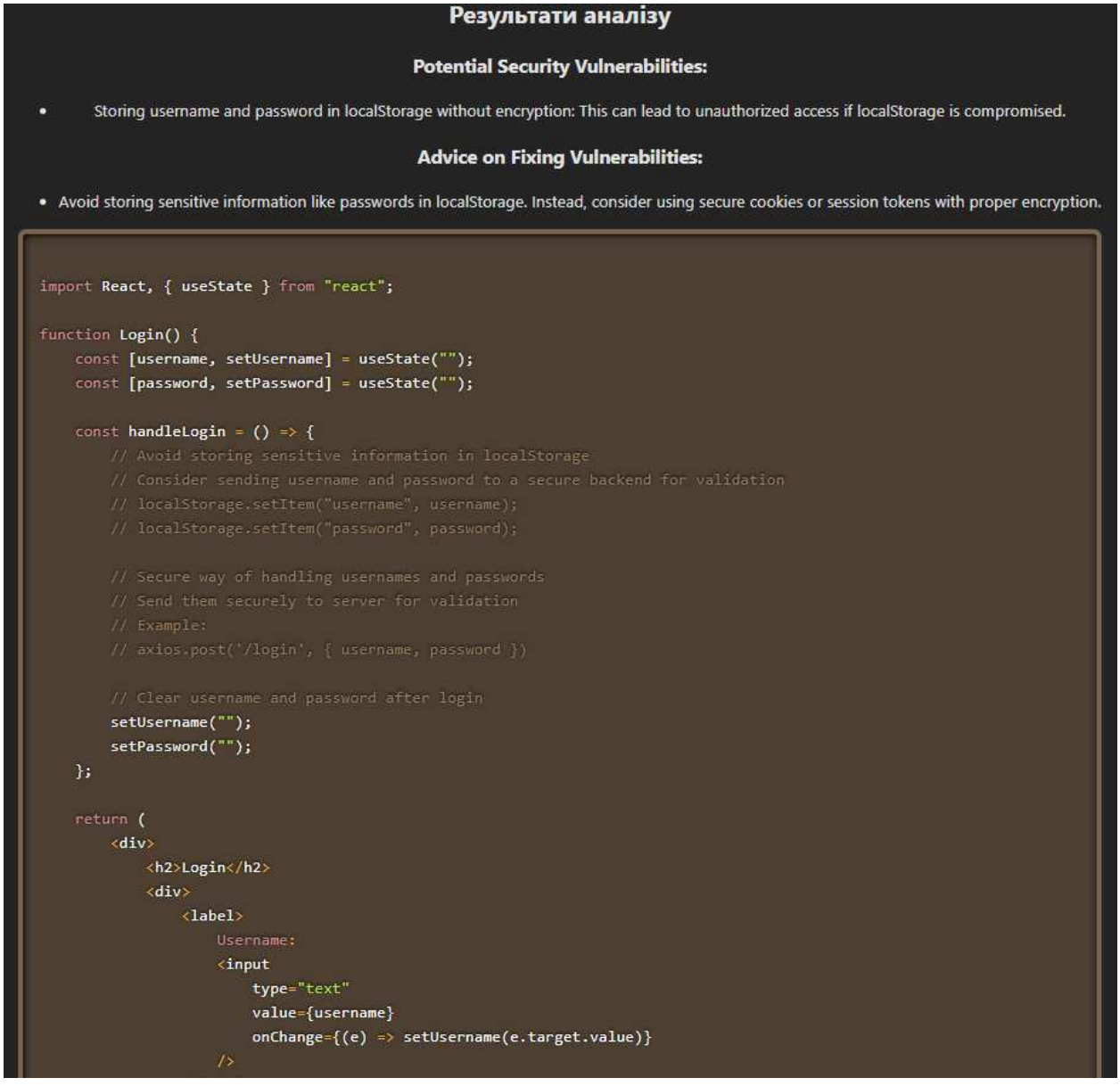


Рисунок 3.10 - Результат виконання аналізу коду, зображеного на рисунку 3.9

3.3 Результат роботи існуючих аналізаторів та порівняння з розробленим методом захисту

Для порівняння результатів статичного аналізу з розробленим методом було обрано безкоштовні версії чотирьох статичних аналізаторів безпеки застосунків: Aikido, Codacy, Snyk та Semgrep. Для порівняння були використані ті ж компоненти з вразливостями, що і для перевірки роботи розробленого методу.

