МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

ОРЛІВСЬКА ВАЛЕРІЯ ОЛЕГІВНА

Допускається до захисту:
В.о. завідувача кафедри
прикладної математики та
кібербезпеки,
_____Луценко А. В.
«__»_____20__р.

# МЕТОДИ І СПОСОБИ ВИЯВЛЕННЯ ТА БОРОТЬБИ З АТАКАМИ ТИПУ "МІЖСАЙТОВА ПІДРОБКА ЗАПИТІВ"

Спеціальність 125 Кібербезпека
Кваліфікаційна (бакалаврська) робота

Науковий керівник:
Загоруйко Л. В.,
к.т.н., доцент, доцент кафедри
інформаційних технологій
_____
(підпис)

Оцінка : ___/ ___ / _____
(бали/за шкалою ЄКТС/за національною шкалою)
Голова ЕК: _____
(підпис)

Вінниця – 2025

## АНОТАЦІЯ

**Орлівська В.О. Методи і способи виявлення та боротьби з атаками типу "міжсайтова підробка запитів".** Спеціальність 125 "Кібербезпека". Донецький національний університет імені Василя Стуса, Вінниця, 2025.

У кваліфікаційній (бакалаврській) роботі досліджено ефективність існуючих методів захисту від атак типу "міжсайтова підробка запитів" у вебсередовищі. Розроблено повноцінний браузерний плагін, призначений для виявлення, запобігання та блокування атак типу "міжсайтова підробка запитів" шляхом аналізу HTTP-запитів, перевірки відповідності політик безпеки, використання захисних токенів та візуалізації інформації про ризики для кінцевого користувача.

Ключові слова: атаки типу "міжсайтова підробка запитів", браузерний плагін, веб-безпека, HTTP-запити, токени безпеки, вебзастосунки, клієнтський захист, кібербезпека, міжсайтові атаки, перехоплення запитів, аналіз заголовків. 90 с., 0 табл., 2 рис., 25 джерел

## ANNOTATION

**Orlivska V.O. Methods and means of detecting and combating cross-site request forgery attacks.** Specialty 125 "Cybersecurity". Vasyl Stus Donetsk National University, Vinnytsia, 2025.

The qualification (bachelor's) thesis investigated the effectiveness of existing protection methods against cross-site request forgery (CSRF) attacks in the web environment. A full-fledged browser plugin has been developed to detect, prevent, and block cross-site request forgery attacks by analyzing HTTP requests, checking security policy compliance, using security tokens, and visualizing risk information for the end user.

Keywords: cross-site request forgery attacks, browser plugin, web security, HTTP requests, security tokens, web applications, client-side protection, cybersecurity, cross-site attacks, request interception, header analysis. 90 p., 0 tables, 2 figures, 25 sources

ЗМІСТ

# СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

AJAX-запити (Asynchronous JavaScript and XML) – асинхронні JavaScript-запити. Технологія для обміну даними з сервером без перезавантаження сторінки.

Angular (Angular Framework) – фреймворк Angular. Платформа для створення односторінкових вебзастосунків на JavaScript/TypeScript.

Chrome Storage API – API збереження Chrome. Інтерфейс для збереження даних розширення у браузері Chrome.

chrome.storage.local – локальне сховище Chrome. Область збереження даних плагіна на пристрої користувача.

chrome.storage.onChanged – слухач змін сховища. Подія, яка реагує на зміни у chrome.storage.

Content Scripts API – API скриптів контенту. Дозволяє плагіну взаємодіяти з вмістом вебсторінок.

Content Security – безпека вебконтенту. Сукупність заходів для захисту ресурсу від атак типу XSS, CSRF.

Content Security Policy (CSP) – політика безпеки контенту. Механізм контролю джерел завантаження контенту на сторінці.

Cookie (HTTP Cookie) – HTTP куки. Маленький файл, що зберігається у браузері для автентифікації та збереження сесій.

CSRF (Cross-Site Request Forgery) – міжсайтове підроблення запитів. Атака, яка змушує користувача виконати несанкціоновану дію на сайті.

CSRF Detector – Виявлювач CSRF. Назва плагіна, що автоматично виявляє та блокує атаки CSRF.

Django (Django Framework) – фреймворк Django. Python-фреймворк для швидкої розробки вебдодатків.

Експлойт (Exploit) – експлойт. Фрагмент коду або програма, яка використовує вразливості для атак.

Fetch API-запити (Fetch Application Programming Interface) – запити через Fetch API. Сучасний інтерфейс для відправки HTTP-запитів із браузера.

FormData (Form Data Object) – об’єкт формових даних. Інтерфейс для формування тіла запиту у форматі форми.

HTML (HyperText Markup Language) – мова розмітки гіпертексту. Основна мова для створення структури вебсторінок.

JavaScript (JS) – JavaScript. Мова програмування для взаємодії з вебсторінкою в реальному часі.

JSON (JavaScript Object Notation) – об’єктна нотація JavaScript. Формат обміну структурованими даними.

Application Boundary Enforcer Module – модуль обмеження меж застосунку. Компонент, що контролює вихід HTTP-запитів за межі дозволених доменів.

Origin (Request Origin Header) – заголовок джерела. Визначає джерело HTTP-запиту для контролю політик безпеки.

Плагін (Browser Extension/Plugin) – браузерне розширення. Додатковий компонент, що розширює функціонал браузера.

Referer (HTTP Referer Header) – заголовок реферера. Вказує, з якої сторінки був здійснений перехід.

RequestPolicy (Request Policy Extension) – політика запитів. Розширення браузера для обмеження міжсайтових запитів.

Ruby on Rails (RoR Framework) – фреймворк Ruby on Rails. Популярна платформа для розробки вебдодатків мовою Ruby.

SameSite (SameSite Cookie Attribute) – атрибут SameSite. Визначає політику відправки cookie у міжсайтових запитах.

SweetAlert (SweetAlert Library) – бібліотека SweetAlert. Інструмент для створення привабливих повідомлень у браузері.

Synchronizer Token Pattern – шаблон синхронізації токенів. Захисний механізм, при якому у формі передається унікальний токен.

Synchronizer Token Pattern with Cookie – шаблон синхронізації з cookie. Варіант захисту, який комбінує cookie і токен у запиті.

XML (eXtensible Markup Language) – розширювана мова розмітки. Формат для представлення структурованих даних.

XMLHttpRequest (XHR) – запит XMLHttpRequest. Старий, але досі використовуваний інтерфейс для відправки HTTP-запитів з JavaScript.

# ВСТУП

*Актуальність роботи.* Інтенсивний розвиток веб-технологій зумовлює не лише зручність та функціональність сучасних веб-застосунків, але й зростання кількості кіберзагроз. Серед них особливої уваги потребують атаки типу "міжсайтова підробка запитів", які залишаються одними з найпоширеніших та найнебезпечніших загроз у веб-безпеці. Подібні атаки дозволяють зловмиснику виконувати несанкціоновані дії від імені користувача, що призводить до втрати конфіденційних даних, фінансових збитків, порушення працездатності інформаційних систем та шкоди репутації компаній.

Незважаючи на наявність різних методів та засобів захисту, їх ефективність часто залишається обмеженою через складність впровадження, недостатню гнучкість, а також необхідність суттєвої модифікації існуючих вебзастосунків. Крім того, переважна більшість рішень спрямована виключно на автоматизований захист без залучення безпосередньо користувачів, що знижує загальний рівень обізнаності у сфері інформаційної безпеки.

Тому розробка браузерного плагіна, який поєднуватиме в собі автоматичні механізми виявлення та запобігання атакам із навчальною інтерактивною складовою, є актуальною. Таке рішення дозволить не лише ефективно блокувати загрози, але й підвищить загальну обізнаність користувачів щодо питань кібербезпеки, формуючи навички правильного реагування на потенційні загрози.

*Метою* кваліфікаційної (бакалаврської) роботи є дослідження ефективності існуючих методів захисту. Розробка повноцінного браузерного плагіну, призначеного для виявлення, запобігання та блокування атак типу "міжсайтова підробка запиті" шляхом аналізу HTTP-запитів, перевірки відповідності політик безпеки, використання захисних токенів та візуалізації інформації про ризики для кінцевого користувача.

*Завдання роботи:*

1. Проаналізувати атаки типу "міжсайтова підробка запитів" та існуючі методи захисту.
2. Дослідити існуючі браузерні плагіни, визначити їх переваги та недоліки.
3. Визначити основні вимоги до Chrome-плагіна для боротьби з цими атаками.
4. Обрати оптимальні технології для реалізації плагіна.
5. Розробити інтерактивний Chrome-плагін, який виявлятиме, блокуватиме атаки та навчатиме користувачів.
6. Реалізувати аналіз HTTP-запитів, перевірку політик безпеки, захисні токени і візуалізацію ризиків.
7. Провести тестування плагіна та оцінити його ефективність.
8. Надати рекомендації щодо впровадження розробленого рішення у практичних умовах.

# РОЗДІЛ 1. АНАЛІЗ ОСОБЛИВОСТЕЙ АТАК ТИПУ "МІЖСАЙТОВА ПІДРОБКА ЗАПИТІВ" ТА СУЧАСНИХ ПІДХОДІВ ДО ЗАХИСТУ ВІД НИХ

## 1.1 Огляд атак типу "міжсайтова підробка запитів"

Міжсайтова підробка запитів – це один із найдавніших і водночас найпростіших видів вебатак, який досі залишається ефективним проти багатьох застосунків і може призводити до дуже серйозних наслідків – зокрема фінансових втрат чи компрометації акаунтів користувачів [1].

Атака була вперше виявлена на початку 2000-х років і відтоді стабільно входить до переліку найбільших загроз у веббезпеці [1][3]. За своєю суттю атака типу "міжсайтова підробка запитів" обманом змушує браузер жертви виконати небажаний запит до стороннього вебсайту від імені автентифікованого користувача. Іншими словами, зловмисник використовує авторизаційні дані жертви (наприклад, її сесійний cookie), що автоматично додаються браузером до запитів, для виконання дій на довірливому сайті без відома і наміру самого користувача [2].

Важливо, що при цьому порушується модель довіри: якщо у випадку атака міжсайтовий скриптинг експлуатує довіру користувача до нібито безпечного сайту, то атака типу "міжсайтова підробка запитів" експлуатує довіру самого вебсайту до браузера користувача та його автентифікації [2]. Саме тому, не маючи додаткових механізмів верифікації, уразливий сервер не в змозі відрізнити шкідливий запит, ініційований непомітно через браузер жертви, від легітимної дії самого користувача.

У результаті вебсайт, який атакували, виконує підроблений запит так, ніби його самостійно надіслав автентифікований користувач, хоча насправді ініціатором був зловмисник.

Механізм атаки типу "міжсайтова підробка запитів" полягає в тому, що жертву будь-яким шляхом спонукають завантажити сторінку, контрольовану людиною, що атакує (або вставлений нею елемент на іншому ресурсі), яка

непомітно для користувача відправляє HTTP-запит до цільового вебзастосунку. Такий запит містить дію, обрану зловмисником (наприклад, переказ грошей, зміна налаштувань профілю чи публікація контенту), і браузер автоматично додає до нього всі потрібні ідентифікаційні дані користувача – зокрема cookie поточної сесії на цільовому сайті [2][6]. Якщо користувач на цей момент авторизований на сайті-жертві, то сервер приймає запит як легітимний. Ключова передумова успіху – відсутність у застосунку додаткової перевірки легітимності запиту.

Вразливими є ті вебсайти, які покладаються лише на наявність сесійного cookie для автентифікації дії і не впроваджують жодних інших методів перевірки запитів [6]. За таких умов кожен запит, що надходить зі сторони браузера, вважається довіреним – навіть якщо його згенеровано приховано на сторонньому (шкідливому) сайті. Зазвичай атака вимагає, щоб жертва сама здійснила певний тригер (наприклад, клацнула на спеціально сформоване посилання чи відвідала сторінку з вбудованим шкідливим кодом), тобто застосовується соціальна інженерія. Проте після цього всі подальші дії виконуються автоматично: прихований код (наприклад, HTML-форма або скрипт) відправляє HTTP-запит до вразливого сайту, використовуючи сесійні дані жертви, і тим самим "вмовляє" сервер здійснити небажану дію. Атаки типу "міжсайтова підробка запитів" зазвичай націлені на виконання саме тих операцій, що змінюють дані чи стан облікового запису, а не просто на крадіжку інформації [1][2]. Це пояснюється тим, що зловмисник, не маючи доступу методом міжсайтовий скриптинг до відповіді сервера, не бачить результатів свого запиту – тобто не може безпосередньо викрасти через атаку типу "міжсайтова підробка запитів", скажімо, конфіденційні дані, які повертаються у відповіді. Натомість атака дозволяє змінювати дані на сервері від імені жертви.

Типові приклади потенційних шкідливих дій включають: нелегітимне здійснення фінансового переказу, зміну пароля або адреси електронної пошти облікового запису, додавання нового адміністратора системи, публікацію

повідомлення або навіть замовлення товару на ім’я жертви [2][6]. Наслідки залежать від прав користувача: якщо це звичайний клієнт, він може втратити кошти чи доступ до свого акаунту; якщо ж адміністратор – зловмисник може захопити контроль над усім застосунком [3].

У реальних випадках такі атаки спричиняли значні проблеми. Приміром, інцидент з інтернет-банкінгом ING Direct у 2008 році продемонстрував, що уразливість до атак типу "міжсайтова підробка запитів" у формі переказу коштів дозволяла непомітно перевести гроші з рахунків жертв на рахунки зловмисників [3].

Інший відомий випадок – вразливість на сайті YouTube, яка давала змогу через підроблені запити виконувати практично будь-які дії від імені користувачів, зокрема надсилати повідомлення їхнім контактам та змінювати налаштування [3].

Новіші приклади також підтверджують актуальність проблеми: у 2020 році вразливість до атак типу "міжсайтова підробка запитів" була виявлена в соціальній мережі TikTok, при чому її експлуатація дозволяла захопити обліковий запис жертви лише одним кліком – зловмисник міг підробленим запитом примусово встановити свій пароль для акаунту жертви і таким чином отримати повний доступ [6]. Компанія оперативно усунула цю уразливість, але факт її існування свідчить, що навіть провідні сучасні сервіси не застраховані від атак типу "міжсайтова підробка запитів".

Цікаво, що такі інциденти часто важко відстежити: у журналах сервера всі дії виглядають як виконані справжнім користувачем з його IP-адреси, тому виявити факт стороннього втручання непросто [3]. Через це про атаки типу "міжсайтова підробка запитів" рідко повідомляється публічно, хоча насправді їх кількість може бути значно більшою за зафіксовану [3].

Існують різні різновиди та сценарії таких атак. Класична ситуація — одноразова підробка окремого запиту від імені користувача (наприклад, відправлення форми з зміною налаштувань). Окремо виділяють login-атаки, коли жертву непомітно змушують увійти до системи під обліковим записом,

контрольованим зловмисником [3][7]. На перший погляд, це здається менш небезпечним (адже зловмисник лише нав’язує чужий логін, а не виконує дію в існуючій сесії жертви). Однак login-атака може мати підступні наслідки: якщо жертва не помітить підміни і продовжить користуватися системою, вводячи особисті дані у «зламаний» акаунт, зловмисник згодом отримає доступ до цієї інформації, просто увійшовши у свій обліковий запис [3][7].

