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

РУЩАК Владислав Михайлович

Методика тестування та захисту веб-додатків на TypeScript / Methodology for Testing and Securing Web Applications Using TypeScript

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

Кваліфікаційна робота

Виконав студент групи КБм -21
В. М. Рущак

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

Кваліфікаційну роботу допущено
до захисту:

«____» _____ ___2025 р.

Завідувач кафедри
_____ В.В.Яцків

ТЕРНОПІЛЬ - 2025

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

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

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
РУЩАКУ ВЛАДИСЛАВУ МИХАЙЛОВИЧУ
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи:
Методика тестування та захисту веб-додатків на TypeScript / Methodology for Testing and Securing Web Applications Using TypeScript
керівник роботи д.т.н., доцент С.В. Івасьєв
затверджені наказом по університету від 20 грудня 2024 року № 938
2. Строк подання студентом закінченої випускної кваліфікаційної роботи 5 грудня 2025 року.
3. Вихідні дані до кваліфікаційної роботи: завдання на випускну кваліфікаційну роботу студента, наукові статті, технічна література.
4. Основні питання, які потрібно розробити:
– дослідити сучасні підходи до тестування веб-додатків та визначити їхні ключові особливості;
– дослідити специфічні вразливості веб-додатків, що виникають у середовищі TypeScript;
– проаналізувати механізм виникнення та експлуатації вразливості Prototype Pollution;
– дослідити вразливість у бібліотеці dot-diver та визначити причини її появи;
– розробити алгоритм тестування TypeScript-застосунків з урахуванням вимог безпеки;
– проаналізувати залежності та ризики програмного ланцюга постачання;
– дослідити можливості вбудованих інструментів безпеки екосистеми Node.js та оцінити їх ефективність.
5. Перелік графічного матеріалу у роботі:
– техніки тестування застосунків на TypeScript,
– класифікація типів тестування unit, integration, e2e, performance, security testing,
– аналіз TypeScript / Node вразливостей з OWASP Top 10,
– архітектура засобів безпеки TypeScript додатку,
– механізм роботи setByPath,
– схема алгоритму використання вразливості setByPath,
– схема алгоритму тестування TypeScript-застосунку.

6. Консультанти розділів кваліфікаційної роботи

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

7. Дата видачі завдання 20 грудня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строки виконання етапів кваліфікаційної роботи	Примітка
1	Теоретичні основи тестування та захисту веб-додатків	12.2024 р. – 03.2025 р.	
2	Вразливості забруднення прототипу	03.2025 р. – 06.2025 р.	
3	Аналіз залежностей Typescript-застосунків	06.2025 р. – 11.2025 р.	

Студент _____ Владислав РУЩАК
(підпис)

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

АНОТАЦІЯ

Рущак В. М. Методика тестування та захисту веб-додатків на TypeScript – Рукопис.

Дослідження на здобуття освітнього ступеня «магістр» за спеціальністю 125 «Кібербезпека та захист інформації», освітньо-професійна програма «Кібербезпека». – Західноукраїнський національний університет, Тернопіль, 2025.

У ході дослідження встановлено, що сучасні підходи до тестування веб-додатків ґрунтуються на поєднанні автоматизованих інструментів, методів статичного та динамічного аналізу, а також інтеграції тестування у весь життєвий цикл розробки, що дозволяє своєчасно виявляти помилки та зменшувати ризики експлуатації вразливостей. Встановлено, що TypeScript підвищує надійність коду завдяки статичній типізації, проте сам по собі не усуває загрози, характерні для JavaScript-екосистеми, зокрема проблеми з прототипним наслідуванням, залежностями та unsafe-конструкціями. Проведений аналіз підтвердив, що причина вразливості в dot-diver полягає у неконтрольованій обробці ключів шляху у функції setByPath, що дозволяло атакувальнику вносити зміни у глобальний об’єкт Object.prototype, використовуючи шкідливі поля на зразок __proto__.

Запропонована комплексна методика тестування та захисту веб-додатків, розроблених на основі технології TypeScript, яка поєднує сучасні підходи до статичного та динамічного аналізу, оцінювання залежностей і дослідження специфічних для JavaScript/TypeScript уразливостей.

Ключові слова: TypeScript, веб-додатки, інформаційна безпека, тестування безпеки, Prototype Pollution, вразливості, залежності, supply-chain, статичний аналіз коду, динамічний аналіз, SCA, npm audit, dot-diver.

ABSTRACT

Ruschak V. M. Methodology for Testing and Securing Web Applications Using TypeScript – Manuscript.

Research for the degree of “Master” in specialty 125 “Cybersecurity and information protection”, educational and professional program “Cybersecurity”. – Western Ukrainian National University, Ternopil, 2025.

The study found that modern approaches to testing web applications are based on a combination of automated tools, static and dynamic analysis methods, as well as the integration of testing into the entire development life cycle, which allows for timely detection of errors and reducing the risks of exploiting vulnerabilities. It was found that TypeScript increases the reliability of the code due to static typing, but by itself it does not eliminate the threats characteristic of the JavaScript ecosystem, in particular problems with prototypical inheritance, dependencies and unsafe constructions. The analysis confirmed that the cause of the vulnerability in dot-diver is uncontrolled processing of path keys in the setByPath function, which allowed the attacker to make changes to the global Object.prototype object using malicious fields such as __proto__.

A comprehensive methodology for testing and protecting web applications developed based on TypeScript technology is proposed, which combines modern approaches to static and dynamic analysis, dependency assessment and research of vulnerabilities specific to JavaScript/TypeScript.

Keywords: TypeScript, web applications, information security, security testing, Prototype Pollution, vulnerabilities, dependencies, supply-chain, static code analysis, dynamic analysis, SCA, npm audit, dot-diver.

.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API — Application Programming Interface, інтерфейс програмування застосунків.

AST — Abstract Syntax Tree, абстрактне синтаксичне дерево.

CI/CD — Continuous Integration / Continuous Delivery, безперервна інтеграція та доставка.

CRLF — Carriage Return Line Feed, тип впливу на структуру HTTP-заголовків.

CSRF — Cross-Site Request Forgery, міжсайтове підроблення запиту.

CSP — Content Security Policy, політика безпеки контенту.

CVE — Common Vulnerabilities and Exposures, каталог відомих уразливостей.

DAST — Dynamic Application Security Testing, динамічне тестування застосунків.

DOM — Document Object Model, об’єктна модель документа.

DoS — Denial of Service, відмова в обслуговуванні.

DTO — Data Transfer Object, об’єкт передавання даних.

ESM — ECMAScript Modules, модульна система JavaScript.

HTTP — HyperText Transfer Protocol, протокол передавання гіпертексту.

JS — JavaScript, мова сценаріїв.

JWT — JSON Web Token, токен аутентифікації.

ВСТУП

У сучасному середовищі веб-розробки мова програмування TypeScript набуває дедалі більшої популярності. Це пов’язано з тим, що TypeScript додає статичну типізацію до класичного JavaScript, що підвищує надійність, підтримуваність та масштабованість коду. У багатьох проєктах — від невеликих SPA до великих корпоративних систем — TypeScript став стандартом. Водночас зростає складність таких систем: вони часто містять велику кількість залежностей (npm-пакети, сторонні модулі), реалізують багаторівневу архітектуру (frontend, backend, API, мікросервіси), працюють з критичними бізнес-даними, потребують авторизації, обробки чутливої інформації, взаємодії з базами даних, користувацьким вводом тощо. У такому контексті забезпечення безпеки та стабільності веб-додатків набуває вирішального значення. При цьому простого підходу: «пишемо код і сподіваємося, що все буде добре» — недостатньо. Навіть якщо TypeScript гарантує коректність типів, він не захищає від логічних помилок, неправильного використання залежностей, атак на вразливості типу “prototype pollution”, SQL-/NoSQL-ін’єкцій, XSS, CSRF, неправильного розмежування доступу, неправильної обробки прав та ін. Окрім того, застосунки на TypeScript часто мають глибоке дерево залежностей, де одна невелика npm-бібліотека може містити сотні транзитивних пакетів — ризик появи вразливості в такому ланцюгу дуже високий.

З огляду на це, виникає гостра потреба у системній, методичній, систематизованій і реляційній методиці тестування та захисту таких веб-додатків. Сучасні підходи до безпеки мають охоплювати не лише тестування функціоналу, а комплексний аналіз: залежностей, коду, поведінки в runtime, конфігурацій, захист від типових веб-атак.

Мета і завдання дослідження. Метою роботи є розробка комплексної методики тестування та захисту веб-додатків на TypeScript.

Досягнення визначеної мети передбачає вирішення таких завдань:

– дослідити сучасні підходи до тестування веб-додатків та визначити їхні ключові особливості;

– дослідити специфічні вразливості веб-додатків, що виникають у середовищі TypeScript;

– проаналізувати механізм виникнення та експлуатації вразливості Prototype Pollution;

– дослідити вразливість у бібліотеці dot-diver та визначити причини її появи;

– розробити алгоритм тестування TypeScript-застосунків з урахуванням вимог безпеки;

– проаналізувати залежності та ризики програмного ланцюга постачання;

– дослідити можливості вбудованих інструментів безпеки екосистеми Node.js та оцінити їх ефективність.

Об’єкт дослідження – процеси розробки, тестування та забезпечення безпеки веб-додатків, створених з використанням мови програмування TypeScript та супутньої екосистеми інструментів.

Предмет дослідження – підходи, принципи, техніки та інструменти, що застосовуються для виявлення уразливостей, тестування функціональної та нефункціональної безпеки, аналізу залежностей, статичного і динамічного контролю коду веб-додатків на TypeScript.

Методи досліджень. Для розв’язання поставлених задач у даній кваліфікаційній роботі використано: стандарти безпеки, рекомендації OWASP, офіційна документація TypeScript та інструменти екосистеми.

Наукова новизна одержаних результатів. Розроблена комплексна методика тестування та захисту веб-додатків, розроблених на основі технології TypeScript, яка поєднує сучасні підходи до статичного та динамічного аналізу, оцінювання залежностей і дослідження специфічних для JavaScript/TypeScript уразливостей.

Практичне значення отриманих результатів. Практичне значення полягає у створенні цілісної методики, яка забезпечує підвищення рівня безпеки та надійності веб-додатків, розроблених на TypeScript, через впровадження систематизованих підходів до їхнього аналізу, тестування та захисту.

Публікації та апробація кваліфікаційної роботи.

1. Рущак В.М. Порівняння Flow та Typescript в JavaScript/ Матеріали

науков-практичного симпозіуму «ЗАХИСТ ІНФОРМАЦІЇ», Тернопіль, 2025. – С. 86-91.

2. Рущак В.М., Івасьєв С.В. Дослідження вразливостей бібліотеки CLICKBAR/DOT-DIVER / Збірник матеріалів науково-практичної конференції молодих вчених, аспірантів та студентів «Кібербезпека та комп’ютерно-інтегровані технології»(КБКІТ-2025), Тернопіль, 2025. - С. 68-72.

1 ТЕОРЕТИЧНІ ОСНОВИ ТЕСТУВАННЯ ТА ЗАХИСТУ ВЕБ-ДОДАТКІВ

1.1. Аналіз сучасних підходів до тестування веб-додатків

Розвиток веб-технологій і зростання складності веб-додатків зумовлюють необхідність системного підходу до їх тестування. Тестування є ключовим етапом життєвого циклу програмного забезпечення, який забезпечує виявлення помилок, перевірку відповідності функціональних вимог і підвищення рівня надійності та безпеки застосунку.

Сучасні підходи до тестування веб-додатків базуються на концепціях DevOps та CI/CD, де тестування інтегрується безпосередньо у процес безперервної розробки та розгортання. Це дає змогу забезпечити автоматизовану перевірку змін у коді, зменшити кількість людських помилок та прискорити вихід продукту на ринок.

Залежно від рівня деталізації та цілей, тестування веб-додатків поділяється на кілька основних типів.

Модульне (unit testing) — перевірка окремих функцій або компонентів. Найчастіше використовується разом із фреймворками Jest, Mocha, Vitest або AVA для середовища Node.js і TypeScript.

Інтеграційне тестування (integration testing) — перевірка взаємодії між компонентами системи, включно з API, базами даних та сервісами. Інструменти: SuperTest, Postman, Jest + Testing Library.

Системне та end-to-end тестування (E2E) — імітує дії користувача у браузері для перевірки повної бізнес-логіки. Застосовуються інструменти Cypress, Playwright, Selenium, Puppeteer.

Навантажувальне та стрес-тестування (performance testing) — визначає продуктивність та масштабованість системи, зазвичай із використанням k6, JMeter, Artillery.

Тестування безпеки (security testing) — спрямоване на виявлення вразливостей і перевірку захисту від атак, зокрема XSS, SQL Injection, CSRF

тощо. У цьому контексті застосовуються інструменти OWASP ZAP, Burp Suite, SonarQube, npm audit.

Однією з тенденцій є поєднання різних типів тестування в межах єдиної інфраструктури автоматизації. Сучасні фреймворки підтримують Behavior-Driven Development (BDD) та Test-Driven Development (TDD), що дозволяють створювати тести ще до реалізації функціональності. Ці підходи сприяють підвищенню якості коду та зменшенню кількості дефектів у фінальному продукті.

Особливу роль відіграє статичний аналіз коду, який забезпечує раннє виявлення потенційних проблем без виконання програми. Для TypeScript це може бути реалізовано через ESLint, TypeScript Compiler Diagnostics, або інтеграцію зі системами як SonarQube.

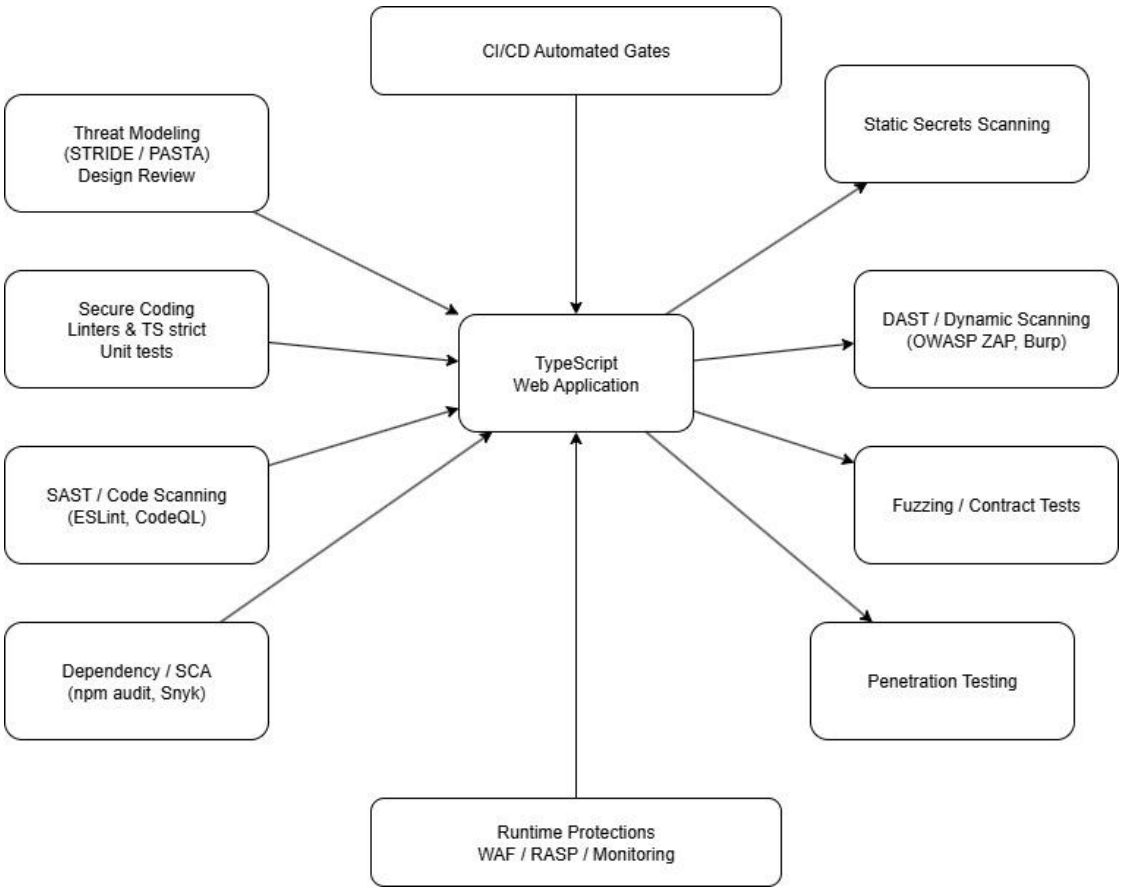

Рисунок 1.1 – Техніки тестування застосунків на TypeScript

Таким чином, сучасні підходи до тестування веб-додатків орієнтовані на автоматизацію та інтеграцію в CI/CD-процеси, застосування типізованих мов (зокрема TypeScript) для зниження кількості помилок на етапі компіляції, підвищення рівня безпеки через поєднання функціонального й безпекового

тестування, використання відкритих фреймворків і інструментів, що підтримують гнучкість і масштабованість процесу тестування.

1.2. Класифікація типів тестування

Тестування веб-додатків є багаторівневим процесом, який охоплює різні аспекти функціональності, взаємодії компонентів, продуктивності та безпеки системи. Класифікація типів тестування базується на рівнях перевірки програмного забезпечення, цілях тестування та методах його виконання. У межах сучасної парадигми розробки програмного забезпечення на TypeScript виділяють п’ять основних категорій тестування: модульне, інтеграційне, наскрізне (end-to-end), продуктивності та безпеки.

Модульне тестування є базовим рівнем тестування, спрямованим на перевірку коректності окремих функцій, класів або компонентів програми. Його мета — виявлення логічних помилок у найменших одиницях коду ще до інтеграції з іншими частинами системи.