Статичний аналізатор Aikido виявив тільки одну вразливість - зберігання конфіденційної інформації у програмному коді. Для виявленої вразливості були описані загальні рекомендації щодо її виправлення.

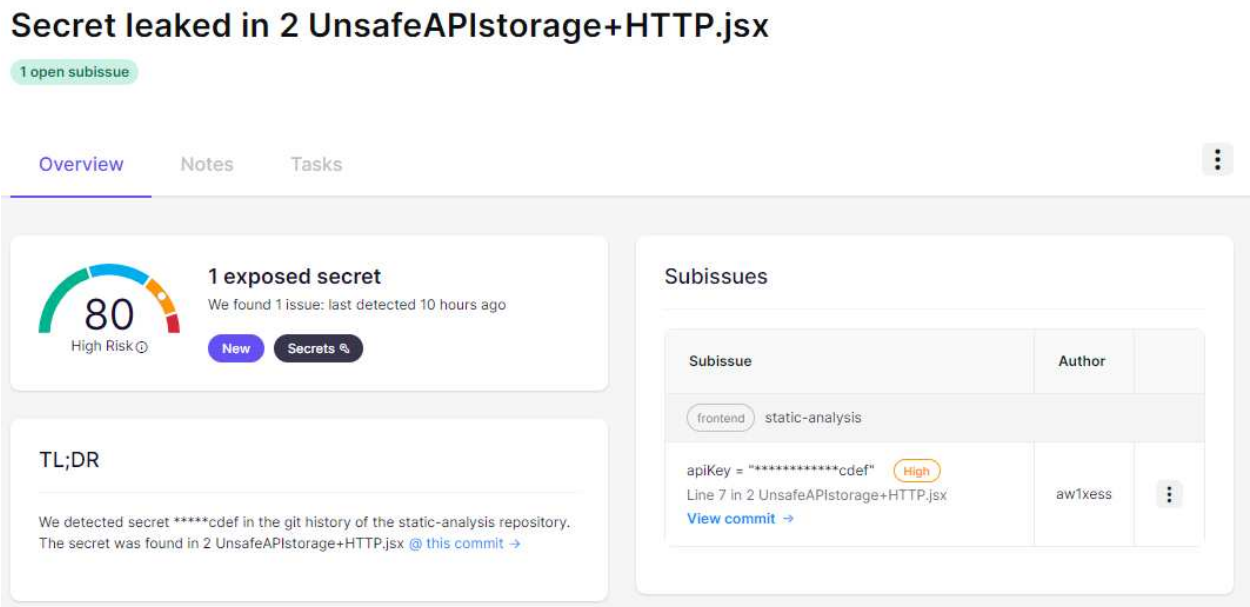


Рисунок 3.11 – Результат сканування аналізатором Aikido

Аналізатор Codacy виявив дві вразливості: використання незахищених каналів комунікації та вразливість XSS через використання атрибуту dangerouslySetInnerHTML. Для цих двох вразливостей аналізатор також надав загальні рекомендації щодо їх виправлення.