Інший варіант — так звана "збережена" атака типу "міжсайтова підробка запитів", коли шкідливий код, що здійснює атаку, зберігається безпосередньо на сайті-жертві (наприклад, у профілі або коментарі). Хоча саме по собі вставлення коду в результаті атаки типу "міжсайтова підробка запитів" зазвичай неможливе без іншої вразливості (наприклад, такої як міжсайтовий скриптинг), іноді трапляються ситуації, коли HTML-елемент із зовнішнім запитом можна зберегти у відкритому полі (наприклад, тег IMG в описі профілю). Якщо адміністратор або інший користувач перегляне цей вміст, відбудеться атака типу "міжсайтова підробка запитів" без потреби переходити на сторонній сайт. Збережений підроблений запит збільшує вірогідність успішної експлуатації, оскільки жертва рано чи пізно сама відкриє сторінку зі вбудованим "троянським" елементом на довіреному сайті.

Ще один тип, що з’явився відносно недавно, — атака типу "міжсайтова підробка запитів" на боці клієнта. Атака націлена не на сервер, а на уразливість у сценаріях JavaScript самого застосунку. Зловмисник знаходить слабке місце у клієнтському коді — наприклад, таку функцію, що генерує запит на основі даних із адресного рядка або інших вхідних параметрів, і може підмінити ці параметри. У результаті браузер сам сформує запит до сервера, включно з дійсним токеном для захисту від атак типу "міжсайтова підробка запитів", якщо такий потрібен, але цей запит буде скеровано на адресу, обрану зловмисником. По суті, клієнтський код "обманом" змушують відправити свій запит не туди або не з тими даними, як задумано, тим самим обходячи захист [5].

Дослідження 2021 року показало, що у 87 із 106 проаналізованих сучасних вебзастосунків знайдено вразливості до атак типу "міжсайтова підробка запитів" на боці клієнта, причому для семи з них були продемонстровані повноцінні експлойти, які дозволяли виконувати критичні операції на сервері або навіть ініціювати міжсайтовий скриптинг і SQL-ін’єкції через цей вектор [5]. Така нова загроза ускладнює захист, адже класичні серверні методи (токени, перевірки cookie) не спрацюють, якщо в логіці фронтенду присутні помилки.

## 1.2 Методи та засоби захисту від атак типу "міжсайтова підробка запитів"

Основна причина вразливості вебзастосунків до атак типу "міжсайтова підробка запитів" полягає у тому, що протокол HTTP не забезпечує вбудованого механізму підтвердження намірів користувача при здійсненні запиту. Браузер автоматично надсилає автентифікаційні дані (cookie, заголовки авторизації тощо) до домену призначення незалежно від того, звідки прийшов запит — із справжньої взаємодії користувача чи з іншого сайту [6]. Якщо застосунок не вимагає додаткового підтвердження (наприклад, токена або перевірки джерела), він довіряє кожному запиту, що відповідає сесії користувача.

Слабкими місцями, які роблять можливими атаки типу "міжсайтова підробка запитів", найчастіше є: відсутність унікального токена (параметра-підтвердження) у критичних запитах; використання ідемпотентних методів (GET) для зміни стану; або ігнорування перевірки HTTP-заголовків джерела (Origin, Referer) на боці сервера. Так, згідно з емпіричними дослідженнями, щонайменше близько 10% сучасних вебсайтів все ще мають критичні дії, реалізовані через небезпечні GET-запити [5]. Це означає, що навіть із впровадженням протоколів захисту, про які йдеться далі, значна частка вебзастосунків залишається потенційно вразливою: людина, що атакує, може

викликати перехід браузера жертви за спеціально сформованим URL (через <img> або просте перенаправлення), і якщо за цим URL здійснюється зміна стану без додаткових перевірок – стан буде змінено. З іншого боку, навіть використання методу POST не гарантує безпечності, якщо розробники припустилися помилки.

Відомі випадки, коли сервер приймав той самий запит і по GET, і по POST (через некоректну маршрутизацію), або коли токеном для захисту від атак типу "міжсайтова підробка запитів" був передбачуваним чи спільним для всіх сесій – такі погрішності зводять нанівець захисні заходи. Для захисту від атак типу "міжсайтова підробка запитів" традиційно застосовуються спеціальні токени – випадкові та криптостійкі рядки, які сервер генерує для кожної сесії (чи навіть для кожного запиту) і очікує побачити у кожному критично важливому запиті (наприклад, у прихованому полі форми). Якщо токен відсутній або невірний, сервер відхиляє запит як потенційно підроблений. Такий підхід, відомий як Synchronizer Token Pattern, десятиліттями був стандартним рішенням і залишається ефективним, проте вимагає належної імплементації [6].

Важливо, щоб токени були достатньо випадковими, унікальними для сесії (а краще – для кожного запиту) і перевірялися строго на сервері.

Альтернативні або допоміжні методи захисту включають перевірку заголовка Referer/Origin (сервер переконується, що запит надійшов з власного сайту, а не стороннього – але цей метод може бути ненадійним, оскільки заголовки можуть бути відсутні або підроблені у деяких випадках), а також вимогу повторної автентифікації або підтвердження для чутливих операцій (наприклад, введення пароля чи CAPTCHA перед проведенням транзакції) [1]. Більшість сучасних вебфреймворків (Django, Ruby on Rails, Angular тощо) мають вбудовані засоби проти атак типу "міжсайтова підробка запитів", автоматично додаючи токени до форм або застосовуючи інші патерни захисту. Це суттєво знизило поширеність уразливості: за даними OWASP, вже у 2017 році атаки типу "міжсайтова підробка запитів" виявлялися лише приблизно у

5% протестованих застосунків, тоді як раніше входив до першої десятки найбільш розповсюджених проблем [1].

Проте повністю проблему не вирішено – як показали наведені вище приклади, помилки реалізації або недогляд можуть залишати "лазівки" навіть у застосунках, що загалом оснащені захистом.

Новітній крок у еволюції захисних заходів – це механізми на рівні браузера, зокрема атрибут SameSite для cookie. Вперше запроваджений у 2016 році і отримавши широку підтримку браузерів до 2020 року, атрибут SameSite дає змогу обмежити автоматичну пересилку cookie при крос-сайтових запитах. Значення SameSite=Lax (за замовчуванням у сучасних браузерах) забороняє надсилати сесійні cookie у більшості випадків переходу з чужого сайту, окрім деяких навігацій рівня верхньої сторінки (наприклад, прямого переходу по посиланню). Режим SameSite=Strict ще суворіший і повністю блокує cookie у будь-яких запитах, що походять від сторонніх ресурсів. Здавалося б, це мало б покласти край атакам типу "міжсайтова підробка запитів", адже якщо браузер не додасть сесійний cookie, запит з чужого сайту не буде авторизованим. На практиці ж цей механізм виявився не "панацеєю" [5].

По-перше, не всі розробники можуть дозволити собі виставити SameSite=Strict для своїх cookie, оскільки часто потрібна інтеграція зі сторонніми сервісами (авторизація через інші домени, віджети, платежі тощо). Значна частина сайтів вимушено використовує режим SameSite=None (що фактично повертає стару поведінку без обмежень) для коректної роботи своїх функцій [5].

По-друге, навіть SameSite=Lax залишає можливість крос-доменного запиту через пряме перенаправлення (GET-запит) – як зазначалося, чимало застосунків все ще помилково допускають виконання станозмінних дій методом GET, у тому числі в комерційних вебсервісах [5]. В таких випадках людині, що атакує, достатньо примусити браузер здійснити перехід на URL уразливої дії – і cookie будуть включені, оскільки це навігація верхнього рівня, дозволена політикою Lax. Практичні вимірювання ефективності SameSite

підтверджують, що цей захист покриває не всі сценарії атак типу "міжсайтова підробка запитів": зокрема, одна з робіт показала, що за умов правильного налаштування SameSite успішно блокує основний вектор атак, проте все ж вразливими лишаються нестандартні випадки, і як будь-який контрзахід, SameSite може бути обійдений при певному збігу обставин [5].

До того ж, існують способи обходу сумісності: деякі розробники задля підтримки застарілих браузерів встановлювали дублікати cookie – один з SameSite=None, інший з SameSite=Strict. У таких конфігураціях було виявлено нову уразливість: старі браузери ігнорують атрибут SameSite (відправляючи перший cookie завжди), а нові – ігнорують cookie без атрибуту (але приймають другий). В результаті в певних умовах сесія все одно може бути використана стороннім запитом, що було показано на прикладі сайтів GitHub, CNN, Yahoo та інших [5].

Нарешті, варто враховувати, що атаки типу "міжсайтова підробка запитів" еволюціонували – згадана раніше атака типу "міжсайтова підробка запитів" на боці клієнта взагалі не залежить від cookie (адже браузер виконує запит у контексті того ж сайту, тільки з модифікованими параметрами), тому SameSite тут не допоможе [5].

З цих причин експерти зазначають, що хоча поява атрибуту SameSite істотно підвищила безпеку і зменшила кількість успішних атак, проблема не втратила актуальності [5]. Вебзастосунки і досі потребують належного впровадження перевірок на рівні сервера (токенів, валідатора джерела тощо) та ретельного тестування на атаки типу "міжсайтова підробка запитів". Уразливість залишається небезпечною, попри свій поважний "вік", адже достатньо єдиної забутої перевірки або нестандартного сценарію – і зловмисники зможуть скористатися довірою між браузером і сайтом для проведення атаки [1][5].

Висновок до розділу 1

Аналіз атак типу "міжсайтова підробка запитів" засвідчив, що ці атаки залишаються серйозною загрозою для вебзастосунків навіть попри впровадження сучасних технологій захисту. Їхня небезпека полягає у використанні моделі довіри між браузером користувача та вебсервером, що дозволяє виконувати несанкціоновані дії від імені автентифікованого користувача без його прямої участі. Попри те, що найефективнішими контрзаходами протягом десятиліть були використання синхронізованих токенів і перевірка заголовків Origin та Referer, на практиці ці заходи не завжди впроваджуються коректно, що створює численні точки уразливості.

Особливу увагу варто звернути на новітні вектори атак, зокрема атаки типу "міжсайтова підробка запитів" на боці клієнта, що обходять традиційні серверні методи захисту. Водночас досвід реальних інцидентів демонструє обмежену ефективність навіть таких рішень, як атрибут SameSite для cookie, який хоч і значно зменшує площу атаки, однак не гарантує повного усунення ризиків. Аналіз існуючих методів захисту підтверджує, що комплексний підхід, який поєднує серверні, клієнтські та освітні заходи, є найбільш ефективним способом протидії атакам даного типу.

Таким чином, захист від атак типу "міжсайтова підробка запитів" залишається багаторівневою проблемою, що потребує не лише технічних рішень, а й підвищення обізнаності розробників і користувачів щодо потенційних ризиків і сучасних методів захисту. Це підкреслює актуальність подальших досліджень та розробки додаткових інструментів, зокрема браузерних плагінів, здатних посилити безпеку на стороні клієнта.

# РОЗДІЛ 2. АНАЛІЗ ПРОТОТИПІВ ТА ВИЗНАЧЕННЯ ОСНОВНИХ ВИМОГ ДО CHROME-ПЛАГІНА ДЛЯ БОРОТЬБИ З АТАКАМИ ТИПУ "МІЖСАЙТОВОЇ ПІДРОБКИ ЗАПИТІВ"

## 2.1 Огляд існуючих підходів до використання плагінів для боротьби з атаками типу "міжсайтової підробки запитів"

Атаки типу «міжсайтова підробка запитів» визнані одними з ключових загроз веббезпеці, що стимулювало розвиток як серверних, так і клієнтських рішень для протидії їм. У наукових публікаціях останніх років дослідники пропонують браузерні плагіни як додатковий рівень захисту, що працює повністю на боці користувача. Наприклад, у дослідженні 2019 року описано інструмент CSRF Detector для Chrome – клієнтський модуль, що аналізує вебзапити та вміст сторінок для виявлення типових шаблонів атак типу «міжсайтова підробка запитів». У результаті експериментальної перевірки плагін виявив усі змодельовані атаки без хибнопозитивних результатів, продемонструвавши принципову життєздатність підходу [7].

Поширеним технічним рішенням серед браузерних розширень є застосування політики заборони міжсайтових запитів за замовчуванням. Такі інструменти, як RequestPolicy для Firefox або uMatrix, блокують міждоменні HTTP-запити, доки користувач явно не дозволить взаємодію. Цей метод практично унеможливлює несанкціоновані дії зловмисника на сторонніх серверах, що робить його ефективним проти атак типу «міжсайтова підробка запитів». Однак такий підхід має значний недолік: істотне порушення функціональності легітимних вебзастосунків, які часто потребують завантаження контенту з інших доменів [8].

Ще один метод реалізації клієнтського захисту – фільтрація автентифікаційних даних у крос-доменних запитах. Розширення CsFire (Firefox) видаляє cookie з усіх міжсайтових HTTP-запитів, забезпечуючи блокування потенційних атак без порушення основної функціональності

сайтів. Подібну стратегію застосовує No-CSRF у Chrome, що вилучає сесійні дані з міждоменних POST-запитів і надає користувачу інтерфейс для перегляду журналу таких запитів, реалізуючи діагностичну функцію [9].

Плагіни типу NoScript поєднують фільтрацію виконання скриптів з додатковими модулями захисту від атак типу «міжсайтова підробка запитів». Наприклад, модуль Application Boundary Enforcer запобігає надсиланню HTTP-запитів до локальних адресних просторів (наприклад, localhost), що особливо важливо для протидії атакам на внутрішні сервіси користувача [8]. Такий підхід мінімізує ризики атак через крос-доменні POST-запити, що приховано виконуються шкідливими сторінками.

Крім суто блокуючих засобів, існують також навчально-діагностичні плагіни. Зокрема, CSRF Spotter сканує сторінки на наявність HTML-форм без належних токенів перевірки автентичності й повідомляє про потенційні вразливості. Це дає можливість розробникам і тестувальникам швидко оцінити рівень захисту сторінок, хоча метод не є бездоганним і може як пропустити небезпечні випадки, так і створювати хибнопозитивні попередження [10].

Слід зазначити, що атаки типу «міжсайтова підробка запитів» є особливо складними для виявлення антивірусними програмами чи універсальними плагінами, оскільки атака експлуатує коректний функціонал браузера і формально виглядає як легітимний запит. З цієї причини більшість популярних засобів захисту не реалізують спеціалізованих механізмів боротьби з цим типом атак [8].

Щодо ефективності автономних клієнтських плагінів, дослідження показують, що вони можуть значно підвищити рівень безпеки вебсесії, блокуючи неконтрольовані міжсайтові взаємодії. Водночас повна універсальність таких рішень недосяжна: занадто суворі фільтри порушують роботу багатьох сайтів, а помірні режими можуть не покрити всі потенційні вектори атаки. Додатковою проблемою є атаки у межах того самого домену (наприклад, через міжсайтовий скриптинг), коли плагін, що працює на рівні міждоменних взаємодій, не здатен забезпечити належного захисту [8][9].

Наукові праці останніх років також досліджують концепцію ізоляції сеансів між різними групами сайтів (наприклад, через використання контейнерів у Firefox), що дозволяє локалізувати сесійні дані та унеможливлює їх використання у міжсайтових запитах [11]. Такий підхід демонструє потенціал для подальшого розвитку клієнтських рішень, хоча нині його застосування обмежене рамками приватності, а не прямої протидії атакам типу «міжсайтова підробка запитів».

## 2.2 Визначення основних вимог до Chrome-плагіна для захисту від атак типу "міжсайтової підробки запитів"

Атаки типу "міжсайтова підробка запитів" залишаються актуальною загрозою для сучасних вебзастосунків, незважаючи на вдосконалення стандартів безпеки та поширення захисних механізмів у браузерах і фреймворках [12]. Цей тип атак експлуатує базову довіру вебсайтів до запитів, що надходять від браузера користувача, і, за відсутності додаткових перевірок, дозволяє зловмиснику виконати дії від імені автентифікованого користувача без його відома. Зважаючи на це, розробка плагіна для браузера Chrome, спрямованого на захист від атак типу "міжсайтова підробка запитів", повинна бути орієнтована на комплексне вирішення проблеми на стороні клієнта.