У TypeScript модульні тести мають перевагу завдяки статичній типізації, що дозволяє знизити кількість помилок під час компіляції. Найбільш поширені інструменти — Jest, Mocha, Chai, Vitest. Для фронтенд-застосунків використовуються бібліотеки React Testing Library або Vue Test Utils, що забезпечують імітацію користувацької взаємодії з компонентами.

Інтеграційне тестування перевіряє взаємодію між кількома компонентами системи, модулями або сервісами. Основна мета — виявити дефекти, що виникають під час обміну даними або при комбінуванні функціональних частин.

Для веб-додатків, побудованих на TypeScript, інтеграційне тестування часто охоплює взаємодію REST API, баз даних і зовнішніх сервісів. Типовими інструментами є SuperTest, Postman, Jest із використанням тестових контейнерів (Testcontainers) або Docker Compose для імітації середовища.

Наскрізне тестування забезпечує перевірку системи як єдиного цілого, імітуючи дії реального користувача у браузері або мобільному інтерфейсі. Метою E2E-тестів є перевірка бізнес-процесів і користувацьких сценаріїв, включно з навігацією, формами, авторизацією та роботою з API.

Сучасні інструменти, такі як Cypress, Playwright, Puppeteer та Selenium WebDriver, дають змогу автоматизувати такі перевірки. Вони інтегруються в CI/CD-пайплайни, надають можливість створення детальних звітів та знімків екрана у разі помилок. Перевагою Playwright є рідна підтримка TypeScript і паралельне виконання тестів у різних браузерах.

Цей тип тестування визначає здатність веб-додатку ефективно обробляти запити при різному навантаженні. Основними параметрами є час відгуку, пропускна здатність, використання ресурсів і стабільність під навантаженням.

Для тестування продуктивності застосовуються інструменти Apache JMeter, k6, Artillery, Locust. Вони дозволяють симулювати велику кількість одночасних користувачів і виявляти «вузькі місця» в архітектурі. У контексті TypeScript тести продуктивності можуть бути частиною автоматизованих CI/CD-завдань, що допомагає виявляти деградацію швидкодії після кожного оновлення.

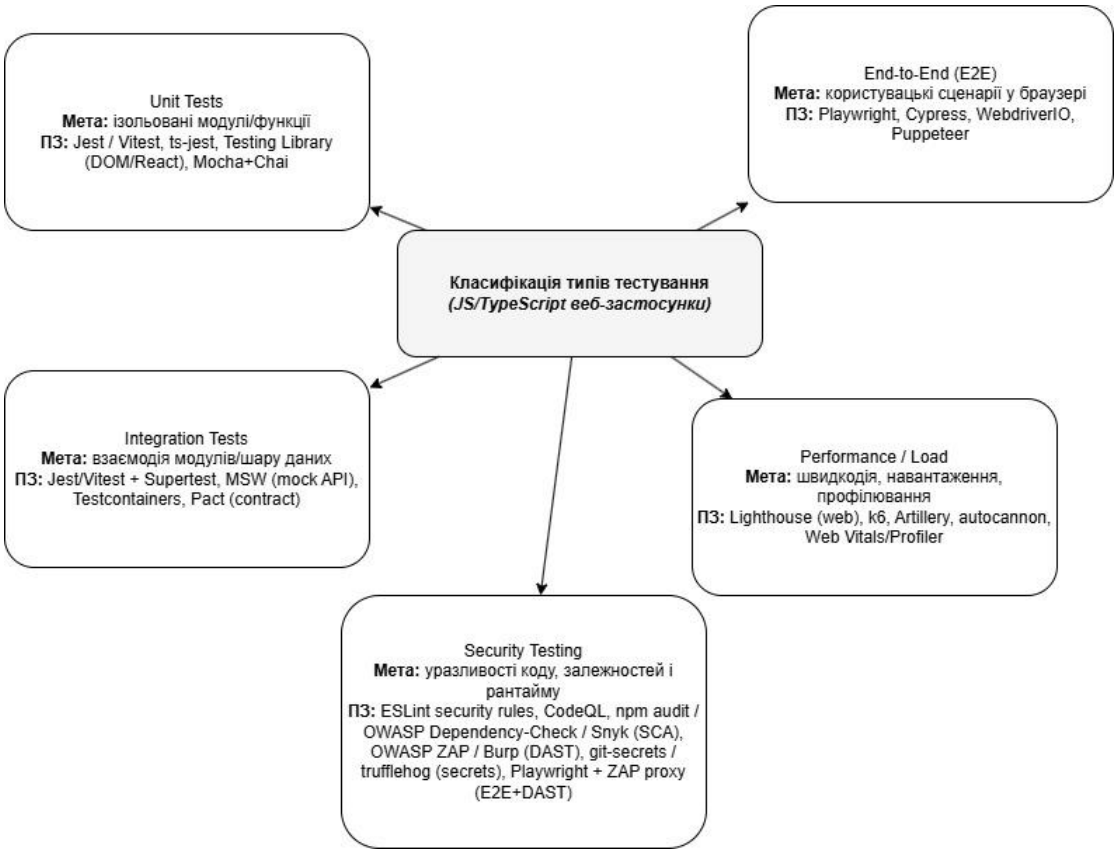

Рисунок 1.2 - Класифікація типів тестування unit, integration, e2e, performance, security testing

Тестування безпеки є критично важливим етапом для веб-додатків, оскільки дозволяє виявити потенційні вразливості до кібератак. Основними напрямами є перевірка на SQL Injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), перевірка автентифікації та авторизації (JWT, OAuth 2.0) та аналіз залежностей і бібліотек на наявність відомих вразливостей.

Для автоматизації цього типу тестування застосовуються OWASP ZAP, Burp Suite, npm audit, Snyk, SonarQube. У випадку TypeScript, статичний аналіз коду допомагає виявляти небезпечні патерни вже під час розробки.

Отже, класифікація типів тестування веб-додатків дозволяє систематизувати процес перевірки програмного забезпечення, розподілити відповідальність між рівнями контролю якості та інтегрувати тестування в усі етапи життєвого циклу проєкту. Поєднання різних підходів (unit, integration, e2e, performance, security) забезпечує повне покриття можливих ризиків і сприяє підвищенню надійності та безпечності веб-додатків, розроблених із використанням TypeScript.

1.3. Основи інформаційної безпеки веб-додатків

Інформаційна безпека веб-додатків є ключовим аспектом їхнього життєвого циклу, оскільки саме через веб-інтерфейси здійснюється основна взаємодія користувача з інформаційними ресурсами та сервісами. Недостатній рівень захисту призводить до витоку персональних даних, несанкціонованого доступу, фінансових збитків та репутаційних ризиків. Тому забезпечення безпеки веб-додатків повинно розглядатися не як окремий етап розробки, а як безперервний процес, інтегрований у всі стадії життєвого циклу програмного продукту (підхід Security by Design).

Поняття та принципи інформаційної безпеки можна відобразити у вигляді кількох основних визначень.

Основу концепції інформаційної безпеки становить тріада CIA (Confidentiality, Integrity, Availability), яка визначає три головні властивості захищеності інформаційних систем:

Конфіденційність (Confidentiality) — забезпечення доступу до даних лише уповноваженим користувачам.

Цілісність (Integrity) — гарантія незмінності інформації під час зберігання або передавання.

Доступність (Availability) — забезпечення можливості доступу до системи та її функцій у будь-який момент часу.

Додатково виділяють принципи аутентифікації, авторизації, невідмовності (non-repudiation) та аудиту подій, які дозволяють контролювати ідентичність користувачів і фіксувати всі дії у системі.

Сучасні веб-додатки постійно піддаються атакам, спрямованим на використання вразливостей у коді або конфігурації. Найпоширеніші типи загроз систематизовано у списку OWASP Top 10, який оновлюється міжнародною організацією Open Web Application Security Project.

До основних категорій належать:

– Ін’єкційні атаки (Injection, SQL/NoSQL/Command Injection) — виконання зловмисних запитів до бази даних або інтерпретатора команд.

– Порушення автентифікації (Broken Authentication) — компрометація облікових даних або сесій користувачів.

– Витік конфіденційних даних (Sensitive Data Exposure) — неправильне шифрування або зберігання чутливої інформації.

– Cross-Site Scripting (XSS) — виконання довільного JavaScript-коду у браузері користувача.

– Cross-Site Request Forgery (CSRF) — виконання несанкціонованих дій від імені користувача без його відома.

– Недостатня валідація введення та помилки конфігурації безпеки.

Технологічні аспекти захисту веб-додатків включають в себе багато механізмів, що стосуються різних частин архітектури веб застосунку.

Для побудови безпечного веб-додатку необхідно впроваджувати багаторівневі механізми захисту, що включають:

Захист клієнтської частини (frontend) складається з застосування Content Security Policy (CSP) для запобігання XSS, використання HTTPOnly і Secure Cookies та впровадження CORS для контролю міждоменних запитів.

Захист серверної частини (backend) повинен складатись із шифрування з'єднань за допомогою HTTPS/TLS, перевірки та фільтрації введення користувачів, контролю прав доступу на рівні API (рольова модель RBAC, JSON Web Tokens).

Захист даних і середовища виконання полягає в використанні ORM для запобігання SQL-ін'єкціям, зберіганні паролів у вигляді хешів (bcrypt, Argon2) та регулярному оновленні бібліотек і залежностей.

Особливу увагу слід приділяти захисту від атаки на залежності (Dependency Confusion), яка є актуальною для проектів на TypeScript і Node.js, де використовуються зовнішні пакети з NPM-репозиторію. Для цього застосовуються інструменти npm audit, Snyk, Dependabot.

Організаційні та процесні аспекти безпеки включають в себе методики тестування та безперервний моніторинг веб застосунку.

Захист веб-додатків не обмежується лише технічними заходами. Ефективна безпека досягається завдяки впровадженню процесів управління безпекою. Наведемо основні техніки що застосовуються.

Security Testing Lifecycle — інтеграція тестування безпеки у всі етапи розробки (SDLC).

Code Review та Static Analysis — регулярна перевірка коду на вразливості.

Penetration Testing — імітація реальних атак для оцінки захищеності.

Безперервний моніторинг (SIEM, логування, IDS/IPS).

У рамках концепції DevSecOps безпека впроваджується як невід'ємна складова процесів розробки, тестування та розгортання. Це дозволяє виявляти вразливості ще до потрапляння коду у виробниче середовище.

Отже, основи інформаційної безпеки веб-додатків полягають у комплексному підході, який об’єднує технологічні, організаційні та процедурні заходи. Використання TypeScript як мови розробки надає додаткові переваги у забезпеченні безпеки завдяки статичній типізації, суворій структурі коду та можливості інтеграції з інструментами статичного аналізу, що загалом підвищує стійкість веб-додатків до сучасних кіберзагроз.

1.4. Типові вразливості веб-додатків

Забезпечення безпеки веб-додатків неможливе без розуміння типових вразливостей, які найчастіше експлуатуються зловмисниками. Організація OWASP (Open Web Application Security Project) розробила міжнародний стандарт — OWASP Top 10, який відображає найпоширеніші категорії ризиків у веб-додатках. Остання редакція OWASP Top 10 (2021 року) акцентує увагу на змішаних технічних і концептуальних загрозах, пов’язаних із проектуванням, розробкою та експлуатацією програмного забезпечення.

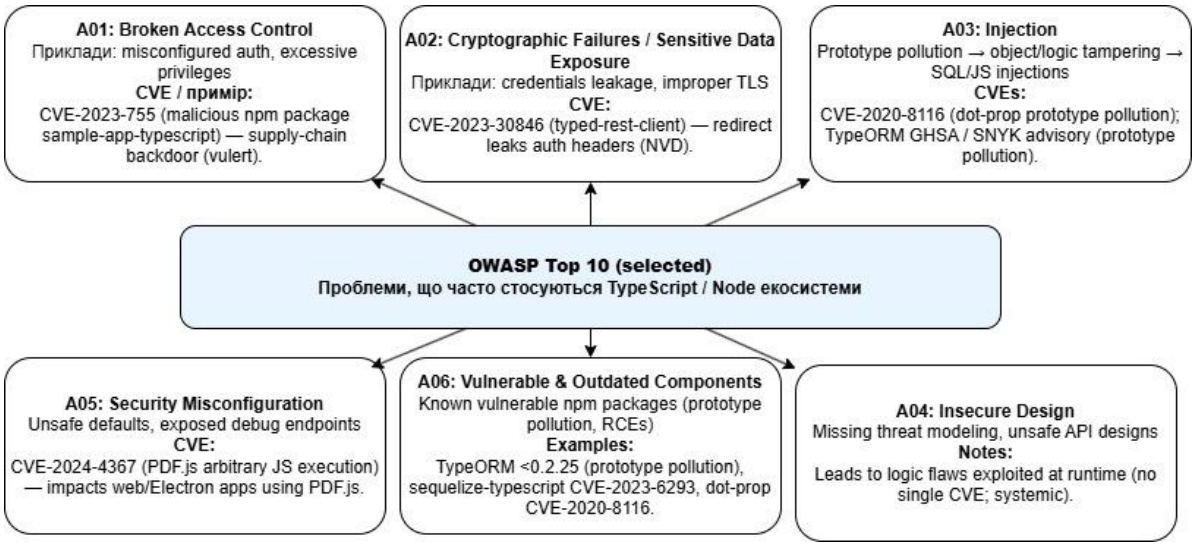

Рисунок 1.3 – Аналіз TypeScript / Node вразливостей з OWASP Top 10

Broken Access Control (Порушення контролю доступу) - найпоширеніша вразливість сучасних веб-додатків. Вона виникає, коли користувач отримує доступ до функцій або ресурсів, які йому не призначені.

Прикладом такої вразливості може бути доступ до чужих облікових записів через зміну ID у URL (/user/2 → /user/1), недостатня перевірка прав користувача на сервері (авторизація лише на фронтенді) та відсутність розмежування ролей у REST API.

У TypeScript-додатках на Node.js це часто трапляється через відсутність middleware-перевірок у фреймворках Express або NestJS. Рішенням є впровадження рольової моделі доступу (RBAC) та централізованих політик авторизації.

Cryptographic Failures (Криптографічні помилки) - тип вразливості охоплює неправильне використання або відсутність шифрування.

Проблеми виникають при зберіганні паролів у відкритому вигляді або без “солі”. Використання застарілих алгоритмів (MD5, SHA1) та передавання даних через HTTP без TLS також призводить до потенційної вразливості.

Для TypeScript слід застосовувати сучасні бібліотеки (bcrypt, Argon2, crypto) і зберігати ключі шифрування в безпечному середовищі (наприклад, dotenv, Vault).

Ін’єкції (SQL, NoSQL, LDAP, Command Injection) — класична проблема, коли введення користувача некоректно фільтрується і потрапляє в інтерпретатор.

У Node.js це проявляється при безпосередньому використанні рядків запитів до баз даних, наприклад:

db.query(`SELECT * FROM users WHERE name='${userInput}'`);

Такі конструкції відкривають можливість для SQL Injection. Запобігти цьому можна через використання ORM (TypeORM, Prisma) або параметризованих запитів.

Insecure Design (Небезпечне проектування) - вразливість пов’язана не з конкретним кодом, а з архітектурними рішеннями — відсутністю механізмів обмеження дій користувача, перевірки введення чи політики безпеки.

Прикладом може бути REST API, що дозволяє масове оновлення даних без валідації.

У TypeScript-проектах доцільно впроваджувати Data Transfer Objects (DTO) та схеми перевірки (через Zod, Joi, class-validator) для контролю вхідних даних.

Security Misconfiguration (Помилки конфігурації безпеки) - одна з найпоширеніших причин компрометації веб-додатків. Вона охоплює використання стандартних паролів або відкритих портів та увімкнені режими налагодження (debug, dev mode) на продуктивному сервері;

Надмірно детальні повідомлення про помилки також додають ризиків для безпеки веб застосунку.

У середовищі Node.js важливо відключати CORS за замовчуванням, правильно налаштовувати Helmet.js для безпечних HTTP-заголовків і обмежувати доступ до адміністративних панелей.

Загроза Vulnerable and Outdated Components (Вразливі або застарілі компоненти) також досить розповсюджена. Більшість веб-додатків використовує зовнішні бібліотеки. Якщо такі компоненти не оновлюються, вони можуть містити відомі вразливості.

У TypeScript/Node.js-проектах це виявляється через залежності з npm. Інструменти npm audit, Snyk або OWASP Dependency-Check допомагають виявити небезпечні пакети. Рекомендується впровадити автоматичну перевірку в CI/CD.

Помилки при реалізації механізмів входу, сесій і токенів часто призводять до компрометації акаунтів. Такий вид вразливостей називають Identification and Authentication Failures (Порушення автентифікації).

Прикладом цих вразливостей може бути зберігання JWT-токенів у LocalStorage (доступні через XSS) та відсутність механізму “session timeout”. Також варто приділити увагу шифруванню cookie. Для захисту необхідно використовувати HTTPOnly, Secure cookie, обмеження терміну дії токенів і двофакторну автентифікацію.

Вразливості типу Software and Data Integrity Failures (Порушення цілісності ПЗ та даних) виникають при неконтрольованому оновленні або підвантаженні коду. Наприклад, якщо веб-додаток завантажує зовнішні скрипти без перевірки підпису. Рекомендується використовувати Subresource Integrity (SRI), перевірку хешів і контроль версій через package-lock.json.

Security Logging and Monitoring Failures (Недостатній моніторинг і логування) вказує на відсутність адекватного моніторингу не дозволяє своєчасно виявити інциденти безпеки. Рішенням є впровадження централізованого логування (Winston, Morgan) і систем моніторингу (ELK Stack, Prometheus, Grafana). Додатково необхідно забезпечити аудит дій користувачів і адміністраторів.

