МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

КОВАЛЬЧУК ЛІЗА ПЕТРІВНА

Допускається до захисту:
Завідувач кафедри прикладної
математики та кібербезпеки, д-р
філософії з математики
_____ Луценко А. В.
«__»_____20__р.

МЕТОДИ І СПОСОБИ ЗАХИСТУ ВІД ШКІДЛИВИХ МІЖДОМЕННИХ ЗАПИТІВ

Спеціальність 125 Кібербезпека
Кваліфікаційна (бакалаврська) робота

Керівник:
Загоруйко Л. В.,
доцент кафедри прикладної
математики та кібербезпеки

_____
(підпис)

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

Вінниця – 2025

АНОТАЦІЯ

Ковальчук Л. П. Методи і способи захисту від шкідливих міждоменних запитів. Спеціальність 125 «Кібербезпека». Донецький національний університет імені Василя Стуса, Вінниця, 2025.

У кваліфікаційній (бакалаврській) роботі досліджено механізми міждоменної взаємодії веб-сайтів, зосереджуючись на проблематиці шкідливих міждоменних запитів. Проаналізовано методи і способи захисту від шкідливих міждоменних запитів. Показано технології тестування веб-додатків на вразливість до шкідливих міждоменних запитів та наведено приклад ефективного підходу.

Ключові слова: міждоменні запити, веб-сайт, CORS, налаштування параметрів, тестування.

44 с., 2 табл., 8 рис., 1 дод., 26 джерел.

ABSTRACT

Kovalchuk L. P. Methods and Techniques of Protection Against Malicious Cross-Domain Requests. Specialty 125 "Cybersecurity". Vasyl Stus Donetsk National University, Vinnytsia, 2025.

This bachelor’s qualification paper investigates the mechanisms of cross-domain interaction between websites, focusing on the issue of malicious cross-domain requests. The study analyzes methods and techniques for protecting against malicious cross-domain requests. It also presents technologies for testing web applications for vulnerabilities to such requests and provides an example of an effective approach.

Keywords: cross-domain requests, website, CORS, configuration parameters, testing.

44 pp, 2 tables, 8 figures, 1 appendix, 26 sources.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface

Cookie – невеликий фрагмент даних, відправлений веб-сервером і збережений на комп’ютері користувача.

CORS – Cross-Origin Resource Sharing

CSRF – Cross-Site Request Forgery

DOM – Document Object Model

HTML – Hyper Text Markup Language

HTTP – Hypertext Transfer Protocol

MitM — man-in-the-middle

SDK – Software Development Kit

SSL — Secure Sockets Layer

URL – Uniform Resource Locator

OWASP – The Open Worldwide Application Security Project

W3C – World Wide Web Consortium

XML – EXtensible Markup Language

XSS – Cross-Site Scripting

ВСТУП

Актуальність дослідження. У контексті стрімкого розвитку веб-технологій та широкого впровадження розподілених архітектур, міждоменна взаємодія стала ключовим елементом сучасних веб-застосунків.

Водночас міждоменна взаємодія створює додаткові вектори атак, зокрема через механізми, що використовують браузерні особливості, такі як автоматичне надсилання cookie або незахищені API-запити.

Особливо актуальною ця проблема стає для сучасних архітектурних рішень при розробці веб-сайтів. Відсутність належного захисту у таких сценаріях створює критичні ризики як для користувачів, так і для бізнесу. Тому дослідження існуючих методів і способів захисту від шкідливих міждоменних запитів має важливе теоретичне й практичне значення. Це дозволяє підвищити рівень безпеки веб-систем і сформувати комплексний підхід до захисту міждоменної взаємодії.

Метою дослідження є дослідити існуючі методи і способи захисту від шкідливих міждоменних запитів. Запропонувати ефективний спосіб їх тестування оцінити рівень захищеності, який вони надають.

Для досягнення поставленої мети потрібно виконати наступні **завдання**:

1. Аналіз шкідливих міждоменних запитів.
2. Огляд існуючих методів і засобів захисту від шкідливих міждоменних запитів.
3. Аналіз підходів до тестування на вразливість до шкідливих міждоменних запитів.
4. Створення скрипту для сканування на вразливість до шкідливих міждоменних запитів.

Об’єкт дослідження є методи і способи захисту від шкідливих міждоменних запитів.

Предмет дослідження є оцінка ефективності поєднання проаналізованих методи і способи захисту від шкідливих міждоменних запитів.

Апробація. Стаття на тему “Шкідливі міждоменні запити” була опубліковані у науковому журналі “Вісник студентського наукового товариства ДонНУ імені Василя Стуса” Том 1 № 17 (2025) м. Вінниця.

Структура роботи. Бакалаврська робота складається із вступу, трьох розділів, висновків, списку використаних джерел та додатків. Загальний обсяг роботи 44 сторінок, містить 2 таблиць, 8 рисунки.

РОЗДІЛ 1 АНАЛІЗ ДОСЛІДЖЕНЬ ШКІДЛИВИХ МІЖДОМЕННИХ ЗАПИТІВ

1.1 Застосування міждоменних запитів

На початку створення глобальної мережі та перших сайтів міждоменні запипити на використання вмісту були неможливі. Архітектура статичного веб-сайту була простою. Вона містила лише один HTML файл який містив DOM-дерево, яке не містило сторонніх підключень. Такі сайти не приймають запитів від користувачів на сервер та не містять інтерактивних елементів.

Але із зростанням кількості користувачів у Інтернеті почали змінюватися інструменти для створення веб-сайтів. Так для створення інтерактивних елементів та зручності підтримки й масштабування сайтів з'явилися технології міждоменних запитів.

Якщо ми зайдемо, наприклад, на сучасний сайт ми з високою ймовірністю побачимо там фото, відео, карти міста. Усі ці дані мають різний формат та не містяться в головному статичному DOM-дереві. Такі дані містяться на сторонніх серверах і відображаються тільки коли користувач відкриває сторінку тим самим посилає запит на отримання вмісту сторонього веб-ресурсу.

Для відтворення аудіо та відеофайлів у мові розмітки HTML створили тег "<source>" із атрибутом "src=", який містить URL-адреса на сторінку, яка визначає зміст елемента.

Передача даних відбувається за допомогою протоколу HTTP. HTTP – це протокол для отримання ресурсів, наприклад, за допомогою браузера . Він лежить в основі обміну даними в Інтернеті та є протоколом клієнт-серверної взаємодії, що означає ініціювання запитів до сервера самим одержувачем, зазвичай веб-браузером [1].

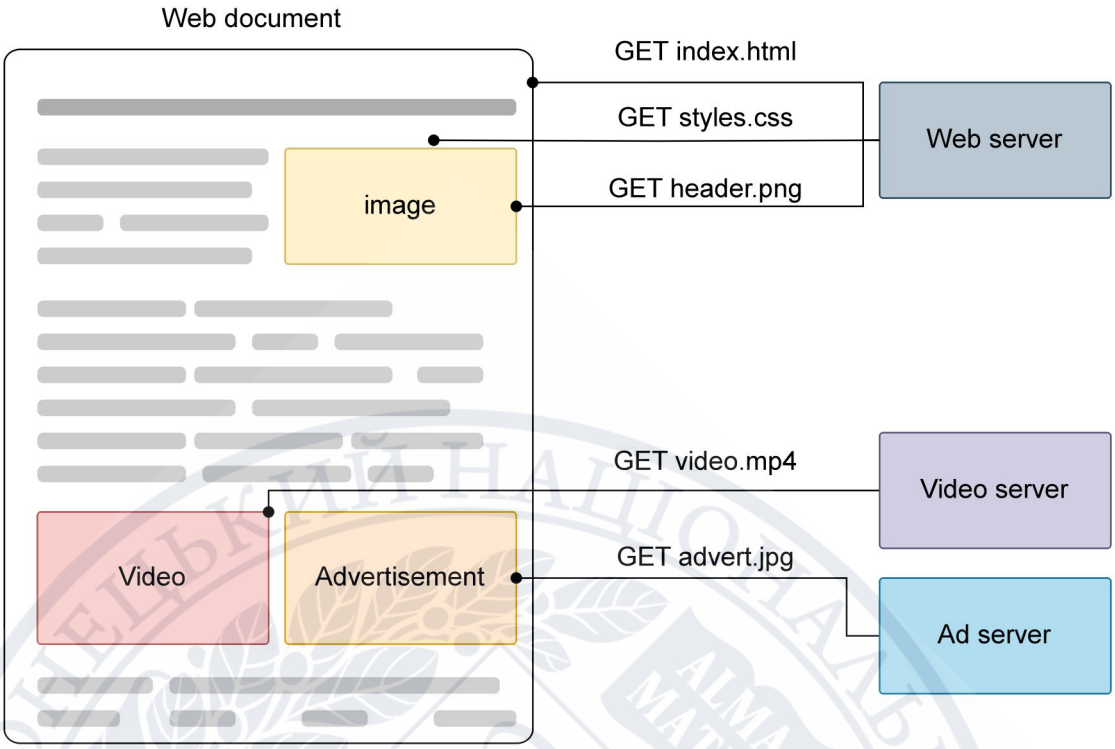

Рис. 1.1.1 — Структура сучасного веб-сайту

Міждоменні запити — це важливий механізм для реалізації багатьох архітектурних рішень у веб-розробці. Він дозволяє реалізувати розподілене розміщення всіх ресурсів, що в свою чергу дозволяє легко підтримувати, масштабувати та розвивати проєкти великою командою. Ми можемо це побачити коли міждоменні запити використовуються у фронтенд додатках для взаємодії з бекендом, розміщеним на іншому домені, для інтеграції сторонніх сервісів, у мікросервісній архітектурі, коли різні сервіси мають власні домени або піддомени.