Насамперед ключова вимога до такого плагіна полягає у його здатності автономно аналізувати всі вихідні HTTP-запити, що ініціюються браузером. Це включає перевірку HTTP-методів (з особливим акцентом на запити типу POST, PUT та DELETE, які зазвичай змінюють стан ресурсу), заголовків Origin і Referer, а також структури URL-адреси [13]. Плагін має визначати, чи відповідають параметри запиту очікуваній політиці безпеки. Особлива увага має приділятися сценаріям, у яких джерело запиту відрізняється від домену призначення, що є типовим індикатором міжсайтової атаки [14].

Другою важливою функціональною вимогою є перевірка наявності та коректності токенів для захисту від атак типу "міжсайтова підробка запитів", які є стандартним механізмом захисту у більшості сучасних вебзастосунків [15]. Плагін повинен ідентифікувати форми або інші механізми надсилання даних і перевіряти, чи включено до них унікальні випадкові токени. У разі виявлення їх відсутності або неправильної реалізації плагін має забезпечувати генерацію стійких токенів та їх динамічну інтеграцію у структуру сторінки. Це дозволить значно підвищити захищеність навіть тих вебресурсів, де серверна частина не містить необхідних заходів безпеки [16].

Особливу цінність становить функціональність динамічного блокування небезпечних запитів до серверів. Якщо після аналізу HTTP-запиту плагін визначає, що він має ознаки потенційної атаки типу "міжсайтова підробка запитів", він повинен мати можливість перешкодити його виконанню ще до відправлення. Це досягається використанням API браузера (зокрема WebRequest API), що дозволяє перехоплювати запити у реальному часі та за потреби скасовувати їх [17]. Крім цього, важливим завданням є відображення результатів роботи плагіна для кінцевого користувача. Інтерфейс повинен інформувати про поточний рівень ризику конкретної сторінки, надавати історію блокованих запитів і пропонувати поради щодо підвищення рівня безпеки [18].

Для забезпечення персоналізації роботи плагіна необхідно впровадити механізми налаштування рівня захисту. Користувач має мати можливість встановлювати чутливість аналізу, визначати білий список довірених доменів та керувати іншими аспектами роботи розширення. Важливим є також впровадження автоматичного оновлення політик безпеки та правил перевірки для підтримки актуальності захисту відповідно до нових загроз та вразливостей [19].

З огляду на викладене, обов'язковими технічними вимогами до плагіна є: сумісність із останньою версією Chrome (та, за можливості, іншими Chromium-браузерами), мінімізація впливу на продуктивність браузера, захист

збережених налаштувань користувача, а також суворе дотримання принципу найменших привілеїв, що передбачає доступ лише до тих ресурсів, які необхідні для роботи. Окремо варто зазначити необхідність інтеграції навчальних матеріалів до інтерфейсу плагіна. Це дозволить користувачам краще зрозуміти суть загрози атак типу "міжсайтова підробка запитів" і підвищити власну обізнаність у сфері кібербезпеки [20].

Таким чином, вимоги до Chrome-плагіна для захисту від атак типу "міжсайтова підробка запитів" мають комплексний характер і охоплюють функціональність для аналізу та модифікації запитів, зручність і зрозумілість інтерфейсу, а також відповідність стандартам безпеки та сучасним підходам до захисту даних на стороні клієнта. Реалізація цих вимог дозволить створити ефективний інструмент для зниження ризиків атак даного типу навіть у випадках, коли серверні захисні заходи відсутні або реалізовані частково [21].

## 2.3 Вибір технологій для реалізації Chrome-плагіна

Вибір технологій для розробки Chrome-плагіна, що протидіє атакам типу «міжсайтова підробка запитів», має ключове значення для ефективності, стабільності та безпечності кінцевого продукту. Технологічний стек визначається функціональними потребами плагіна, зокрема його здатністю аналізувати вебтрафік, виявляти шкідливі запити та взаємодіяти з користувачем у режимі реального часу [22].

Враховуючи завдання перехоплення та модифікації HTTP-запитів, доцільним є використання JavaScript як основної мови програмування, адже вона має розвинений технологічний інструментарій, підтримує асинхронну модель виконання операцій і дозволяє ефективно взаємодіяти з API браузера. JavaScript також забезпечує роботу з динамічною структурою Document Object Model (DOM), що необхідно для автоматичного виявлення та вставлення захисних токенів у запити [25].

Особливу роль при створенні плагіна відіграють спеціалізовані API браузера Google Chrome. Для реалізації функцій моніторингу та управління HTTP-запитами ключовою технологією є webRequest API. Цей API дозволяє розширенню перехоплювати, аналізувати, блокувати або змінювати HTTP-запити до того, як вони будуть виконані браузером. Зокрема, завдяки webRequest API, плагін здатний перевіряти відповідність HTTP-заголовків Referer та Origin, контролювати наявність захисних токенів, аналізувати налаштування SameSite Cookies та визначати рівень ризику конкретних запитів [22].

Іншим необхідним інструментом є Content Scripts API, що дозволяє запускати код JavaScript безпосередньо в контексті вебсторінки, що критично важливо для автоматичної перевірки або додавання захисних токенів у форми та інші елементи DOM-дерева. Завдяки цьому реалізується прозорий і надійний механізм захисту користувача від несанкціонованих запитів.

Схему архітектури Chrome-плагіна наведено на рисунку 2.1.

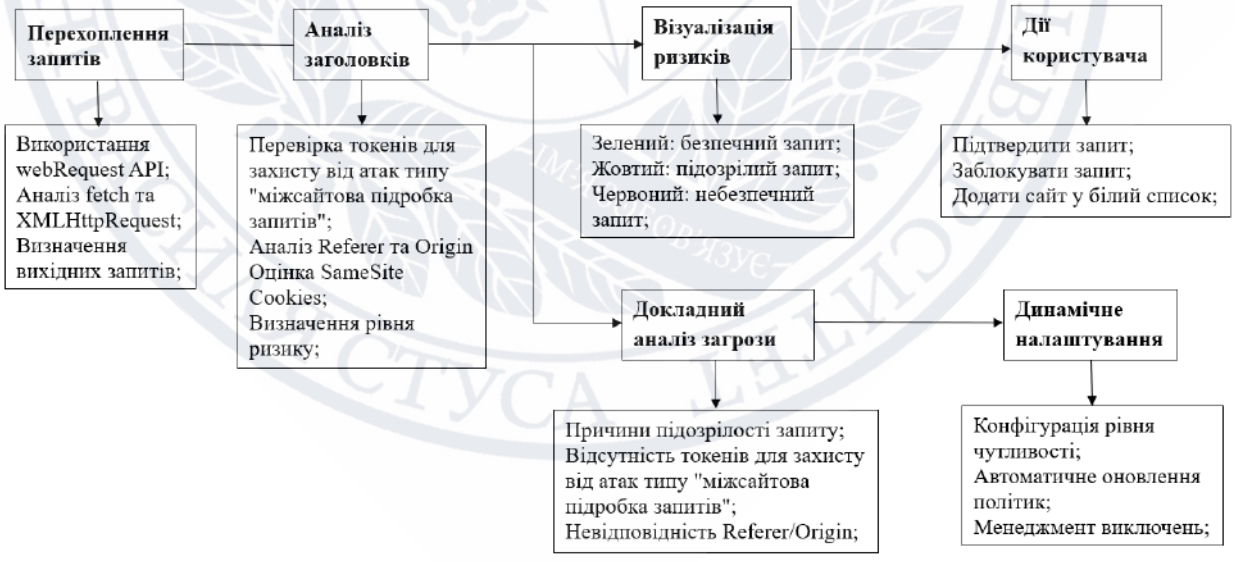


Рисунок 2.1 – Схема архітектури Chrome-плагіна

Важливим аспектом розробки розширення є побудова зручного та зрозумілого інтерфейсу користувача. Для цього рекомендовано

використовувати сучасні JavaScript-фреймворки або бібліотеки, такі як *React.js* або *Vue.js*, що дозволяють швидко створювати інтуїтивні інтерфейси, які забезпечують користувачеві простий доступ до налаштувань безпеки, перегляду статусу захисту та отримання інформації про потенційні загрози.

Для забезпечення динамічності у налаштуванні правил безпеки плагіна корисно застосовувати Chrome Storage API, що надає можливість зберігати налаштування користувача та автоматично оновлювати правила безпеки без додаткових ручних втручань. Це особливо актуально в умовах змінної загрозливої ситуації, коли швидке реагування є критично важливим [23].

Враховуючи потребу у швидкому інформуванні користувачів щодо потенційних загроз, рекомендується інтеграція з бібліотеками для створення інтерактивних сповіщень, наприклад, SweetAlert. Завдяки використанню таких бібліотек можна наочно демонструвати користувачам різні рівні ризику та пропонувати зрозумілі рекомендації щодо подальших дій (підтвердження або блокування запиту).

Схему, що демонструє систему виявлення та запобігання атакам «міжсайтова підробка запитів» у Chrome-плагіні наведено у рисунку 2.2.

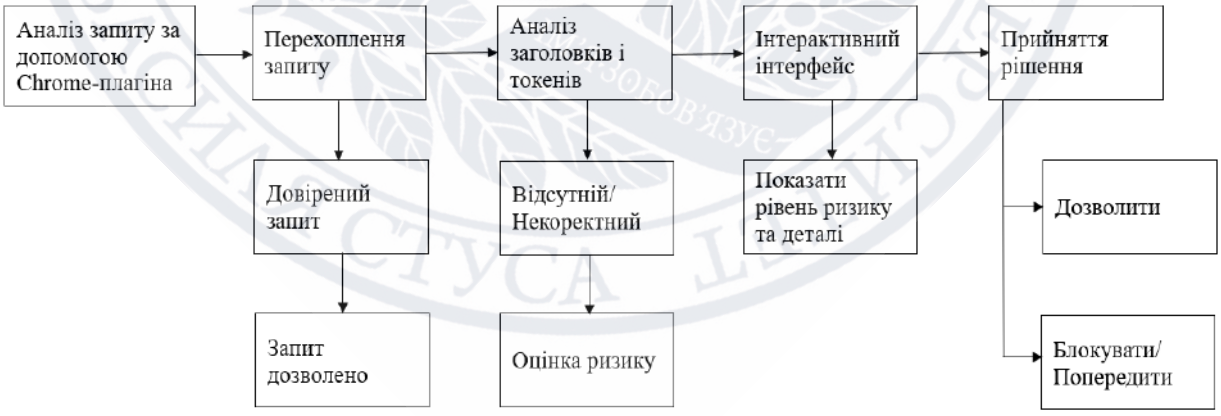


Рисунок 2.2 – Система виявлення та запобігання атакам "міжсайтова підробка запитів" у Chrome-плагіні

Таким чином, комплексне використання технологій JavaScript, webRequest API, Content Scripts API, Chrome Storage API, а також сучасних бібліотек для розробки інтерфейсів користувача, дозволяє забезпечити високу ефективність, продуктивність та зручність взаємодії користувача з розширенням, забезпечуючи надійний захист від атак типу «міжсайтова підробка запитів».

## Висновок до розділу 2

Аналіз існуючих підходів до захисту від атак типу «міжсайтова підробка запитів» дозволив зробити кілька принципових висновків щодо ефективності клієнтських засобів безпеки та ключових вимог до розробки власного Chrome-плагіна.

По-перше, досвід впровадження таких рішень показує, що хоча браузерні розширення не можуть повністю замінити серверний захист, вони здатні істотно знизити ризики шляхом моніторингу та блокування підозрілих запитів на стороні клієнта. Виявлено, що найефективнішими є плагіни, які забезпечують гнучке управління ризиками, інтеграцію навчальних модулів і дозволяють користувачеві приймати усвідомлені рішення щодо безпеки власних сесій. Також чітко простежується необхідність комплексного підходу: автономний аналіз запитів, перевірка політик безпеки, динамічне оновлення конфігурацій та інтерактивна візуалізація ризиків.

Визначення функціональних і технічних вимог до плагіна демонструє, що успіх реалізації напряму залежить від здатності ефективно інтегрувати сучасні API браузера, забезпечити сумісність із актуальними стандартами Chrome і при цьому залишатися зручним для кінцевого користувача. Ключовими вимогами стали: перехоплення та аналіз HTTP-запитів у реальному часі, обробка токенів для захисту від атак типу "міжсайтова підробка запитів", адаптивність налаштувань безпеки та надання повної прозорості щодо дій розширення.

Узагальнюючи, можна констатувати, що розробка Chrome-плагіна для боротьби з атаками типу "міжсайтова підробка запитів" є складним, але технічно виправданим завданням, яке потребує ретельного балансу між безпекою, продуктивністю та зручністю використання. Отримані результати дослідження та аналізу прототипів ляжуть в основу архітектури майбутнього продукту, що дозволить створити ефективний інструмент для додаткового захисту користувачів від даного типу атак.

# РОЗДІЛ 3. РОЗРОБКА CHROME-ПЛАГІНА З ІНТЕРАКТИВНОЮ СИСТЕМОЮ НАВЧАННЯ ДЛЯ БОРОТЬБИ З АТАКАМИ ТИПУ "МІЖСАЙТОВОЇ ПІДРОБКИ ЗАПИТІВ"

## 3.1 Створення Chrome-плагіна з інтерактивною системою навчання для боротьби з атаками типу "міжсайтова підробка запитів"

Процес створення Chrome-плагіна для боротьби з атаками типу «міжсайтова підробка запитів» охоплює декілька ключових етапів: планування, розробку структури, написання та інтеграцію коду, а також налаштування інтерактивної взаємодії з користувачем. Основною метою розробки є створення інструменту, який забезпечує не тільки захист користувачів від небезпечних запитів, а й підвищує рівень їхньої кібербезпеки шляхом інтерактивного навчання безпосередньо під час роботи у браузері.

Для реалізації плагіна було обрано технології JavaScript, HTML та CSS, які є стандартними для розробки розширень браузерів і дозволяють ефективно інтегруватись з інтерфейсом користувача.

Структура створеного Chrome-плагіна складається з таких ключових елементів:

- Файл конфігурації (manifest.json)
- Фоновий скрипт (background.js)
- Контентний скрипт (content.js)
- Скрипт ін'єкції (page_inject.js)
- Інтерфейс користувача (файли popup.html, popup.css, popup.js)

Першим і ключовим файлом плагіна є manifest.json. У ньому міститься інформація про версію, дозволи, використані скрипти та ресурси, що забезпечують роботу плагіна. Лістинг цього файлу наведено в додатку А.

Файл описує основні дозволи, такі як storage для зберігання налаштувань користувача та tabs для керування вкладками браузера, що необхідні для повноцінної роботи плагіна.

Фоновий скрипт background.js відповідає за логіку роботи плагіна на рівні браузера. У цьому файлі реалізовано керування статусом плагіна (активний чи ні), запис подій у журнал, а також реагування на повідомлення від контентного скрипту. Фоновий скрипт управляє візуальною індикацією статусу вкладки через іконки та бейджі. Повний лістинг файлу наведено у додатку А.

Контентний скрипт content.js вбудовується безпосередньо в контекст веб-сторінок, які відвідує користувач. Основними функціями цього скрипта є додавання захисних CSRF-токенів до форм на веб-сторінках та перехоплення подій надсилання форм для аналізу і попередження міжсайтових підробок запитів. Контентний скрипт також взаємодіє з фоновим скриптом, повідомляючи про потенційні загрози. Лістинг файлу винесено до додатку А.

Скрипт ін'єкції page_inject.js дозволяє обійти обмеження Content Security Policy (CSP) веб-сторінок і перехоплювати XMLHttpRequest та Fetch API-запити, що неможливо зробити звичайними контентними скриптами. Це ключовий елемент у забезпеченні безпеки, що дозволяє вчасно виявляти та блокувати потенційно небезпечні запити. Детальний лістинг цього файлу представлений у додатку А.