Server-Side Request Forgery (SSRF)-атаки виникають, коли сервер здійснює запити до вказаних користувачем URL, що може дати зловмиснику доступ до внутрішніх ресурсів. Для запобігання цих дій слід обмежувати цільові адреси, використовувати allowlist, перевіряти схему URL і блокувати запити до локальних IP-адрес (наприклад, 127.0.0.1, 169.254.*.*).

Таким чином, вразливості з переліку OWASP Top 10 охоплюють більшість типових ризиків, з якими стикаються веб-додатки. Для середовища TypeScript важливо поєднувати захист на рівні коду, статичний аналіз, тестування безпеки та постійне оновлення компонентів. Впровадження цих підходів дозволяє не лише мінімізувати ризики, але й створити культуру безпечної розробки (Secure Coding Practices) у команді.

1.5. Огляд технологій TypeScript і його особливостей щодо безпеки

TypeScript — це надбудова над мовою JavaScript, розроблена компанією Microsoft, яка додає статичну типізацію та сучасні можливості об’єктно-орієнтованого програмування. Код на TypeScript компілюється у стандартний

JavaScript, що забезпечує сумісність із будь-яким браузером або середовищем виконання, таким як Node.js.

Основна мета TypeScript полягає в підвищенні надійності, читабельності та передбачуваності коду, що має особливе значення для великих веб-додатків із складною архітектурою.

TypeScript забезпечує можливість використовувати сувору типізацію для зменшення логічних помилок. Мова дозволяє застосовувати класи, інтерфейси, дженерики та модулі, що полегшують структуризацію коду та інтегруватися з сучасними фреймворками — Angular, React, NestJS, Next.js. Є можливість забезпечити підтримку асинхронного програмування з використанням async/await.

TypeScript не є інструментом безпеки у вузькому сенсі, однак він суттєво зменшує ризики вразливостей, пов’язаних із помилками програмної логіки та некоректною обробкою даних. Серед ключових безпекових особливостей варто виділити статичну типізацію, суворий режим, контроль над структурами даних, інтеграція з системами безпеки, покращена підтримка IDE та аналізу коду.

Статична типізація полягає в виявленні помилок типів на етапі компіляції запобігає непередбачуваній поведінці коду, що часто призводить до XSS або логічних вразливостей. Наприклад, визначення точних типів для даних, отриманих із запитів, допомагає уникнути маніпуляцій із введенням користувача.

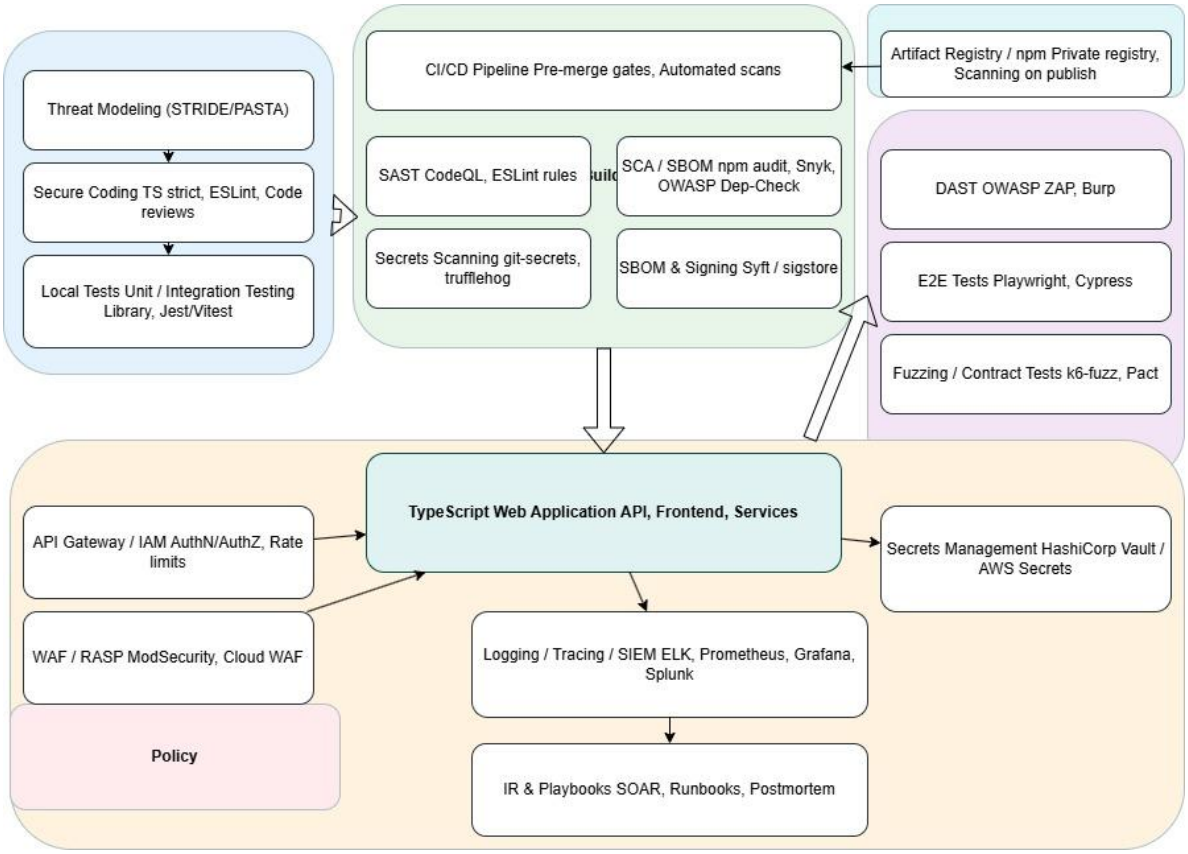

Рисунок 1.5 – Архітектура засобів безпеки TypeScript додатку

Увімкнення строгого режиму компілятора ("strict": true у tsconfig.json) зменшує ймовірність помилок через неініціалізовані змінні, неочікувані null або undefined, що є потенційними джерелами уразливостей.

Використання інтерфейсів і типів забезпечує чітке визначення структури об’єктів, що дозволяє реалізувати перевірку відповідності даних схемам без додаткових бібліотек.

TypeScript легко інтегрується з бібліотеками, що підвищують рівень безпеки — наприклад, helmet (для HTTP заголовків), express-validator (валідація даних запиту), або zod (схемна типізація для API).

Завдяки потужним засобам розробки (VS Code, WebStorm) забезпечується автоматичне виявлення потенційно небезпечних конструкцій, зокрема неконтрольованих викликів функцій або неправильного використання API.

Технологічна екосистема TypeScript для безпечних веб-додатків складається з властивостей комплексів та фреймворків, що застосовуються з мовою в процесі розробки.

TypeScript використовується в комплексі з фреймворками та бібліотеками, що підтримують високий рівень безпеки:

Angular — має вбудований механізм захисту від XSS, строгі шаблони та автоматичне екранування даних.

NestJS — фреймворк серверного боку на TypeScript із підтримкою модульності, валідації даних і middleware для автентифікації.

Next.js / React з TypeScript — дозволяють реалізовувати безпечні інтерфейси з використанням перевірених компонентів і типобезпеки.

Також важливою складовою є інтеграція з інструментами тестування, такими як Jest, Mocha або Playwright, які дозволяють проводити автоматизовані тести безпеки та валідації у середовищі TypeScript.

Попри суттєві переваги, TypeScript не гарантує повного захисту від уразливостей, пов’язаних із помилками у бізнес-логіці та вразливими залежностями у npm-пакетах. Також загрозу становлять неправильні налаштування серверного середовища.

Тому TypeScript слід розглядати як інструмент профілактики, який допомагає розробникам уникати частини типових помилок, але не замінює повноцінного безпекового аудиту.

TypeScript є потужним засобом підвищення надійності та безпеки веб-додатків завдяки статичній типізації, суворій перевірці коду й інтеграції з сучасними фреймворками. Його використання сприяє формуванню культури безпечного програмування та мінімізації людського фактору при розробці.

Проте ефективність безпеки досягається лише в поєднанні TypeScript із комплексними підходами — аудитом залежностей, тестуванням, DevSecOps і практиками безпечного кодування.

1.6 Аналіз типових вразливостей у веб-додатках, розроблених на TypeScript

Попри широкі можливості TypeScript у підвищенні надійності програмного коду, більшість вразливостей веб-додатків пов’язана не з особливостями мови, а з помилками реалізації логіки, некоректною валідацією даних та небезпечною взаємодією із зовнішніми системами.

TypeScript мінімізує ризики, спричинені динамічною природою JavaScript, однак не може повністю усунути загрози, пов’язані з архітектурними недоліками або людським фактором.

Основними джерелами уразливостей є недбале поводження з даними користувача та використання застарілих або вразливих npm-пакетів. Також загрозу становлять відсутність належних механізмів контролю доступу і помилки при налаштуванні серверного середовища або HTTP-заголовків.

Вразливість XSS залишається актуальною навіть у TypeScript-додатках, особливо при динамічному формуванні HTML. Наприклад некоректне відображення вмісту, отриманого з API або введеного користувачем без очищення, може призвести до виконання шкідливого скрипта.

Шляхом запобігання цієї вразливості є автоматичне екранування в Angular, використання бібліотек DOMPurify або xss-filters.

У додатках, що використовують токени або сесійні cookie, CSRF-атаки можуть змусити користувача виконати дію без його згоди. Дії щодо запобігання складаються з впровадження CSRF-токенів (csurf), застосування заголовків SameSite для cookie.

Використання ORM (наприклад, TypeORM або Prisma) значно знижує ризик SQL-ін’єкцій, однак ручне складання запитів через query builder може бути небезпечним.

Для запобігання виникнення загрози потрібне використання параметризованих запитів, відмова від динамічного конкатенування рядків.

Використання неконтрольованої серіалізації/десеріалізації JSON-даних може дозволити зловмиснику вставити неочікувані об’єкти або викликати помилки виконання. Щоб усунути можливу вразливість варто впровадити перевірку структури даних через Zod, class-transformer або Joi.

Помилки при роботі з JWT або неправильне налаштування токенів можуть призвести до підміни користувача. Для того щоб уникнути цього потрібне встановлення часу життя токенів (exp), використання безпечних алгоритмів (RS256), зберігання секретів у .env.

Часто вразливості виникають через неправильне налаштування серверів або фреймворків. Наприклад залишені відкритими endpoints /swagger, /admin та відсутність HTTPS або слабкі CORS-політики. Серед засобів запобігання є використання бібліотек helmet, dotenv, обмеження доступу до службових маршрутів.

Надмірна кількість запитів або неконтрольована обробка великих JSON-об’єктів може спричинити перевантаження сервера.

Застосування express-rate-limit, контроль розміру запитів, кешування можуть усунути цб вразливість.

Веб-додатки TypeScript значною мірою залежать від npm-екосистеми. Небезпека полягає в тому, що навіть популярні пакети можуть містити шкідливий код або вразливості, виявлені після релізу.

Типові загрози що виникають при побудові додатків з використанням сторонніх пакетів:

– ін’єкції у бібліотеках логування;

– неконтрольовані оновлення залежностей;

– використання небезпечних версій пакетів.

npm audit, Snyk, Dependabot, Retire.js — автоматично перевіряють залежності на відомі уразливості та рекомендують оновлення.

TypeScript не може запобігти логічним вразливостям, які виникають через неправильне проектування бізнес-процесів, наприклад:

– відсутність перевірки ролей користувачів (authorization bypass);

– помилки в реалізації багатофакторної автентифікації;

– недостатня ізоляція даних користувачів у багатосесійних системах.

Для мінімізації таких ризиків необхідно впроваджувати аудит коду, рев'ю бізнес-логіки та пентестинг на рівні прототипу.

Практики мінімізації вразливостей у TypeScript-додатках включає в себе ряд етапів:

– застосування ESLint security rules і TypeScript strict mode;

– впровадження DevSecOps-підходу з автоматичним аналізом коду;

– використання контейнеризації (Docker) із обмеженими правами доступу;

– впровадження CI/CD-перевірок безпеки перед розгортанням;

– регулярне оновлення залежностей і моніторинг CVE-звітів.

Попри високий рівень надійності TypeScript, основна частина загроз залишається актуальною через людські помилки та складність веб-екосистеми.

Для забезпечення безпеки необхідно поєднувати технологічні засоби (TypeScript, фреймворки, бібліотеки) з процесними підходами (аудит, тестування, DevSecOps).

Таким чином, безпечна розробка на TypeScript можлива лише за умови системного впровадження контролю якості коду, управління залежностями та тестування безпеки на всіх етапах життєвого циклу веб-додатку.

2 ВРАЗЛИВОСТІ ЗАБРУДНЕННЯ ПРОТОТИПУ

2.1 Аналіз вразливості забруднення прототипу

У JavaScript усі сутності є об’єктами, зокрема функції та визначення класів. Наслідування реалізується через модель прототипів. Кожен об’єкт у JavaScript пов’язаний зі спеціальним об’єктом — прототипом. Об’єкт успадковує всі властивості свого прототипу. Для доступу до прототипу об’єкта використовується вбудоване поле __proto__. Пошук будь-якого поля здійснюється по ланцюгу прототипів (prototype chain): спочатку поле шукають у самому об’єкті, потім у його прототипі й далі — до найвищого рівня наслідування.

Prototype Pollution дозволяє зловмисникові «забруднити» властивість глобального об’єкта, яка може бути успадкована користувацькими об’єктами й створювати загрозу безпеці застосунку.

Умови, необхідні для успішного втілення атаки Prototype Pollution:

1. Наявність недовірених вхідних даних, що використовуються для «забруднення» глобального об’єкта (prototype pollution source).
2. Наявність вразливих функцій або місць у коді, які можуть призвести до безпекових проблем при використанні змінених властивостей (sink).
3. Можливість використати «забруднену» властивість глобального об’єкта у вразливій функції без її фільтрації або валідації (exploitable gadget).

Механізм реалізації атаки зазвичай виглядає так:

1. Зловмисник через доступний інтерфейс модифікує (забруднює) властивість глобального об’єкта.
2. У застосунку вразлива функція отримує або оперує об’єктом, який наслідує від цього «забрудненого» прототипу.
3. Вразлива функція використовує змінену властивість прототипу, встановлену зловмисником, що призводить до порушення безпеки (ескалація прав, обходи валідації, виконання небажаної логіки тощо).

Типові властивості та шляхи доступу, що використовуються в атаках:

- object.constructor.prototype.pollutedField
- object.__proto__.pollutedField

У прикладах ідентифікатори властивостей часто мають умовний характер; важливо розуміти сам механізм — зміну прототипу й подальше несанкціоноване використання цих змін у коді.

2.2 Дослідження вразливості в бібліотеці dot-diver

Бібліотека @clickbar/dot-diver, реалізована на TypeScript (відкритий вихідний код), надає зручний API для зчитування значення поля об’єкта (функція getByPath) та запису значення у поле об’єкта (функція setByPath).

Уразливість виявлено у функції setByPath, яка приймає три аргументи:

- object — об’єкт, у властивість якого встановлюється значення;
- path — шаблон шляху до властивості, що підлягає зміні;
- value — значення, яке має бути записане у вказане поле.

При виклику setByPath відбувається рекурсивний обхід елементів у шаблоні шляху. Після знаходження цільового поля йому присвоюється нове значення.

Нижче зазначено, що у вихідному повідомленні наводилося загальне пояснення механіки роботи setByPath.

```
import { getByPath, setByPath } from '@clickbar/dot-diver'

// Define a sample object with nested properties
const object = {
  a: 'hello',
  b: {
    c: 42,
    d: {
      e: 'world',
    },
  },
  f: [{ g: 'array-item-1' }, { g: 'array-item-2' }],
}
// Example 2: Set a value by path
setByPath(object, 'a', 'new hello')
console.log(object.a) // Output: 'new hello'

setByPath(object, 'f.1.g', 'new array-item-2')
console.log(object.f[1].g) // Output: 'new array-item-2'
```

Рисунок 2.1 - Механізм роботи setByPath

Основна проблема застосування цієї функції полягає в тому, що якщо значення шляху містить посилання на прототип, з'являється можливість встановити властивості прототипу, що може призвести до забруднення глобального прототипу.

```
252  function setByPath<
253    T extends SearchableObject,
254    P extends PathEntry<T, 10> & string,
255    V extends PathValueEntry<T, P, 10>
256  >(object: T, path: P, value: V): void {
257    //
258    const pathArray = (path as string).split('.')
259    const lastKey = pathArray.pop()
260
261    if (lastKey === undefined) {
262      throw new Error('Path is empty')
263    }
264    //
265    const objectToSet = pathArray.reduce(
266      (accumulator: any, current) => accumulator?.[current],
267      object
268    )
269
270    if (objectToSet === undefined) {
271      throw new Error('Path is invalid')
272    }
273    //
274    objectToSet[lastKey] = value
275  }
```

Рисунок 2.2 - Код функції до виправлення (версія 1.0.1)

Цей же код після виправлення (версія 1.0.2) приведено на рисунку 2.3. У версії 1.0.1 функція setByPath дозволяє встановлювати властивості за шляхом, що подається у вигляді рядка (наприклад, "a.b.c"). Через відсутність перевірок спеціальних ключів (наприклад, __proto__, constructor.prototype) зловмисник може змінювати властивості прототипу — тобто відбувається prototype pollution. Це відкриває шлях до обходу механізмів контролю доступу й іншої небажаної модифікації поведінки застосунку.

setByPath виконує рекурсивний або ітеративний обхід шляхового шаблона (split('.') → reduce/loop) і без додаткових перевірок записує значення у знайдене поле.

Немає захисту проти модифікації властивостей прототипу: не відкидаються ключі на зразок __proto__, prototype, constructor.

Відсутній код, щоб перевірити, чи властивість є власною (own property) об’єкта, або чи операція створює нове поле на самому об’єкті, а не змінює глобальний прототип.