Міждоменні запити у фронтенд-додатках виконують критично важливу роль у сучасній архітектурі веб-застосунків, де інтерфейс користувача та серверна логіка часто розміщені на різних доменах. Такий поділ є типовим для хмарних інфраструктур, а також при використанні сторонніх API. Наприклад, коли фронтенд додатку розміщений на окремому сервері, а бекенд, який обробляє запити, — на іншому, необхідна можливість комунікації між цими двома компонентами. Такі запити виникають, коли веб-додаток, що завантажений у браузері з одного походження (тобто домену, порту та протоколу), намагається

отримати ресурси з іншого походження. Політика однакового походження, яку реалізують браузери, забороняє доступ до відповідей з інших походжень, щоб запобігти несанкціонованому доступу до даних користувача. Проте в багатьох реальних сценаріях потрібно, щоб різні домени взаємодіяли між собою. Для цього застосовується механізм міждоменних запитів, який дозволяє серверу явно вказати, яким саме походженням дозволено звертатися до його ресурсів [2].

Міждоменні запити широко використовуються для інтеграції сторонніх сервісів у веб-додатках. Це дозволяє розробникам швидко та ефективно додавати функціональність, яка вимагає значних ресурсів або спеціалізованих знань, без необхідності реалізовувати її самостійно. Наприклад, сервіси на кшталт Google Maps, Stripe, Firebase або Facebook Login надають потужні інструменти для реалізації картографії, обробки платежів, автентифікації користувачів, хмарного збереження даних та інших ключових можливостей через API, до яких доступ здійснюється за допомогою міждоменних запитів.

Коли веб-додаток інтегрує сторонній сервіс, він, як правило, взаємодіє з API, який розміщений на іншому домені, ніж сам додаток. Наприклад, при використанні Stripe для обробки платежів, клієнтська частина додатку звертається до серверів Stripe, що знаходяться на домені stripe.com. Браузер при цьому застосовує політику одного походження і дозволяє такі запити лише у випадку, якщо сервер стороннього сервісу налаштований відповідно до стандарту CORS і явно дозволяє звернення з походження додатку. Аналогічно, коли користувач входить через Facebook Login, фронтенд надсилає запит до доменів Facebook, а той повертає відповідь, що містить токен або дані профілю користувача — знову ж таки, це міждоменна комунікація, яку потрібно обробити безпечно та відповідно до стандартів.

Інтеграція сторонніх сервісів за допомогою міждоменних запитів вирішує низку проблем. По-перше, вона дозволяє делегувати складну та чутливу логіку (наприклад, зберігання платіжних даних, автентифікацію, облік сесій) спеціалізованим провайдерам, які гарантують високу надійність і відповідність

сучасним вимогам безпеки. По-друге, це скорочує час розробки, оскільки більшість таких сервісів надають добре задокументовані API та SDK, адаптовані під популярні фреймворки. По-третє, така інтеграція підвищує масштабованість і надійність додатку, оскільки обробка ресурсомістких запитів покладається на зовнішню інфраструктуру[3].

У той час як фронтенд-додатки активно використовують міждоменні запити для зв'язку з бекендом, а сторонні сервіси — для розширення функціональності веб-застосунків, мікросервісна архітектура ставить ці механізми на службу внутрішній взаємодії всередині однієї розподіленої системи. У таких архітектурах функціональність розбивається на незалежні сервіси, кожен з яких відповідає за окрему ділянку бізнес-логіки — наприклад, керування користувачами, обробку платежів, відправлення повідомлень, генерацію звітів тощо. Ці сервіси можуть бути реалізовані різними командами, на різних технологіях, і навіть розгортатися в різних середовищах. Внаслідок цього взаємодія між ними часто відбувається не в межах одного походження, а через окремі домени або піддомени.

У цьому контексті міждоменні запити стають внутрішнім інфраструктурним інструментом, що підтримує цілісність розподіленої системи. Вони забезпечують комунікацію між логічно пов'язаними, але фізично ізольованими компонентами, не порушуючи безпекової моделі браузера. На відміну від інтеграції зі сторонніми API, де доступ до ресурсів має бути обмежений і вибірковий, у мікросервісах міждоменна взаємодія часто налаштована на тісну і постійну співпрацю між службами. Проте навіть у такому середовищі важливо чітко регламентувати дозволені походження, автентифікацію сервісів, обмін токенами та обробку запитів з урахуванням можливих вразливостей [4].

Завдяки міждоменним запитам мікросервісна архітектура зберігає модульність та взаємозамінність компонентів. У разі потреби певний сервіс можна оновити або замінити без істотного впливу на всю систему — достатньо лише зберегти його інтерфейс і дозволені шляхи взаємодії. Це робить систему більш гнучкою до змін, розширюваною, а також стійкою до збоїв, оскільки

відмова одного сервісу не обов’язково призводить до повного зупинення роботи [5].

Отже, міждоменні запити є незамінним інструментом у сучасній веб-розробці, що використовується практично в усіх актуальних архітектурних підходах — від класичних клієнт-серверних рішень до інтеграції сторонніх сервісів і мікросервісних систем. Їхнє широке застосування зумовлено гнучкістю, яку вони надають у побудові масштабованих, розподілених і функціонально насичених веб-застосунків. Водночас, хоча міждоменна взаємодія пов’язана з низкою потенційних вразливостей у сфері безпеки, ці ризики не нівелюють загальну користь та необхідність цього механізму. Навпаки, вони лише підкреслюють важливість наявності компетентного підходу до реалізації міждоменних запитів, який враховує сучасні вимоги до захисту даних та контролю доступу. Відмова від міждоменної взаємодії означала б суттєве обмеження функціональності та інтеграційних можливостей сучасного вебу, тоді як грамотне використання цього інструмента дозволяє ефективно вирішувати широкий спектр завдань, забезпечуючи при цьому необхідний рівень безпеки.

1.2 Структура міждоменних запитів

API (Application Programming Interface) — це інтерфейс прикладного програмування, який визначає спосіб взаємодії між різними програмними компонентами. У веб-розробці API найчастіше використовується як набір правил та протоколів, за якими один додаток (наприклад, фронтенд) може обмінюватися даними з іншим — зазвичай бекендом або стороннім сервісом. Ця взаємодія відбувається через HTTP-запити, які дозволяють отримувати, надсилати, змінювати або видаляти дані на сервері [6].

При розробці власного сайту можна знайти багато доступних API на офіційних сайтах великих компаній. Наприклад, легко можна реалізувати актуальний курс валют чи погодні умови.

Розглянемо особливості реалізації міждоменних запитів у веб-продуктах, розроблених з використанням мови програмування JavaScript. JavaScript як основна мова для клієнтської частини веб-застосунків забезпечує широкий набір інструментів для ініціації HTTP-запитів, зокрема за допомогою fetch, XMLHttpRequest, або бібліотек типу Axios. Саме через ці механізми відбувається взаємодія з API, що розміщені на зовнішніх доменах. Реалізація міждоменної взаємодії в JavaScript вимагає не лише технічного налаштування запитів, але й розуміння безпекових обмежень та принципів обробки відповідей з інших походжень.

XMLHttpRequest. XMLHttpRequest — це стандартний об'єкт, вбудований у більшість сучасних браузерів, який дає змогу JavaScript коду надсилати HTTP-запити до сервера без необхідності перезавантаження сторінки. Його поява стала ключовим етапом у розвитку динамічних веб-додатків, оскільки дозволила реалізовувати так звану "асинхронну" взаємодію між клієнтом і сервером. Хоча його назва містить слово "XML", цей об'єкт здатен працювати з різноманітними форматами даних, а не лише з XML. З його допомогою можна надсилати та отримувати файли з сервера, контролювати хід запиту та реалізовувати інші аспекти асинхронної взаємодії з серверною частиною.