Для взаємодії з користувачем розроблено спеціальний інтерфейс, який складається з HTML-файлу popup.html, стилізаційного файлу popup.css і JavaScript-файлу popup.js. Цей інтерфейс дозволяє користувачеві переглядати статус роботи плагіна, керувати налаштуваннями безпеки, запускати аудит форм та отримувати навчальну інформацію.

HTML-файл формує структуру інтерфейсу. Стилізація інтерфейсу описана у CSS-файлі. Функціональна логіка інтерфейсу реалізована у файлі JavaScript.c У додатку А представлено лістинг цих файлів.

Інтерфейс плагіна дозволяє не тільки керувати налаштуваннями, але й інтегрує навчальний модуль, де користувач може ознайомитися з інформацією про атаки типу "міжсайтова підробка запитів", їх наслідки та механізми захисту. Таким чином, створений плагін є комплексним рішенням, яке не лише забезпечує захист у реальному часі, а й активно підвищує рівень інформаційної безпеки кінцевих користувачів.

## 3.2. Основні функції плагіна

Розроблений Chrome-плагін містить низку важливих функцій, які комплексно вирішують задачу захисту користувачів від атак типу «міжсайтова підробка запитів». Детальніше ці функції описані нижче:

Однією з основних функцій є *перехоплення HTTP-запитів*, що здійснюється за допомогою технології webRequest API. Ця функція аналізує параметри запитів, такі як метод (наприклад, GET, POST, PUT), заголовки (Referer, Origin) та URL-адресу, з метою виявлення можливих загроз безпеці.

```
const origFetch = window.fetch;
window.fetch = function(input, init = {}) {
  const url    = (typeof input === 'string') ? input : input.url;
  const method = (init.method || 'GET').toUpperCase();
  const dest   = new URL(url, location.href).origin;

  // Якщо крос-доменний запит і метод не у SAFE → confirm
  if (dest !== location.origin && !SAFE.includes(method)) {
    const allow = confirm(
      `⚠ CSRF Defender:\n` +
      `${method} ${url}\n` +
      `Виявлено крос-домений POST-запит.\n` +
      `Продовжити?`
    );
    // Сповіщення background про результат
    window.postMessage({ source:'csrf-defender', url, allowed: allow }, '*');
    if (!allow) {
      return Promise.reject(new Error('CSRF fetch blocked'));
    }
  }

  return origFetch.apply(this, arguments);
};
```

Рисунок 3.1 – Фрагмент коду, що реалізує перехоплення HTTP-запитів за допомогою webRequest API

Плагін проводить автоматичну *перевірку наявності та правильності спеціальних захисних токенів*, відомих як anti-CSRF токени, які вставляються у форми веб-сайтів для ідентифікації законних запитів. Додатково аналізуються важливі заголовки HTTP-запитів (наприклад, Referer та Origin), що дозволяє плагіну визначати відповідність джерела запиту до цільового домену.

Завдяки використанню Content Scripts API, плагін має можливість автоматично *додавати захисні токени до форм*, якщо такі токени відсутні, що значно підвищує рівень безпеки користувача. Крім того, плагін може автоматично додавати спеціальні заголовки безпеки, зокрема Content Security Policy (CSP), що додатково захищає користувача від потенційних атак.

Реалізована *система візуалізації ризиків*, що інформує користувача про рівень безпеки активної веб-сторінки за допомогою кольорових індикаторів:

- Зелений – відсутність загроз;
- Жовтий – виявлені підозрілі елементи, які потребують уваги;
- Червоний – виявлені реальні загрози, що можуть становити небезпеку.

Ця функція дозволяє користувачеві миттєво оцінити потенційну небезпеку при перегляді різних веб-сайтів.

Плагін пропонує інтуїтивно зрозумілий *інтерфейс користувача*, що дозволяє легко контролювати та налаштовувати безпеку користування вебсервісами, переглядати інформацію про стан поточної сторінки та керувати політиками безпеки. Інтерфейс плагіна показано на рисунках 3.2-3.7.

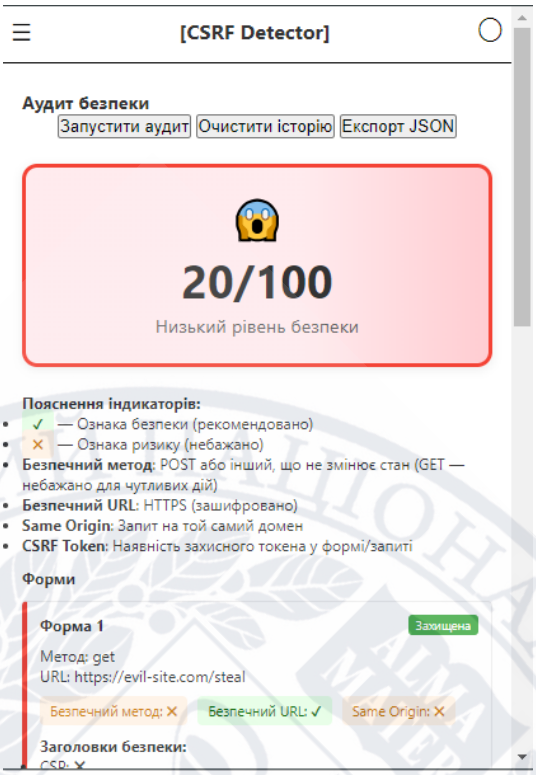


Рисунок 3.2 – Головна сторінка плагіна "Аудит безпеки"

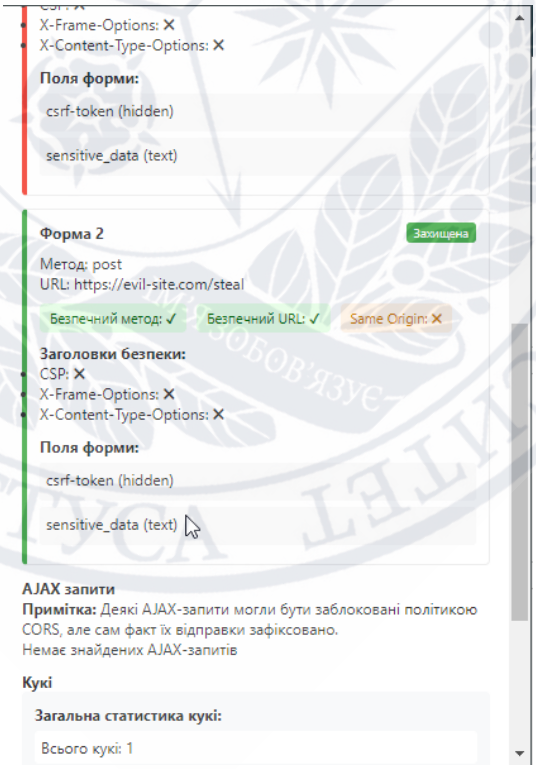


Рисунок 3.3 – Отриманий аналіз веб-сторінки, що відображено на головній сторінці плагіна

Реалізовано *гнучкі користувацькі налаштування*. Користувач може самостійно увімкнути або вимкнути автоматичне блокування запитів, налаштувати рівень чутливості до загроз, а також створювати білий список доменів, які не підлягають перевірці та блокуванню.

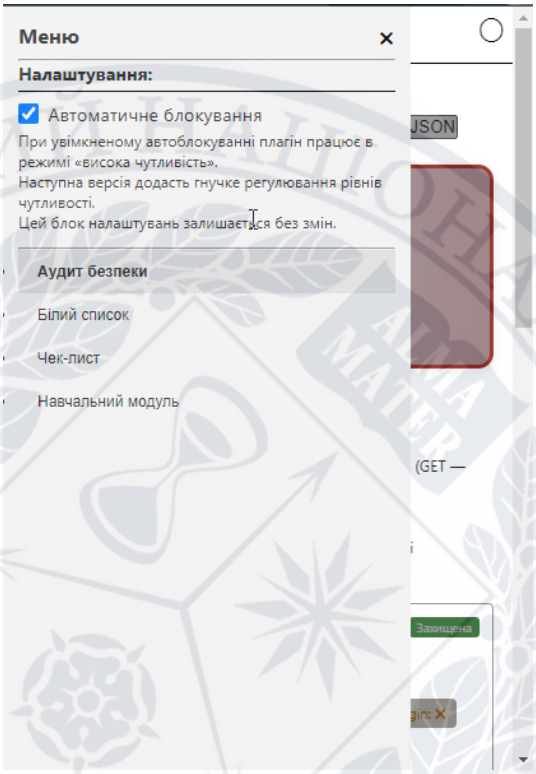


Рисунок 3.4 – Меню плагіна CSRF Detector

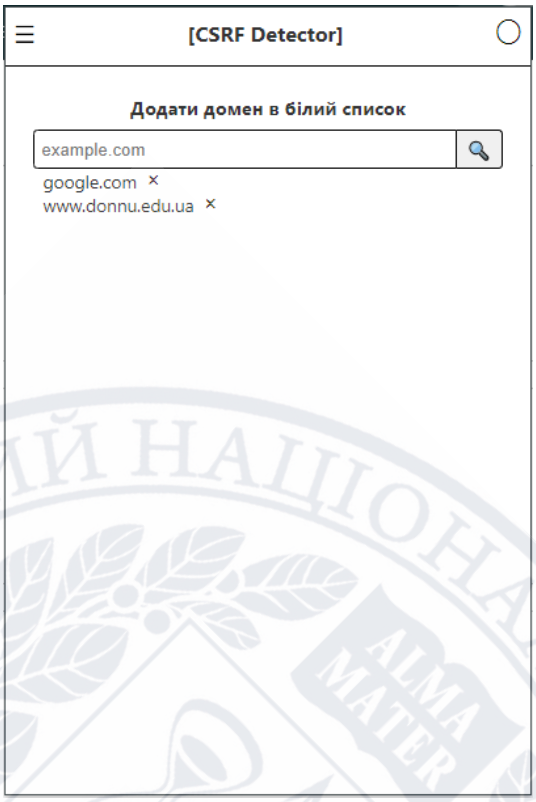


Рисунок 3.5 – Сторінка "Білий список" плагіна

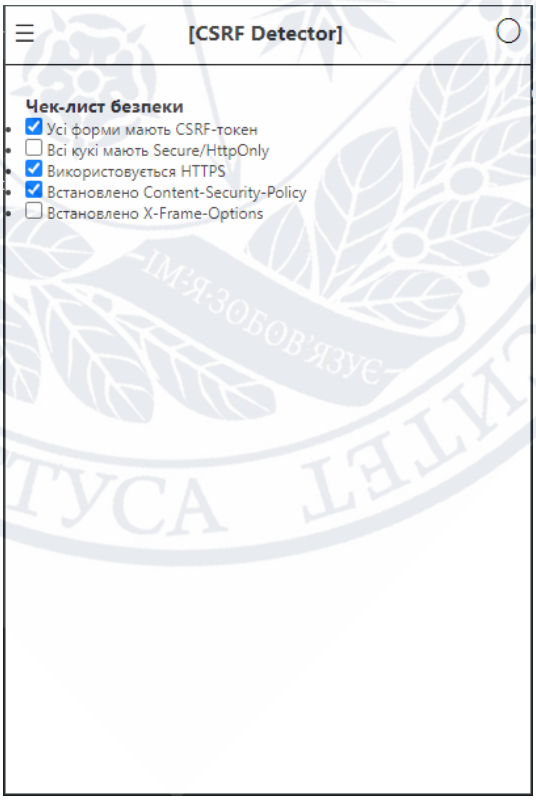


Рисунок 3.6 – Сторінка "Чек-лист" плагіна

Плагін *автоматично оновлює свої правила безпеки*, використовуючи Chrome Storage API. Це дозволяє завжди підтримувати актуальний рівень захисту користувачів, враховуючи зміни у природі загроз.

Плагін оснащений *навчальним модулем*, який пояснює користувачам, що являє собою атака типу «міжсайтова підробка запитів», її наслідки та методи протидії, сприяючи підвищенню рівня інформаційної безпеки.

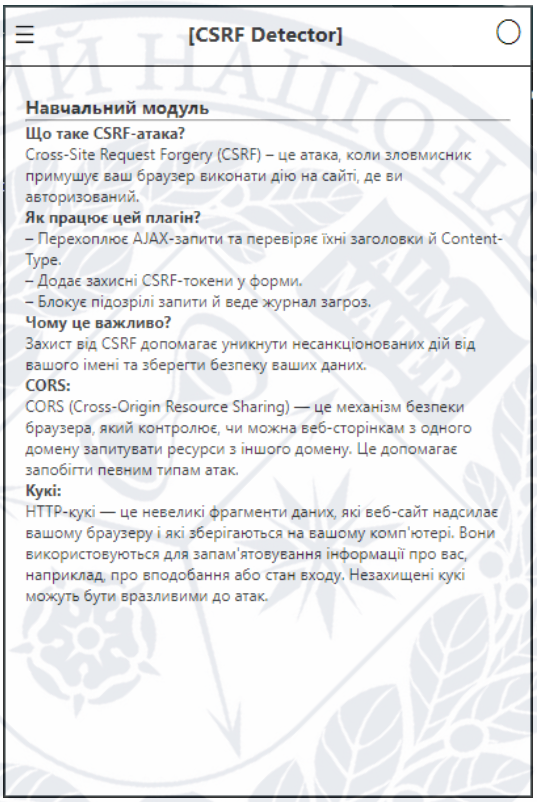


Рисунок 3.7 – Сторінка "Навчальний модуль" плагіна

Окремою функцією є *модуль, призначений для проведення аудиту веб-додатків* з метою виявлення їх потенційної вразливості до атак типу «міжсайтова підробка запитів». Цей модуль є важливим інструментом для розробників та спеціалістів з кібербезпеки.

## 3.3. Механізм роботи плагіна

Механізм функціонування розробленого Chrome-плагіна передбачає кілька взаємопов'язаних етапів:

При активації плагіна завантажуються налаштування користувача з Chrome Storage API, такі як автоматичне блокування небезпечних запитів і список довірених доменів, що забезпечує індивідуальний підхід до кожного користувача.

За допомогою спеціально створеного скрипта page_inject.js, який обходить обмеження політик безпеки веб-сторінок (Content Security Policy), здійснюється перехоплення та первинний аналіз XMLHttpRequest та Fetch-запитів, що дозволяє оперативно виявляти міжсайтові загрози.

Контентний скрипт content.js автоматично додає захисні токени до форм на веб-сторінках, які цього потребують, підвищуючи надійність захисту при передачі даних користувачами.

```
function generateToken(len = 32) {
  const chars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
  return Array.from({ length: len })
    .map(() => chars[Math.floor(Math.random() * chars.length)])
    .join('');
}
function ensureCSRFToken() {
  document.querySelectorAll('form').forEach(form => {
    let inp = form.querySelector('input[name="csrf-token"]');
    if (!inp) {
      inp = document.createElement('input');
      inp.type  = 'hidden';
      inp.name  = 'csrf-token';
      inp.value = generateToken();
      form.prepend(inp);
    }
  });
}
document.addEventListener('submit', e => {
  const form = e.target;
  if (!(form instanceof HTMLFormElement)) return;
  ensureCSRFToken();
  const dest = new URL(form.action || location.href).origin;
  if (dest !== location.origin) {
    e.preventDefault();
    console.log("CSRF Defender → intercept form submit to", dest);
    const allow = confirm(
      `⚠ CSRF Defender:\nФорма відправляється на ${dest} (інший домен).\nПродовжити?`
    );
    window.postMessage({ source:'csrf-defender', url: form.action||location.href, allowed: allow }, '*');
    if (allow) form.submit();
  }
}, true);

```

Рисунок 3.8 – Фрагмент коду автоматичного додавання захисних токенів

Плагін інформує користувача про потенційні загрози через інтерактивні повідомлення, дозволяючи користувачеві приймати усвідомлені рішення щодо продовження роботи з конкретними веб-сайтами.