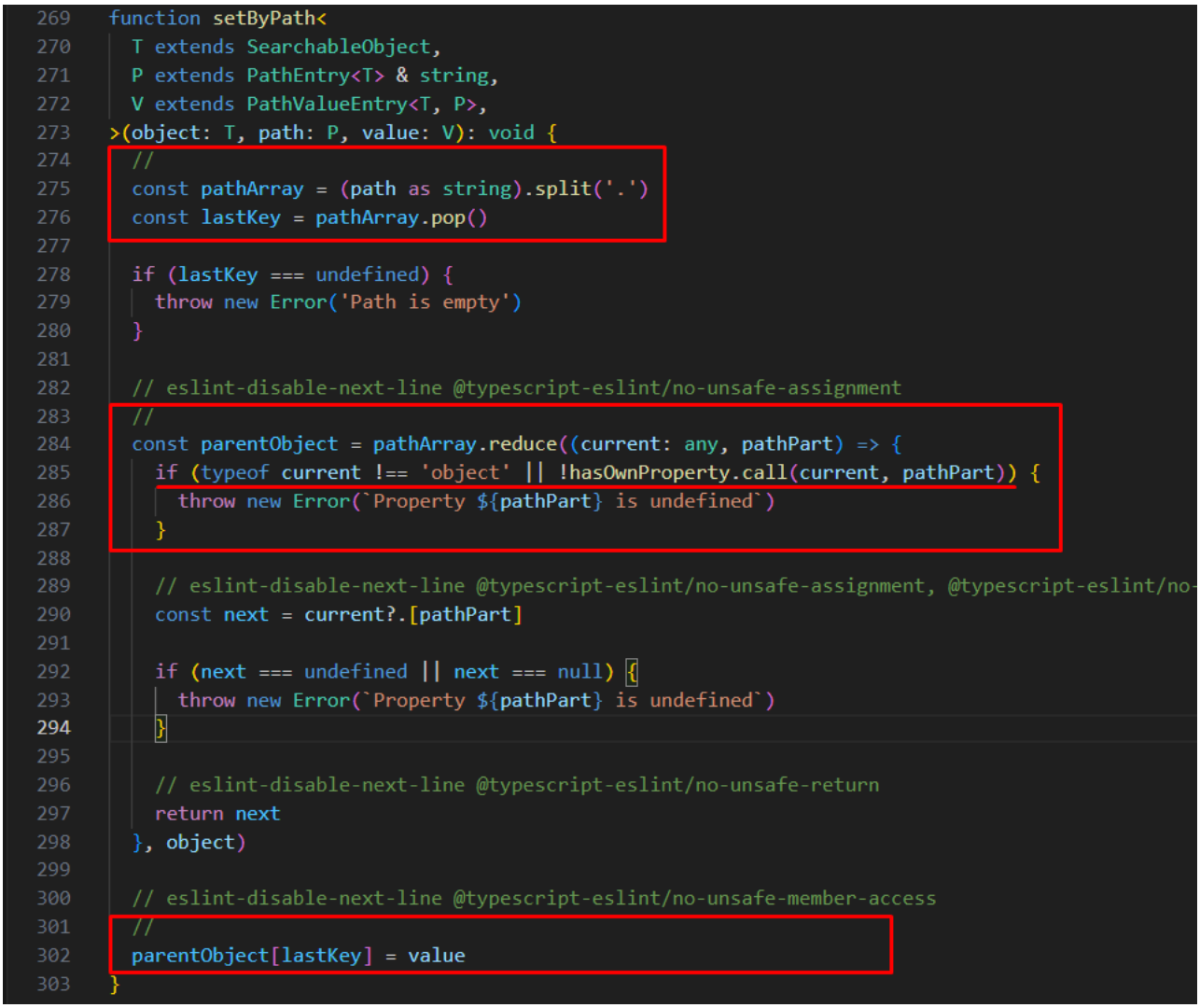

Рисунок 2.3 - Код функції після виправлення(версія 1.0.2)

У виправленій версії було додано перевірку наявності власної властивості в об’єкті (за допомогою методу Object.prototype.hasOwnProperty) перед внесенням змін.

Якщо потрібне поле відсутнє, викликається виняток із відповідним повідомленням.

Вектор атаки можна описати наступними кроками. Зловмисник подає шлях, який містить __proto__ або інший спеціальний сегмент (через користувацькі поля/JSON).

setByPath проходить шлях і в кінці записує задане значення не в локальний об’єкт, а в прототип (Object.prototype або інший спільний прототип).

Після цього будь-який інший код, що покладається на наявність певної властивості або перевіряє її присутність через спадкування, може побачити змінену поведінку (наприклад, user.isAdmin === true), що призводить до ескалації привілеїв або обходу авторизації. Алгоритм реалізації вразливості приведено на рисунку 2.4.

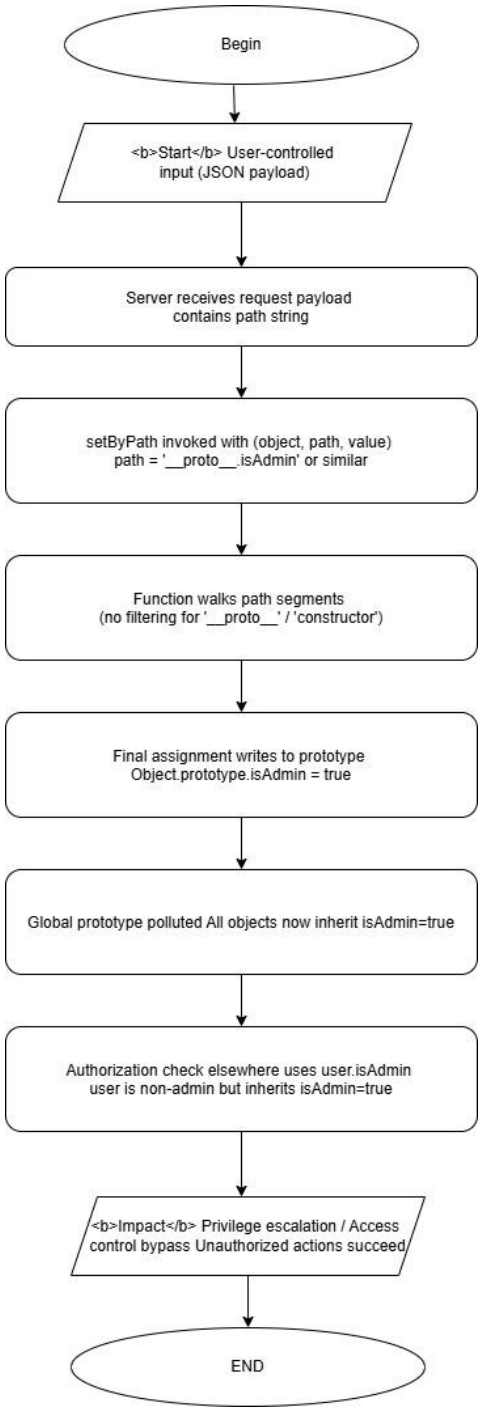

Рисунок 2.4 – Схема алгоритму використання вразливості setByPath

У прикладі розглядається застосунок для обліку прочитаних книг, у якому реалізовано механізм розмежування користувачів із різними ролями:

- user — має право додавати дані про прочитані книги та переглядати їх;
- admin — має право видаляти книги зі списку.

Розмежування доступу реалізовано за допомогою додаткової властивості isAdmin, яка присутня лише в об’єкта користувача з успішною автентифікацією для ролі admin.

В інших користувачів це поле відсутнє, як показано на рисунку 2.5.

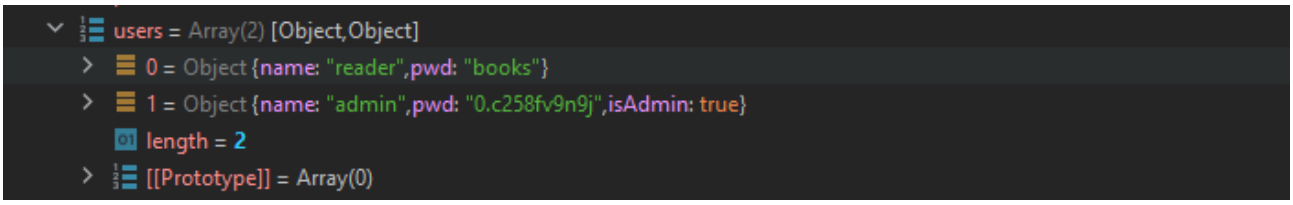

Рисунок 2.5 - Механізм роботи setByPath

Фрагмент коду, який виконує обробку запиту на видалення книги за ключем title та реалізує контроль прав доступу, може виглядати так, як показано на рисунку 2.6.

```
app.delete('/books/:title', (req, res) => {
  const { title } = req.params;
  const user = req.user; // об’єкт користувача, отриманий після автентифікації

  // Перевірка прав доступу
  if (!user || !user.isAdmin) {
    return res.status(403).json({
      error: 'Access denied: insufficient privileges.'
    });
  }

  // Пошук книги за назвою
  const index = books.findIndex((book) => book.title === title);

  if (index === -1) {
    return res.status(404).json({
      error: `Book with title '${title}' not found.`
    });
  }

  // Видалення книги
  books.splice(index, 1);
  return res.status(200).json({
    message: `Book '${title}' was successfully deleted.`
  });
});
```

Рисунок 2.6 - Фрагмент коду, який виконує обробку запиту на видалення книги

У застосунку використовуються такі API-команди:

отримати список книг:

$ curl -v -X GET -H http://192.169.27.1:6000/

оновити список книг:

$ curl -v -X PUT -H "Content-Type:application/json" --data '{"auth":{"name":"reader", "pwd":"books"},"title":"Tom Sawyer"}' http://192.169.27.1:6000/

видалити книгу з певною назвою:

$ curl -v -X DELETE -H "Content-Type:application/json" --data '{"auth":{"name":"reader", "pwd":"books"},"title":"Tom Sawyer"}' http://192.169.27.1:6000/

Після спроби видалення книги від імені користувача reader повертається відповідь із кодом 403 та повідомленням “Access denied”, що свідчить про відсутність прав на виконання цієї операції.

До запиту додається корисне навантаження для атаки:

$ curl -v -X PUT -H "Content-Type:application/json" --data '{"auth":{"name":"reader", "pwd":"books"},"title":"Tom Sawyer", "note":"__proto__.isAdmin", "text":true}' http://192.169.27.1:6000/

У відповіді на цей запит відображається результат успішного оновлення списку книг. Окрім того, у глобальному об’єкті Object з’явилося поле isAdmin, як показано на рисунку 2.7.

Рисунок 2.7 - Object з полем isAdmin

Після повторного запиту на видалення книги від імені користувача reader у відповіді повертається повідомлення про успішне видалення книги, що

свідчить про реалізацію атаки та обхід механізму розмежування доступу, реалізованого в застосунку.

2.3 Механізм протидії вразливості бібліотеки dot-diver

Основні рекомендації щодо усунення проблем безпеки, пов’язаних із вразливістю Prototype Pollution. Наступна послідовність кроків дозволить зменшити ризик виникнення вразливостей.

Регулярне оновлення пакетів до останніх стабільних версій.

Виявлення вразливих компонентів за допомогою SCA (Software Composition Analysis) та SAST (Static Application Security Testing).

Валідація й санітизація недовірених вхідних даних на межі застосунку (input validation / sanitization);

Уникнення небажаного наслідування, якщо воно не потрібно — створювати об’єкти без прототипу:

```
let obj = Object.create(null);
let obj = { __proto__: null };
```

Захист від модифікації глобальних прототипів за допомогою API: Object.freeze() та Object.seal(). Слід враховувати, що застосування цих методів може порушити роботу бібліотек, які свідомо змінюють атрибути глобальних прототипів, тому необхідно ретельно тестувати сумісність перед впровадженням.

Фільтрувати/забороняти спеціальні ключі. Приміром, відкидати __proto__, prototype, constructor та будь-які імена, що містять prototype/__proto__.

Перевіряти власні властивості перед встановленням (якщо мета — змінювати тільки власні поля):

```
if (!Object.prototype.hasOwnProperty.call(targetObject, key)) {
  throw new Error(`Property ${String(key)} not allowed to set`);
}
```

При створенні нових властивостей — не дозволяти створювати їх за допомогою шляхів, що містять прототипні сегменти.

Запобігати створенню ланцюгів, що ведуть у прототип:

Перевіряти при кожному кроці walk, що наступний крок не є __proto__ або constructor.

Використовувати об'єкти без прототипу для внутрішніх структур, якщо наслідування не потрібно:

const safe = Object.create(null);

Додавати unit-тести та інтеграційні тести, що перевіряють реакцію на вхідні шляхи з __proto__ та подібними.

Документувати поведінку: чітко вказати, чи дозволяється створення нових властивостей і як обробляються спеціальні ключі. Приклад безпечного фрагменту коду приведено на рисунку 2.8.

```
function isForbiddenKey(key: string) : boolean{
  return key == = '__proto__' || key == = 'prototype' || key == = 'constructor' || key.includes('prototype');
}

function setByPathSafe(obj : any, path : string, value : any) {
    const parts = path.split('.');
    let cur = obj;

    for (let i = 0; i < parts.length; i++) {
        const key = parts[i];
        if (isForbiddenKey(key)) {
            throw new Error(`Forbidden path segment : ${ key }`);
        }

        if (i == = parts.length - 1) {
            // final segment — встановлюємо лише якщо це власне поле або дозволено створення
            if (!Object.prototype.hasOwnProperty.call(cur, key) && cur != = Object(cur)) {
                // приклад політики: заборонити створювати нові властивості у прототипних ланцюгах
                throw new Error(`Cannot set non - own property ${ key }`);
            }
            cur[key] = value;
        }
        else {
            // пересуваємось глибше — створюємо проміжний об'єкт тільки як plain object
            if (cur[key] == null || typeof cur[key] != = 'object') {
                cur[key] = Object.create(null);
            }
            cur = cur[key];
        }
    }
}
```

Рисунок 2.8– Приклад безпечного фрагменту коду

Було розроблено набір unit тестів для перевірки «безпечної» реалізації setByPath. Тести приведені в додатку А.

3 АНАЛІЗ ЗАЛЕЖНОСТЕЙ TYPESCRIPT-ЗАСТОСУНКІВ

3.1 Алгоритм тестування TypeScript-застосунків

Процес тестування TypeScript-застосунку починається з формулювання вимог і моделювання загроз. На цьому етапі команда визначає функціональні та нефункціональні вимоги, уточнює сценарії використання і будує карту потенційних атак — наприклад, які ресурси мають бути захищені, які активи представляють цінність і які ролі користувачів існують у системі. Ретельне threat-modeling допомагає виявити конструкційні вразливості на рівні архітектури, такі як ненадійний дизайн авторизації, відсутність розмежування привілеїв або можливість інжекцій через неочищені компоненти інтерфейсу; також це дозволяє заздалегідь спланувати захист від supply-chain-ризиків і визначити критичні місця для подальшого тестування.

Під час розробки (development) пріоритетом є застосування безпечних практик кодування та інструментальної підтримки. Впровадження строгих налаштувань TypeScript, налаштування лінтерів і правил безпеки, регулярні перевірки рев'ю коду та модульні тести мінімізують клас помилок, що призводять до логічних вад, неправильної обробки даних або помилок типізації. На цьому рівні усуваються часті проблеми: небезпечна обробка вхідних даних, некоректні припущення щодо форматів, ненавмисне використання глобальних змінних, а також ризики, пов'язані з неправильним застосуванням API (наприклад, неналежна валідація даних перед їх використанням у запитах).

Локальні тести, що виконуються розробниками, забезпечують швидкий зворотний зв'язок. Unit-тести фокусуються на ізольованій логіці функцій і дозволяють виявляти регресії та помилки реалізації, що можуть стати джерелом більш серйозних вад у продуктивному середовищі. Тести на рівні компонентів або service-layer допомагають переконатися, що бізнес-логіка не допускає небезпечних умов, таких як некоректні сценарії авторизації, потенційні race-condition або неправильно реалізовані граничні випадки.

CI/CD-петля є наступною ступенем контролю якості й безпеки: у ній автоматично запускаються лінтинги, SAST-сканування та SCA-аналіз залежностей, а також перевірки на витік секретів. Ці інструменти дозволяють виявити уразливі залежності, відомі CVE у бібліотеках, небезпечні патерни у коді (наприклад, використання небезпечних функцій, потенційна SQL/NoSQL-ін’єкція або XSS-паспортизація) і випадкове закомічування ключів чи паролів. У CI також відпрацьовуються політики — наприклад, блокування мерджу при виявленні високої критичності вразливостей або при невиконанні критичних тестів.

Після проходження автоматичних перевірок відбуваються інтеграційні тести, які перевіряють коректну взаємодію компонентів, підсистем та зовнішніх сервісів. Цей етап виявляє проблеми, що не проявляються в ізольованих тестах, — некоректну роботу фасадних шарів, помилки у схемах API, непридатність конфігурацій бази даних, а також уразливості, пов’язані з логікою обробки транзакцій або неконсистентністю станів між сервісами. Інтеграційні тести допомагають виявити сценарії, за яких може відбутися витік даних або обхід логіки контролю доступу через несподівані ланцюги викликів.

E2E-тестування і симуляція користувацьких сценаріїв у тестовому середовищі перевіряють поведінку фронтенду і бекенду під реальними умовами. Прогон сценаріїв у браузері або через емуляцію API-запитів дозволяє виявляти кросс-сайт скриптинг, помилки в обробці сесій, невідповідності валідації форм і вразливі місця в навігації або відображенні даних. Ручні або автоматизовані E2E-тести також корисні для перевірки захисту від атак, що експлуатують послідовності кроків користувача, наприклад обходи авторизації через послідовність запитів.

