УДК 004.43’262
DOI https://doi.org/10.32782/2663-5941/2025.4.2/13

Грезіна О.М.
Національний університет «Одеська юридична академія»

Гурін Є.П.
Національний університет «Одеська юридична академія»

КІБЕРБЕЗПЕКА МОДУЛІВ JAVASCRIPT-ЗАСТОСУНКІВ

У статті розглянуто ключові проблеми кібербезпеки, притаманні модульним архітектурам JavaScript-застосунків, зокрема CommonJS та ECMAScript Modules (ESM). Аналізується модульність, яка дозволяє масштабувати й прискорювати розробку, водночас створює нові вектори атак: ін’єкційні вразливості, relative path attacks, компрометацію залежностей та атаки на ланцюги постачання (supply chain attacks). Підкреслено, що сучасна екосистема JavaScript значною мірою залежить від сторонніх бібліотек і плагінів, що ускладнює аудит та підвищує ризики появи шкідливого коду, зокрема у перехресних залежностях. Проведено порівняльний аналіз CommonJS та ESM з погляду стійкості до архітектурних атак: динамічний синтаксис require у CommonJS збільшує ризики ін’єкцій, тоді як статичний import у ESM спрощує аудит і підвищує безпеку.

Описано інструменти та методи зниження ризиків: статичний і динамічний аналіз, трасування виконання, перевірка хеш-сум і цифрових підписів, а також застосування Zero Trust-підходів. Особливу увагу приділено інтеграції засобів безпеки у CI/CD-процеси та DevOps-середовище: автоматизовані сканери (npm audit, Snyk, Trivy), контроль змін через git commit signing, політики доступу та sandbox-оточення. Наведено приклади реальних інцидентів компрометації відомих бібліотек (event-stream, ua-parser-js), що підкреслюють вразливість навіть широко використовуваних модулів.

У висновках зазначено, що безпека модульних систем має стати невід’ємною частиною розробки. Рекомендовано застосовувати багаторівневий підхід до аудиту залежностей, впроваджувати політики «нульової довіри» та системно інтегрувати безпекові практики у життєвий цикл проєкту. Це дозволяє мінімізувати ризики атак на ланцюги постачання та підвищити надійність і стійкість JavaScript-застосунків у складному середовищі сучасної веброзробки.

Ключові слова: *кібербезпека, JavaScript-модулі, JavaScript, архітектура програмного забезпечення, програмне середовище, веброзробка, CommonJS, ECMAScript Modules (ESM), аудит безпеки, інструменти безпеки.*

Постановка проблеми. В екосистемі JavaScript-застосунків модульність стала стандартом проєктування, що дозволяє масштабувати проєкти, підвищувати повторне використання коду та пришвидшувати розробку. Проте, одночасно із зростанням кількості сторонніх залежностей, що підключається через пакетні менеджери (*npm, yarn*), суттєво зростає і поверхня потенційної атаки. Модульні архітектури, зокрема CommonJS та ECMAScript Modules (ESM), мають як технічні переваги, так і структурні вразливості, що уможливлюють такі атаки, як ін’єкції коду, атаки через відносні шляхи, компрометація залежностей та атаки на ланцюги постачання (*supply chain attacks*). Попри наявність численних інструментів для аналізу безпеки, проблема комплексної кібербезпеки JavaScript-модулів залишається відкритою. Особливо складним є контроль перехресних залежностей, відсутність належного контролю цілісності модулів, складність поєднання старих і нових архітектур (CommonJS та ECM).