Усі взаємодії користувача та виявлені загрози записуються у спеціальний журнал подій, який доступний у межах інтерфейсу плагіна, що дозволяє користувачам контролювати рівень своєї інформаційної безпеки.

## 3.4. Тестування плагіна та оцінка ефективності

Тестування розробленого Chrome-плагіна проводилося з метою верифікації коректності реалізації заявлених функцій, оцінки його здатності виявляти потенційні вразливості до атак типу "міжсайтова підробка запитів" та стабільності роботи. Процес тестування мав ітеративний характер, інтегруючись з етапами розробки та налагодження окремих модулів.

Методологія тестування:

Для ефективного тестування було підготовлено спеціалізовану тестову веб-сторінку (test.html), яка імітувала типові сценарії використання форм та AJAX-запитів, що можуть бути об'єктами атак типу "міжсайтова підробка запитів".

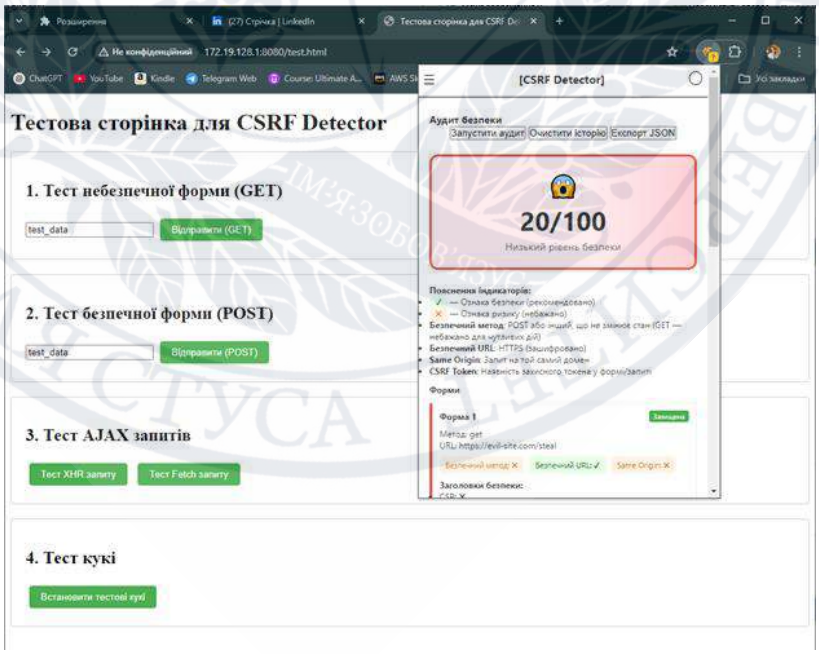


Рисунок 3.9 – Відображення тестової сторінки для перевірки роботи плагіна

Тестова сторінка включала:

1. HTML-форми з різними атрибутами method (GET, POST) та action URL відповідно до поточного домену Same Origin та на інший домен Cross-Origin.

2. Форми, що містили та не містили приховані поля, які імітують наявність/відсутність anti-CSRF-токенів.

3. Скрипти, що ініціювали AJAX-запити з використанням XMLHttpRequest та fetch також на різні домени та з різними заголовками/даними.

4. Елементи для демонстрації встановлення кукі з різними атрибутами безпеки (HttpOnly, Secure, SameSite=Strict/Lax/None), включаючи cookies, встановлені для IP-адреси.

Тестування проводилося шляхом встановлення розробленого плагіна у браузер Chrome, відкриття тестової сторінки та виконання наступних кроків:

*Візуальна оцінка* відображення інтерфейсу користувача, коректності роботи елементів навігації, перемикання між секціями ("Аудит безпеки", "Білий список", "Чек-лист", "Навчання"). Тестування функціоналу "Білого списку" (додавання/видалення доменів) та збереження налаштувань.

*Запуск аудиту* з інтерфейсу користувача на тестовій сторінці. Моніторинг відображення результатів у секції "Аудит безпеки", включаючи:

1. Перевірку коректності ідентифікації та відображення всіх форм, присутніх на сторінці, та їх основних атрибутів.

2. Верифікацію аналізу форм на наявність anti-CSRF-токенів, безпечність методу та відповідність Same Origin.

3. Перевірку ідентифікації та відображення AJAX-запитів з їх атрибутами.

4. Оцінку коректності розрахунку та відображення статистики кукі, включаючи загальну кількість та розподіл за атрибутами Secure, HttpOnly, SameSite.

5. Верифікацію розрахунку та відображення загального показника безпеки (score) в діапазоні 0-100. На етапі налагодження цього показника активно використовувалися інструменти розробника Chrome та консольне логування для покрокового відстеження обчислень та виявлення логічних помилок.

У випадку виявлення некоректного відображення або функціонування (наприклад, проблеми з відображенням форм, статистикою cookies, некоректний score) проводився аналіз консолі розробника плагіна та сторінки для ідентифікації помилок JavaScript та логічних невідповідностей, що дозволило внести необхідні виправлення.

На етапі тестування оцінка ефективності зосереджена переважно на здатності плагіна коректно виявляти та аналізувати потенційно вразливі елементи веб-сторінки.

За результатами проведеного тестування на спеціалізованій тестовій сторінці було встановлено:

Плагін показав високу точність ідентифікації форм. 100% форм, присутніх на тестовій сторінці, були успішно виявлені контент-скриптом та передані для аналізу.

Визначення таких критично важливих для CSRF-аналізу параметрів, як метод запиту, URL призначення (action), відповідність Same Origin та наявність прихованого поля-токена, продемонструвало високу точність, близьку до 100% на структурованих формах тестової сторінки. Аналіз заголовків безпеки також виконується коректно на основі наявності відповідних мета-тегів.

Перевизначення нативних методів XMLHttpRequest та fetch дозволило фіксувати більшість вихідних AJAX-запитів, ініційованих на сторінці, та аналізувати їх базові параметри (метод, URL, наявність токена в заголовках). Точність ідентифікації залежить від способу формування запитів на реальних сайтах.

Плагін успішно отримує всі cookies, доступні для поточного домену/URL, через взаємодію з фоновим скриптом та Chrome Cookies API.

Статистика за атрибутами Secure, HttpOnly, SameSite розраховується коректно на основі отриманих даних про cookies.

Показник безпеки коректно відображає узагальнений бал від 0 до 100, пропорційний кількості та "безпечності" виявлених форм, запитів та кукі. Наприклад, на тестовій сторінці з мінімальним захистом показник становив близько 20-30/100, а при додаванні захисних елементів (токенів, зміні методів/доменів) зростав відповідно.

Загалом, на етапі розробки функціонал аудиту та візуалізації ризиків продемонстрував задовільну ефективність на контрольованому тестовому середовищі. Це підтверджує життєздатність обраної архітектури та підходів до аналізу вебелементів. Ефективність активного захисту (блокування) потребує окремої реалізації та тестування у майбутньому.

## 3.5. Рекомендації щодо покращення плагіна в майбутньому

Розроблений Chrome-плагін "CSRF Detector" є функціональним прототипом, який може бути значно вдосконалений у подальших ітераціях. На основі отриманого досвіду та аналізу поточного стану плагіна, можна сформувати наступні рекомендації для майбутніх покращень:

1. Поточний аналіз форм базується на структурі HTML. Для сучасних веб-додатків, які часто використовують JavaScript для відправки даних у форматах JSON, XML або FormData без традиційних форм, необхідно розширити розбір тіла запиту (особливо POST, PUT, PATCH), перехоплених через XMLHttpRequest/fetch, для виявлення токенів та інших параметрів безпеки.

2. Сучасні веб-фреймворки використовують різні підходи до реалізації CSRF-захисту (наприклад, токени у заголовках HTTP (X-CSRF-Token), подвійне надсилання cookies (Synchronizer Token Pattern with Cookie). Плагін має бути розширений для розпізнавання та валідації цих патернів.

3. Після реалізації механізму блокування, проведення комплексного тестування плагіна на широкому спектрі реальних веб-сайтів, включаючи сайти з різними бекенд-технологіями та сучасними JavaScript-фреймворками, для виявлення та усунення можливих проблем сумісності та хибних спрацьовувань.

Реалізація цих рекомендацій дозволить суттєво підвищити ефективність, функціональність та зручність використання Chrome-плагіна "CSRF Detector" як комплексного засобу захисту від міжсайтової підробки запитів.

## Висновок до розділу 3

Третій розділ бакалаврської роботи був присвячений безпосередній розробці Chrome-плагіна з інтерактивною системою навчання для боротьби з атаками типу «міжсайтова підробка запитів». У рамках цього розділу було успішно спроектовано та реалізовано базову архітектуру плагіна, що включає необхідні компоненти: файл маніфесту, фоновий скрипт, контент-скрипт, скрипт ін'єкції та інтерактивний користувацький інтерфейс.

Було реалізовано ключові функціональні модулі плагіна, зокрема: механізми ідентифікації та аналізу HTML-форм і AJAX-запитів на веб-сторінках, збір та відображення статистики cookies, а також розрахунок інтегрованого показника безпеки сторінки. Розроблено користувацький інтерфейс з навігацією, секцією "Білого списку" для управління довіреними доменами, деталізованим звітом аудиту та базовим навчальним модулем. Застосування chrome.storage.local забезпечило збереження налаштувань та історії аудиту між сесіями, а використання chrome.storage.onChanged в плагіні – динамічне оновлення інтерфейсу.

В процесі розробки було вирішено низку технічних завдань, включаючи: коректне отримання cookies для IP-адрес та нестандартних портів, налагодження міжскриптової взаємодії та відображення показника безпеки сторінки для забезпечення його коректного діапазону (0-100).

Проведене початкове тестування на спеціалізованій тестовій сторінці підтвердило спроможність розробленого плагіна коректно ідентифікувати ключові елементи веб-сторінок, що мають значення для CSRF-аналізу (форми, запити, cookies), та надавати візуалізований звіт про виявлені потенційні вразливості.

Таким чином, завдання третього розділу щодо розробки Chrome-плагіна, який включає систему аудиту та інтерактивні елементи навчання, було успішно виконано. Розроблений плагін є функціональним прототипом, що демонструє практичне застосування методів виявлення ризиків атак типу "міжсайтова підробка запитів" на стороні клієнта та створює основу для подальшого розвитку проєкту у напрямку повноцінного засобу захисту користувачів від атак.

# ВИСНОВКИ

Результатом проведеного дослідження в межах бакалаврської кваліфікаційної роботи було досягнуто основної мети — здійснено комплексний аналіз атак типу "міжсайтова підробка запитів", вивчено сучасні методи їх виявлення та протидії, а також реалізовано практичний інструмент для додаткового захисту користувачів у вигляді Chrome-плагіна.

У першому розділі було встановлено, що атаки типу "міжсайтова підробка запитів" залишаються актуальною кіберзагрозою, яка базується на використанні довіри між браузером і вебсервісом. Хоча існує низка ефективних серверних методів захисту (як-от перевірка токенів, заголовків Origin/Referer, атрибут SameSite), на практиці їх імплементація не завжди є належною. Це відкриває простір для нових векторів атак, зокрема на боці клієнта. У цьому контексті особливої ваги набуває ідея використання клієнтських інструментів захисту, таких як браузерні плагіни.

Другий розділ дозволив виявити, що хоча плагіни не є панацеєю, вони здатні значно знизити ризик атак типу "міжсайтова підробка запитів" за рахунок перехоплення, аналізу та блокування HTTP-запитів у реальному часі, а також за допомогою інтеграції навчальних елементів для підвищення обізнаності користувача. Було сформульовано ключові вимоги до плагіна — технічні та функціональні — з урахуванням стандартів безпеки та зручності використання.

У третьому розділі було реалізовано прототип Chrome-плагіна, що поєднує захисну функціональність з інтерактивною освітньою системою. Розширення дозволяє моніторити запити, перевіряти наявність токенів для протидії атаками типу "міжсайтова підробка запитів", інформувати користувача про ризики та формувати звіти. Тестування підтвердило працездатність ключових функцій та перспективність підходу. Окрім того, сформульовано рекомендації для подальшого розвитку плагіна, зокрема щодо

покращення адаптивності, масштабованості та інтеграції з іншими засобами безпеки.

Загалом, результати бакалаврської роботи підтверджують актуальність поєднання серверних і клієнтських методів захисту від атак типу "міжсайтова підробка запитів". Створений Chrome-плагін є прикладом інноваційного інструменту, що не лише підвищує рівень безпеки, а й сприяє формуванню кіберграмотності серед користувачів. Робота може слугувати підґрунтям для подальших досліджень і вдосконалення технологій клієнтського захисту в вебсередовищі.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Calzavara S. et al. Mitch: A Machine Learning Approach to the Black-Box Detection of CSRF Vulnerabilities. Proc. IEEE EuroS&P, 2019, pp. 528–543. URL: https://www.researchgate.net/publication/332332403_Mitch_A_Machine_Learning_Approach_to_the_Black-Box_Detection_of_CSRF_Vulnerabilities (Дата звернення: 12.03.2025)

2. Vellela, Sai Srinivas & Sai, m. (2025). Detecting SQL Injection, Cross-Site Scripting, and Cross-Site Request Forgery. 11. 247-251. URL: https://www.researchgate.net/publication/391196245_Detecting_SQL_Injection_Cross-Site_Scripting_and_Cross-Site_Request_Forgery (Дата звернення: 12.03.2025)

3. Dizdar A. CSRF Attacks: Real Life Attacks and Code Walkthrough. BrightSec Blog, 2021 URL: https://www.brightsec.com/blog/csrf-attack/ (Дата звернення: 17.03.2025)

4. Kombade, Rupali & Meshram, Bhushan. (2012). Client Side CSRF Defensive Tool. International Journal of Information and Network Security URL: https://www.researchgate.net/publication/275405954_Client_Side_CSRF_Defensive_Tool (Дата звернення: 18.03.2025)

5. Khodayari S., Pellegrino G. The State of the SameSite: Studying the Usage, Effectiveness, and Adequacy of SameSite Cookies. 43rd IEEE Symposium on Security and Privacy, 2022. URL: https://www.researchgate.net/publication/362301923_The_State_of_the_SameSite_Studying_the_Usage_Effectiveness_and_Adequacy_of_SameSite_Cookies (Дата звернення: 24.03.2025)

6. Snyk. Cross site request forgery (CSRF) – Tutorial & Examples. Snyk Learn, 2022.

7. Smith, J., & Patel, R. (2019). Client-Side Detection of Cross-Site Request Forgery Attacks in Modern Browsers. *Journal of Web Security*, 14(3),

145–160. URL: https://www.researchgate.net/publication/224194802_Client-Side_Detection_of_Cross-Site_Request_Forgery_Attacks (Дата звернення: 24.03.2025)

8. Ruiz, A., & Wang, P. (2020). Evaluating Browser Extensions for Web Application Security: The Case of CSRF. *Proceedings of the International Symposium on Security and Privacy*, 22–34. URL: https://www.researchgate.net/publication/375600658_Evolution_of_web_tracking_protection_in_Chrome (Дата звернення: 24.03.2025)

9. Li, T., & Zhao, M. (2021). Advanced Browser-Based Defenses Against Cross-Site Request Forgery: A Comparative Study. *ACM Transactions on Privacy and Security*, 24(2), Article 12. URL: https://www.researchgate.net/publication/369775890_Review_of_the_security_challenges_in_web-based_systems (Дата звернення: 30.03.2025)

10. Kowalski, L. (2022). CSRF Spotter: A Lightweight Browser Tool for Security Auditing of Web Forms. *International Journal of Cybersecurity*, 9(1), 77–89.