Тестування продуктивності та навантаження є невід’ємною частиною підготовки релізу: під час цих прогонів перевіряють, як система поводиться під різним навантаженням, чи немає бенчмарків, що призводять до деградації, та чи не відкриваються нові вектори атак через ресурсоємні операції. Навантажувальні тести дозволяють виявити вузькі місця, що можуть бути використані для DoS-

атаки, некоректну обробку таймаутів або витоки пам’яті, які укупі впливають на стійкість системи.

Окремий напрямок — спеціалізоване безпекове тестування. Динамічний аналіз (DAST), ручні пентести та перевірки залежностей зосереджуються на виявленні класичних вразливостей: ін’єкцій різного типу, XSS, неправильного налаштування CORS, уразливостей у десеріалізації, проблем з контролем доступу, а також prototype pollution і вразливостей ланцюга постачання. Ці перевірки часто виявляють комбінації проблем, котрі не помітні у статичному аналізі, і дають змогу перевірити працездатність механізмів захисту у реальних сценаріях.

Перед розгортанням у staging-середовищі проводять повні проміжні прогони, де перевіряють роботу системи в умовах, що максимально наближені до production. Тут відпрацьовуються сценарії відкату, перевіряється коректність конфігурації середовища, налаштувань безпеки та інтеграції з інфраструктурними сервісами (secrets managers, WAF, CDN). Також на цій стадії виконуються перевірки сумісності патчів безпеки і остаточне сканування на вразливості, які могли з’явитися внаслідок змін.

Релізний етап передбачає контрольований rollout, наприклад канарне розгортання, із посиленим моніторингом. У процесі релізу важливо мати налаштовані метрики та алерти, що дозволяють оперативно виявляти аномалії в поведінці застосунку — показники помилок, підвищення часу відповіді, незвичні шаблони трафіку. Налаштовані playbooks для реакції на інциденти забезпечують швидке відновлення і мінімізацію впливу подій безпеки.

Постійний моніторинг і операційна підтримка забезпечують довготривалу стійкість системи. Логи, трасування та кореляція подій у SIEM дозволяють виявляти інциденти, проводити розслідування і валідацію гіпотез щодо атак; RASP та WAF додають додатковий шар захисту у рантаймі. Моніторинг також сприяє виявленню тих вразливостей, що проявляються лише під навантаженням або у взаємодії з реальними користувачами, наприклад витоки інформації через помилки у форматуванні або поведінкові вразливості. На рисунку 3.1 приведена схема алгоритму тестування TypeScript-застосунку.

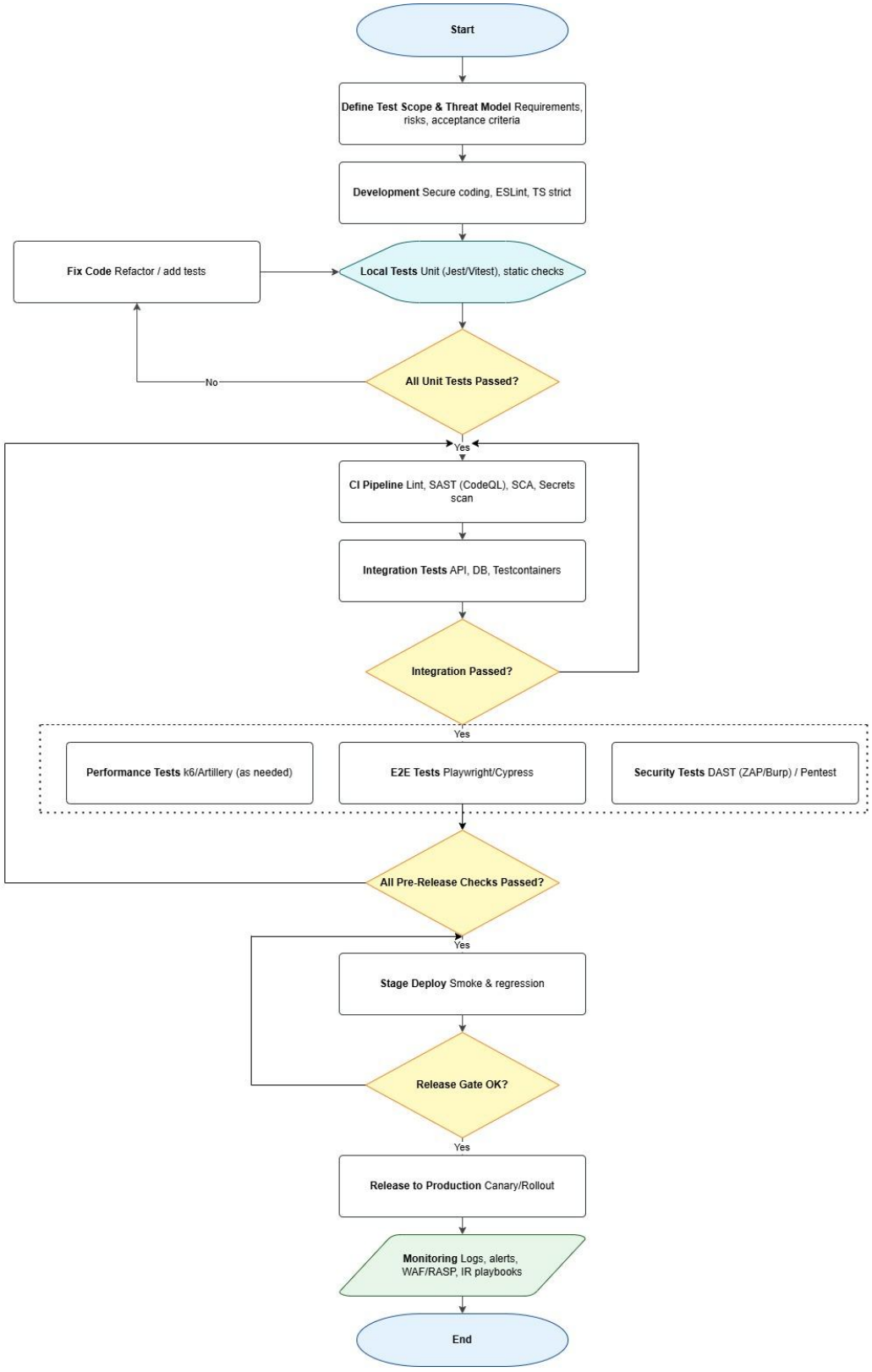

Рисунок 3.1 – Схема алгоритму тестування TypeScript-застосунку

На всіх етапах важливо підтримувати feedback-loop: знайдені вади повинні бути швидко повернені до розробки, задокументовані та закриті з

урахуванням тестів регресії. Такий підхід гарантує, що виправлення не створюють нових проблем і що політики безпеки адаптуються до еволюції коду та бібліотек. В результаті поєднання формального моделювання загроз, інструментальної автоматизації й ретельного ручного аналізу дозволяє системно зменшувати ймовірність експлуатації вразливостей у TypeScript-застосунку.

3.2 Аналіз залежностей та supply-chain

Snyk — це хмарна платформа, створена для інтеграції безпеки безпосередньо в робочий процес розробників. Її ключова особливість полягає у тому, що вона орієнтована не стільки на післяфактум-аудит, скільки на запобігання проблемам ще під час написання коду. Платформа була задумана як інструмент, який допомагає розробникам самостійно виявляти та усувати вразливості у програмному коді, бібліотеках і контейнерах, не потребуючи глибоких знань з кібербезпеки. Її головна мета — зробити безпеку частиною звичного циклу розробки, а не додатковим бар’єром.

Після входу до системи користувач потрапляє на панель керування, де Snyk одразу пропонує підключити свої проєкти або репозиторії з GitHub, GitLab чи інших систем контролю версій. Після цього інструмент автоматично сканує структуру залежностей і виявляє всі відомі вразливості на основі власної бази даних, яка постійно оновлюється та синхронізується з загальновизнаними джерелами, такими як CVE та NVD. Усі знайдені проблеми систематизуються за рівнем критичності, що дозволяє команді концентруватися насамперед на тих, які можуть реально вплинути на безпеку застосунку.

На головному екрані (приклад показано на скріні нижче) розробник бачить короткий огляд стану проєкту: скільки вразливостей знайдено, які бібліотеки їх викликають і які дії можна виконати для усунення. Snyk не лише ідентифікує проблему, але й пропонує спосіб її виправлення — оновлення версії пакета,

заміну залежності або внесення змін у конфігурацію. На рисунку 3.2 приведено головне вікно платформи Snyk з виявленими вразливостями.

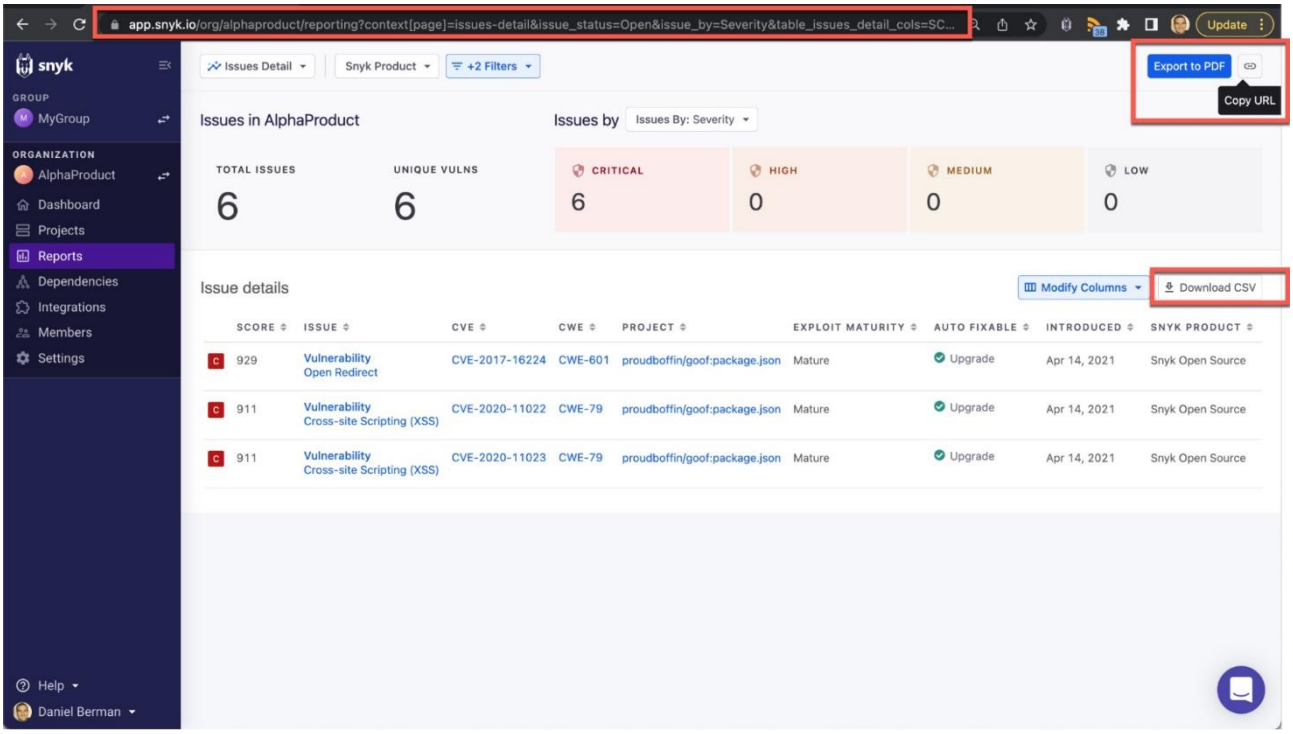

Рисунок 3.2 – Виявлення загроз платформою Snyk

Особливо важливою функцією є глибокий аналіз TypeScript-кодів, адже Snyk не обмежується лише перевіркою залежностей. Його модуль Snyk Code виконує статичний аналіз вихідного коду, виявляючи небезпечні шаблони використання API, некоректну обробку користувацьких даних, невдалу валідацію або ін'єкційні ризики. Такий підхід дозволяє виявити не лише вразливості у сторонніх бібліотеках, а й логічні помилки, що потенційно призводять до XSS, SQL-ін'єкцій або проблем із контролем доступу. У контексті TypeScript це особливо цінно, оскільки багато проблем, не видимих під час компіляції, можуть бути пов'язані з динамічною природою JavaScript-середовища, яке використовується під капотом.

На іншому зображенні можна побачити, як інструмент виводить знайдені проблеми безпосередньо у вигляді дерева залежностей, де поруч з назвою пакета відображено рівень небезпеки, опис вразливості та посилання на відповідний CVE-ідентифікатор. Завдяки цьому розробник розуміє не лише факт наявності проблеми, а й контекст її виникнення, що значно полегшує усунення.

Після первинного сканування Snyk переходить у режим моніторингу. Це означає, що система постійно відстежує оновлення бібліотек і, якщо якась із них отримує нову вразливість, повідомляє розробників автоматично. Таким чином, навіть після релізу продукт не залишається без нагляду. Усі ці сповіщення можна отримувати у вигляді листів, повідомлень у Slack чи як інтегровані нотифікації в CI/CD-консолі.

Окрім статичного аналізу, Snyk активно використовується і як компонент конвеєра CI/CD. Його можна інтегрувати в GitHub Actions, GitLab CI або Jenkins, де він виконує роль автоматичного «охоронця» — кожен коміт чи злиття проходить через перевірку на наявність критичних вразливостей. Якщо інструмент виявляє проблему високого рівня небезпеки, він може заблокувати злиття або створити пул-реквест із готовим виправленням. Така інтеграція дозволяє вбудувати безпеку у сам процес розробки, а не відкладати її на окрему фазу після тестування.

Для прикладу, під час сканування TypeScript-застосунку Snyk може виявити, що один із пакетів має відомий prototype pollution, який дозволяє змінювати глобальні об’єкти через властивість __proto__. У звіті інструмент пояснює, як ця вразливість може бути використана для ескалації привілеїв або обходу авторизації, і водночас пропонує рішення — оновити бібліотеку до конкретної версії, у якій помилку виправлено.

Інтеграція з IDE, наприклад Visual Studio Code, робить цей процес ще зручнішим: Snyk може підсвічувати проблеми прямо в редакторі, як це робить ESLint, але зі специфічним фокусом на безпеку. Це дозволяє розробникам реагувати одразу під час написання коду, а не лише після коміту.

Коли проєкт розгортається у production, Snyk продовжує виконувати роль спостерігача. Він постійно перевіряє базу відомих CVE, співставляє її з уже наявними залежностями й у разі виявлення нових загроз повідомляє про це команду. Таким чином забезпечується безперервний контроль безпеки у всьому життєвому циклі розробки.

На зображенні нижче показано приклад використання Snyk у CI-середовищі — коли інструмент перевіряє код на наявність ризиків ще до того, як

він буде розгорнутий у staging або production. Це гарантує, що будь-яка потенційна вразливість буде виявлена і виправлена до того, як вона потрапить у реальне середовище користувачів.

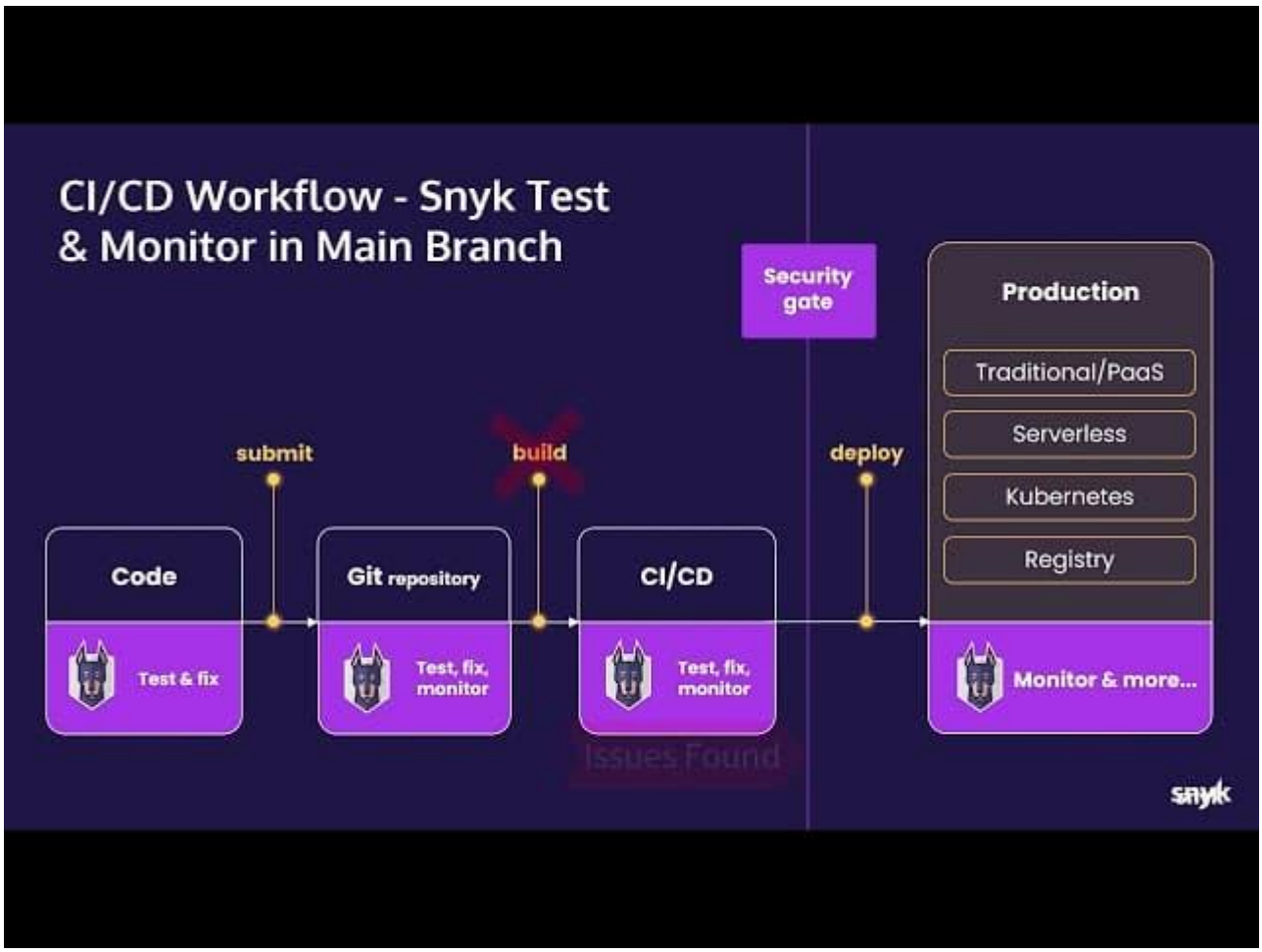

Рисунок 3.3 - Схема використання Snyk у CI-середовищі

У контексті TypeScript-застосунків Snyk має особливу цінність, оскільки більшість сучасних веб-додатків базуються на екосистемі npm, де кількість залежностей може сягати сотень або навіть тисяч. У такому середовищі вручну відстежувати оновлення і зміни у бібліотеках практично неможливо. Snyk автоматизує цей процес, надаючи не лише аналітику, а й конкретні кроки для усунення проблем, що зменшує ризики експлуатації відомих вразливостей і забезпечує більш стабільний розвиток продукту.