За замовчуванням об’єкт XMLHttpRequest не передає стороннім джерелам такі дані, як HTTP-заголовки авторизації та cookie. Це зроблено з міркувань безпеки, щоб запобігти несанкціонованій передачі облікової інформації під час міждоменних запитів.

Однак, якщо потрібно забезпечити автентифікований доступ до ресурсу на іншому домені, можна вручну дозволити передачу таких даних. Для цього необхідно встановити властивість xhr.withCredentials у значення true. Це дає змогу включати cookie, токени сесії або інші облікові дані у запит, але лише за умови, що сервер також налаштований приймати такі запити [7].

Розглянемо нижче код, який демонструє повноцінний цикл роботи з XMLHttpRequest: ініціалізацію, надсилання, обробку відповіді, відстеження

прогресу та обробку помилок. Такий підхід є базовим прикладом реалізації асинхронної взаємодії з сервером у JavaScript.

```
1   let xhr = new XMLHttpRequest();
2
3   xhr.open('GET', '/article/xmlhttprequest/example/load');
4
5   xhr.send();
6
7   xhr.onload = function() {
8     if (xhr.status != 200) {
9       alert(`Помилка ${xhr.status}: ${xhr.statusText}`);
10    } else {
11      alert(`Запит завершено, отримано ${xhr.response.length} байт`);
12    }
13  };
14
15  xhr.onprogress = function(event) {
16    if (event.lengthComputable) {
17      alert(`Отримано ${event.loaded} із ${event.total} байт`);
18    } else {
19      alert(`Отримано ${event.loaded} байт`);
20    }
21  };
22
23  xhr.onerror = function() {
24    alert("Не вдалося виконати запит");
25  };
```

Рис. 1.2.1 — Програмний код виконаний мовою програмування JavaScript

Рядок 1 створює новий екземпляр об’єкта XMLHttpRequest, який дозволяє надсилати HTTP-запити до сервера та обробляти відповіді.

Рядок 3 — ініціалізація.

Основні параметри запиту, які методи приймає:

- method – HTTP-метод. Зазвичай "GET" або "POST".
- URL – URL для запиту, зазвичай це рядок, але може бути і об’єктом URL.
- async – якщо явно встановлено значення false, тоді запит буде синхронним, ми розглянемо це трохи пізніше.
- user, password – логін та пароль для базової HTTP-аутентифікації (якщо потрібно).

Метод open, незважаючи на свою назву, не встановлює з’єднання з сервером. Його завдання — лише підготувати параметри запиту. Реальна мережева активність розпочинається лише після виклику методу send.

Рядок 5 — надсилання запиту.

Цей метод встановлює з’єднання та ініціює надсилання запиту до сервера. Параметр body, який є необов’язковим, може містити вміст запиту.

Деякі HTTP-методи, зокрема GET, не передбачають наявності тіла запиту. Натомість інші, наприклад POST, активно використовують body для передавання даних на сервер. Приклади такого використання ми розглянемо далі.

Рядки 7-13 — обробка завершення запиту.

Цей блок коду виконується після завершення запиту:

Якщо статус відповіді не 200 (успішно), виводиться повідомлення про помилку із кодом статусу та описом (statusText).

Якщо статус 200, виводиться повідомлення про успішне завершення запиту та кількість отриманих байтів у відповіді.

Рядки 15-21 — відстеження прогресу завантаження.

Цей обробник дозволяє відстежувати прогрес завантаження:

Якщо відома загальна кількість байтів (lengthComputable), показується співвідношення завантаженого об’єму до загального.

Інакше показується лише обсяг вже отриманих даних.

Рядки 23-25 — обробка помилок.

Цей обробник викликається у разі виникнення помилки під час виконання запиту (наприклад, якщо сервер не відповідає або відсутнє інтернет-з’єднання).

Об’єкт XMLHttpRequest також підтримує велику кількість подій, які суттєво спрощують роботу зі станами запитів, обробкою HTTP-заголовків, а також із формуванням POST-запитів, наприклад, на основі даних з HTML-форми. Ці події надають розробнику гнучкий інструментарій для побудови складної логіки взаємодії з сервером у реальному часі.

Однак у межах цієї роботи ми не будемо детально зупинятися на всіх можливостях та подіях XMLHttpRequest, оскільки їх розгляд виходить за межі основної теми. Замість цього ми зосередимося на загальних принципах використання та ключових аспектах, що стосуються міждоменної взаємодії.

На сьогодні об'єкт XMLHttpRequest усе ще залишається працездатним і активно підтримується більшістю сучасних веб-браузерів. Проте у нових проєктах його дедалі частіше замінюють більш сучасним та зручним інструментом — вбудованим інтерфейсом Fetch.

Fetch. Fetch вважається сучасною альтернативою XMLHttpRequest, оскільки пропонує спрощений синтаксис, базується на використанні промісів та забезпечує кращу інтеграцію з іншими сучасними JavaScript технологіями. Його використання робить код чистішим, більш зрозумілим і легким для підтримки.

Інтерфейс Fetch реалізовано на основі об'єкта Promise, що дає змогу організовувати асинхронні HTTP-запити з мінімальними витратами на обробку. Такий підхід дозволяє використовувати синтаксис then/catch або async/await, що істотно покращує читабельність і підтримуваність коду. Асинхронність Fetch API усуває необхідність у багаторівневих зворотних викликах (callback hell), які були типовими для XMLHttpRequest.

Метод fetch(input, init) приймає два аргументи: ресурс, до якого здійснюється запит, та об’єкт параметрів. Останній дозволяє налаштовувати ключові характеристики запиту:

- HTTP-метод (method: GET, POST, PUT, DELETE тощо),
- заголовки (headers),
- тіло запиту (body),
- автентифікаційні дані (credentials),
- політику CORS (mode),
- поведінку кешування (cache),
- правила переадресації (redirect).

Така структура сприяє універсальності застосування API як у простих, так і у високонавантажених розподілених системах.

Fetch API забезпечує повноцінну підтримку усіх стандартних HTTP-методів, включаючи HEAD, OPTIONS та PATCH, що дозволяє реалізовувати повний набір CRUD-операцій згідно з принципами RESTful-архітектури.

Однією з технічних переваг Fetch є можливість передавання вмісту тіла запиту у багатьох форматах:

- текстові значення (типу text/plain),
- JSON (із застосуванням JSON.stringify()),
- об’єкти типу FormData для HTML-форм,
- бінарні дані (Blob, ArrayBuffer),
- потоки (ReadableStream).

Це дозволяє адаптувати API до різних сценаріїв: від надсилання простих форм до завантаження великих медіафайлів або роботи з потоковими даними.

Об’єкт Response, який повертається після виконання запиту, надає доступ до таких методів, як .json(), .text(), .blob(), .formData() та .arrayBuffer(). Це дає змогу ефективно працювати з різними типами відповідей сервера без необхідності додаткового парсингу або перетворень. Крім того, Response надає метаінформацію, зокрема статус відповіді, заголовки, тип контенту тощо [8].

Fetch API дотримується політики Cross-Origin Resource Sharing (CORS) за замовчуванням. Для того, щоб дозволити надсилання cookie, токенів або інших облікових даних у міждоменних запитах, необхідно встановити параметр credentials: 'include'. Сумісність зі специфікацією CORS робить Fetch безпечним та надійним засобом у побудові розподілених систем із захищеним обміном даними між клієнтом і сервером.

Для прикладу розглянемо реалізацію POST запиту за допомогою Fetch.

```javascript
async function postData(url = "", data = {}) {

  const response = await fetch(url, {
    method: "POST", // *GET, POST, PUT, DELETE, etc.
    mode: "cors", // no-cors, *cors, same-origin
    cache: "no-cache", // *default, no-cache, reload, force-cache, only-if-cached
    credentials: "same-origin", // include, *same-origin, omit
    headers: {
      "Content-Type": "application/json",
    },
    redirect: "follow", // manual, *follow, error
    referrerPolicy: "no-referrer", // no-referrer, *client
    body: JSON.stringify(data), // body data type must match "Content-Type" header
  });
  return await response.json(); // parses JSON response into native JavaScript objects
}

postData("https://example.com/answer", { answer: 42 }).then((data) => {
  console.log(data); // JSON data parsed by `response.json()` call
});
```

Рис. 1.2.2 — Програмний код виконаний мовою програмування JavaScript

Коротко розглянемо які налаштування використані в наведеному вище коді:

- fetch(url, options) відправляє HTTP-запит за заданою адресою,
- метод POST використовується для відправки даних,
- встановлено режим cors, що дозволяє робити запити між різними доменами,
- кешування вимкнене через cache: "no-cache",
- дані відправляються в форматі application/json,
- політика реферера no-referrer означає, що браузер не буде надсилати заголовок Referer,
- дані з об'єкта data серіалізуються у JSON через JSON.stringify,
- відповідь парситься в об'єкт через response.json().