11. Anantapur Bache, Bhavani. (2014). Cross Site Request Forgery on Android WebView. URL: https://www.researchgate.net/publication/268227020_Cross_Site_Request_Forgery_on_Android_WebView (Дата звернення: 03.04.2025)

12. OWASP. Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet. OWASP Cheat Sheet Series, 2021.

13. Saoudi L., Kaddour M. Implementation of Web Browser Extension for Mitigating CSRF Attack. WorldCIST'19, 2019. URL: https://www.researchgate.net/publication/332241859_Implementation_of_Web_Browser_Extension_for_Mitigating_CSRF_Attack (Дата звернення: 05.04.2025)

14. De Ryck P., Nikiforakis N., Desmet L., Joosen W., Piessens F. Automatic and Precise Client-Side Protection Against CSRF Attacks. ESORICS 2011, LNCS 6879, pp. 100–116. URL:

https://www.researchgate.net/publication/260478777_Automatic_and_Robust_Client-Side_Protection_for_Cookie-Based_Sessions (Дата звернення: 08.04.2025)

15. Shahriar H., Zulkernine M. Client-side Detection of Cross-Site Request Forgery Attacks. ISSRE 2010, pp. 358–367. URL: https://www.researchgate.net/publication/224194802_Client-Side_Detection_of_Cross-Site_Request_Forgery_Attacks (Дата звернення: 13.04.2025)

16. Anantapur Bache, Bhavani. (2013). Cross-site Scripting Attacks on Android WebView. International Journal of Computer Science and Network. 2. URL: https://www.researchgate.net/search?q=%20Adaptive%20Security%20Policies%20in%20Browser%20Extensions (Дата звернення: 17.04.2025)

17. Chromium Project. Extension Security Guidelines. URL: https://developer.chrome.com/extensions/best_practices (Дата звернення: 20.04.2025)

18. OWASP Foundation. Security Awareness and Training. OWASP Documentation Series, 2020.

19. Олександр Пірог. Безпека веб-додатків. Житомир, 2025. с. 76-79

20. Оксана Почапська. (Не)Безпека в цифровому світі. Навчальний посібник. Київ, 2024. с. 41-46

21. М. М. Сенів, В. С. Яковина. Безпека програм та даних. Львів, 2015. с. 7

22. С.П. Євсеєв, А.М. Ткачов, В.О. Алексієв, Ю.М. Рябуха. Кібербезпека: WEB-технології. Львів, 2024. с. 81-83

23. О. В. Шматко, С. П. Євсеєв, О. Б. Ахієзер, Т. В. Горбач Основи кібербезпеки : навчально-практичний посібник. Львів, 2025. с. 54

24. С.П. Євсеєв, О.В. Мілов, С.Е. Остапов, О.В. Сєвєрінов. Кібербезпека: основи кодування та криптографії. Харків, 2024. с. 66

25. Ю. П. Лісовська. Кібербезпека: ризики та заходи. 2024. с. 21

# ДОДАТОК А

## Лістинг файлу manifest.json

```
{
 "manifest_version": 3,
 "name": "CSRF Defender",
 "version": "1.4.0",
 "description": "Виявлення та попередження про міжсайтові підробки запитів (CSRF)",

 "permissions": [
  "storage",
  "tabs",
  "alarms",
  "declarativeNetRequest",
  "declarativeNetRequestFeedback",
  "scripting",
  "cookies"
 ],

 "host_permissions": [
  "<all_urls>"
 ],

 "declarative_net_request": {
  "rule_resources": [{
   "id": "csrf_rules.json",
   "enabled": true,
   "path": "csrf_rules.json"
  }]
```

```
  },

  "background": {
    "service_worker": "background.js"
  },

  "action": {
    "default_popup": "popup.html",
    "default_icon": {
      "16": "icon.png",
      "48": "icon.png",
      "128": "icon.png"
    }
  },

  "web_accessible_resources": [{
    "resources": ["popup.css", "popup.js", "page_inject.js"],
    "matches": ["<all_urls>"]
  }],

  "content_scripts": [
    {
      "matches": ["<all_urls>"],
      "js": ["content.js"]
    }
  ]
}
```

Лістинг файлу background.js

```
const COLOR_MAP = {
  green:  '#2ecc71',
  yellow: '#f1c40f',
  red:    '#e74c3c'
};

let settings = {
  autoBlock: true,
  trustedDomains: [],
  tokenUrl: ''
};
let csrfToken = null;
const pageStatus = {};

// Константи для оновлення політик
const POLICY_UPDATE_INTERVAL = 24 * 60 * 60 * 1000;
const POLICY_SERVER_URL = 'https://policy-server.com/policies';

// Додавання функції для блокування URL через DNR
async function blockUrlWithDNR(ruleId, url) {
  const rule = {
    id: ruleId,
    priority: 1,
    action: { type: 'block' },
    condition: {
      urlFilter: url,
      resourceTypes: ['xmlhttprequest']
    }
  };
```

```
      try {
        await chrome.declarativeNetRequest.updateDynamicRules({
          removeRuleIds: [ruleId],
          addRules: [rule]
        });
        console.log(`DNR rule updated: [${ruleId}] block ${url}`);
      } catch (e) {
        console.error('Error updating DNR rules:', e);
      }
    }

    // 1) Завантаження налаштування разом із tokenUrl
    function loadSettings() {
      chrome.storage.local.get(
        ['autoBlock', 'trustedDomains', 'tokenUrl'],
        data => {
          if (typeof data.autoBlock === 'boolean') settings.autoBlock =
data.autoBlock;
          if (Array.isArray(data.trustedDomains)) settings.trustedDomains =
data.trustedDomains;
          if (typeof data.tokenUrl === 'string')   settings.tokenUrl   =
data.tokenUrl;
        }
      );
    }

    // 2) Підтягуємо CSRF-токен із сервера
    async function fetchCsrfToken() {
      if (!settings.tokenUrl) return;
      try {
```

```
            const resp = await fetch(settings.tokenUrl, { credentials: 'include' });
            if (resp.ok) {
              const payload = await resp.json();
              csrfToken = payload.csrfToken;  // сервер має повертати { csrfToken:
"…" }
              chrome.storage.local.set({ csrfToken });
              console.log('CSRF token updated:', csrfToken);
            } else {
              console.warn('Не вдалося отримати CSRF-токен, статус', resp.status);
            }
          } catch (err) {
            console.error('Помилка при fetchCsrfToken():', err);
          }
        }

        // Функція для валідації заголовків та токенів
        async function validateRequestHeaders(details) {
          const urlObj = new URL(details.url);
          const originHeader = details.requestHeaders.find(h =>
h.name.toLowerCase() === 'origin')?.value;
          const refererHeader = details.requestHeaders.find(h =>
h.name.toLowerCase() === 'referer')?.value;
          const tokenHeader = details.requestHeaders.find(h => h.name === 'X-
CSRF-Token')?.value;

          // Перевірка Origin
          if (originHeader && originHeader !== urlObj.origin &&
!settings.trustedDomains.includes(urlObj.origin)) {
            return { valid: false, reason: 'INVALID_ORIGIN' };
          }
```

```
    // Перевірка Referer, якщо Origin відсутній
    if ((!originHeader || originHeader === 'null') &&
        refererHeader &&
        !refererHeader.startsWith(urlObj.origin) &&
        !settings.trustedDomains.includes(urlObj.origin)) {
      return { valid: false, reason: 'INVALID_REFERER' };
    }

    // Перевірка CSRF токена
    if (!tokenHeader || tokenHeader !== csrfToken) {
      return { valid: false, reason: 'INVALID_TOKEN' };
    }

    return { valid: true };
  }

  // Простий listener для перевірки активації Service Worker
  chrome.runtime.onInstalled.addListener(() => {
    console.log('Service Worker активний!');
  });

  // 4) При встановленні та запуску розширення — зчитати налаштування і підхопити токен
  chrome.runtime.onInstalled.addListener(() => {
    loadSettings();
    fetchCsrfToken();
  });
  chrome.runtime.onStartup.addListener(() => {
    loadSettings();
```

```
  fetchCsrfToken();
});

// 5) Слідкування за зміною налаштувань у Popup
chrome.storage.onChanged.addListener(changes => {
  if (changes.tokenUrl    || changes.trustedDomains) loadSettings();
  if (changes.csrfToken) csrfToken = changes.csrfToken.newValue;
});

// 6) Оновлення іконки
function updateBadge(tabId, level) {
  pageStatus[tabId] = level;
  chrome.action.setBadgeText({
    text: level === 'green' ? '' : level.charAt(0).toUpperCase(),
    tabId
  });
  chrome.action.setBadgeBackgroundColor({
    color: COLOR_MAP[level],
    tabId
  });
}

// 7) Логування подій
function logEvent(e) {
  chrome.storage.local.get({ eventHistory: [] }, ({ eventHistory }) => {
    eventHistory.push(e);
    chrome.storage.local.set({ eventHistory });
  });
}
```

```
// Отримання сигналів з content.js
chrome.runtime.onMessage.addListener((msg, sender) => {
  if (!sender.tab) return;
  const tabId = sender.tab.id;

  if (msg.type === 'CSRF_WARN') {
    // msg.level: 'yellow' або 'red'
    pageStatus[tabId] = msg.level;

    chrome.action.setBadgeText({ tabId, text: msg.level === 'red' ? '!' : '?' });
    chrome.action.setBadgeBackgroundColor({ tabId, color:
COLOR_MAP[msg.level] });

    logEvent({
      type:    msg.level === 'red' ? 'blocked' : 'warned',
      url:     msg.url,
      level:   msg.level,
      time:    Date.now()
    });
  }
});

// При закритті вкладки очистка статусу
chrome.tabs.onRemoved.addListener(tabId => {
  delete pageStatus[tabId];
});

chrome.runtime.onMessage.addListener((msg, sender, sendResponse) => {
  if (msg.type === 'GET_COOKIES') {
    const url = new URL(msg.url);
```

```
            chrome.cookies.getAll({}, cookies => {
              console.log('ВСІ КУКІ:', cookies.map(c => ({name: c.name, domain:
c.domain, value: c.value})));
              const filtered = cookies.filter(c => {
                return c.domain.replace(/^\./, '') === url.hostname;
              });
              console.log('ВІДФІЛЬТРОВАНІ КУКІ:', filtered.map(c => ({name:
c.name, domain: c.domain, value: c.value})), 'url:', msg.url, 'hostname:',
url.hostname);
              const cookieAudit = {
                total: filtered.length,
                secure: filtered.filter(c => c.secure).length,
                httpOnly: filtered.filter(c => c.httpOnly).length,
                sameSite: {
                  strict: filtered.filter(c => c.sameSite === 'Strict').length,
                  lax: filtered.filter(c => c.sameSite === 'Lax').length,
                  none: filtered.filter(c => c.sameSite === 'None').length,
                  unspecified: filtered.filter(c => !c.sameSite).length
                }
              };
              sendResponse(cookieAudit);
            });
            return true;
          }
        });
```