Впровадження Snyk у робочий процес допомагає командам перейти від реактивного підходу до безпеки до проактивного, коли ризики усуваються на етапі їхнього виникнення. Завдяки цьому розробка стає не лише швидшою, а й більш контрольованою. Продукт, який проходить перевірку через Snyk, набуває

вищого рівня довіри як з боку розробників, так і з боку кінцевих користувачів, оскільки безпека в ньому не є додатком, а невід'ємною складовою всього процесу створення програмного забезпечення.

3.3 Вбудований інструмент безпеки в екосистемі Node.js

Середовище Node.js і його менеджер пакетів npm створили екосистему, у якій переважна більшість TypeScript-застосунків ґрунтується на відкритих бібліотеках. Кожна з них може мати власні залежності, утворюючи складне дерево з тисяч компонентів. Коли один із таких компонентів виявляється уразливим, ризик автоматично поширюється на всі застосунки, що його використовують. Саме тому команда npm створила вбудований механізм безпеки — npm audit, який дозволяє швидко оцінити стан безпеки проєкту, порівнюючи всі встановлені пакети з базою відомих вразливостей.

Інструмент npm audit не потребує окремої інсталяції — він інтегрований безпосередньо в клієнт npm і викликається звичайною командою npm audit у терміналі. У цей момент система аналізує файл package-lock.json, де зафіксовано всі конкретні версії залежностей, та надсилає запит до централізованої бази даних npm Advisory. Ця база містить тисячі записів про вразливості, кожен з яких має ідентифікатор, опис проблеми, рівень критичності, посилання на CVE та вказівку, з якої версії пакета помилка була усунена. Після отримання відповіді інструмент формує короткий звіт просто у консолі.

Типовий результат виглядає як таблиця, де кожен рядок відповідає окремій вразливості. Для кожної з них зазначається рівень небезпеки — низький, середній, високий або критичний — а також назва пакета, шлях у дереві залежностей і рекомендація щодо оновлення. Наприклад, у проєкті може бути виявлено, що використовується версія бібліотеки minimist до 1.2.6, яка має відому уразливість до Prototype Pollution. У звіті буде вказано, що потрібно оновити пакет до версії 1.2.7 або новішої.

Команда npm audit дає можливість не лише перевірити залежності, а й одразу застосувати виправлення. Запуск команди npm audit fix автоматично оновлює всі пакети, для яких існують безпечні версії, не порушуючи сумісності. Якщо ж уразливість не можна усунути без потенційного ризику, інструмент виводить повідомлення з порадою переглянути залежності вручну. Таким чином розробник одразу бачить, які проблеми можна виправити безболісно, а які вимагають детальнішого аналізу.

У TypeScript-проєктах цей підхід особливо зручний, оскільки більшість збірних систем — наприклад, Webpack, Vite чи NestJS — мають довгі ланцюги залежностей. Використання npm audit дозволяє контролювати їх без додаткових сторонніх інструментів. Це стає першою лінією оборони: кожного разу, коли команда виконує npm install, автоматично перевіряється, чи не додається до проєкту вразливий пакет. У нових версіях npm ця перевірка навіть відбувається автоматично після інсталяції залежностей — користувач отримує попередження, якщо встановлений модуль має відомі проблеми безпеки.

Після встановлення залежностей виконується команда

npm audit

У відповідь система проводить сканування проєкту, звіряє всі пакети з базою npm Advisory і повертає звіт. Типовий результат у консолі виглядає так:

 # npm audit report

minimist <1.2.7

Severity: high

Prototype Pollution - https://github.com/advisories/GHSA-7fhm-mqm4-2wp7

fix available via `npm audit fix`

node_modules/minimist

mkdirp <=0.5.5

Depends on vulnerable versions of minimist

node_modules/mkdirp

2 high severity vulnerabilities

To address all issues, run:

npm audit fix

Цей звіт означає, що у проєкті виявлено дві вразливості високого рівня у пакеті minimist. npm audit пояснює, що проблема стосується Prototype Pollution — класичної вразливості, через яку зловмисник може змінювати властивості глобальних об’єктів JavaScript. Далі система вказує шлях, яким ця бібліотека потрапила до проєкту: її використовує пакет mkdirp. Наприкінці звіту наведено рекомендацію — виконати команду npm audit fix, щоб автоматично оновити minimist до безпечної версії.

Після застосування цієї команди npm переглядає усі залежності й оновлює лише ті пакети, де це не порушує сумісність. Якщо проблема вимагає мажорного оновлення (що може вплинути на API), npm попереджає про це і пропонує розробнику розв’язати питання вручну. Повторне виконання npm audit показує, що вразливість усунено, а звіт закінчується повідомленням:

found 0 vulnerabilities

Крім простого текстового звіту, npm audit підтримує детальний формат JSON. Виконання команди

npm audit --json > audit-report.json

зберігає повний список проблем з усіма полями — рівнем серйозності, шляхом у дереві залежностей, описом, порадами з оновлення та посиланнями на ресурси. Цей файл можна використовувати у власних системах звітності або інтегрувати у CI/CD для автоматичного аналізу.

У процесі безпечної розробки TypeScript-додатків npm audit найчастіше інтегрують у CI як перевірку залежностей перед збіркою або деплоєм. У GitHub Actions це робиться за допомогою простого кроку:

- name: Security audit

run: npm audit --audit-level=high

Якщо під час сканування виявлено вразливість з рівнем небезпеки «high» або «critical», збірка завершується з помилкою, і розробник не може змерджити код, поки не буде виконано оновлення. Таким чином реалізується принцип *shift-left security*, коли проблеми безпеки усуваються до того, як код потрапить до production.

У більш складних пайплайнах результати npm audit об’єднують із іншими перевірками — наприклад, з ESLint і TypeScript-компіляцією. У такому сценарії пайплайн послідовно виконує перевірку якості коду, статичний аналіз і аудит залежностей, що гарантує комплексну оцінку безпеки ще на етапі CI.

Для великих команд доцільно періодично виконувати повний аудит у нічних збірках. Наприклад, раз на добу запускається npm audit --json, результати експортуються у централізовану систему моніторингу (наприклад, Grafana або Kibana), де можна відстежувати динаміку вразливостей за часом. Це дозволяє швидко реагувати на появу нових проблем у будь-якому проєкті компанії.

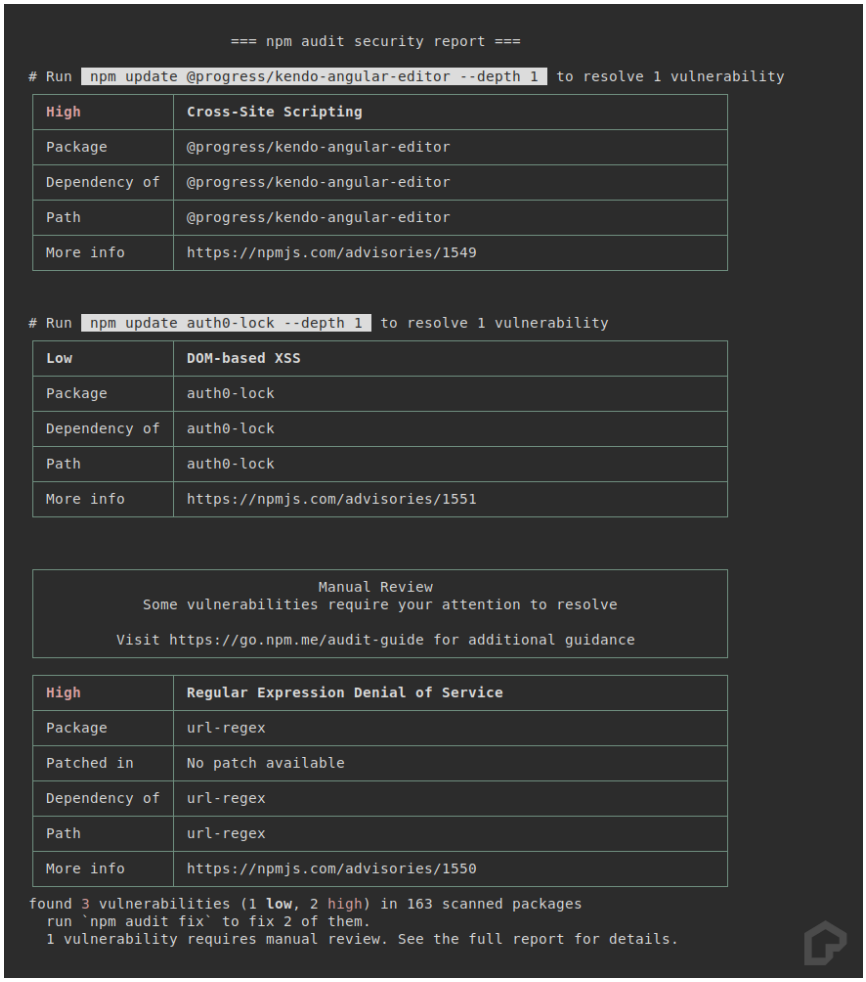

Рисунок 3.4 – Звіт про виконання npm audit

Для проєктів на TypeScript використання npm audit має особливе значення. Цей інструмент не вимагає додаткових ресурсів, працює швидко й автоматично інтегрований у весь ланцюг npm-пакетів. Оскільки більшість TypeScript-додатків

мають велику кількість транзитивних залежностей, навіть одна вразлива бібліотека може стати точкою компрометації. npm audit дозволяє запобігти таким ситуаціям, роблячи аудит частиною звичного робочого циклу.

У поєднанні зі сторонніми інструментами, такими як Snyk або OWASP Dependency-Check, npm audit забезпечує базовий, але систематичний рівень контролю. Він не замінює повноцінний аналіз коду, проте ідеально підходить для щоденного сканування залежностей. У TypeScript-середовищі, де кожен новий пакет може впливати на стабільність і безпеку системи, саме цей механізм часто стає першим запобіжним бар'єром на шляху до експлуатації вразливостей.

Для більш масштабних проєктів npm audit стає основою безпекового етапу CI/CD. Його часто включають у пайплайн перед розгортанням або перед злиттям гілки. Наприклад, у GitHub Actions чи GitLab CI створюють крок із командою *npm audit --audit-level=high*, яка завершує збірку з помилкою, якщо знайдено критичні уразливості. Це дисциплінує команду: код не переходить у production, доки всі пакети не оновлені до безпечних версій.

У практиці розробки можна навести типовий сценарій: під час перевірки бекенду на TypeScript інструмент виявив, що бібліотека express використовує в ланцюгу залежностей стару версію qs, у якій була вразливість до Denial of Service через нескінченні вкладені об'єкти. Розробник запускає npm audit fix, система оновлює пакет, і після повторного сканування звіт показує, що вразливість усунено. Таким чином процес не потребує ні додаткових налаштувань, ні зовнішніх сервісів — все відбувається всередині звичної екосистеми Node.js.

У звітах npm audit вказано не лише конкретну проблему, а й контекст, наприклад:

found 2 vulnerabilities (1 high, 1 critical)

run `npm audit fix` to fix them, or `npm audit` for details

Після виконання команди npm audit --json можна отримати деталізований звіт у форматі JSON для подальшої інтеграції з іншими системами моніторингу чи аналітики. Це зручно для побудови власних дашбордів безпеки або для автоматизованої звітності в CI.

Попри свою ефективність, npm audit має певні обмеження. Його база даних охоплює лише відомі публічні вразливості й не проводить статичного аналізу самого коду. Тобто він не виявить логічних помилок чи неправильних перевірок у TypeScript-файлах — для цього потрібні SAST-інструменти, такі як Snyk чи Semgrep. Проте npm audit ідеально підходить для швидкої діагностики стану залежностей, а завдяки своїй інтегрованості в npm його можна застосовувати навіть на ранніх етапах розробки.

Коли цей інструмент використовується систематично, він формує здорову практику контролю безпеки у команді. Розробники починають сприймати оновлення бібліотек не як «несвоєчасне втручання», а як природну частину циклу підтримки. Це знижує технічний борг і запобігає накопиченню застарілих залежностей, які часто стають основною причиною компрометацій.

У комплексі з більш глибокими рішеннями, такими як Snyk або OWASP Dependency-Check, npm audit відіграє роль першої, оперативної лінії оборони, забезпечуючи щоденний контроль за станом пакунків. Для TypeScript-додатків, що постійно розвиваються і мають значну кількість бібліотек, його використання не просто бажане, а необхідне. Це той базовий інструмент, який дозволяє впровадити принцип *security by default* у щоденний процес розробки — коли безпечне середовище є не додатковою метою, а стандартом роботи команди.

ВИСНОВКИ

У результаті дослідження встановлено, що сучасні підходи до тестування веб-додатків ґрунтуються на поєднанні автоматизованих інструментів, методів статичного та динамічного аналізу, а також інтеграції тестування у весь життєвий цикл розробки, що дозволяє своєчасно виявляти помилки та зменшувати ризики експлуатації вразливостей.

Безпека веб-систем визначається не окремими механізмами, а комплексною взаємодією політик, архітектурних рішень і процедур контролю доступу, що формують стійкість системи до типових кіберзагроз.

Досліджено вразливість Prototype Pollution, котра є однією з найнебезпечніших вразливостей JavaScript-середовища, оскільки дозволяє змінювати поведінку програми через модифікацію глобальних прототипів, що може призводити до ескалації прав, обходу логіки та компрометації даних.

Експериментальні результати засвідчили, що ефективним механізмом захисту є перевірка власних властивостей об’єктів, блокування небезпечних ключів та обмеження доступу до прототипів, що унеможливлює повторну експлуатацію аналогічної вразливості.

Розроблений алгоритм тестувння застосунків TypeScript довів ефективність інтеграції тестування безпеки у всі етапи розробки, забезпечивши послідовну перевірку коду, залежностей, конфігурацій та поведінки застосунку, що суттєво зменшує ймовірність появи критичних помилок.

У ході дослідження підтверджено, що найбільші ризики безпеки походять саме від сторонніх та транзитивних залежностей, тому регулярний SCA-аналіз і контроль ланцюга постачання є ключовою передумовою захисту сучасних TypeScript-додатків.

Аналіз інструменту npm audit показав, що він є ефективним первинним засобом виявлення вразливих пакетів і забезпечує оперативний контроль безпеки залежностей, хоча потребує поєднання з розширеними SCA-рішеннями для повноцінного захисту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. OWASP Foundation. OWASP Top 10: 2021. – Режим доступу: https://owasp.org/Top10/ (дата звернення: 12.12.2025).
2. OWASP Foundation. Web Security Testing Guide v5.0. – Режим доступу: https://owasp.org/www-project-web-security-testing-guide/ (дата звернення: 12.12.2025).
3. Microsoft. TypeScript Documentation. – Режим доступу: https://www.typescriptlang.org/docs/ (дата звернення: 12.12.2025).
4. Node.js Foundation. npm Documentation. – Режим доступу: https://docs.npmjs.com/ (дата звернення: 12.12.2025).
5. NPM Audit. Security auditing for npm packages. – Режим доступу: https://docs.npmjs.com/cli/v9/commands/npm-audit (дата звернення: 12.12.2025).
6. Snyk Ltd. Snyk Open Source Security Platform. – Режим доступу: https://snyk.io/ (дата звернення: 12.12.2025).
7. GitHub Security Advisory. Prototype Pollution in @clickbar/dot-diver (GHSA-9w5f-mw3p-pj47). – Режим доступу: https://github.com/clickbar/dot-diver/security/advisories/GHSA-9w5f-mw3p-pj47 (дата звернення: 12.12.2025).
8. National Vulnerability Database (NVD). CVE-2023-45827: dot-diver Prototype Pollution vulnerability. – Режим доступу: https://nvd.nist.gov/vuln/detail/CVE-2023-45827 (дата звернення: 12.12.2025).
9. Snyk Vulnerability DB. CVE-2023-45827 Prototype Pollution in @clickbar/dot-diver. – Режим доступу: https://security.snyk.io/vuln/SNYK-JS-CLICKBARDOTDIVER-6046429 (дата звернення: 12.12.2025).
10. Acunetix / Invicti. Vulnerability description CVE-2023-45827. – Режим доступу: https://www.acunetix.com/vulnerabilities/ (дата звернення: 12.12.2025).
11. MDN Web Docs. JavaScript Prototype Inheritance. – Режим доступу: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain (дата звернення: 12.12.2025).

12. OWASP Cheat Sheet Series. JavaScript Security Cheat Sheet. – Режим доступу: https://cheatsheetseries.owasp.org/ (дата звернення: 12.12.2025).