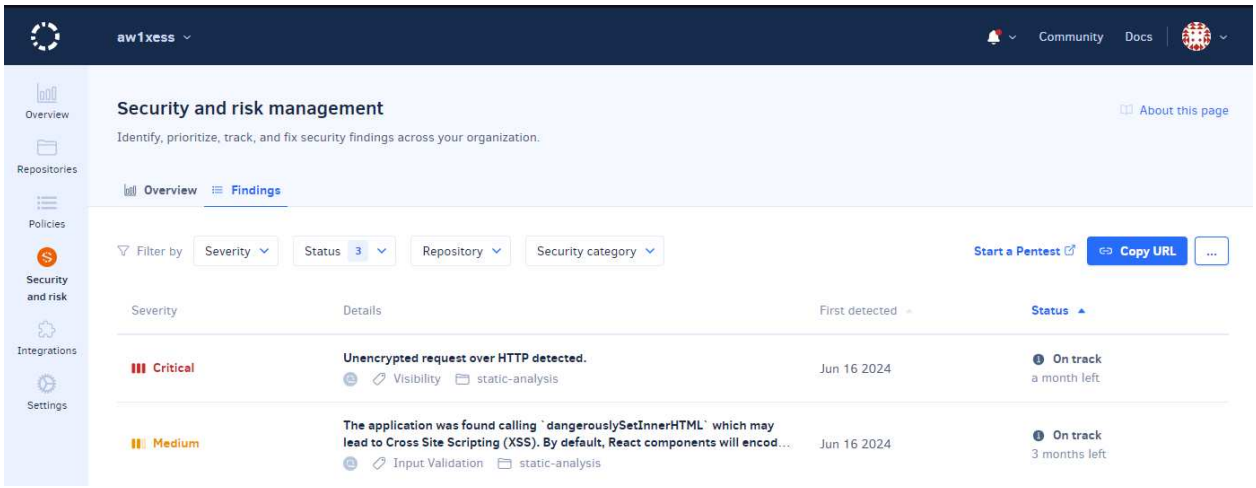


Рисунок 3.12 – Результат сканування аналізатором Codacy

Аналізатор Snyk також виявив дві вразливості – зберігання конфіденційної інформації у програмному коді та вразливість XSS через використання атрибуту dangerouslySetInnerHTML. Для обох вразливостей були надані рекомендаціїї для їх усунення.