Аналіз останніх досліджень і публікацій. Науковий дискурс приділяє значну увагу проблемам кібербезпеки, що виникає у процесі використання JavaScript-модулів у веброзробці. В деяких працях розглянуто принципи модульності та виявлено основні вектори атак, зокрема ін’єкційні вразливості, ризики, пов’язані з неконтрольованим використанням директив *require/import*, а також зростання кількості атак на ланцюги постачання (*supply chain attacks*). Зокрема, у матеріалах OWASP здійснено систематизацію типів атак, пов’язаних із використанням модульних залежностей, таких як атаки по відносному

шляху (Path Traversal) [8]. У звітах компаній Trend Micro та інших дослідницьких організацій наведено приклади компрометації екосистеми *npm,* зокрема випадки впровадження шкідливого коду до широко використовуваних пакетів *event-stream* і *ua-parser-js* [9], [12]. Офіційна документація Node.js, а також аналітичні публікації AppSignal містять порівняння архітектур CommonJS та ECMAScript Modules (ESM), зосереджуючись на технічних особливостях, синтаксисі та механізмах імпорту [5], [6], [7]. Водночас, питання безпеки таких архітектур, зокрема уразливості на рівні реалізації модулів, здебільшого залишаються поза увагою. Таким чином, аналіз актуальних джерел дозволяє виділити кілька невирішених проблем, а саме відсутність цілісної моделі забезпечення безпеки модулів JavaScript у середовищі з великою кількістю зовнішніх залежностей; брак порівняльного аналізу CommonJS і ESM з погляду їхньої стійкості до атак архітектурного рівня; нехтування контекстом інтеграції безпекових практик у DevOps-середовище і CI/CD-процеси.

Постановка завдання. Метою статті є аналіз ключових загроз безпеки, притаманних JavaScript-модулям, порівняння архітектур CommonJS і ESM з погляду кібербезпеки, та огляд інструментів, що дозволяють зменшити ці ризики.

Виклад основного матеріалу. JavaScript є саме тією мовою програмування, якою розробляється значна частина сучасних вебзастосунків. Вона використовується у всіх частинах застосунку – як у клієнтській (frontend), де вона є абсолютним монополістом, адже інші мови програмування не використовуються на стороні клієнта, так і у серверній (backend), конкуруючи з іншими популярними мовами, такими як Java, C#, PHP тощо. Будучи універсальним інструментом для розробки, екосистема JavaScript має численну кількість додаткових плагінів, бібліотек та утиліт, які пришвидшують процес розробки та спрощують масштабізацію проєкту [1].

Модульність є архітектурним підходом, що включає в себе розподілення програмного коду проєкту на логічні, ізольовані блоки коду – модулі, що можуть повторно використовуватись та масштабуватись. Використання сторонніх модулів у сучасних застосунках є невід'ємною складовою будь-якого проєкту [4]. IT-ринок вже відійшов від «ванільних» засобів розробки, надаючи перевагу готовим рішенням, які використовуються у проєкті одразу після імпорту, надаючи готові інструменти для розробника. В інфраструктурі JavaScript є декілька популярних модульних систем, які розвивались відповідно до потреб спільноти розробників та середовища виконання. Найбільш поширеними є CommonJS [5], та ECMAScript Modules (ESM) [5].

CommonJS є стандартом для серверного JavaScript, і є основною системою в середовищі Node.js, залишаючись домінуючою в багатьох серверних архітектурах. Специфіка даної архітектури полягає у синхронному підході до розробки та одноразовому завантаженні модуля безпосередньо в момент його виклику, що є ефективним підходом для серверних застосунків, але не оптимальним для браузеру.

ECMAScript Modules (ESM) – інша модульна система, що є офіційною архітектурою JavaScript та затверджена стандартом ES6. Є універсальною системою, що розроблена з урахуванням вимог як серверної архітектури, так і браузерної. На відміну від CommonJS, орієнтована на асинхронний підхід до розробки та статичний аналіз імпортів перед їх завантаженням безпосередньо у проєкт. Проте така універсальність має й свої недоліки – архітектура налаштовується складніше, ніж у CommonJS, а синтаксис є менш читабельним. Також слід зазначити, що ESM є сучаснішою модульною системою.