13. INCIBE-CERT. Advisory on CVE-2023-45827. – Режим доступу: https://www.incibe.es/ (дата звернення: 12.12.2025).

14. ECMA International. ECMAScript Language Specification. – Режим доступу: https://tc39.es/ecma262/ (дата звернення: 12.12.2025).

15. GitHub Security Advisory. Prototype Pollution (PP) vulnerability in setByPath (GHSA-9w5f-mw3p-pj47) [Електронний ресурс] // GitHub – clickbar/dot-diver. – Опубл.: 02.11.2023. – Режим доступу: https://github.com/clickbar/dot-diver/security/advisories/GHSA-9w5f-mw3p-pj47

16. (дата звернення: 10.11.2025).

17. National Vulnerability Database (NVD). CVE-2023-45827: dot-diver – Prototype Pollution у setByPath; виправлення у релізі 1.0.2 [Електронний ресурс]. – Опубл.: 06.11.2023. – Режим доступу: https://nvd.nist.gov/vuln/detail/CVE-2023-45827 (дата звернення: 10.11.2025).

18. Snyk Vulnerability DB. Prototype Pollution in @clickbar/dot-diver | CVE-2023-45827 [Електронний ресурс]. – Опубл.: 05.11.2023. – Режим доступу: https://security.snyk.io/vuln/SNYK-JS-CLICKBARDOTDIVER-6046429

19. (дата звернення: 10.11.2025).

20. Acunetix (Invicti) SCA. CVE-2023-45827 vulnerability in npm package @clickbar/dot-diver [Електронний ресурс]. – Режим доступу: https://www.acunetix.com/vulnerabilities/sca/cve-2023-45827-vulnerability-in-npm-package-clickbar-dot-diver/ (дата звернення: 10.11.2025).

21. INCIBE-CERT (ES). CVE-2023-45827: Prototype Pollution у dot-diver; рекомендація оновлення до 1.0.2 [Електронний ресурс]. – Режим доступу: https://www.incibe.es/en/incibe-cert/early-warning/vulnerabilities/cve-2023-45827 (дата звернення: 10.11.2025).

22. CVE Details. Clickbar – Dot-diver: статистика вразливостей (включно з CVE-2023-45827) [Електронний ресурс]. – Режим доступу: https://www.cvedetails.com/product/164402/Clickbar-Dot-diver.html (дата звернення: 10.11.2025).

23. Clickbar/dot-diver – CHANGELOG. Запис про включення виправлення (commit 98daf567) та посилання на GHSA [Електронний ресурс]. – Режим доступу: https://raw.githubusercontent.com/clickbar/dot-diver/master/CHANGELOG.md (дата звернення: 10.11.2025).

24. Sploitus (агрегатор PoC). Exploit for Prototype Pollution in Clickbar Dot-Diver – CVE-2023-45827 / GHSA-9w5f-mw3p-pj47 [Електронний ресурс]. – Режим доступу: https://sploitus.com/exploit?id=D9459CD0-82CF-5A84-AB9F-477EF6AE249A (дата звернення: 10.11.2025).

25. npm Registry. Сторінка пакета @clickbar/dot-diver: відомості про версії та публікації (для звірки безпечної версії ≥1.0.2) [Електронний ресурс]. – Режим доступу: https://www.npmjs.com/package/%40clickbar%2Fdot-diver (дата звернення: 10.11.2025).

Додаток А

Набір Unit тестів для перевірки вразливості бібліотеки dot-diver

setByPathSafe.spec.ts

```
import { describe, it, beforeEach, afterEach, expect } from 'vitest';
// Варіант 1: тестуємо власну (патчену) реалізацію
import { setByPathSafe, getByPath } from '../setByPathSafe';

// Варіант 2: якщо тестуєте безпосередньо патчену функцію з бібліотеки
// import { setByPath, getByPath } from '@clickbar/dot-diver';
// і замініть усюди setByPathSafe -> setByPath

/**
 * Хелпер: перевіряє ознаки prototype pollution.
 * Якщо глобальний прототип змінено, у порожнього літерала {} з’являється властивість.
 */
function hasPolluted(prop: string) {
  // Використовуємо оператор in, щоб виявити властивість через прототип
  return prop in ({} as any);
}

// Для ізоляції: у разі, якщо якась реалізація таки записала в Object.prototype,
// чистимо слід після кожного тесту.
const POLLUTION_FLAG = '__pp_test_flag__';

afterEach(() => {
  // Прибрати побічні ефекти від прототипного забруднення
  // eslint-disable-next-line @typescript-eslint/no-explicit-any
  delete (Object.prototype as any)[POLLUTION_FLAG];
  // Аналогічно для загальновживаних імен
  // eslint-disable-next-line @typescript-eslint/no-explicit-any
  delete (Object.prototype as any).isAdmin;
});

describe('setByPathSafe (baseline behavior)', () => {
```

```
    it('writes a simple nested value (safe path)', () => {
      const obj: Record<string, unknown> = {};
      setByPathSafe(obj, 'a.b.c', 123);

      expect((obj as any).a).toBeTruthy();
      expect((obj as any).a.b).toBeTruthy();
      expect((obj as any).a.b.c).toBe(123);
      expect(hasPolluted('c')).toBe(false);
    });

    it('creates intermediate objects safely (without polluting prototypes)', () => {
      const obj: any = { a: 1 };
      setByPathSafe(obj, 'a2.b.c', 'ok');

      expect(obj.a).toBe(1);
      expect(obj.a2).toBeTruthy();
      expect(obj.a2.b.c).toBe('ok');
      expect(Object.getPrototypeOf(obj.a2)).toBe(Object.getPrototypeOf({})); // або
Object.create(null) — залежно від вашої політики
    });
  });

  describe('setByPathSafe (forbidden segments)', () => {
    it('rejects "__proto__" in path', () => {
      const obj: Record<string, unknown> = {};
      expect(() => setByPathSafe(obj, `__proto__.${POLLUTION_FLAG}`, true)).toThrow();

      // не має з’явитись у {} через прототип
      expect(hasPolluted(POLLUTION_FLAG)).toBe(false);
    });

    it('rejects "constructor" in path', () => {
      const obj: Record<string, unknown> = {};
      expect(() =>
        setByPathSafe(obj, `constructor.prototype.${POLLUTION_FLAG}`, true)
      ).toThrow();
```

```
    expect(hasPolluted(POLLUTION_FLAG)).toBe(false);
  });

  it('rejects "prototype" in path (in any segment)', () => {
    const obj: Record<string, unknown> = {};
    expect(() =>
      setByPathSafe(obj, `user.prototype.${POLLUTION_FLAG}`, true)
    ).toThrow();
    expect(hasPolluted(POLLUTION_FLAG)).toBe(false);
  });
});

describe('setByPathSafe (own-property policy)', () => {
  it('throws if final key is not an own property when policy requires own property', () => {
    // Цей тест відповідає політиці: дозволяється встановлення ТІЛЬКИ існуючих власних властивостей.
    // Якщо ви дозволяєте створення нових ключів – приберіть або змініть цей тест.
    const obj: any = { user: { name: 'alice' } };
    // спроба створити нову властивість user.admin (якої немає)
    expect(() => setByPathSafe(obj, 'user.admin', true)).toThrow();

    // але наявну власну властивість – дозволяє змінити
    expect(() => setByPathSafe(obj, 'user.name', 'bob')).not.toThrow();
    expect(obj.user.name).toBe('bob');
  });
});

describe('setByPathSafe (resilience)', () => {
  it('throws on non-string or empty path', () => {
    const obj: Record<string, unknown> = {};
    // @ts-expect-error — перевірка захисту від некоректного типу
    expect(() => setByPathSafe(obj, null, 1)).toThrow();
    // Порожній шлях також має бути заборонений
    expect(() => setByPathSafe(obj, '', 1)).toThrow();
  });
```

```
    it('handles intermediate non-objects by replacing/creating safe containers (policy-dependent)', () => {
      const obj: any = { a: 42 };
      // Залежно від політики — або створюємо безпечний контейнер, або кидаємо помилку.
      try {
        setByPathSafe(obj, 'a.b', 'x');
        expect(typeof obj.a).toBe('object');
        expect(obj.a.b).toBe('x');
      } catch {
        // Якщо реалізація воліє кидати помилку — це теж прийнятно, але перевіримо, що немає pollution
        expect(hasPolluted('b')).toBe(false);
      }
    });
  });

  describe('getByPath + setByPathSafe integration (optional)', () => {
    it('getByPath returns the value set by setByPathSafe', () => {
      const obj: any = {};
      setByPathSafe(obj, 'x.y', 777);
      expect(getByPath ? getByPath(obj, 'x.y') : obj.x.y).toBe(777);
    });
  });



  // setByPathSafe.ts
  export function isForbiddenKey(key: string): boolean {
    return (
      key === '__proto__' ||
      key === 'prototype' ||
      key === 'constructor' ||
      key.includes('prototype')
    );
  }
```

```
export function assertValidPath(path: unknown): asserts path is string {
  if (typeof path !== 'string' || path.trim() === '') {
    throw new Error('Invalid path');
  }
}

export function setByPathSafe(obj: any, path: string, value: unknown) {
  assertValidPath(path);

  const parts = path.split('.');
  let cur = obj;

  for (let i = 0; i < parts.length; i++) {
    const key = parts[i];

    if (isForbiddenKey(key)) {
      throw new Error(`Forbidden path segment: ${key}`);
    }

    const isLast = i === parts.length - 1;

    if (isLast) {
      // Політика «власної властивості»: змінюємо лише якщо поле вже існує на самому об’єкті
      if (!Object.prototype.hasOwnProperty.call(cur, key)) {
        throw new Error(`Refusing to set non-own property: ${key}`);
      }
      cur[key] = value;
      return;
    }

    // Рухаємось углиб: якщо немає об’єкта — створюємо безпечний контейнер
    if (cur[key] == null || typeof cur[key] !== 'object') {
      // Можна обрати Object.create(null) або {}
      cur[key] = {};
    }
```

```
    cur = cur[key];
  }
}

export function getByPath(obj: any, path: string) {
  assertValidPath(path);
  return path.split('.').reduce((acc, seg) => (acc == null ? acc : acc[seg]), obj);
}
```

Додаток Б

Копії публікацій




ЗАХІДНОУКРАЇНСЬКИЙНАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КІБЕРБЕЗПЕКИ
ГРОМАДСЬКА ОРГАНІАЦІЯ «КІБЕРБЕЗПЕКА І АВТОМАТИЗАЦІЯ»

Матеріали
науково-практичного симпозіуму
"ЗАХИСТ'ІНФОРМАЦІЇ'2025"

28 листопада 2025
Тернопіль

Збірник матеріалів науково-практичного симпозіуму «Захист інформації'2025», Тернопіль, 2025. – 118с.

Адреса редакції:
Громадська організація «Кібербезпека і автоматизація»
м. Тернопіль
Контактний телефон: (066)043-42-10
e-mail: conferencekb@gmail.com

УДК 004.056.5

Владислав РУЩАК

Західноукраїнський національний університет

ПОРІВНЯННЯ FLOW ТА TYPESCRIPT В JAVASCRIPT

Вступ. Екосистема JavaScript за останні роки перетворилася з простої мови для браузерів на повноцінне середовище для розробки складних веб– і серверних застосунків. Одним із ключових викликів при цьому стала відсутність статичної типізації – механізму, що дозволяє виявляти помилки ще до запуску коду. Відповіддю на цю потребу стали два підходи до типізації JavaScript: Flow, запропонований Facebook (Meta), і TypeScript від Microsoft.

Обидва інструменти прагнули розв'язати одну й ту саму проблему – зробити JavaScript безпечнішим та передбачуванішим, зберігаючи його гнучкість. Обидва реалізують статичний аналіз типів, інтегруються в процес розробки та транспілюють код у стандартний JavaScript, сумісний із будь–яким браузером, навіть такими застарілими, як Internet Explorer 8. Попри схожість у підходах, ці системи розвивалися у різних напрямах і сьогодні займають нерівні позиції на ринку. У цьому контексті порівняння Flow та TypeScript не лише виявляє технічні відмінності, але й дозволяє зрозуміти, чому одна з технологій набула широкої підтримки, а інша поступово втратила актуальність.

Мета: Проаналізувати та порівняти можливості застосування Typescript та flow.

1. Безпека типів в JavaScript

Як і в більшості скриптових (інтерпретованих) мов програмування, у JavaScript можна написати код, що не працює. Якщо виконання цього коду не ініціюється браузером, помилка не виникає – відсутні будь–які повідомлення або попередження. З одного боку, це може бути перевагою: навіть якщо на великому вебсайті в обробнику натискання кнопки присутня синтаксична помилка, це не завадить повному завантаженню сторінки для користувача.

Водночас така поведінка є потенційно небезпечною. Наявність коду, що не працює, негативно впливає на якість проєкту. Щоб уникнути таких ситуацій, ще до розгортання сайту бажано перевірити всі скрипти на відсутність помилок – принаймні синтаксичних. У ідеалі слід також впевнитися, що код працює коректно. Для цього використовуються різноманітні інструменти – зокрема, такі як npm, webpack, babel або tsc, а також засоби для тестування (karma, jsdom, mocha, chai тощо).

У теоретично ідеальному середовищі всі скрипти, включно з однорядковими, покриті автоматизованими тестами. У реальності це рідкість, і значна частина коду залишається без тестового покриття. Для такої частини необхідно застосовувати автоматизовані засоби перевірки, які дозволяють:

Перевірити коректність синтаксису JavaScript. Інструменти аналізу переконуються, що текст програми може бути розпізнаний інтерпретатором: що всі дужки закриті, рядки – правильно обмежені лапками тощо. Таку перевірку зазвичай виконують збирачі, транспайлери, мініфікатори та обфускатори.

Перевірити правильність використання семантики мови. Наприклад, у коді:

```
var x = null;
x.foo();
```

синтаксис формально правильний, але семантично код некоректний – спроба викликати метод у null призведе до помилки під час виконання.

Окрім семантичних, існують і логічні помилки – найнебезпечніший тип, адже програма виконується без збоїв, але результат не відповідає очікуванням. Класичний приклад:

```
console.log(input.value);    // "1"
console.log(input.value + 1); // "11"
```

Інструменти статичного аналізу коду, як–от ESLint, здатні виявити значну кількість потенційних помилок, наприклад:

– некоректні умови завершення циклу for;
– заборона використання async–функцій як аргументів конструктора Promise;
– присвоєння в умовах;
– дублікати ключів у літералах об'єктів;
– та інші подібні ситуації.

Загалом, усі подібні правила є свого роду обмеженнями, які лінтер накладає на розробника. Вони зменшують гнучкість JavaScript, аби знизити ризик помилок. Наприклад, за активованих правил не допускається присвоєння в умовах, хоча стандарт мови це дозволяє. У деяких конфігураціях забороняється навіть використання console.log().

Додавання типів змінних і перевірка викликів з урахуванням типів – це ще один рівень обмежень, покликаний зменшити кількість можливих помилок у коді JavaScript.

```
//  - A place to
//  - A sandbox t
const anExampleVa
console.log( 5 * "abc" )
```

The right-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type. (2363)

Peek Problem (Alt+F8) No quick fixes available

Спроба помножити число на рядок

```
const a = {
    key: "value
}
console.log( a.otherKey );
```

any

Property 'otherKey' does not exist on type '{ key: string; }'. (2339)

Peek Problem (Alt+F8) No quick fixes available

Спроба звернутися до неіснуючої (неописаної у типі) властивості об'єкта

```
function foo( bar : number ) {}
console.log( foo( "abc" ) );
```

Argument of type 'string' is not assignable to parameter of type 'number'. (2345)

Peek Problem (Alt+F8) No quick fixes available

Спроба викликати функцію з незбігаючим типом аргументу. Якщо ми напишемо цей код без засобів перевірки типів, код успішно траспилюється. Ніякі засоби статичного аналізу коду, якщо вони не використовують (явно чи неявно) інформацію про типи об'єктів, не зможуть знайти ці помилки.