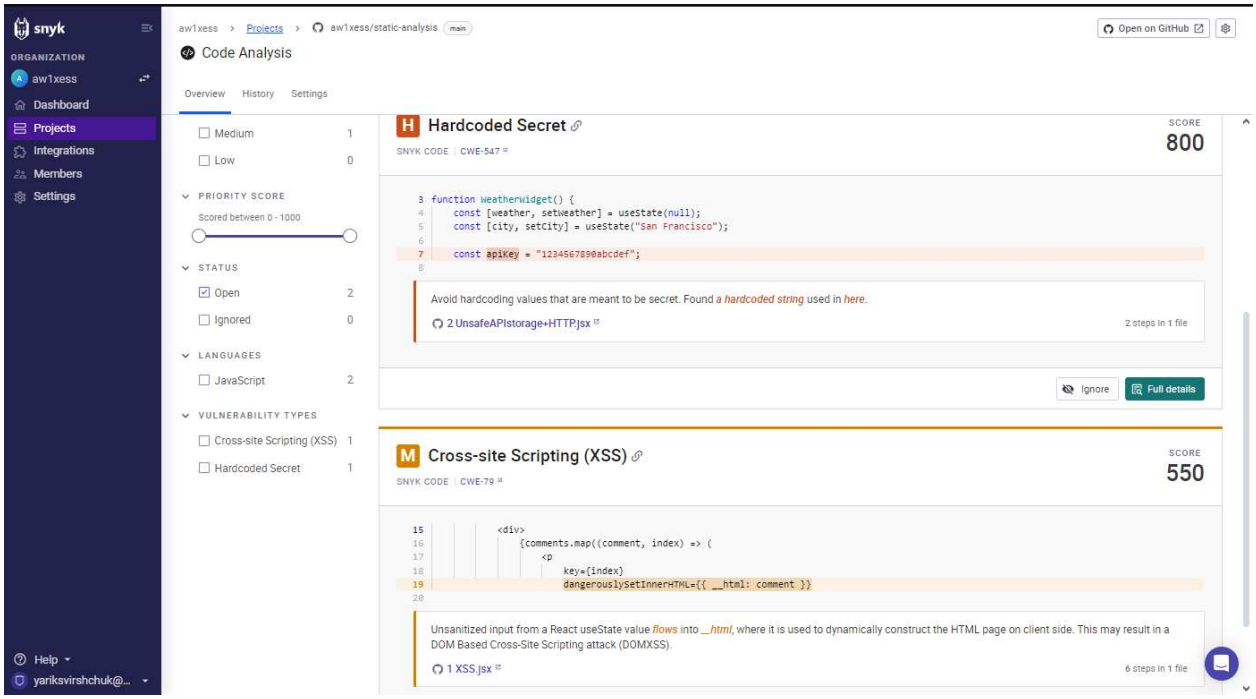


Рисунок 3.13 – Результат сканування аналізатором Snyk

Останній розглянутий аналізатор Semgrep знайшов тільки одну вразливість - використання незахищених каналів комунікації. Для знайденої вразливості були надані рекомендації з усунення.

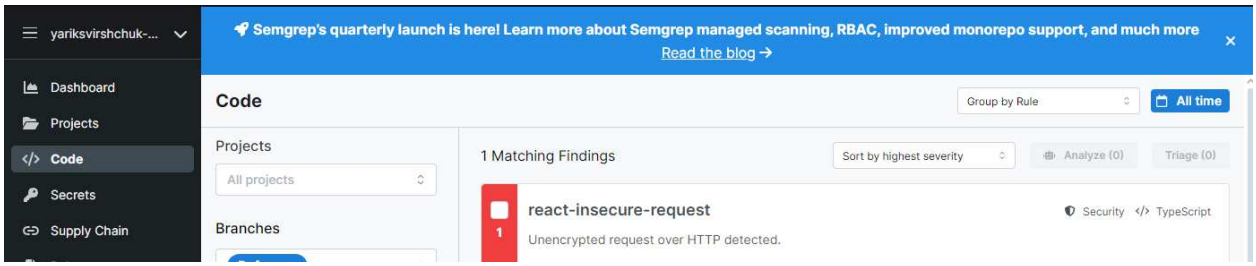


Рисунок 3.14 – Результат сканування аналізатором Semgrep

Отже, результати сканування вразливостей розробленим статичним аналізатором та існуючими можна сформувати у вигляді таблиці. В цій таблиці можна побачити, що існуючі аналізатори виявили не всі вразливості, що були виявлені розробленим статичним аналізатором. Варто зазначити, що для порівняння були використані безкоштовні та пробні версії аналізаторів, у яких відсутні деякі функції сканування, наявні лише у платній версії.

Таблиця 3.1 - Вразливості, виявлені розробленим та існуючими статичними аналізаторами безпеки веб-застосунків

	Розроблений аналізатор	Aikido	Codacy	Snyk	Semgrep
Вразливість XSS	+		+	+	
Неналежне зберігання конфіденційної інформації в клієнтській частині веб-застосунку	+	+		+	
Неналежне зберігання конфіденційної інформації в localStorage	+				
Використання незахищеного способу комунікації з сервером	+		+		+
Відсутність CSRF-токену при запиті до серверу	+				

Хоча й розроблений метод виявив усі вразливості у наданих прикладах, аналіз за допомогою штучного інтелекту не завжди видає правильні результати сканування. Для того, щоб оцінити наскільки часто розроблений аналізатор видає False Positive чи False Negative результати, було створено 25 вразливих компонентів та 25 безпечних для кожної з вразливостей, та було проведено аналіз кожного по 5 разів. Результати оформлені в таблицю 3.2.