Узагальнюючи, Fetch API є технічно прогресивним інструментом для роботи з HTTP-запитами у веб-середовищі. Він надає широкий набір налаштувань та функцій. Чим усуває необхідність використання сторонніх бібліотек.

Axios. Axios — це популярна JavaScript бібліотека з відкритим кодом для здійснення HTTP-запитів із клієнтської або серверної частини застосунку. Вона побудована поверх стандартного API XMLHttpRequest (у браузері) або http/https модулів Node.js (на сервері) і пропонує спрощений та уніфікований інтерфейс для реалізації запитів до зовнішніх ресурсів.

Axios активно використовується в сучасних фронтенд-фреймворках, таких як React, Vue.js, Angular, а також у серверних застосунках, що працюють на Node.js. Однією з ключових причин його популярності є високий рівень абстракції, який дає змогу розробникам зосередитися на бізнес-логіці, зменшуючи потребу в обробці низькорівневих деталей HTTP-взаємодії.

Бібліотека підтримує проміси, автоматичну трансформацію відповідей, гнучке налаштування заголовків, інтерцептори запитів і відповідей, а також глобальне конфігурування, що робить її зручною для побудови масштабованих архітектур. Axios особливо ефективний у випадках, коли проєкт потребує обробки великої кількості однотипних запитів або інтеграції з REST API та зовнішніми сервісами.

Axios, як і Fetch, використовує механізм промісів, що дає змогу обробляти запити в асинхронному режимі за допомогою then/catch або async/await. Це дозволяє реалізовувати логіку взаємодії з сервером у більш декларативному та зрозумілому вигляді.

Axios автоматично перетворює відповідь сервера з JSON у JavaScript-об’єкти, що усуває необхідність у виклику .json() як у Fetch. При надсиланні запиту бібліотека також перетворює передані об'єкти в JSON-рядки без додаткових дій з боку розробника.

Axios має розширену модель обробки помилок: якщо запит завершився з кодом помилки HTTP (навіть 404 чи 500), проміс буде відхилено (rejected). Це дозволяє легко реалізовувати централізовану обробку виняткових ситуацій.

Однією з найсильніших функцій Axios є можливість додавати інтерцептори до запитів і відповідей. Інтерцептори дозволяють автоматично модифікувати запити (наприклад, додавати токени авторизації) або перехоплювати відповіді для обробки помилок чи логування.

Axios дозволяє встановити глобальні налаштування для всіх запитів: базову URL-адресу, тайм-аут, заголовки, політику обробки облікових даних тощо. Це особливо зручно при роботі з одним API, до якого надсилається багато запитів.

Axios підтримує повний спектр HTTP-методів: GET, POST, PUT, DELETE, PATCH, HEAD, OPTIONS. Для кожного методу існує відповідна функція, що робить API інтуїтивно зрозумілим.

Axios спрощує відправку FormData, включно з файлами, формами та іншими мультимедійними ресурсами. Крім того, бібліотека підтримує завантаження та вивантаження файлів з використанням blob або arraybuffer типів даних.

На відміну від Fetch, який не підтримується у Node.js без сторонніх полів, Axios є універсальним інструментом як для браузера, так і для серверного середовища. Це забезпечує єдину точку роботи з HTTP на обох рівнях програми [9].

Для прикладу розглянемо реалізацію POST запиту за допомогою Axios.

```
1   import axios from 'axios';
2
3   async function postData(url = '', data = {}) {
4     try {
5       const response = await axios.post(url, data, {
6         headers: {
7           'Content-Type': 'application/json',
8         },
9         withCredentials: false,
10      });
11      console.log(response.data);
12      return response.data;
13    } catch (error) {
14      console.error('Помилка при відправці запиту:', error);
15      throw error;
16    }
17  }
18
19  // Виклик функції:
20  postData('https://example.com/answer', { answer: 42 });
```

Рис. 1.2.3 — Програмний код виконаний мовою програмування JavaScript

Коротко розглянемо які налаштування використані в наведеному вище коді:

- axios.post(url, data, config) — відправляє POST-запит,
- headers: { 'Content-Type': 'application/json' } — вказує формат даних,

- withCredentials: false — не надсилає куки. Якщо сервер вимагає автентифікацію через сесію, змінюй на true,
- обробка помилок через try-catch.

XMLHttpRequest, fetch та axios — три поширені способи виконання HTTP-запитів у JavaScript, кожен із яких має свої переваги й обмеження. XMLHttpRequest з'явився першим і досі використовується, але його синтаксис вважається застарілим та громіздким для сучасної розробки.

Зі зростанням популярності асинхронного програмування було запроваджено fetch, який спростив написання запитів за рахунок промісів, зробивши код лаконічнішим. Водночас він потребує додаткової обробки для таких речей, як тайм-аути або перевірка статусу відповіді.

Axios — це бібліотека, яка поєднує зручність fetch із додатковими функціями, що часто потрібні на практиці: автоматична обробка JSON, вбудована підтримка тайм-аутів, зручні перехоплювачі запитів і відповідей, а також краща підтримка старих браузерів. Завдяки цьому він часто є вибором за замовчуванням у багатьох проєктах, де важлива швидкість розробки та гнучкість.

Хоча всі три інструменти вирішують одну задачу — надсилання HTTP-запитів — вибір між ними залежить від конкретного контексту: потреб проєкту, вимог до сумісності, розміру коду та особистих вподобань розробника.

У порівнянні між цими трьома підходами помітно, що XMLHttpRequest вимагає більше коду для досягнення базових результатів, тоді як fetch і axios дозволяють працювати з меншими обсягами логіки завдяки використанню промісів. Наприклад, обробка відповіді у fetch виглядає значно простіше, ніж у XMLHttpRequest, де потрібно вручну відстежувати зміни стану запиту.

У порівнянні fetch і axios мають багато спільного за зручністю, проте axios надає більше можливостей із коробки — таких як автоматична серіалізація/десеріалізація даних, тайм-аути, глобальні налаштування заголовків та перехоплювачі, які корисні у більших проєктах. Fetch, хоч і

легший та нативний, потребує додаткових обгорток або функцій для досягнення такого ж рівня функціональності.

Таким чином, якщо потрібна простота і сучасний нативний API — доцільно використовувати fetch. Якщо ж важлива широка функціональність, зручність обробки запитів та розширення — axios зазвичай є кращим вибором. XMLHttpRequest здебільшого залишився як спадщина для підтримки старого коду або специфічних сценаріїв.

1.3 Шкідливі міждоменні запити

Неправильна реалізація CORS є однією з найпоширеніших причин міждоменних атак, особливо в застосунках, які мають відкриті або напіввідкриті API. На відміну від CSRF, де атака відбувається через "довіру" браузера до cookie, у випадку CORS зловживань мова йде про довіру сервера до клієнта, яка виникає внаслідок неправильно налаштованих HTTP-заголовків доступу.

Коли сервер дозволяє всім джерелам надсилати запити, не перевіряючи його, будь-який сторонній сайт може виконувати запити до API, отримувати повноцінні відповіді включаючи конфіденційні дані, і навіть взаємодіяти з авторизованими сесіями користувача [10].

Такі вразливості можуть виникати з декількох причин.

Ведення білого списку авторизованих доменів може стати виснажливим завданням, особливо коли програма має взаємодіяти з великою кількістю доменів. Щоб спростити цей процес, деякі програми обирають більш дозвільний підхід, дозволяючи доступ до ресурсів з будь-якого домену. Вони досягають цього, зчитуючи значення заголовка «Origin» вхідного запиту і відображаючи його безпосередньо у відповіді сервера.

Коли користувач платформи відвідує сайт, на якому розміщено шкідливий код, вона робить запит і повертає відповідь на сервері зловмисника [11].