Тобто додавання типізації JavaScript додає додаткові обмеження на код, який пише програміст, зате дозволяє знайти такі помилки, які в іншому випадку

трапилися б під час виконання скрипта (тобто швидше за все у користувача в браузері). Обидва популярні рушії типізації для JavaScript – TypeScript (Microsoft) і Flow (Meta / Facebook) – надають приблизно однакові можливості. Водночас існує важлива концептуальна відмінність від мов із суворою класичною типізацією, таких як Java чи C++. У JavaScript всі типи фактично є інтерфейсами – тобто описами набору властивостей (разом із їх типами й аргументами функцій). Якщо два інтерфейси сумісні за складом властивостей, вони вважаються взаємозамінними. В таблиці 2 приведено можливості типізації JavaScript. Наприклад, наведений нижче код є абсолютно коректним у TypeScript, хоча був би некоректним у Java чи C++:

```
type MyTypeA = { foo: string; bar: number; };
type MyTypeB = { foo: string; };
function myFunction(arg: MyTypeB): string {
  return `Hello, ${arg.foo}!`;}
const myVar: MyTypeA = { foo: "World", bar: 42 };
console.log(myFunction(myVar)); // "Hello, World!"
```

Цей приклад демонструє структурну типізацію: достатньо, щоб тип аргументу myVar мав усі обов'язкові поля типу MyTypeB – наявність додаткових полів не є проблемою. Код також можна скоротити, використавши літеральний об’єкт без явного зазначення типу:

```
type MyTypeB = { foo: string };
function myFunction(arg: MyTypeB): string {
  return `Hello, ${arg.foo}!`;}
const myVar = { foo: "World", bar: 42 };
console.log(myFunction(myVar)); // "Hello, World!"
```

Таблиця 1 – Можливості типізації JavaScript

	Flow	TypeScript

Можливість задати тип змінної, аргументу або тип значення функції, що повертається	`a : number = 5;`
	`function foo( bar : string) : void {`
	`  /*...*/`
	`}`	

Можливість описати свій тип об'єкта (інтерфейс)	`type MyType {`
	`  foo: string,`
	`  bar: number`
	`}`	

Обмеження допустимих значень для типу	`type Suit = "Diamonds" | "Clubs" | "Hearts" | "Spades";`	

Окремий type–level extension для перерахувань		`enum Direction { Up, Down, Left, Right }`
«Складання» типів	`type MyType = TypeA & TypeB;`	
Додаткові типи для складних випадків	`$Keys<T>, $Values<T>, $ReadOnly<T>, $Exact<T>, $Diff<A, B>, $Rest<A, B>, $PropertyType<T, k>, $ElementType<T, K>, $NonMaybeType<T>, $ObjMap<T, F>, $ObjMapi<T, F>, $TupleMap<T, F>, $Call<F, T...>, Class<T>, $Shape<T>, $Exports<T>, $Supertype<T>, $Subtype<T>, Existential Type (*)`	`Partial<T>, Required<T>, Readonly<T>, Record<K,T>, Pick<T, K>, Omit<T, K>, Exclude<T, U>, Extract<T, U>, NonNullable<T>, Parameters<T>, ConstructorParameters<T>, ReturnType<T>, InstanceType<T>, ThisParameterType<T>, OmitThisParameter<T>, ThisType<T>`

У цьому випадку інтерфейс об'єкта myVar – { foo: string, bar: number } – є надмножиною MyTypeB, тому сумісність зберігається. Такий підхід спрощує взаємодію з бібліотеками, зовнішнім кодом та різними API. Часто це дає змогу передавати об'єкти з потрібною структурою без необхідності явно імпортувати відповідні типи:

```
// У бібліотеці interface OptionsType {
        optionA?: string;
        optionB?: number;
}export function libFunction(arg: number, options = {} as OptionsType) {
         /* ... */}// У коді користувача
        import { libFunction } from "lib";
        libFunction(42, { optionA: "someValue" });
```

У цьому прикладі інтерфейс OptionsType не імпортовано в код користувача, але система типізації автоматично перевіряє сумісність об'єкта з очікуваною структурою. У класичних мовах програмування, таких як Java, така поведінка призводить до помилок компіляції. TypeScript та Flow не підтримуються браузерами напряму. Як і багато нових можливостей JavaScript, типи потребують попереднього перетворення – транспіляції. Це процес, під час якого код з нестандартними можливостями перетворюється на звичайний JavaScript, який розуміють браузери. Що стосується типів – усі описи інтерфейсів, анотації типів та інші елементи типізації просто видаляються.

Наприклад, наведений TypeScript–код:

```
type MyTypeA = { foo: string; bar: number };
type MyTypeB = { foo: string };
function myFunction(arg: MyTypeB): string {
  return `Hello, ${arg.foo}!`;}
const myVar: MyTypeA = { foo: "World", bar: 42 };
console.log(myFunction(myVar));
```

після транспіляції перетворюється на:

```
function myFunction(arg) {
  return `Hello, ${arg.foo}!`;} const myVar = { foo: "World", bar: 42 };
console.log(myFunction(myVar)); // "Hello, World!"
```

Тобто вся інформація про типи видаляється ще до виконання коду браузером. Саме ця особливість дозволяє використовувати нові синтаксичні можливості мови без втрати сумісності. Для перетворення коду, що містить типи, використовуються різні інструменти:

Flow – з Babel використовується плагін @babel/plugin–transform–flow–strip–types, який видаляє типи Flow. TypeScript: можна використовувати Babel з плагіном @babel/plugin–transform–typescript, який також видаляє анотації типів; або застосовувати офіційний транспілятор tsc від Microsoft – окрему утиліту, яка може виконувати як видалення типів, так і компіляцію проекту. Таким чином, використання типів у JavaScript (через TypeScript або Flow) – це не про виконання, а про розробку: вони дозволяють виявляти помилки ще до запуску коду, значно підвищуючи якість програмного забезпечення без шкоди для сумісності з браузерами.

Висновок. TypeScript і Flow мають схожі можливості для додавання статичної типізації в JavaScript, проте TypeScript отримав значно ширшу підтримку спільноти та індустрії. Його екосистема активно розвивається, інтегрується з популярними редакторами коду та підтримується більшістю сучасних фреймворків. Flow, хоча й був технічно потужним, поступово втратив актуальність через меншу гнучкість у конфігурації та слабшу підтримку сторонніх бібліотек. У реальних проектах TypeScript демонструє кращу сумісність і простоту використання, що робить його переважним вибором для більшості команд.

Перелік використаних джерел.

1. S. Kushwah, C. Bansal, S. Rathore, V. Kate, D. Bargal and D. Vishwakarma, "TypeScript: An Open–Source Programming Language with Options for Robust Development and Large–Scale Applications," 2024 International Conference on Advances in Computing Research on Science Engineering and Technology (ACROSET), Indore, India, 2024, pp. 1–5, doi: 10.1109/ACROSET62108.2024.10743426.

2. Carl Rippon, ASP.NET Core 5 and React: Full–stack web development using .NET 5, React 17, and TypeScript 4 , Packt Publishing, 2021.

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

КІБЕРБЕЗПЕКА ТА КОМП'ЮТЕРНО-ІНТЕГРОВАНІ ТЕХНОЛОГІЇ (КБКІТ – 2025)

науково-практична конференція
молодих вчених, аспірантів та студентів

28–29 серпня 2025
Тернопіль

Збірник матеріалів науково-практичної конференції молодих вчених, аспірантів та студентів «Кібербезпека та комп'ютерно-інтегровані технології» (КБКІТ - 2025), Тернопіль, 2025. - 154 с.

Адреса редакції:
Західноукраїнський національний університет, кафедра кібербезпеки,
вул. Олени Теліги 8, м. Тернопіль 46003
Контакти:
*e-mail:*conferencekb@gmail.com

УДК 004.056.5

Владислав РУЩАК, Степан ІВАСЬЄВ

Західноукраїнський національний університет

ДОСЛІДЖЕННЯ ВРАЗЛИВОСТІ БІБЛІОТЕКИ CLICKBAR/DOT-DIVER

Вступ. Бібліотека @clickbar/dot-diver, реалізована на TypeScript (відкритий вихідний код), надає зручний API для зчитування значення поля об'єкта (функція getByPath) та запису значення у поле об'єкта (функція setByPath), має широке застосування. В бібліотеці було виявлено вразливість типу prototype pollution, що критично для безпеки вебзастосунків на TypeScript, що її використовують.

Мета: дослідити шляхи реалізації вразливості prototype pollution бібліотеки dot-diver.

1. Аналіз вразливості функції `setByPath`

Уразливість виявлено у функції setByPath, яка приймає три аргументи:

- object - об'єкт, у властивість якого встановлюється значення;
- path - шаблон шляху до властивості, що підлягає зміні;
- value - значення, яке має бути записане у вказане поле.

При виклику setByPath відбувається рекурсивний обхід елементів у шаблоні шляху. Після знаходження цільового поля йому присвоюється нове значення. Нижче зазначено, що у вихідному повідомленні наводилося загальне пояснення механіки роботи setByPath.

```
1  import { getByPath, setByPath } from '@clickbar/dot-diver'
2
3  // Define a sample object with nested properties
4  const object = {
5    a: 'hello',
6    b: {
7      c: 42,
8      d: {
9        e: 'world',
10     },
11   },
12   f: [{ g: 'array-item-1' }, { g: 'array-item-2' }],
13 }
14 // Example 2: Set a value by path
15 setByPath(object, 'a', 'new hello')
16 console.log(object.a) // Output: 'new hello'
17
18 setByPath(object, 'f.1.g', 'new array-item-2')
19 console.log(object.f[1].g) // Output: 'new array-item-2'
```

Рисунок 1 - Механізм роботи setByPath

Основна проблема застосування цієї функції полягає в тому, що якщо значення шляху містить посилання на прототип, з'являється можливість встановити властивості прототипу, що може призвести до забруднення глобального прототипу.

Цей же код після виправлення (версія 1.0.2) приведено на рисунку 3.

У версії 1.0.1 функція setByPath дозволяє встановлювати властивості за шляхом, що подається у вигляді рядка (наприклад, "a.b.c").

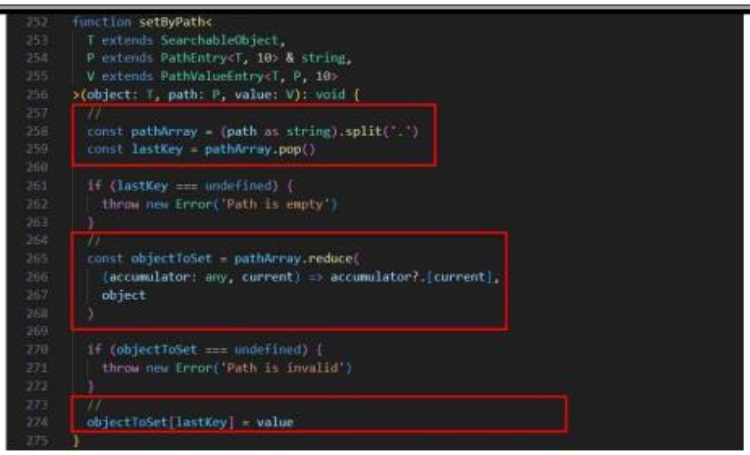

Рисунок 2 - Код функції до виправлення (версія 1.0.1)

Через відсутність перевірок спеціальних ключів (наприклад, __proto__, constructor.prototype) зловмисник може змінювати властивості прототипу - тобто відбувається prototype pollution. Це відкриває шлях до обходу механізмів контролю доступу й іншої небажаної модифікації поведінки застосунку. setByPath виконує рекурсивний або ітеративний обхід шляхового шаблона (split('.') → reduce/loop) і без додаткових перевірок записує значення у знайдене поле.

2. Аналіз виправлень вразливості

Проблемою що викликала вразливість є відсутність захисту проти модифікації властивостей прототипу: не відкидаються ключі на зразок __proto__, prototype, constructor та відсутній код, щоб перевірити, чи властивість є власною (own property) об'єкта, або чи операція створює нове поле на самому об'єкті, а не змінює глобальний прототип.

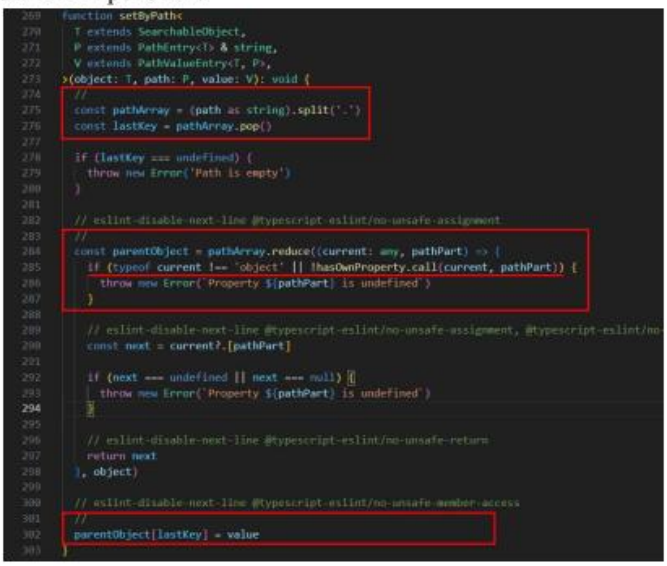

Рисунок 3 - Код функції після виправлення(версія 1.0.2)

У виправленій версії було додано перевірку наявності власної властивості в об'єкті (за допомогою методу Object.prototype.hasOwnProperty) перед внесенням змін. Якщо потрібне поле відсутнє, викликається виняток із відповідним повідомленням.

3. Вектор атаки забруднення прототипу

Вектор атаки можна описати наступними кроками. Зловмисник подає шлях, який містить __proto__ або інший спеціальний сегмент (через користувацькі поля/JSON). setByPath проходить шлях і в кінці записує задане значення не в локальний об'єкт, а в прототип (Object.prototype або інший спільний прототип).

Після цього будь-який інший код, що покладається на наявність певної властивості або перевіряє її присутність через спадкування, може побачити змінену поведінку (наприклад, user.isAdmin === true), що призводить до ескалації привілеїв або обходу авторизації. Алгоритм реалізації вразливості приведено на рисунку 4.

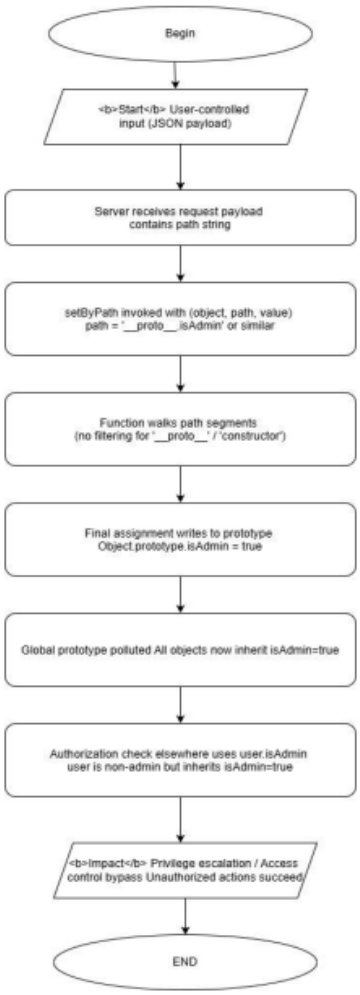

Рисунок 4 – Схема алгоритму використання вразливості setByPath

У прикладі розглядається застосунок для обліку прочитаних книг, у якому реалізовано механізм розмежування користувачів із різними ролями: user - має право додавати дані про прочитані книги та переглядати їх, admin - має право видаляти книги зі списку.

Розмежування доступу реалізовано за допомогою додаткової властивості `isAdmin`, яка присутня лише в об'єкта користувача з успішною автентифікацією для ролі admin. В інших користувачів це поле відсутнє, як показано на рисунку 5.

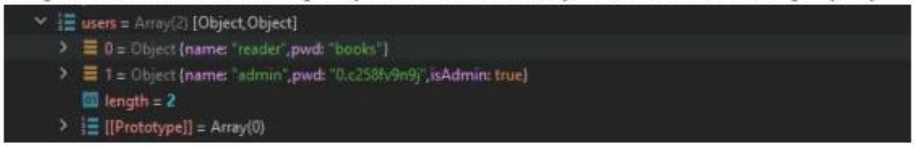

Рисунок 5 - Механізм роботи setByPath

Фрагмент коду, який виконує обробку запиту на видалення книги за ключем title та реалізує контроль прав доступу, може виглядати так, як показано на рисунку 6.

```
app.delete('/books/:title', (req, res) => {
  const { title } = req.params;
  const user = req.user; // об'єкт користувача, отриманий після автентифікації

  // Перевірка прав доступу
  if (!user || !user.isAdmin) {
    return res.status(403).json({
      error: 'Access denied: insufficient privileges.'
    });
  }

  // Пошук книги за назвою
  const index = books.findIndex((book) => book.title === title);

  if (index === -1) {
    return res.status(404).json({
      error: `Book with title '${title}' not found.`
    });
  }

  // Видалення книги
  books.splice(index, 1);
  return res.status(200).json({
    message: `Book '${title}' was successfully deleted.`
  });
});
```

Рисунок 6 - Фрагмент коду, який виконує обробку запиту на видалення книги

У застосунку використовуються такі API-команди, як отримати список книг:

$ curl -v -X GET -H http://192.169.27.1:6000/

Оновити список книг:

$ curl -v -X PUT -H "Content-Type:application/json" --data '{"auth":{"name":"reader", "pwd":"books"},"title":"Tom Sawyer"}' http://192.169.27.1:6000/

Видалити книгу з певною назвою:

$ curl -v -X DELETE -H "Content-Type:application/json" --data '{"auth":{"name":"reader", "pwd":"books"},"title":"Tom Sawyer"}' http://192.169.27.1:6000/

Після спроби видалення книги від імені користувача reader повертається відповідь із кодом 403 та повідомленням "Access denied", що свідчить про відсутність прав на виконання цієї операції.

До запиту додається корисне навантаження для атаки:

$ curl -v -X PUT -H "Content-Type:application/json" --data '{"auth":{"name":"reader", "pwd":"books"},"title":"Tom Sawyer", "note":"__proto__.isAdmin", "text":true}' http://192.169.27.1:6000/

У відповіді на цей запит відображається результат успішного оновлення списку книг. Окрім того, у глобальному об'єкті Object з'явилося поле isAdmin, як показано на рисунку 7.

Рисунок 7 - Object з полем isAdmin

Після повторного запиту на видалення книги від імені користувача reader у відповіді повертається повідомлення про успішне видалення книги, що свідчить про реалізацію атаки та обхід механізму розмежування доступу, реалізованого в застосунку.

Висновок. Необхідно забезпечити регулярне оновлення всіх бібліотек та залежностей до останніх стабільних версій. Більшість випадків «забруднення прототипу» виникає через застарілі пакети, що містять уразливі функції для глибокого копіювання чи об'єднання об'єктів (наприклад, у lodash, jQuery, dot-prop, deepmerge). Важливу роль відіграє також ретельна перевірка та фільтрація вхідних даних, які надходять від користувачів або зовнішніх API. Усі параметри, що використовуються для побудови об'єктів або динамічних шляхів властивостей, мають проходити валідацію.

Перелік використаних джерел.

1. GitHub Security Advisory. Prototype Pollution (PP) vulnerability in setByPath (GHSA-9w5f-mw3p-pj47). [Електронний ресурс]. – Режим доступу: // GitHub – clickbar/dot-diver. – Опубл.: 02.11.2023.

2. National Vulnerability Database (NVD). CVE-2023-45827: dot-diver – Prototype Pollution у setByPath; виправлення у релізі 1.0.2 Опубл.: 06.11.2023. [Електронний ресурс]. –Режим доступу: https://nvd.nist.gov/vuln/detail/CVE-2023-45827.