Таблиця 3.2 – Статистика виявлених вразливостей розробленим статичним аналізатором

	Кількість аналізів	Вірно визначено	False Positive	False Negative
Вразливість XSS	50	48	2	0
Неналежне зберігання конфіденційної інформації в клієнтській частині веб-застосунку	50	41	4	5
Неналежне зберігання конфіденційної інформації в localStorage	50	42	6	2
Використання незахищеного способу комунікації з сервером	50	47	2	1
Відсутність CSRF-токену при запиті до серверу	50	40	7	3

Можемо бачити, що розроблений статичний аналізатор не у всіх випадках правильно спрацьовував, та для деяких небезпечних компонентів не були виявлені вразливості, які в них знаходились, а для безпечних – були визначені вразливості, яких насправді там немає. Однак, результати аналізу є доволі точними, на мою думку, через наявність додаткового контексту для запиту до моделі штучного інтелекту.

Висновки до розділу 3

У даному розділі були описані етапи розробки статичного аналізатора безпеки коду React з використання моделі OpenAI «gpt-3.5-turbo». У ході перевірки функціональності аналізатора були розглянуті приклади реалізації основних вразливостей затосунків на фреймворку React. Статичний аналізатор виявив усі розглянуті вразливості та навів рекомендації для їх усунення. Перевагою використання штучного інтелекту є гнучкість конфігурації запиту, за допомогою якої можна налаштувати різні аспекти перевірки, а також можна дописати додатковий контекст веб-застосунку, щоб штучний інтелект міг точніше зрозуміти, що відбувається в тому чи іншому компоненті, для якого проводиться перевірка.

Також було проведене порівняння розробленого статичного аналізатору безпеки веб-застосунків та існуючих. Жоден з існуючих аналізаторів не виявив усі вразливості, однак причиною цього може бути безкоштовна версія аналізаторів, яка може включати не всі перевірки, які доступні у платній версії. Розроблений аналізатор виявив усі вразливості, наявні у протестованих компонентах, однак аналіз за допомогою штучного інтелекту не завжди видає правильні результати, тому кожен результат аналізу повинен додатково перевірятися користувачем аналізатору.

ВИСНОВКИ

Було проведено аналіз найпоширеніших вразливостей та загроз для веб-застосунків створених на будь-яких технологіях, та були обрані вразливості, притаманні застосункам на React. Також були проаналізовані наявні методи безпеки, які включають вбудовані методи, сторонні бібліотеки та комплексні застосунки, такі як статичний та динамічний аналізатори безпеки коду. Результатом роботи є розроблений на фреймворкі React метод захисту для веб-застосунків заснованих на React. Створений метод захисту – це статичний аналізатор безпеки застосунків з використання штучного інтелекту, який переглядає програмний код, який був введений користувачем, виявляє можливі вразливості у коді та надає рекомендації щодо усунення проблем безпеки, разом з рекомендованим програмним кодом, де знайдені вразливості усунені. Також було проведене порівняння створеного статичного аналізатору безпеки веб-застосунків з існуючими.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Most used web frameworks among developers worldwide, as of 2023 – Statista [Електронний ресурс] - https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/
2. WPScan 2024 Website Threat Report [Електронний ресурс] - https://wpscan.com/2024-website-threat-report/
3. React State Management in 2024 [Електронний ресурс] - https://dev.to/nguyenhongphat0/react-state-management-in-2024-5e7l
4. React Reference [Електронний ресурс] - https://react.dev/reference/react
5. MDN References [Електронний ресурс] - https://developer.mozilla.org/en-US/docs/Web
6. ReactJS Virtual DOM [Електронний ресурс] - https://www.geeksforgeeks.org/reactjs-virtual-dom/
7. Cross-site scripting – PortSwigger [Електронний ресурс] - https://portswigger.net/web-security/cross-site-scripting
8. Cross Site Request Forgery (CSRF) – OWASP [Електронний ресурс] - https://owasp.org/www-community/attacks/csrf
9. Cross-Site Request Forgery Prevention Cheat Sheet – OWASP [Електронний ресурс] - https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html
10. SQL injection – PortSwigger [Електронний ресурс] - https://portswigger.net/web-security/sql-injection
11. Insecure direct object references (IDOR) – PortSwigger [Електронний ресурс] - https://portswigger.net/web-security/access-control/idor
12. Broken Authentication: Impact, Examples, and How to Fix It [Електронний ресурс] - https://brightsec.com/blog/broken-authentication-impact-examples-and-how-to-fix-it/