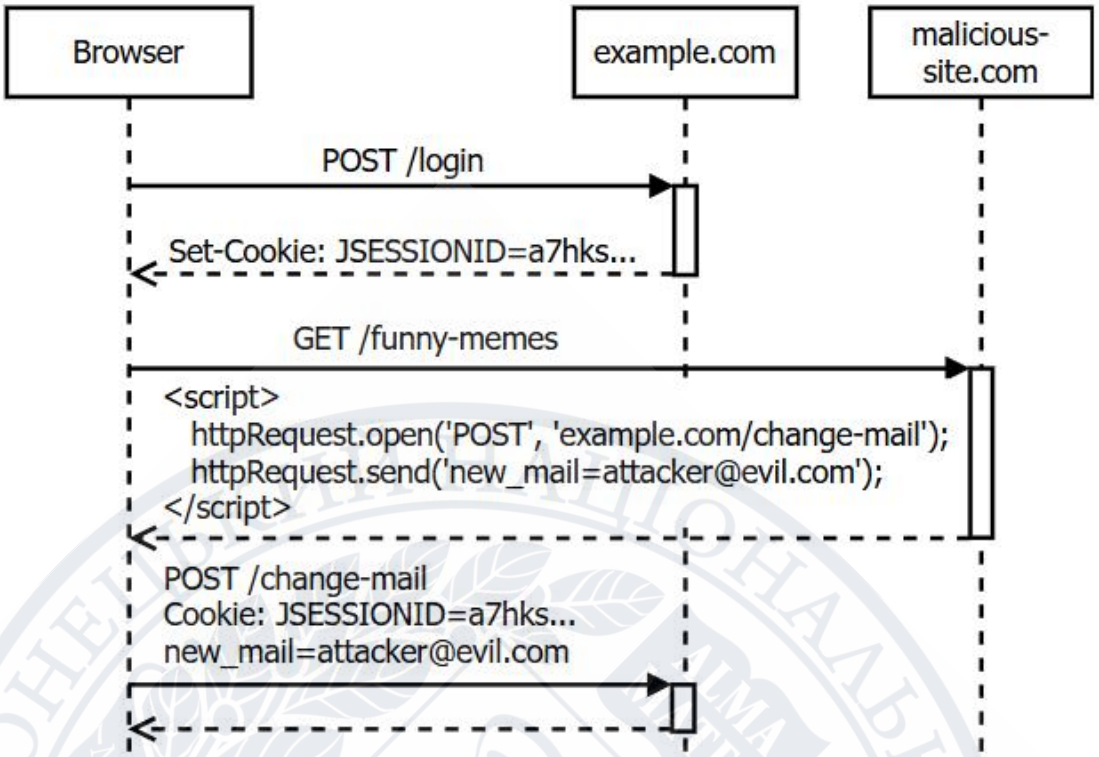

Рис. 1.3.1 — Приклад реалізації атаки [12]

Авторизація всіх джерел шляхом відображення вмісту заголовка є неправильною конфігурацією, яка ставить під загрозу безпеку програми. Тому розробники можуть вирішити додати перевірку на основі регулярного виразу. Це рішення може працювати, якщо ви добре володієте регулярними виразами. Однак іноді це не так, тому ви можете опинитися в ситуації, коли перевірку можна обійти [13-14].

При локальній розробці веб-додатків нерідко до білого списку авторизованих джерел додають «нульове» походження. Включення «нульового» джерела до білого списку може бути поширеною практикою для полегшення розробки та налагодження додатків у локальному середовищі. Однак, коли додаток запускається у виробництво, виникає проблема: іноді розробники можуть забути видалити «null» з білого списку.

Іноді правила CORS можна налаштувати так, щоб дозволити доступ до ресурсів з будь-якого субдомену веб-додатку.Такий підхід сам по собі не є шкідливим, але дозволяє хакеру дістатися до менш захищених субдоменів [15].

Таблиця 1.3.1. Шкідливі міждоменні запити

Неправильна конфігурація	Вразливості
Бекдор для розробників	Незахищені джерела розробника/налагодження, такі як JSFiddler CodePen, мають доступ до ресурсу.
Відображення походження	Походження просто відображається в заголовку, будь-якому сайту дозволено доступ до ресурсу.
Нульова неправильна конфігурація	Будь-якому сайту дозволено доступ, примусово використовуючи нульове походження через ізольований iframe.
Підстановка перед доменом	notdomain.com має доступ, який зловмисник може просто зареєструвати
Підстановочний знак після домену	Домен domain.com.evil.com дозволено доступ, його може просто налаштувати зловмисник
Дозволені субдомени	sub.domain.com дозволений доступ, можливе використання, якщо зловмисник знаходить XSS у будь-якому піддомені
Дозволено сайти без SSL	Походження HTTP має доступ до ресурсу HTTPS, що дозволяє MitM зламати шифрування.

Висновки до розділу 1

Після дослідження останніх тенденцій у веб-розробці. А саме створення сучасних підходів до архітектури сайтів та веб-додатків. Було встановлено, що міждоменна взаємодія є базовим механізмом, який підтримує зв'язок між веб-компонентами, що працюють у різних середовищах. У ході аналізу стало зрозумілим, що міждоменні запити стали основою для реалізації таких рішень, як клієнт-серверна модель, інтеграція сторонніх API, мікросервісна архітектура тощо. Цей механізм є незамінним. І він несе в собі великий вплив на розвиток веб-розробки та роботі в браузері.

Було розглянуто програмні інструменти для реалізації міждоменних запитів на прикладі мови програмування JavaScript. Проаналізовано особливості двох вбудованих методів: XMLHttpRequest, Fetch. Та одну з найпопулярніших бібліотек — Axios.

Аналізу наукових досліджень на тему створення та реалізації міждоменних запитів дозволив нам виділили їх вразливості, які дають змогу реалізувати кібератаку.

РОЗДІЛ 2 ОГЛЯД МЕТОДІВ І СПОСОБІВ ЗАХИСТУ ВІД ШКІДЛИВИХ МІЖДОМЕННИХ ЗАПИТІВ

2.1 Політика єдиного походження

Політика єдиного походження (Same-Origin Policy, SOP) — це базовий механізм безпеки, реалізований у сучасних браузерах, який обмежує взаємодію між ресурсами, завантаженими з різних джерел. Його основна мета полягає в тому, щоб дозволити майже необмежене виконання скриптів та взаємодію між сторінками, що належать до одного сайту (визначеного як однакове доменне ім’я, протокол і порт), водночас майже повністю забороняючи взаємодію з несуміжними сайтами [16].

Термін "походження" визначається сукупністю доменного імені, протоколу прикладного рівня та (у більшості сучасних браузерів) номера порту HTML документа, з якого виконується скрипт. Два ресурси вважаються такими, що мають спільне походження, лише в тому випадку, якщо всі три параметри збігаються. Наприклад, у таблиці нижче наведено типові результати перевірки походження відносно URL «http://www.example.com/dir/page.html».

Таблиця 2.1.1. Результати перевірки одного походження

URL для порівняння	Результат	Причина
http://www.example.com/dir/page.html	успішно	Однаковий протокол і хост
http://www.example.com/dir2/other.html	успішно	Однаковий протокол і хост
http://www.example.com:81/dir/other.html	невдало	Різні порти
https://www.example.com/dir/other.html	невдало	Різні протоколи
http://en.example.com/dir/other.html	невдало	Різні хости
http://example.com/dir/other.html	невдало	Різні хости (потрібен повний збіг)
http://v2.www.example.com/dir/other.html	невдало	Різні хости (потрібен повний збіг)

На практиці не існує єдиної політики єдиного походження — замість цього діє набір механізмів, які мають схожу загальну логіку, але суттєво відрізняються у деталях реалізації залежно від типу ресурсу або контексту. Ці варіації політики розглядаються окремо в технічній документації та специфікаціях.

У цій роботі розглянуто сценарій політики одного походження для XMLHttpRequest.

Попри широкі функціональні можливості XMLHttpRequest, цей інструмент також потребує належного контролю безпеки, оскільки він дає змогу скриптам взаємодіяти із серверною частиною застосунку на досить глибокому рівні. Усі HTTP-запити, що надсилаються через XMLHttpRequest, автоматично включають cookie, які зберігає браузер, що може становити загрозу у випадку некоректно налаштованої політики доступу до ресурсів.

Для обмеження потенційних ризиків сучасні браузери реалізують спеціалізовану версію політики єдиного походження, яка регулює запити, що

виконуються через XMLHttpRequest. Вона має кілька відмінностей від стандартного механізму DOM доступу:

- Ігнорування параметра document.domain: у разі взаємодії скриптів через DOM можливе налаштування document.domain, що дозволяє певну взаємодію між піддоменами. У випадку XMLHttpRequest цей параметр ігнорується, що унеможливлює навмисне послаблення обмежень на міждоменну комунікацію.
- Обмеження на протоколи, заголовки та методи: деякі реалізації браузерів вводять додаткові обмеження щодо використання певних HTTP-методів (наприклад, PUT або DELETE), специфічних заголовків або навіть відповідей з нетиповими кодами статусу. Це ще більше звужує можливості сторонніх ресурсів у спробах зловживання механізмом запитів.
- Урахування номера порту: у браузері Microsoft Internet Explorer політика походження для XMLHttpRequest враховує також порт, на відміну від DOM перевірок. Це посилює контроль за міжпортовими запитами, проте не є універсальним правилом для всіх браузерів [17].

З огляду на це, належне використання XMLHttpRequest передбачає не лише обмеження доступу до ресурсів, а й точне налаштування міждоменних політик, а також контроль над тим, які саме запити дозволено виконувати з клієнтського боку.