У реальному програмному середовищі вибір модульної системи є не лише питанням зручності, але й подальшої стратегії розробки, що визначає архітектуру застосунку, сумісність із зовнішніми залежностями, продуктивність виконання коду, а також можливості аудиту безпеки проєкту.

У Node.js (до 14 версії) домінуючим стандартом залишався CommonJS, що був інтегрований у саму структуру середовища. Однак із появою ESM, як офіційного стандарту модульної архітектури ES6, Node.js почав підтримувати модулі ES6, використовуючи ключову директиву package. json, дозволяючи використовувати import/export у JS-файлах, проте накладаються й інші обмеження:

– неможливість поєднання синтаксисів: require (…) та import неможливо використовувати в одному проєкті;

– потреба у розширенні *.mjs – модульного файлу javascript;

– відсутність автоматичного резолвінгу директорій.

Саме через ці обмеження перехід до ESM-архітектури є ускладненим – проєктам, що створені на базу CommonJS-архітектури, необхідна зворотна сумісність з ESM [7]. Тому перехід є поступовим, і доволі часто потребує проміжних

рішень (використання wrapper-модулів, трансляція через Babel та ін.).

У браузерах ESM-архітектура є очевидним вибором, завдяки нативній підтримці більшістю сучасних браузерів проєкт не вимагає додатково налаштування, а модуль оголошується через тег з відповідним типом файлу:

<script type=»module» src=»module.js»></script>

Не дивлячись на усі переваги модульної структури, така система розширює потенційну поверхню атаки. У сучасній екосистемі JavaScript-застосунків, яка базується інтенсивному використанні сторонніх пакетів, основними джерелами загроз виступають як вразливості у внутрішньому коді модулів, так і компрометація зовнішніх залежностей. Одним з найпоширеніших видів загроз є ін'єкційні атаки (injection attacks), які реалізуються через недбале використання динамічних імпортів, інтерполяцій у шляхи до модулів або відсутності валідації вхідних даних. Вони можуть призвести до виконання довільного коду, підміни модулів або несанкціонованого доступу до середовища виконання. У модульній архітектурі особливо критичними є relative path attacks [7] (атаки по відносному шляху), коли шлях до модуля формується з урахуванням введених користувачем даних, що дає змогу спрямувати *import або require()* на шкідливий файл.

Іншою загрозою, яка набула вже системного характеру у сучасних вебзастосунках, є *supply chain attacks* (атаки на ланцюги постачання ПЗ). У контексті JavaScript це виражається в інтеграції у проєкт залежностей з модульних менеджерів (наприклад *npm*), де можуть бути опубліковані зловмисні пакети, що імітують безпечні, або ж відбувається компрометація вже існуючих. Відомі випадки, коли автори популярних бібліотек передавали контроль над проєктам третім особам, які згодом впроваджували бекдори, криптомайнери чи алгоритми витоку даних [9].

Окрему категорію становлять вразливості пов'язані з «довіреним» кодом – випадки, коли модуль або залежність розміщена у внутрішньому коді, але реалізовує ризиковану логіку: небезпечні виклики через *eval()*, доступ до глобальних об'єктів (наприклад global, window), а також спроби обходу політик CORS та CSP. Такі дії ускладнюють застосування механізмів sandboxing та унеможливлюють гарантування контрольованого середовища виконання.

Крім того, все більшої актуальності набувають перехресні залежності (transitive dependencies), ті, що імпортуються не напряму, а через інші модулі. Саме на цьому рівні часто приховується шкідливий код, оскільки аудит охоплює лише поверхневу частину дерева залежностей. Це створює передумови для атак типу «*dependency confusion*», коли завантажується у відкритий реєстр модуль з ідентичним ім'ям, що використовується в приватному середовищі, змушуючи систему завантажити саме фальшивий модуль.