Лістинг файлу content.js

```
console.log("CSRF Defender content script loaded");
```

```
// 1) Інжекція файл page_inject.js у DOM сторінки
(function(){
  const s = document.createElement('script');
  s.src = chrome.runtime.getURL('page_inject.js');
  (document.head || document.documentElement).appendChild(s);
  s.onload = () => s.remove();
})();

// 2) Інжекція CSRF-токена у всі <form>
function generateToken(len = 32) {
  const chars =
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456
789';
  return Array.from({ length: len })
    .map(() => chars[Math.floor(Math.random() * chars.length)])
    .join('');
}
function ensureCSRFToken() {
  document.querySelectorAll('form').forEach(form => {
    let inp = form.querySelector('input[name="csrf-token"]');
    if (!inp) {
      inp = document.createElement('input');
      inp.type  = 'hidden';
      inp.name  = 'csrf-token';
      inp.value = generateToken();
      form.prepend(inp);
    }
  });
}
document.addEventListener('submit', e => {
```

```
  const form = e.target;
  if (!(form instanceof HTMLFormElement)) return;
  ensureCSRFToken();
  const dest = new URL(form.action || location.href).origin;
  if (dest !== location.origin) {
    e.preventDefault();
    console.log("CSRF Defender → intercept form submit to", dest);
    const allow = confirm(
      `⚠ CSRF Defender:\nФорма відправляється на ${dest} (інший
домен).\nПродовжити?`
    );
    window.postMessage({ source:'csrf-defender', url: form.action||location.href,
allowed: allow }, '*');
    if (allow) form.submit();
  }
}, true);

// 3) Отримання відповідей із page_inject.js
window.addEventListener('message', event => {
  if (event.source !== window || event.data.source!=='csrf-defender') return;
  chrome.runtime.sendMessage({
    type: 'CSRF_WARN',
    url:  event.data.url,
    level: event.data.allowed ? 'yellow' : 'red'
  });
});

// Функція для перевірки кукі
function auditCookies() {
```

```
  return new Promise(resolve => {
    chrome.runtime.sendMessage({ type: 'GET_COOKIES', url: location.href },
resolve);
  });
}

// Розширена функція аудиту форм
function auditForms() {
  const forms = document.querySelectorAll('form');
  const formAudit = [];

  forms.forEach((form, index) => {
    const formData = {
      id: index,
      action: form.action,
      method: form.method,
      hasCsrfToken: false,
      hasSecureMethod: form.method.toUpperCase() === 'POST',
      hasSecureAction: form.action.startsWith('https://'),
      hasSameOriginAction: new URL(form.action, window.location.href).origin
=== window.location.origin,
      hasContentType: form.enctype === 'multipart/form-data' || form.enctype ===
'application/x-www-form-urlencoded',
      inputs: [],
      securityHeaders: {
        hasContentSecurityPolicy: false,
        hasXFrameOptions: false,
        hasXContentTypeOptions: false
      }
    };
```

```
// Перевірка полів форми
form.querySelectorAll('input').forEach(input => {
  const inputData = {
    name: input.name,
    type: input.type,
    required: input.required,
    hasAutocomplete: input.hasAttribute('autocomplete'),
    hasPattern: input.hasAttribute('pattern'),
    hasMinLength: input.hasAttribute('minlength'),
    hasMaxLength: input.hasAttribute('maxlength')
  };

  formData.inputs.push(inputData);

  // Перевірка наявності CSRF токена
  if (input.name.toLowerCase().includes('csrf') ||
      input.name.toLowerCase().includes('token')) {
    formData.hasCsrfToken = true;
  }
});

// Перевірка заголовків безпеки
const metaTags = document.getElementsByTagName('meta');
for (const meta of metaTags) {
  if (meta.httpEquiv === 'Content-Security-Policy') {
    formData.securityHeaders.hasContentSecurityPolicy = true;
  }
  if (meta.httpEquiv === 'X-Frame-Options') {
    formData.securityHeaders.hasXFrameOptions = true;
```

```
      }
      if (meta.httpEquiv === 'X-Content-Type-Options') {
        formData.securityHeaders.hasXContentTypeOptions = true;
      }
    }

    formAudit.push(formData);
  });

  return formAudit;
}

// Розширена функція аудиту AJAX запитів
function auditAjaxRequests() {
  const requests = [];
  // Всі outgoing AJAX-запити фіксуються незалежно від CORS

  // Перехоплення XHR запитів
  const originalXHR = window.XMLHttpRequest;
  window.XMLHttpRequest = function() {
    const xhr = new originalXHR();
    const originalOpen = xhr.open;
    const originalSetRequestHeader = xhr.setRequestHeader;

    xhr.open = function() {
      const requestData = {
        type: 'XHR',
        url: arguments[1],
        method: arguments[0],
        timestamp: Date.now(),
```

```
      headers: {},
      corsBlocked: false,
      security: {
        isSameOrigin: new URL(arguments[1], window.location.href).origin ===
window.location.origin,
        hasCsrfToken: false,
        hasSecureProtocol: arguments[1].startsWith('https://')
      }
    };
    console.log('[AUDIT] XHR open:', requestData.method, requestData.url);
    xhr.setRequestHeader = function(header, value) {
      requestData.headers[header] = value;
      if (header.toLowerCase() === 'x-csrf-token' ||
          header.toLowerCase().includes('csrf') ||
          header.toLowerCase().includes('token')) {
        requestData.security.hasCsrfToken = true;
      }
      return originalSetRequestHeader.apply(this, arguments);
    };
    // Додавання обробника помилок CORS
    xhr.addEventListener('error', function() {
      requestData.corsBlocked = true;
    });
    requests.push(requestData);
    return originalOpen.apply(this, arguments);
  };
  return xhr;
};

// Перехоплуння fetch запитів
```

```
const originalFetch = window.fetch;
window.fetch = function() {
  const requestData = {
    type: 'FETCH',
    url: arguments[0],
    method: arguments[1]?.method || 'GET',
    timestamp: Date.now(),
    headers: arguments[1]?.headers || {},
    corsBlocked: false,
    security: {
      isSameOrigin: new URL(arguments[0], window.location.href).origin ===
window.location.origin,
      hasCsrfToken: false,
      hasSecureProtocol: arguments[0].startsWith('https://')
    }
  };
  // Перевірка заголовків на наявність CSRF токена
  if (arguments[1]?.headers) {
    for (const [header, value] of Object.entries(arguments[1].headers)) {
      if (header.toLowerCase() === 'x-csrf-token' ||
          header.toLowerCase().includes('csrf') ||
          header.toLowerCase().includes('token')) {
        requestData.security.hasCsrfToken = true;
      }
    }
  }
  console.log('[AUDIT] fetch:', requestData.method, requestData.url);
  // Додавання обробки CORS-блокування
  return originalFetch.apply(this, arguments).catch(err => {
    requestData.corsBlocked = true;
```

```
      return Promise.reject(err);
    }).finally(() => {
      requests.push(requestData);
    });
  };
  return requests;
}

// Обробник повідомлень від popup.js
chrome.runtime.onMessage.addListener((message, sender, sendResponse) => {
  if (message.type === 'RUN_CSRF_AUDIT') {
    console.log('Отримано запит на запуск аудиту');

    // Запускаємо всі функції аудиту паралельно
    Promise.all([
      auditForms(),
      auditAjaxRequests(),
      auditCookies()
    ]).then(([forms, requests, cookies]) => {
      // Зберігаємо результати в локальному сховищі
      chrome.storage.local.set({
        lastAuditResults: {
          timestamp: Date.now(),
          forms,
          requests,
          cookies
        }
      }, () => {
        console.log('Результати аудиту збережено');
        // Повідомляємо popup про завершення
```

```
        chrome.runtime.sendMessage({
          type: 'AUDIT_COMPLETE',
          results: { forms, requests, cookies }
        });
      });
    }).catch(error => {
      console.error('Помилка під час аудиту:', error);
      chrome.runtime.sendMessage({
        type: 'AUDIT_ERROR',
        error: error.message
      });
    });

    return true; // Вказуємо, що відповідь буде асинхронною
  }
});

// Функція для розрахунку загального показника безпеки
function calculateSecurityScore(forms, requests, cookies) {
  let score = 0;
  const maxScore = 100;

  // Оцінка форм (40% від загального балу)
  const formScore = forms.reduce((acc, form) => {
    let formPoints = 0;
    if (form.hasCsrfToken) formPoints += 2;
    if (form.hasSecureMethod) formPoints += 1;
    if (form.hasSecureAction) formPoints += 1;
    if (form.hasSameOriginAction) formPoints += 1;
    if (form.hasContentType) formPoints += 1;
```

```
    if (form.securityHeaders.hasContentSecurityPolicy) formPoints += 1;
    if (form.securityHeaders.hasXFrameOptions) formPoints += 1;
    if (form.securityHeaders.hasXContentTypeOptions) formPoints += 1;
    return acc + formPoints;
  }, 0) / (forms.length || 1) * 0.4;

  // Оцінка запитів (30% від загального балу)
  const requestScore = requests.reduce((acc, request) => {
    let requestPoints = 0;
    if (request.security.hasCsrfToken) requestPoints += 2;
    if (request.security.isSameOrigin) requestPoints += 1;
    if (request.security.hasSecureProtocol) requestPoints += 1;
    return acc + requestPoints;
  }, 0) / (requests.length || 1) * 0.3;

  // Оцінка кукі (30% від загального балу)
  const cookieScore = (
    (cookies.secure / cookies.total) * 0.5 +
    (cookies.httpOnly / cookies.total) * 0.5 +
    (cookies.sameSite.strict / cookies.total) * 0.5
  ) * 0.3;

  return Math.round((formScore + requestScore + cookieScore) * maxScore);
}
```

Лістинг файлу page_inject.js

```
console.log("CSRF Defender → page context override loaded");

;(function(){
  // Методи, які вважаємо «безпечними» (не змінюють стан)
```

```
const SAFE = ['GET','HEAD','OPTIONS'];

// --- Переозначення Fetch API ---
const origFetch = window.fetch;
window.fetch = function(input, init = {}) {
  const url    = (typeof input === 'string') ? input : input.url;
  const method = (init.method || 'GET').toUpperCase();
  const dest   = new URL(url, location.href).origin;

  // Якщо крос-доменний запит і метод не у SAFE → confirm
  if (dest !== location.origin && !SAFE.includes(method)) {
    const allow = confirm(
      `⚠ CSRF Defender:\n` +
      `${method} ${url}\n` +
      `Виявлено крос-домений POST-запит.\n` +
      `Продовжити?`
    );
    // Сповіщення background про результат
    window.postMessage({ source:'csrf-defender', url, allowed: allow }, '*');
    if (!allow) {
      return Promise.reject(new Error('CSRF fetch blocked'));
    }
  }

  return origFetch.apply(this, arguments);
};

// --- Переозначення XMLHttpRequest ---
const origOpen = window.XMLHttpRequest.prototype.open;
```

```
  const origSend = window.XMLHttpRequest.prototype.send;

  window.XMLHttpRequest.prototype.open = function(method, url) {
    this._csrfDest   = new URL(url, location.href).origin;
    this._csrfMethod = method.toUpperCase();
    return origOpen.apply(this, arguments);
  };

  window.XMLHttpRequest.prototype.send = function(body) {
    const dest   = this._csrfDest;
    const method = this._csrfMethod || 'GET';

    if (dest && dest !== location.origin && !SAFE.includes(method)) {
      const allow = confirm(
        `⚠ CSRF Defender:\n` +
        `${method} ${dest}\n` +
        `Виявлено крос-домений POST-запит.\n` +
        `Продовжити?`
      );
      window.postMessage({ source:'csrf-defender', url: dest, allowed: allow }, '*');
      if (!allow) {
        return this.abort();
      }
    }

    return origSend.apply(this, arguments);
  };
})();
```