2.2 Налаштування параметрів доступу до спільних ресурсів

CORS (Cross-Origin Resource Sharing) — це механізм безпеки, який дозволяє веб-серверу визначати, які домени мають право надсилати до нього міждоменні HTTP-запити. Його використання є важливою частиною реалізації сучасних веб-застосунків, де фронтенд і бекенд можуть бути розміщені на різних доменах або портах. CORS ґрунтується на специфікації W3C і підтримується усіма сучасними браузерами.

За замовчуванням браузери блокують запити з одного походження (origin) до іншого, якщо ці запити не є "простими" (наприклад, GET запити без нестандартних заголовків). Ця політика реалізується відповідно до так званого принципу єдиного походження (Same-Origin Policy), який забороняє JavaScript кодові на одній сторінці взаємодіяти з ресурсами з іншого походження. Механізм CORS дозволяє серверу обійти цю заборону вибірково, явно вказуючи, які зовнішні джерела є дозволеними [18].

Для налаштування CORS сервер відповідає на запит спеціальними HTTP-заголовками. Основні з них:

- Access-Control-Allow-Origin — вказує, які домени дозволено.
- Access-Control-Allow-Methods — визначає HTTP-методи, які дозволено використовувати з кросдоменною метою (наприклад, GET, POST, PUT).
- Access-Control-Allow-Headers — зазначає нестандартні заголовки, які дозволено використовувати в запиті.
- Access-Control-Allow-Credentials — якщо встановлено в true, дозволяє надсилання облікових даних (cookies, HTTP-автентифікація).
- Access-Control-Expose-Headers — дозволяє клієнтові читати певні заголовки з відповіді сервера, які зазвичай заблоковані за замовчуванням.
- Access-Control-Max-Age — визначає, як довго результати попереднього запиту можна кешувати браузером [19].

Для "непростих" запитів (наприклад, POST із нестандартним заголовком або з Content-Type: application/json) браузер спочатку надсилає так званий попередній запит методом OPTIONS. Цей запит не містить основного тіла, а лише перевіряє, чи дозволено сервером виконати справжній запит. Лише у разі позитивної відповіді з відповідними CORS заголовками браузер продовжує з основним запитом.

У технології Node.js, яка є написана мовою програмування JavaScript, з використанням фреймворку Express реалізація підтримки CORS є простою завдяки офіційному middleware пакету cors, який дозволяє гнучко налаштовувати міждоменний доступ. Для активації CORS достатньо

встановити цей пакет через npm (npm install cors) та підключити його до застосунка за допомогою app.use(require('cors')()). За потреби можна передати додаткові параметри конфігурації, наприклад, список дозволених доменів, HTTP-методів або заголовків. Такий підхід дозволяє централізовано керувати правилами CORS і запобігати небезпечним міждоменним запитам без потреби вручну задавати відповідні заголовки у кожному маршруті.

Якщо не використовувати фреймворк, CORS можна налаштувати вручну, додаючи відповідні HTTP-заголовки безпосередньо у відповіді на запит. Такий підхід вимагає більше ручної роботи, але дає повний контроль над логікою відповіді [20].

CORS є критично важливою технологією у розробці сучасних багатокомпонентних веб-систем. Його правильне налаштування дозволяє забезпечити безпечну взаємодію між клієнтською та серверною частинами, розташованими на різних доменах, не порушуючи політики безпеки браузера. Водночас навіть незначні помилки у конфігурації можуть відкрити шлях до атак, тому налаштування CORS має бути усвідомленим і точним.

2.3 Налаштування параметрів cookies

У межах міждоменної взаємодії особливу увагу слід приділяти тому, як браузери обробляють cookie при виконанні запитів. Навіть при коректно налаштованому CORS сервер може бути вразливим до атак, якщо не контролює поведінку cookie. Саме тому в сучасній веб-безпеці важливу роль відіграє атрибут SameSite, який є частиною специфікації cookie та визначає, чи має браузер надсилати cookie разом із міждоменним запитом [21].

Атрибут SameSite може приймати три значення:

- Strict — найсуворіший режим. Cookie надсилаються лише для запитів з того самого походження. Це повністю блокує міждоменні запити з

використанням цих cookie, навіть якщо користувач переходить за посиланням.

- Lax — дозволяє cookie в обмежених міждоменних ситуаціях, зокрема при переході за посиланням або при виконанні GET-запитів. Це баланс між зручністю та безпекою, оскільки блокує більшість типів CSRF-атак.
- None — дозволяє повну міждоменну взаємодію, але лише за умови, що cookie передаються через HTTPS-з'єднання. У цьому випадку сервер повинен бути додатково захищений налаштуванням Access-Control-Allow-Credentials: true у CORS [22].

Недостатнє розуміння цього механізму часто призводить до ситуації, коли сервер дозволяє міждоменні запити (Access-Control-Allow-Origin: *) та одночасно передає авторизаційні cookie без SameSite, що відкриває шлях до CSRF атак. Зокрема, шкідливий сайт може змусити браузер користувача виконати запит до вразливого ресурсу (наприклад, змінити пароль), використовуючи cookie, збережене під час справжнього входу.

У цьому прикладі сервер встановлює сесійну cookie після авторизації, яка не буде надсилатися під час переходу з іншого домену — отже, такий підхід запобігає CSRF.

```
1  app.get('/login', (req, res) => {
2    res.cookie('sessionId', 'abc123', {
3      httpOnly: true,
4      secure: true,
5      sameSite: 'strict',
6      path: '/'
7    });
8    res.send('Сесія встановлена');
9  });
```

Рис. 2.3.1 — Програмний код виконаний мовою програмування JavaScript

Використання атрибута SameSite є обов'язковою практикою в сучасних браузерах: з 2020 року більшість з них автоматично застосовують SameSite=Lax, якщо атрибут не вказаний явно. Це означає, що веб-розробникам необхідно

свідомо налаштовувати поведінку cookie відповідно до характеру взаємодії між компонентами системи [23].

Таким чином, атрибут SameSite є важливим інструментом контролю безпеки на стороні клієнта, що доповнює механізм CORS. Його правильне використання дозволяє запобігти критичним типам атак, зокрема CSRF, та значно підвищує стійкість веб-застосунку до шкідливої міждоменної активності.

Висновки до розділу 2

Аналіз сучасних методів захисту від шкідливих міждоменних атак дозволяє зробити висновок, що жоден з них не є універсальним, однак їхнє поєднання може забезпечити високий рівень безпеки.

Політика єдиного походження залишається основним захисним механізмом, реалізованим у всіх сучасних браузерах, однак вона істотно обмежує функціональність взаємодії між різними частинами веб-застосунку, особливо якщо фронтенд і бекенд розміщені на окремих доменах.

Для подолання цих обмежень використовується механізм CORS, який дозволяє гнучко налаштовувати правила міждоменної взаємодії. Проте варто пам’ятати, що CORS сам по собі не забезпечує захист від атак, пов’язаних із крадіжкою облікових даних, зокрема cookie, особливо коли конфігурація CORS є надто лояльною.

Саме тому доцільно доповнювати CORS політики додатковими налаштуваннями cookie на стороні сервера, зокрема обмеженням їхньої видимості (HttpOnly), дозволом передачі лише через захищене з’єднання (Secure) та використанням атрибута SameSite, що обмежує їх участь у міждоменних запитах.

РОЗДІЛ 3 ТЕСТУВАННЯ НА ВРАЗЛИВОСТІ ДО ШКІДЛИВИХ МІЖДОМЕННИХ ЗАПИТІВ

Перевірка конфігурації CORS є важливим етапом аудиту безпеки веб-застосунків, особливо якщо застосунок має публічне API або обробляє конфіденційні дані. Основна мета тестування полягає у виявленні помилкових налаштувань, які дозволяють виконання небезпечних міждоменних запитів, а також зчитування відповідей з іншого походження.

Існує два підходи до перевірки CORS: ручний та автоматизований.

Популярні інструменти, як-от Burp Suite або OWASP ZAP, мають вбудовані модулі аналізу CORS, які автоматично ідентифікують неправильні заголовки та повідомляють про ризики. Автоматизоване тестування має ряд ключових переваг. Розробник програмного коду може не володіти глибокими знаннями шкідливих сценаріїв, так як сканер уже містить типові шаблони. Це дозволяє швидко навчитися та використовувати технології тестування. Такі додатки мають високу швидкість та вбудовані функції для перевірки масиву джерел.

Ручне тестування дозволяє досліднику або аудитору безпосередньо змінювати параметри запитів і спостерігати за поведінкою сервера. Найчастіше для цього використовують інструменти на зразок curl, Postman або власні JavaScript-скрипти, що емулюють міждоменні запити. Такий підхід є більш гнучким та дозволяє за допомогою нетипових випадків більш глибокий аналіз, якщо веб-продукт того потребує. Але такий підхід вимагає вищого рівня кваліфікації спеціаліста [24-26].