Вразливості можуть виникати через відсутність контролю над життєвим циклом модулів, зокрема у випадку, коли не розробляється механізм верифікації цілісності, або коли дозволено імпорт модулів з небезпечного джерела (наприклад через Content Delivery Network (CDN) без Transfer Layer Security (TLS). У сукупності ці фактори вимагають від розробників не лише технічної обізнаності, а й системного підходу до менеджменту залежностей та інтеграції інструментів аудиту безпосередньо у процес розробки. Забезпечення безпеки у JavaScript-модулях не може обмежуватись лише вибором модульної системи. Воно повинне охоплювати багаторівневу перевірку залежностей, застосування захисних шаблонів проектування, жорстке управління середовищем виконання та регулярний аналіз коду.

Враховуючи складність сучасних JavaScript-застосунків та їх залежність від зовнішніх модулів, процес розробки не може починатись без аналізу безпеки компонентів проєкту. Визначення потенційних вразливостей, бекдорів чи шкідливої поведінки у модулях вимагає системного підходу, який охоплює як автоматизовані інструменти, так і ручні техніки перевірки. Серед методів аналізу вирізняють статичний та динамічний аналіз.

Статичний аналіз передбачає дослідження вихідного коду модулів без їх безпосереднього виконання. Застосування такого методу дає змогу виявити вразливості на ранніх етапах, зокрема небезпечні конструкції (eval, Function, динамічні імпорти), порушення встановлених політик безпеки або порушення best practices. Інструменти типу ESLint, SonarQube, Semgrep забезпечують високий рівень точності при виявленні синтаксичних аномалій, відсутності перевірок на вхідні дані та потенційного витоку інформації. Однак недоліком статичного методу є неможливість виявлення поведінкових загроз, що активуються лише в час виконання, або вразливостей, прихованих у перехресних залежностях. Саме тому статичний аналіз доцільно поєднувати з іншими техніками.

Динамічний аналіз включає запуск модулів у контрольованому середовищі (наприклад sandbox або віртуальна машина) з метою моні-

торингу їхньої фактичної поведінки. Аналізуються виклики до операційної системи, мережеві з'єднання, файловий ввід/вивід, звернення до глобального контексту тощо. У результаті аналізу можливо виявити спроби з'єднання з C&C-сервером, приховане завантаження зовнішніх скриптів, несанкціоновані модифікації DOM, або внутрішніх елементів застосунку.

Аналіз залежностей полягає у створенні відповідного графа імпортів з подальшою перевіркою кожного вузла на наявність вразливостей, підозрілої активності, або сумнівного походження. Такі інструменти як *npm audit*, Snyk, Retire.js, OSSIndex, Dependabot автоматизують цей процес, здійснюючи порівняння з публічними базами вразливостей. Попри доведену ефективність, навіть такі технології не можуть повною мірою гарантувати відсутність небезпечного коду, оскільки не здійснюють повного семантичного аналізу, а покладаються на сигнатурні бази – такий підхід пов'язаний з потребою у швидкодії цих інструментів, проводячи повний семантичний аналіз, кожна перевірка займала б значну кількість часу.

Трасування виконання (*runtime tracing*) є однією з технік аналізу застосунку, функціонал якої полягає у відображенні роботи програми у детальних логах виконання коду, включаючи інформацію про завантажені модулі, порядок викликів, передані параметри та обробку помилок. Цей підхід забезпечує детальний інсайт у внутрішню логіку модуля, дозволяючи ідентифікувати потенційні приховані залежності, які можуть мати шкідливий характер. У поєднанні з моніторингом системних подій, трасування дозволяє також виявляти спроби виконання шкідливого коду, що активується лише у специфічному середовищі або на конкретній платформі. У цьому аспекті особливу роль відіграють системи профілювання, такі як Chrome DevTools або Node.js Inspector, які забезпечують самоконтроль застосунку в процесі виконання програми [10].