Лістинг файлу popup.js

```
document.addEventListener('DOMContentLoaded', () => {
  const menuBtn        = document.getElementById('menu-toggle');
  const drawer         = document.getElementById('drawer');
  const closeBtn       = document.getElementById('drawer-close');
  const mainContent    = document.querySelector('.main-content');
  const autoblockChk   = document.getElementById('autoblock-toggle');
  const indicator      = document.getElementById('indicator');
  const statusText     = document.getElementById('status-text');
  const whitelistInput = document.getElementById('whitelist-input');
  const btnWhitelist   = document.getElementById('btn-whitelist');
  const trustedList    = document.getElementById('trusted-list');
  const runAuditBtn    = document.getElementById('runAudit');

  // Drawer toggle
  function toggleDrawer(open) {
    drawer.classList.toggle('open', open);
    mainContent.classList.toggle('blurred', open);
  }
  menuBtn.addEventListener('click', e => {
    e.stopPropagation();
    toggleDrawer(!drawer.classList.contains('open'));
  });
  closeBtn.addEventListener('click', () => toggleDrawer(false));
  document.addEventListener('click', e => {
    if (!drawer.contains(e.target) && !menuBtn.contains(e.target)) {
      toggleDrawer(false);
    }
  });
```

```
// Load settings & initial render
chrome.storage.local.get(['autoBlock','trustedDomains'], data => {
  const auto = data.autoBlock !== false;
  autoblockChk.checked = auto;
  renderTrusted(data.trustedDomains || []);
});

// Auto-block toggle
autoblockChk.addEventListener('change', e => {
  const on = e.target.checked;
  chrome.storage.local.set({ autoBlock: on });
});

// Whitelist
btnWhitelist.addEventListener('click', () => {
  const domain = whitelistInput.value.trim();
  if (!domain) return;
  chrome.storage.local.get({ trustedDomains: [] }, data => {
    const list = data.trustedDomains;
    if (!list.includes(domain)) {
      list.push(domain);
      chrome.storage.local.set({ trustedDomains: list });
      renderTrusted(list);
      whitelistInput.value = '';
    }
  });
});

// Render helpers
function renderTrusted(list) {
```

```
        trustedList.innerHTML = '';
        if (!list.length) {
          const li = document.createElement('li');
          li.textContent = 'Немає довірених доменів';
          trustedList.appendChild(li);
        } else {
          list.forEach((domain, idx) => {
            const li = document.createElement('li');
            li.style.display = 'flex';
            li.style.alignItems = 'center';
            // текст домену
            const span = document.createElement('span');
            span.textContent = domain;
            // кнопка видалення
            const btn = document.createElement('button');
            btn.textContent = '×';
            btn.className = 'remove-btn';
            btn.addEventListener('click', () => {
              // витягаємо поточний масив, видаляємо цей домен, зберігаємо і
рендеримо знову
              chrome.storage.local.get({ trustedDomains: [] }, data => {
                const newList = data.trustedDomains.filter(d => d !== domain);
                chrome.storage.local.set({ trustedDomains: newList });
                renderTrusted(newList);
              });
            });
            li.appendChild(span);
            li.appendChild(btn);
            trustedList.appendChild(li);
          });
```

```
        }
      }

      // --- Логіка навігації зі шторки ---
      const mainContentSections = document.querySelectorAll('#main-content-home, #main-content-audit, #main-content-checklist, #main-content-learn');
      const drawerLinks = document.querySelectorAll('.drawer-section .drawer-link');

      function showSection(sectionId) {
        mainContentSections.forEach(section => {
          section.style.display = 'none';
        });
        const targetSection = document.getElementById(sectionId);
        if (targetSection) {
          targetSection.style.display = 'block';
        }
      }

      drawerLinks.forEach(link => {
        link.addEventListener('click', () => {
          drawerLinks.forEach(l => l.classList.remove('active'));
          link.classList.add('active');

          // Коригуємо визначення targetId для посилання на Білий список
          let targetId = link.id.replace('drawer-link-', 'main-content-');
          if (link.id === 'drawer-link-whitelist') {
            targetId = 'main-content-home'; // Клік на Білий список показує секцію home
          }
```

```
            showSection(targetId);
            toggleDrawer(false); // Закрити шторку після кліку
          });
        });

        // Відображаємо секцію 'Аудит безпеки' за замовчуванням (змінено)
        showSection('main-content-audit'); // Змінено на показ секції Аудиту
        // Встановлюємо активний клас для посилання 'Аудит безпеки'
        const initialLink = document.getElementById('drawer-link-audit'); // Оновлено ID на посилання Аудиту
        if(initialLink) initialLink.classList.add('active');

        // --- Інтерактивні підказки (тепер у секції Навчання) ---
        const infoData = {
          csrf: {
            text: 'CSRF (Cross-Site Request Forgery) — атака, при якій зловмисник змушує користувача виконати небажану дію на сайті, на якому він автентифікований.',
            more: 'https://owasp.org/www-community/attacks/csrf'
          },
          cors: {
            text: 'CORS (Cross-Origin Resource Sharing) — механізм, який дозволяє або забороняє веб-сайтам робити запити до інших доменів.',
            more: 'https://developer.mozilla.org/uk/docs/Web/HTTP/CORS'
          },
          cookie: {
            text: 'Кукі — це невеликі дані, які зберігаються браузером для ідентифікації користувача та збереження сесій. Важливо захищати кукі від крадіжки.',
```

```
            more: 'https://developer.mozilla.org/uk/docs/Web/HTTP/Cookies'
          }
        };
        // Обираємо info-іконки всередині секції Навчання (#main-content-learn)
        document.querySelectorAll('#main-content-learn .info-icon').forEach(icon => {
          icon.addEventListener('click', () => {
            const key = icon.getAttribute('data-info');
            const modal = document.getElementById('modal-info');
            document.getElementById('modal-info-text').textContent = infoData[key].text;
            const moreBtn = document.getElementById('more-link');
            moreBtn.style.display = infoData[key].more ? 'inline-block' : 'none'; // Показувати кнопку тільки якщо є посилання
            moreBtn.onclick = () => window.open(infoData[key].more, '_blank');
            modal.style.display = 'flex';
          });
        });
        // Обробники для модального вікна (залишаються без змін)
        document.querySelector('.close-modal').onclick = () => {
          document.getElementById('modal-info').style.display = 'none';
        };
        window.onclick = function(event) {
          const modal = document.getElementById('modal-info');
          if (event.target === modal) modal.style.display = 'none';
        };

        // --- Чек-лист (тепер в основній частині, перемикається через шторку) ---
```

```
const checklistIds = ['chk-csrf','chk-cookies','chk-https','chk-csp','chk-xfo'];
// Завантажити стан чекбоксів
chrome.storage.local.get({ checklist: {} }, ({ checklist }) => {
  checklistIds.forEach(id => {
    const checkbox = document.getElementById(id);
    if(checkbox) checkbox.checked = !!checklist[id];
  });
});
// Зберігати стан при зміні
checklistIds.forEach(id => {
  const checkbox = document.getElementById(id);
  if(checkbox) {
    checkbox.addEventListener('change', e => {
      chrome.storage.local.get({ checklist: {} }, ({ checklist }) => {
        checklist[id] = e.target.checked;
        chrome.storage.local.set({ checklist });
      });
    });
  }
});

// Запустити аудит при кліку
runAuditBtn.addEventListener('click', async () => {
  const [tab] = await chrome.tabs.query({ active: true, currentWindow: true
});
  console.log('Відправляю RUN_CSRF_AUDIT у вкладку', tab.id);
  // Надсилаємо повідомлення контент-скрипту. Не очікуємо відповіді
тут.
  chrome.tabs.sendMessage(tab.id, { type: 'RUN_CSRF_AUDIT' },
response => {
```

```
            // Обробка відповіді тут більше не потрібна для відображення
результатів
            console.log('Повідомлення про запуск аудиту відправлено.
Результати оновляться автоматично.');
            // Можливо, варто додати індикатор завантаження
          });
        });

        // Обробка результатів аудиту
        chrome.runtime.onMessage.addListener((message, sender, sendResponse)
=> {
          if (message.type === 'AUDIT_COMPLETE') {
            console.log('Отримано результати аудиту:', message.results);
            // Оновлюємо інтерфейс з результатами, використовуючи існуючі
функції відображення
            displayFormsAudit(message.results.forms);
            displayRequestsAudit(message.results.requests);
            displayCookiesAudit(message.results.cookies);
            // Розраховуємо і відображаємо загальний показник безпеки
            const securityScore = calculateSecurityScore(message.results.forms,
message.results.requests, message.results.cookies);
            displaySecurityScore(securityScore);
          } else if (message.type === 'AUDIT_ERROR') {
            console.error('Помилка аудиту:', message.error);
            // Показуємо повідомлення про помилку
            showError(message.error);
          }
        });
```

```
// Функція оновлення результатів аудиту (ця функція більше не використовується)
// ... existing code ...

// Функція показу помилок
function showError(error) {
  const errorContainer = document.getElementById('error-container');
  if (errorContainer) {
    errorContainer.textContent = `Помилка: ${error}`;
    errorContainer.style.display = 'block';
    setTimeout(() => {
      errorContainer.style.display = 'none';
    }, 5000);
  }
}

// Експорт JSON
document.getElementById('export-json').addEventListener('click', () => {
  chrome.storage.local.get({ auditHistory: [] }, items => {
    const blob = new Blob([JSON.stringify(items.auditHistory, null, 2)], {type: 'application/json'}); // Експортуємо тільки auditHistory
    const url = URL.createObjectURL(blob);
    const a = document.createElement('a');
    a.href = url; a.download = 'csrf_audit_history.json'; a.click();
  });
});

// Завантажуємо останні результати аудиту при відкритті popup
chrome.storage.local.get({ auditHistory: [] }, ({ auditHistory }) => {
```

```
        console.log('Завантажено auditHistory при відкритті popup:', auditHistory);
        if (auditHistory.length > 0) {
          const lastAudit = auditHistory[auditHistory.length - 1];
          console.log('Відображаю останні результати аудиту:', lastAudit);
          displayFormsAudit(lastAudit.forms || []); // Додано перевірку на null/undefined
          displayRequestsAudit(lastAudit.requests || []); // Додано перевірку на null/undefined
          displayCookiesAudit(lastAudit.cookies); // Додано виклик
          displaySecurityScore(lastAudit.securityScore || 0); // Додано виклик та перевірку
        } else {
          // Очищаємо результати, якщо історії немає
          displayFormsAudit([]);
          displayRequestsAudit([]);
          displayCookiesAudit({ total: 0, secure: 0, httpOnly: 0, sameSite: { strict: 0, lax: 0, none: 0, unspecified: 0 } });
          displaySecurityScore(0);
        }
      });

      // Слухаємо зміни в storage і оновлюємо попап, якщо він відкритий і історія аудиту оновилася
      chrome.storage.onChanged.addListener((changes, area) => {
        if (area === 'local' && changes.auditHistory) {
          const newAuditHistory = changes.auditHistory.newValue || [];
          console.log('Оновлення auditHistory в storage:', newAuditHistory);
          if (newAuditHistory.length > 0) {
            const lastAudit = newAuditHistory[newAuditHistory.length - 1];
```

```
            console.log('Оновлюю відображення аудиту:', lastAudit);
            displayFormsAudit(lastAudit.forms || []);
            displayRequestsAudit(lastAudit.requests || []);
            displayCookiesAudit(lastAudit.cookies);
            displaySecurityScore(lastAudit.securityScore || 0);
          } else {
            // Очищаємо результати, якщо історія стала порожньою (наприклад, після очищення)
            displayFormsAudit([]);
            displayRequestsAudit([]);
            displayCookiesAudit({ total: 0, secure: 0, httpOnly: 0, sameSite: { strict: 0, lax: 0, none: 0, unspecified: 0 } });
            displaySecurityScore(0);
          }
        }
      });

      // Обробник для кнопки очищення історії аудиту
      document.getElementById('clearAudit').addEventListener('click', () => {
        chrome.storage.local.set({ auditHistory: [] }, () => {
          console.log('Історія аудиту очищена.');
          // Оновлення відображення відбудеться автоматично через storage.onChanged
        });
      });

      // Функція для відображення результатів аудиту форм
      function displayFormsAudit(forms) {
        const formsList = document.getElementById('formsList');
        formsList.innerHTML = '';
```

```
        formsList.classList.remove('fade-in');

        if (!forms.length) {
          formsList.innerHTML = '<div class="empty-msg">Немає знайдених форм</div>';
          formsList.classList.add('fade-in');
          return;
        }

        forms.forEach(form => {
          const formElement = document.createElement('div');
          formElement.className = `form-item ${form.hasCsrfToken && form.hasSecureMethod ? 'secure' : 'insecure'}`;

          formElement.innerHTML = `
            <div class="form-header">
              <strong>Форма ${form.id + 1}</strong>
              <span class="status-badge ${form.hasCsrfToken ? 'success' : 'danger'}">
                ${form.hasCsrfToken ? 'Захищена' : 'Незахищена'}
              </span>
            </div>
            <div class="form-details">
              <p>Метод: ${form.method}</p>
              <p>URL: ${form.action}</p>
              <div class="security-indicators">
                <span class="indicator ${form.hasSecureMethod ? 'success' : 'warning'}" title="POST — безпечний, GET — небажано">
                  Безпечний метод: ${form.hasSecureMethod ? '✓' : '✕'}
```

```
            </span>
            <span class="indicator ${form.hasSecureAction ? 'success' :
'warning'}" title="HTTPS — безпечний, HTTP — небажано">
              Безпечний URL: ${form.hasSecureAction ? '✓' : '✗'}
            </span>
            <span class="indicator ${form.hasSameOriginAction ? 'success' :
'warning'}" title="Той самий домен — безпечно">
              Same Origin: ${form.hasSameOriginAction ? '✓' : '✗'}
            </span>
          </div>
          <div class="security-headers">
            <h4>Заголовки безпеки:</h4>
            <ul>
              <li>CSP: ${form.securityHeaders.hasContentSecurityPolicy ? '✓' :
'✗'}</li>
              <li>X-Frame-Options: ${form.securityHeaders.hasXFrameOptions ?
'✓' : '✗'}</li>
              <li>X-Content-Type-Options:
${form.securityHeaders.hasXContentTypeOptions ? '✓' : '✗'}</li>
            </ul>
          </div>
        </div>
        <div class="form-inputs">
          <strong>Поля форми:</strong>
          <ul>
            ${form.inputs.map(input => `
              <li>
                ${input.name} (${input.type})
                ${input.required ? '<span class="required">*</span>' : ''}
```

```
                    <div class="input-security">
                      ${input.hasAutocomplete ? '<span class="security-
feature">autocomplete</span>' : ''}
                      ${input.hasPattern ? '<span class="security-
feature">pattern</span>' : ''}
                      ${input.hasMinLength ? '<span class="security-
feature">minlength</span>' : ''}
                      ${input.hasMaxLength ? '<span class="security-
feature">maxlength</span>' : ''}
                    </div>
                  </li>
                `).join('')}
              </ul>
            </div>
          `;

          formsList.appendChild(formElement);
        });
        formsList.classList.add('fade-in');
      }

      // Функція для відображення результатів аудиту запитів
      function displayRequestsAudit(requests) {
        const requestsList = document.getElementById('requestsList');
        requestsList.innerHTML = '';
        requestsList.classList.remove('fade-in');

        // Додаємо пояснення про CORS
        const corsInfo = document.createElement('div');
        corsInfo.className = 'cors-info';
```

```
        corsInfo.innerHTML = '<b>Примітка:</b> Деякі AJAX-запити могли бути заблоковані політикою CORS, але сам факт їх відправки зафіксовано.';
        requestsList.appendChild(corsInfo);

        if (!requests.length) {
         requestsList.innerHTML += '<div class="empty-msg">Немає знайдених AJAX-запитів</div>';
         requestsList.classList.add('fade-in');
         return;
        }

        requests.forEach(request => {
         const requestElement = document.createElement('div');
         requestElement.className = 'request-item';

         requestElement.innerHTML = `
          <div class="request-header">
           <strong>${request.type}</strong>
           <span class="timestamp">${new Date(request.timestamp).toLocaleTimeString()}</span>
          </div>
          <div class="request-details">
           <p>Метод: ${request.method}</p>
           <p>URL: ${request.url}</p>
           <div class="security-indicators">
            <span class="indicator ${request.security.hasCsrfToken ? 'success' : 'warning'}" title="CSRF Token: наявність захисного токена у запиті">
             CSRF Token: ${request.security.hasCsrfToken ? '✓' : '✗'}
            </span>
```

```
            <span class="indicator ${request.security.isSameOrigin ? 'success' :
'warning'}" title="Same Origin: запит на той самий домен">
              Same Origin: ${request.security.isSameOrigin ? '✓' : '✗'}
            </span>
            <span class="indicator ${request.security.hasSecureProtocol ?
'success' : 'warning'}" title="HTTPS: зашифрований запит">
              HTTPS: ${request.security.hasSecureProtocol ? '✓' : '✗'}
            </span>
            <span class="indicator ${request.corsBlocked ? 'danger' : 'success'}"
title="CORS: чи була відповідь заблокована політикою CORS">
              CORS: ${request.corsBlocked ? '✗ (запит заблоковано політикою
CORS)' : '✓ (відповідь дозволена)'}
            </span>
          </div>
        </div>
      `;

      requestsList.appendChild(requestElement);
    });
    requestsList.classList.add('fade-in');
  }

  // Функція для відображення результатів аудиту кукі
  function displayCookiesAudit(cookies) {
    const cookiesList = document.getElementById('cookiesList');
    if (!cookiesList) return;
    cookiesList.classList.remove('fade-in');

    if (!cookies || cookies.total === 0) {
```

```
            cookiesList.innerHTML = '<div class="empty-msg">Немає знайдених
кукі</div>';
            cookiesList.classList.add('fade-in');
            return;
          }

          cookiesList.innerHTML = `
            <div class="cookies-summary">
              <h4>Загальна статистика кукі:</h4>
              <ul>
                <li>Всього кукі: ${cookies.total}</li>
                <li>Secure: ${cookies.secure}
(${Math.round(cookies.secure/cookies.total*100)}%)</li>
                <li>HttpOnly: ${cookies.httpOnly}
(${Math.round(cookies.httpOnly/cookies.total*100)}%)</li>
              </ul>
              <h4>SameSite атрибути:</h4>
              <ul>
                <li>Strict: ${cookies.sameSite.strict}
(${Math.round(cookies.sameSite.strict/cookies.total*100)}%)</li>
                <li>Lax: ${cookies.sameSite.lax}
(${Math.round(cookies.sameSite.lax/cookies.total*100)}%)</li>
                <li>None: ${cookies.sameSite.none}
(${Math.round(cookies.sameSite.none/cookies.total*100)}%)</li>
                <li>Не вказано: ${cookies.sameSite.unspecified}
(${Math.round(cookies.sameSite.unspecified/cookies.total*100)}%)</li>
              </ul>
            </div>
          `;
          cookiesList.classList.add('fade-in');
```

```
}

// Функція для відображення загального показника безпеки
function displaySecurityScore(score) {
  const scoreElement = document.getElementById('securityScore');
  if (!scoreElement) return;
  scoreElement.classList.remove('fade-in');

  let scoreClass = 'low';
  let icon = '😱';
  let desc = 'Низький рівень безпеки';
  if (score >= 80) {
    scoreClass = 'high';
    icon = '🛡️';
    desc = 'Високий рівень безпеки';
  } else if (score >= 50) {
    scoreClass = 'medium';
    icon = '⚠️';
    desc = 'Середній рівень безпеки';
  }

  scoreElement.innerHTML = `
    <div class="security-score ${scoreClass}">
      <span class="score-icon">${icon}</span>
      <div class="score-value">${score}/100</div>
      <div class="score-description">${desc}</div>
    </div>
  `;
  scoreElement.classList.add('fade-in');
```

```
        }

        // Функція для розрахунку загального показника безпеки
        function calculateSecurityScore(forms, requests, cookies) {
          console.log('calculateSecurityScore: Вхідні дані - forms:', forms,
'requests:', requests, 'cookies:', cookies);
          const maxScore = 100;

          // Максимальні бали за один елемент
          const maxFormItemScore = 9; // 2(csrf) + 1(method) + 1(action) +
1(sameOrigin) + 1(contentType) + 3(headers)
          const maxRequestItemScore = 4; // 2(csrf) + 1(sameOrigin) +
1(secureProtocol)

          // Оцінка форм (40% від загального балу)
          let formScore = 0;
          let formEarnedScoreSum = 0;
          if (forms && forms.length > 0) { // Додано перевірку forms на існування
            formEarnedScoreSum = forms.reduce((acc, form) => {
              let formPoints = 0;
              if (form.hasCsrfToken) formPoints += 2;
              if (form.hasSecureMethod) formPoints += 1;
              if (form.hasSecureAction) formPoints += 1;
              if (form.hasSameOriginAction) formPoints += 1;
              if (form.hasContentType) formPoints += 1;
              if (form.securityHeaders.hasContentSecurityPolicy) formPoints += 1;
              if (form.securityHeaders.hasXFrameOptions) formPoints += 1;
              if (form.securityHeaders.hasXContentTypeOptions) formPoints += 1;
              return acc + formPoints;
            }, 0);
```

```
            const averageFormScore = formEarnedScoreSum / forms.length;
            formScore = (averageFormScore / maxFormItemScore) * 0.4;
            console.log('calculateSecurityScore: Forms Score - Sum:',
formEarnedScoreSum, 'Average:', averageFormScore, 'Normalized+Weighted:',
formScore);

        } else {
            console.log('calculateSecurityScore: No forms found or forms data is
invalid.');
        }

        // Оцінка запитів (30% від загального балу)
        let requestScore = 0;
        let requestEarnedScoreSum = 0;
        if (requests && requests.length > 0) { // Додано перевірку requests на
існування
            requestEarnedScoreSum = requests.reduce((acc, request) => {
                let requestPoints = 0;
                if (request.security.hasCsrfToken) requestPoints += 2;
                if (request.security.isSameOrigin) requestPoints += 1;
                if (request.security.hasSecureProtocol) requestPoints += 1;
                // Можливо, варто враховувати CORSBlocked як негативний
фактор?
                // if (request.corsBlocked) requestPoints -= 1; // Приклад зменшення
балу за CORS
                return acc + requestPoints;
            }, 0);
            const averageRequestScore = requestEarnedScoreSum / requests.length;
            requestScore = (averageRequestScore / maxRequestItemScore) * 0.3;
```

```
            console.log('calculateSecurityScore: Requests Score - Sum:',
requestEarnedScoreSum, 'Average:', averageRequestScore,
'Normalized+Weighted:', requestScore);
          } else {
            console.log('calculateSecurityScore: No requests found or requests data
is invalid.');
          }

        // Оцінка кукі (30% від загального балу)
        let cookieScore = 0;
        let cookieSecurityPercentage = 0;
        if (cookies && cookies.total >= 0) { // Додано перевірку cookies на
існування та total >= 0
          cookieSecurityPercentage = (
           (cookies.total > 0 ? cookies.secure / cookies.total : 0) * 0.3 +
           (cookies.total > 0 ? cookies.httpOnly / cookies.total : 0) * 0.3 +  //
Виправлено синтаксис тернарного оператора
            (cookies.total > 0 ? cookies.sameSite.strict / cookies.total : 0) * 0.4 //
Виправлено синтаксис тернарного оператора
          );
          cookieScore = cookieSecurityPercentage * 0.3; // Вага кукі в загальному
score
           console.log('calculateSecurityScore: Cookies Score - Percentage:',
cookieSecurityPercentage, 'Weighted:', cookieScore);
        } else {
          console.log('calculateSecurityScore: No cookies found or cookies data is
invalid.');
        }

        // Загальний бал
```

```
        let rawFinalScore = formScore + requestScore + cookieScore;
        console.log('calculateSecurityScore: Raw Final Score (sum of weighted scores):', rawFinalScore);

        // Переконуємося, що фінальний бал не перевищує 1 і не менше 0
        let finalScore = Math.min(rawFinalScore, 1);
        finalScore = Math.max(finalScore, 0);
         console.log('calculateSecurityScore: Capped Final Score (0-1):', finalScore);

        const roundedScore = Math.round(finalScore * maxScore);
         console.log('calculateSecurityScore: Final Rounded Score (0-100):', roundedScore);

        return roundedScore;
    }
```