13. Clickjacking (UI redressing) – PortSwigger [Електронний ресурс] - https://portswigger.net/web-security/clickjacking
14. What is remote code execution? – Cloudflare [Електронний ресурс] - https://www.cloudflare.com/learning/security/what-is-remote-code-execution/
15. Content Security Policy (CSP) [Електронний ресурс] - https://developer.mozilla.org/ru/docs/Web/HTTP/CSP
16. cure53/DOMPurify – GitHub [Електронний ресурс] - https://github.com/cure53/DOMPurify
17. SAST & DAST: який метод тестування безпеки кращий? [Електронний ресурс] - https://qagroup.com.ua/publications/sast-dast-iakyj-metod-testuvannia-bezpeky-krashchyj/
18. SonarQube 10.4 Documentation [Електронний ресурс] - https://docs.sonarsource.com/sonarqube/10.4/
19. What is Dynamic Application Security Testing (DAST)? [Електронний ресурс] - https://www.opentext.com/what-is/dast

ДОДАТОК А

main.jsx

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
import './index.css'

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

App.jsx

```
import React from "react";
import Analyzer from "./Analyzer";
import "./App.css";
// import Example from "./Example";

function App() {
    return (
        <div className="App">
            <header className="App-header">
                <Analyzer />
            </header>
        </div>
    );
}

export default App;
```

Analyzer.jsx

```
import React, { useState } from "react";
import OpenAI from "openai";
import { Prism as SyntaxHighlighter } from "react-syntax-highlighter";
import { dark } from "react-syntax-highlighter/dist/esm/styles/prism";

const OPENAI_API_KEY = import.meta.env.VITE_OPENAI_API_KEY;

const openai = new OpenAI({
  baseURL: "https://api.naga.ac/v1",
  apiKey: OPENAI_API_KEY,
  dangerouslyAllowBrowser: true,
});

const analyzeCodeWithOpenAI = async (code) => {
  try {
    const response = await openai.chat.completions.create({
      messages: [
        {
          role: "user",
          content: `You are given with front-end React code: \n\n${code}\n. If
text after "You are given with front-end React code:"
          is not a React code - just write a string "Not a React code" as an
answer. Else analyze this React code for
          potential front-end security vulnerabilities. This vulnerabilities may
include but not limited to: using dangerouslySetInnerHTML
          without proper sanitization; unsanitized user input; unsafe confidential
information storage in code or localStorage;
          using HTTP in requests over HTTPS; not using CSRF tokens in
requests to server if based on app context, request may perform
```

```
some action on server side, like changing user password, making a transaction in online bank e.t.c.

As an answer, return HTML formatted document with indicated vulnerabilities and pieces of advice, how to fix them.

After that write secure code, wrapped in <pre> HTML tag, where all indicated issues are fixed.

In answer use HTML tag headers <h3> and lower.

Consider that back-end server does not have any security measures and you must offer only client-side solution.`,
            },
          ],
          model: "gpt-3.5-turbo",
        });
        console.log(response.choices[0]);
        return response.choices[0].message.content;
    } catch (error) {
        console.error("Error calling OpenAI API:", error);
        return "Error analyzing code";
    }
};

const Analyzer = () => {
    const [code, setCode] = useState("");
    const [analysis, setAnalysis] = useState("");

    const handleAnalyze = async () => {
        const result = await analyzeCodeWithOpenAI(code);
        setAnalysis(result);
    };
```

```
return (
  <div>
    <h1>Статичний аналізатор коду React</h1>
    <textarea
      rows="40"
      cols="100"
      value={code}
      onChange={(e) => setCode(e.target.value)}
      placeholder="Встав сюди свій React код"
    ></textarea>
    <br />
    <button onClick={handleAnalyze}>Аналізувати</button>
    <h2>Результати аналізу</h2>
    <div>
      <div
        dangerouslySetInnerHTML={{
          __html: analysis.replace(/<pre>([\s\S]*?)<\/pre>/, ""),
        }}
      ></div>
      <SyntaxHighlighter language="javascript" style={dark}>
        {analysis && analysis.match(/<pre>([\s\S]*?)<\/pre>/)
          ? analysis
              .match(/<pre>([\s\S]*?)<\/pre>/)[0]
              .replace(/<\/?pre>/g, "")
              .replace(/&lt;/g, "<")
              .replace(/&gt;/g, ">")
          : ""}
      </SyntaxHighlighter>
    </div>
  </div>
```

```
    );
};

export default Analyzer;
```