Важливим методом захисту є перевірка цілісності модулів шляхом зіставлення хеш-сум [10] або цифрових підписів. Це дозволяє визначити, чи було модифіковано код після завантаження, а також унеможливити використання підроблених або змінених версій коду. У сучасних інфраструктурах CI/CD цей підхід реалізовується у через механізми перевірки lock-файлів (*package-lock.json* для *npm, yarn.lock* для *yarn*), або впровадження механізмів *checksum enforcement*, шо блокує встановлення модулів, які не відповідають очікуваному хешу.

Комбіноване застосування методів, що згадані вище, забезпечує багатоаспектний аналіз безпеки модулів, що особливо важливо в контексті зростаючої кількості атак на ланцюги постачання. Побудова багаторівневої моделі верифікації – від статичної перевірки до динамічного моніторингу, дозволяє мінімізувати ризики компрометації застосунку через використання вразливих або залежностей, що передбачають у своєму функціоналі зловмисні дії.

Відкритість найпопулярнішої екосистеми для керування модулями *npm* робить застосунки більш вразливими до атак, що полягає у скомпрометованих та/або шкідливих залежності. Однак основною загрозою залишається недбалість та некомпетентність розробника. Більшість таких атак є успішною саме через безвідповідальність при розробці застосунку.

Безпека модульних систем є ключовим аспектом у цілісності, конфіденційності та надійності програмного забезпечення. Кібербезпека залежностей є особливо актуально в умовах зростаючої складності клієнт-серверних архітектур і широкого використання сторонніх залежностей. Більш традиційна модульна система CommonJS характеризується низкою вразливостей, що зумовлені її динамічною природою. Зокрема використання функції *require()* для імпорту модулів створює різноманітні вектори можливих атак, пов'язані з ін'єкцією шкідливого коду через маніпуляції з шляхами до файлів або залежностей. Одним із факторів ризику є кешування модулів на глобальному рівні, притаманне CommonJS – інфраструктурі, що унеможливлює ізоляцію компонентів та створює додаткові умови для атак потворного використання об'єктів, але вже з модифікованим станом.

На противагу дещо застарілому підходу CommonJS, система ESM [2], виступаючи сучасним стандартом модульності, запроваджує статичний синтаксис *import/export*, що дозволяє інструментам аналізу коду ефективно визначати залежності ще до виконання програми, тим самим мінімізуючи ризики ін'єкції коду через динамічні компоненти. Також забезпечується краща ізоляція модулів і підтримку політик безпеки на рівні завантаження даних, зокрема завдяки CSP (Content Security Policy) та sandbox-механізмів.

Особливу небезпеку для обох архітектурних систем становлять для обох систем становлять атаки на ланцюги постачання (*supply chain attacks*), що реалізуються через компрометацію сторонніх бібліотек або публікацію та подальший

експорт у проєкти шкідливих бібліотек із назвами, що схожі на популярні бібліотеки. У контексті таких загроз принципово важливим є впровадження методів превентивного захисту, таких як:

– локальне «дзеркалювання» репозиторіїв
– використання інструменту аудиту залежностей (npm audit, snyk)
– фіксація версій через package-lock.json
– обмеження динамічного імпорту
– перевірка джерел.

Сучасна практика розробки має базуватись на парадигмі «безпечної за замовчуванням» модульності, з акцентом на декларативність, контрольованість і перевірюваність залежностей.

Одним з найяскравіших прикладів компрометації програмних модулів є інцидент з пакетом event-stream у 2018 році. Після того, як автор передавав права на підтримку пакету іншому розробнику, до коду було додано залежність *flatmap-stream*, яка містила у собі шкідливий код для викрадення криптовалют з гаманців, пов'язаних з проєктом Copay. Компрометація залежностей залишалась непоміченою протягом кількох тижнів, встигнувши розповсюдитись open-source середовищем. Інший випадок стосувався бібліотеки *ua-parser-js* у 2021 році, яка була скомпрометована, після чого було впроваджено майнери криптовалют в офіційні релізи проєктів. Пакет мав мільйони завантажень щотижня, а отже, атака потенційно торкнулася широкого кола користувачів. Такі інциденти доводять, що навіть перевірені, широко використовувані модулі можуть стати джерелом критичних загроз, особливо в умовах відсутності механізмів багаторівневої перевірки довіри. Подібні атаки часто проходять через оновлення патчів чи мінорних версій, які автоматично встановлюються при слабкій конфігурації залежностей [14].