Тож ми зупинимося на варіанті: написання власного JavaScript скрипту для тестування CORS заголовків.

Це буде консольний застосунок на Node.js, який працюватиме за таким принцепом: отримає URL, Origin та метод із параметрів командного рядка, зробить HTTP-запит, зчитає CORS заголовки й виведе результати в консоль.

Основні етапи ручної перевірки включають:

1. Визначення цільового ресурсу, що використовує міждоменну взаємодію. Насамперед досліджується, чи обробляє сервер міждоменні запити, і якщо так — які ресурси є потенційно вразливими.
2. Надсилання запиту з підробленим заголовком Origin. Змінюється значення заголовка Origin на підозрілий або довільний домен (наприклад, attacker.com). Якщо сервер у відповідь повертає Access-Control-Allow-Origin із віддзеркаленням значення Origin, це може свідчити про вразливість.
3. Аналіз заголовка Access-Control-Allow-Credentials. Якщо вказано true, це означає, що сервер дозволяє надсилання облікових даних (cookie, HTTP-авторизація) з іншого походження — що особливо небезпечно у поєднанні з динамічним Origin.
4. Перевірка поведінки браузера. Створюється HTML-сторінка або скрипт, що надсилає fetch() або XMLHttpRequest до вразливого ресурсу. Якщо відповідь сервера доступна через JavaScript, а запит супроводжується cookie — це підтвердження вразливості.

Проведемо перше тестування сайту URL “https://www.donnu.edu.ua/uk/” і роаналізуємо отриманий результат.

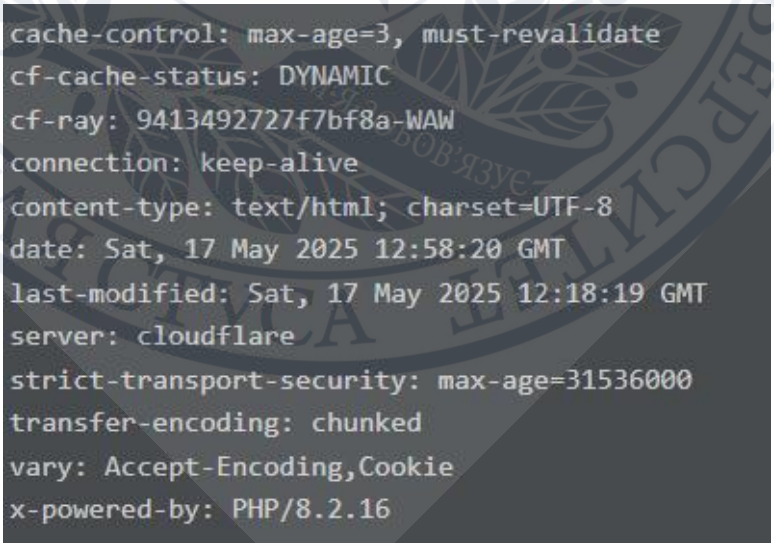

Рис. 3.1 — Результат сканування сайту URL “https://www.donnu.edu.ua/uk/”

1. cache-control: max-age=3, must-revalidate

max-age=3 — кеш може зберігати відповідь максимум 3 секунди.

must-revalidate — після закінчення цього часу клієнт (наприклад, браузер) повинен перевірити у сервера, чи ресурс оновився, перш ніж повторно його використовувати.

2. cf-cache-status: DYNAMIC

Це специфічний заголовок Cloudflare.

DYNAMIC означає, що контент не кешувався Cloudflare, оскільки він динамічно генерується і, ймовірно, змінюється від запиту до запиту.

3. cf-ray: 9413492727f7bf8a-WAW

Унікальний ідентифікатор запиту до Cloudflare.

Частина WAW вказує на дата-центр, через який оброблявся запит (у цьому випадку — Варшава).

4. connection: keep-alive

З'єднання TCP не закривається після відповіді, дозволяючи повторно використовувати його для подальших запитів, що зменшує накладні витрати.

5. content-type: text/html; charset=UTF-8

Вказує, що тіло відповіді — це HTML документ, закодований у форматі UTF-8.

6. date: Sat, 17 May 2025 12:58:20 GMT

Час створення HTTP-відповіді на сервері.

7. last-modified: Sat, 17 May 2025 12:18:19 GMT

Вказує, коли вміст ресурсу востаннє був змінений.

Браузери використовують це значення для умовного кешування (If-Modified-Since).

8. server: cloudflare

Вказує, що сервер, який обробляє запит — це Cloudflare (зазвичай як проксі).

9. strict-transport-security: max-age=31536000

Вказує браузеру використовувати тільки HTTPS протягом 31536000 секунд (тобто 1 рік).

Захист від атак типу downgrade (перехід з HTTPS на HTTP).

10. transfer-encoding: chunked

Сервер надсилає відповідь частинами (чанками), корисно, коли невідомий розмір відповіді наперед.

11. vary: Accept-Encoding,Cookie

Вказує кешу, що варіанти відповіді можуть змінюватися залежно від:

Accept-Encoding — наприклад, gzip або br.

Cookie — тобто відповідь може бути персоналізована (залежить від сесії користувача).

Як ми можемо побачити сайт не містить налаштувань CORS. Тобто ніякий сторонній сервіс не може отримати відповідь із корисним навантаженням від цього сайту. Така конфігураціє є типовою бля більшості веб-сторінок. Звичайно, багато сайтів, таких як Google, активують заголовки CORS лише для певних ресурсів, а не безпосередньо на цільовій сторінці.

Для наступного тестування оберемо API які є у вільному доступі у Інтернеті. Наприклад API курсу валют URL “https://api.minfin.com.ua/mb/”. І проведемо аналіз отриманих результатів.

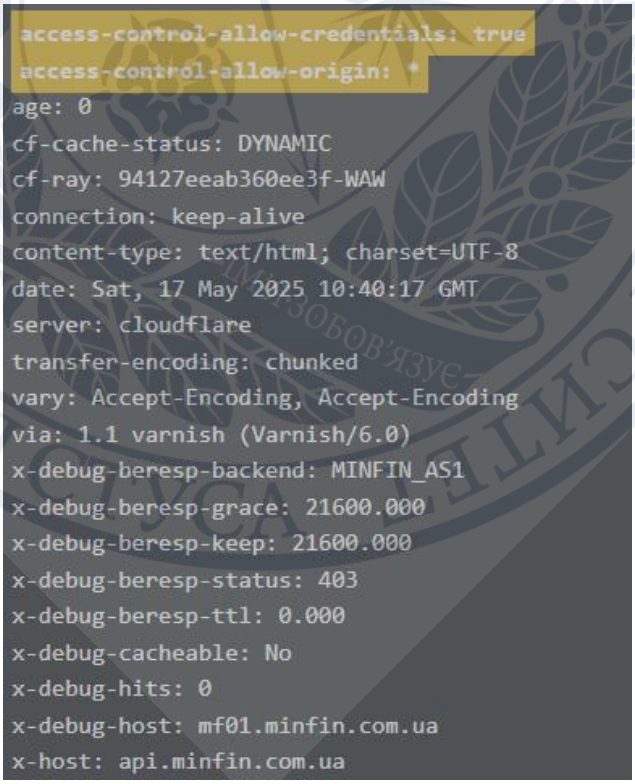

Рис. 3.1 — Результат сканування API курсу валют URL “https://api.minfin.com.ua/mb/”

1. access-control-allow-credentials: true

Дозволяє надсилання cookie, авторизаційних токенів або HTTP-аутентифікаційної інформації в міждоменних запитах.

2. access-control-allow-origin: *

Дозволяє будь-якому походженню (origin) доступ до ресурсу.

Критична вразливість: за стандартом, access-control-allow-credentials: true не може використовуватись разом з access-control-allow-origin: *. Це може призвести до міждоменної атаки з крадіжкою даних (наприклад, cookies користувача), якщо браузер не блокує такі запити.

Отримуємо важливі висновки: контекст має значення.

Така конфігурація цілком підходить для загальнодоступних сайтів або кінцевих точок API, призначених для доступу для всіх. Натомість це може бути катастрофічним для платіжних сайтів або платформ соціальних мереж.

ВИСНОВКИ

У бакалаврській роботі було досліджено механізми міждоменної взаємодії у веб-застосунках, зосереджуючись на проблематиці шкідливих міждоменних запитів. Особливу увагу приділено аналізу типових помилкових конфігурацій, які можуть створити умови для атак через неправильно налаштовану CORS-політику, відкриті API або незахищені cookie.

Огляд сучасних методів і засобів захисту продемонстрував, що не існує універсального рішення, яке б повністю виключало ризики міждоменної уразливості. Найвищий рівень захисту забезпечується при використанні комплексного підходу.