1 XSS.jsx

```
import React, { useState } from "react";

function CommentSection() {
    const [comments, setComments] = useState([]);
    const [newComment, setNewComment] = useState("");

    const handleAddComment = () => {
        setComments([...comments, newComment]);
        setNewComment("");
    };

    return (
        <div>
            <h2>Comments</h2>
            <div>
                {comments.map((comment, index) => (
                    <p
                        key={index}
                        dangerouslySetInnerHTML={{ __html: comment }}
                    />
                ))}
            </div>
            <input
                type="text"
```

```
            value={newComment}
            onChange={(e) => setNewComment(e.target.value)}
            placeholder="Add a comment"
          />
          <button onClick={handleAddComment}>Add Comment</button>
        </div>
    );
}

export default CommentSection;
```

2 UnsafeAPIstorage+HTTP.jsx

```
import React, { useState, useEffect } from "react";

function WeatherWidget() {
    const [weather, setWeather] = useState(null);
    const [city, setCity] = useState("San Francisco");

    const apiKey = "1234567890abcdef";

    useEffect(() => {
        if (apiKey) {
            fetch(
                `http://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${a
piKey}`
            )
                .then((response) => response.json())
                .then((data) => setWeather(data));
        } else {
            console.error("API key is missing");
```

```
    }
  }, [city, apiKey]);

  return (
    <div>
      <h2>Weather in {city}</h2>
      {weather && (
        <div>
          <p>Temperature: {weather.main.temp} °C</p>
          <p>Condition: {weather.weather[0].description}</p>
        </div>
      )}
    </div>
  );
}

export default WeatherWidget;
```

3 NoCSRFtoken.jsx

```
import React, { useState } from "react";

function ChangePassword() {
  const [newPassword, setNewPassword] = useState("");

  const handleChangePassword = () => {
    fetch("http://example.com/api/change-password", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
```

```
        body: JSON.stringify({
          newPassword: newPassword,
        }),
      })
        .then((response) => response.json())
        .then((data) => {
          if (data.success) {
            alert("Password changed successfully!");
          } else {
            alert("Error changing password.");
          }
        })
        .catch((error) => {
          console.error("Error:", error);
          alert("Error changing password.");
        });
    };

    return (
      <div>
        <div>
          <label>
            New Password:
            <input
              type="password"
              value={newPassword}
              onChange={(e) => setNewPassword(e.target.value)}
            />
          </label>
        </div>
```

```
        <button onClick={handleChangePassword}>Change Password</button>
      </div>
    );
}

export default ChangePassword;
```

4 localStorage.jsx

```
import React, { useState } from "react";

function Login() {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");

  const handleLogin = () => {
    localStorage.setItem("username", username);
    localStorage.setItem("password", password);
  };

  return (
    <div>
      <h2>Login</h2>
      <div>
        <label>
          Username:
          <input
            type="text"
            value={username}
            onChange={(e) => setUsername(e.target.value)}
          />
```

```
            </label>
          </div>
          <div>
            <label>
              Password:
              <input
                type="password"
                value={password}
                onChange={(e) => setPassword(e.target.value)}
              />
            </label>
          </div>
          <button onClick={handleLogin}>Login</button>
        </div>
    );
}

export default Login;
```