Аналіз наявних інструментів для виявлення вразливостей, пов’язаних із міждоменними запитами, виявив, що результати автоматизованого сканування часто є неточними або неповними. Це зумовлено складністю контексту, у якому функціонує міждоменна взаємодія, і залежністю від стану автентифікації, ролей користувачів, специфіки API тощо. Таким чином, ефективне тестування вимагає гнучкого підходу, розуміння внутрішньої логіки веб-застосунку та поєднання автоматизованих і ручних методів перевірки.

Загалом, дослідження підтвердило актуальність теми та важливість формування усвідомленої стратегії захисту від шкідливих міждоменних запитів у сучасних веб-системах.

СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ

1. Neumann A., Laranjeiro N., Bernardino J. An Analysis of Public REST Web Service APIs. *IEEE Transactions on Services Computing*. 2018. P. 1.

2. Ahmad I., Suwarni E., Borman R. I. Implementation of RESTful API Web Services Architecture in Takeaway Application Development. *IEEE Transactions on Services Computing*. 2021. P. 490–503.

3. Ranga V., Soni A. API Features Individualizing of Web Services: REST and SOAP. *International Journal of Innovative Technology and Exploring Engineering*. 2019. Vol. 8, no. 9S. P. 664–671.

4. Neumann A., Laranjeiro N., & Bernardino, J. A Framework for the Structural Analysis of REST APIs. *Proceedings of the 2017 IEEE International Conference on Software Architecture (ICSA)*. 2017. 1–10 с.

5. Meshram Sh. An Overview of RESTful Web API Development and Design. International Journal of Research in Engineering. *Science and Management*. 2025. 4(7).

6. Що таке API? Просте пояснення від Петра Газарова. *Dev.ua.* URL: https://dev.ua/news/chto-takoe-api-prostym-yazykom (дата звернення: 23.04.2025).

7. XMLHttpRequest. *XMLHttpRequest Standard*. URL: https://xhr.spec.whatwg.org/ (дата звернення: 23.04.2025).

8. Using the Fetch API with Undici in Node.js. *Node.js*. URL: https://nodejs.org/en/learn/getting-started/fetch (дата звернення: 26.04.2025).

9. Getting Started. *Axios Documentation*. URL: https://axios-http.com/docs/intro (дата звернення: 28.04.2025).

10. Testing Cross Origin Resource Sharing. *OWASP*. URL: https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/11-Client-side_Testing/07-Testing_Cross_Origin_Resource_Sharing (дата звернення: 29.04.2025).

11. Golinelli, M., Arshad, E., Kashchuk, D., & Crispo, B. Mind the CORS. *Proceedings of the 2023 5th IEEE International Conference on Trust, Privacy and Security in Applications, Platforms and Systems (TPS-ISA)*. 2023.

12. Shayakhmetova, A., Litvinenko, N., Mamyrbayev, O., Wójcik, W., & Zhamangarin, D. Design of Wearable EEG Device for Seizures Early Detection. *International Journal of Electronics and Telecommunications*. 2021. 67(2), 187–192 с.

13. Shaji, E., & Subramanian, N. Assessing Non-Intrusive Vulnerability Scanning Methodologies for Detecting Web Application Vulnerabilities on Large Scale. *International Conference on System, Computation, Automation and Networking (ICSCAN)*. 2021. С. 1–5.

14. Nardone, M. Key Security Concerns and Risks for RESTful APIs. *Secure RESTful APIs*. Berkeley, CA: Apress, 2025. С. 15–22.

15. Idris, M., Syarif, I., & Winarno, I. Web Application Security Education Platform Based on OWASP API Security Project EMITTER. *International Journal of Engineering Technology*. 2022. 10(2), С. 246–261.

16. Browser Security Handbook, part 2. *Google Code Archive*. URL: https://code.google.com/archive/p/browsersec/wikis/Part2.wiki#Same-origin_policy (дата звернення: 03.05.2025).

17. van Oorschot, P. C. Web and Browser Security. *Computer Security and the Internet*. Cham: Springer, 2021. С. 245–279.

18. Cross-Origin Resource Sharing (CORS). *Mdn Web Docs*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS (дата звернення: 03.05.2025).

19. Hossain, M. CORS in Action: Creating and consuming cross-origin APIs : навчальний посібник. Simon and Schuster, 2014. 240 с. "Understanding CORS in depth" С. 62–105.

20. Wang, C.-H., & Zhou, Y.-S. A New Cross-Site Scripting Detection Mechanism Integrated with HTML5 and CORS Properties by Using Browser Extensions. *Proceedings of the 2016 International Computer Symposium (ICS)*. 2016. С. 264–269.

21. Wang, Y., Liu, Z., & Xu, W. Understanding and Securing the Web's Most Dangerous Attack: A Comprehensive Study of Cross-Site Request Forgery.

Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP). 2022. С. 1234–1248.

22. SameSite Cookie Attribute: Preventing CSRF SameSite Exploits. *Invicti*. URL: https://www.invicti.com/blog/web-security/same-site-cookie-attribute-prevent-cross-site-request-forgery/ (дата звернення: 04.05.2025).

23. Compagna, L., Jonker, H. L., Krochewski, J., Krumnow, B., & Sahin, M. A preliminary study on the adoption and effectiveness of SameSite cookies as a CSRF defence. *Proceedings of the 2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. Vienna, Austria: IEEE, 2021. С. 49–59.

24. CORS Tester. Codehappy. URL: https://cors-test.codehappy.dev/ (дата звернення: 09.05.2025).

25. Cors-test. *GitHub*. URL: https://github.com/mscoutermarsh/cors-test (дата звернення: 09.05.2025).

26. On Web-Security and -Insecurity. *Blogger*. URL: https://web-in-security.blogspot.com/2017/07/cors-misconfigurations-on-large-scale.html (дата звернення: 08.05.2025).

ДОДАТОК А

Код консольного сканера

```
const fetch = require('node-fetch');
const { URL } = require('url');
// Перевірка, чи URL є дійсним HTTP(S)
function isValidHttpUrl(string) {
  try {
    const url = new URL(string);
    return url.protocol === "http:" || url.protocol === "https:";
  } catch (_) {
    return false;
  }
}
// Екранування HTML символів (на випадок повторного використання)
function encodeHTML(s) {
  return s.replace(/&/g, '&amp;').replace(/</g, '&lt;').replace(/"/g, '&quot;');
}
// Отримання заголовків
async function getHeaders(url, method, origin) {
  try {
    const response = await fetch(url, {
      method: method.toUpperCase(),
      headers: { Origin: origin }
    });
    return response.headers;
  } catch (err) {
    console.error("Помилка запиту:", err.message);
    return null;
```

```
  }
}
// Виведення результатів CORS
function renderCors(headers) {
  const allowOrigin = headers.get("access-control-allow-origin");
  if (allowOrigin === "*") {
    console.log("\x1b[32m✔ CORS дозволено для будь-якого Origin (*).\x1b[0m");
  } else if (allowOrigin !== null) {
    console.log("\x1b[33m⚠ Доступ дозволено лише для: " + allowOrigin + "\x1b[0m");
  } else {
    console.log("\x1b[31m✘ CORS заголовки не знайдені (access-control-allow-origin).\x1b[0m");
  }
}
// Виведення всіх заголовків
function renderHeaders(headers) {
  console.log("\n== Отримані заголовки ==\n");
  for (const [key, value] of headers.entries()) {
    const highlight = /(access-control-allow-methods|access-control-allow-headers|access-control-allow-origin|access-control-max-age|access-control-allow-credentials)/i.test(key);
    if (highlight) {
      console.log(`\x1b[43m${key}: ${value}\x1b[0m`);
    } else {
      console.log(`${key}: ${value}`);
    }
  }
}
// Основна функція
```

```
async function main() {
  const args = process.argv.slice(2);
  const url = args[0];
  const origin = args[1] || 'https://cors-test.codehappy.dev/';
  const method = args[2] || 'get';
  if (!isValidHttpUrl(url)) {
    console.error("❌ Недійсний URL.");
    process.exit(1);
  }
  if (!isValidHttpUrl(origin)) {
    console.error("❌ Недійсний Origin.");
    process.exit(1);
  }
  const validMethods = ["get", "post", "put", "patch", "head", "options"];
  if (!validMethods.includes(method.toLowerCase())) {
    console.error("❌ Недійсний HTTP метод.");
    process.exit(1);
  }
  console.log(`Перевірка CORS для: ${url}`);
  console.log(`→ Origin: ${origin}`);
  console.log(`→ Метод: ${method.toUpperCase()}`);
  const headers = await getHeaders(url, method, origin);
  if (!headers) {
    console.error("❌ Не вдалося отримати заголовки.");
    process.exit(1);
  }
  renderCors(headers);
  renderHeaders(headers);
}
main();
```

Запуск коду

node cors-check.js <url> [origin] [method]