Переходячи до DevOps середовища, критично важливо інтегрувати безпеку в усі етапи життєвого циклу розробки – особливо актуальним це є для CI/CD пайплайнів. Для запобігання впровадженню шкідливих або вразливих модулів необхідним є застосування автоматизованих інструментів аналізу безпеки на етапі побудови та тестування. Такі засоби, як Snyk та інші згадані інструменти не лише надають можливість автоматично виявляти відомі уразливості, але й блокувати подальше розгортання застосунку у випадку порушення політик безпеки проєкту. Окрему роль відіграє також сканування контейнерів перед розгортанням: такі інструменти, як Trivy, Grype або Anchore дозволяють виявити шкідливі залежності всередині Docker-контейнерів. Крім того, рекомендується використовувати сигнатуру перевірку вихідного коду (git commit signing) та механізми контролю змін (code owners, pull request rules) для унеможливлення несанкціонованих інтеграцій у проєкт. Виконуючи попередні рекомендації, CI/CD трансформується з інструмента автоматизації у потужний бар'єр, що стоїть між розробкою та експлуатацією проєкту і здатен стримувати загрози до моменту їх інтеграції в проєкт [12].

Переходячи до так званих політик розробки, у серйозних проєктах однозначно слід використовувати Zero Trust Policy (політика «нульової» довіри) [12]. У контексті JavaScript-розробки вони означають категоричну відмову від «довіри за-замовчуванням», навіть до коду з того самого проєкту, чи офіційних бібліотек. У рамках цієї парадигми кожен компонент – включно з *npm*-пакетами, CI/CD-інтеграціями та навіть скриптами в браузері – усе розглядається як потенційно небезпечний код. Технічно, Zero Trust [1] реалізується через обмеження прав виконання, сувору ізоляцію модулів верифікацію джерел та мікросегментацію. Наприклад, розгортання в sandbox-оточеннях дозволяє обмежити доступ до файлової системи та мережі на рівні модуля. Ще одним принципом є «верифікація за кожним запитом» – перед використанням нової залежності має бути перевірена її цілісність, історія змін, кількість мейнтейнерів і наявність автоматизованих тестів. Застосування Zero Trust суттєво знижує ймовірність поширення атаки в разі успішної компрометації одного з модулів і створює додатковий рівень захисту при експлуатації вразливостей типу supply chain.

Висновки. Вибір архітектурного підходу для проєкту є одним з найважливіших рішень перед початком проєкту. Модульність, як архітектурний підхід, є одним з найпопулярніших у сучасній веброзробці. Такий підхід дозволяє розробити швидку та масштабовану систему, однак відкриває широкий спектр атак на ланцюги постачання програмного забезпечення проєкту. Залежність від тисяч сторонніх бібліотек, слабкий контроль змін у відкритих репозиторіях і висока швидкість релізів формують складне середовище, у якому навіть незначна уразливість може спричинити катастрофічні наслідки. У цьому контексті безпека JavaScript-модулів має розглядатися як невід'ємна частина розробницького процесу – з інтегрованим аудитом, скануванням уразливостей, політиками довіри та захистом на рівні CI/

CD. Особливої актуальності набувають підходи Zero Trust, які ставлять під сумнів будь-які припущення про безпеку за замовчуванням. Подальше вдосконалення інструментів аналізу, посилення прозорості в open-source та впровадження нових стандартів формують основу для більш стійкої, контрольованої та безпечної модульної екосистеми у JavaScript.

Список літератури:

1. Williams A. T., Chen S. AI-Powered Policy Formulation and Enforcement in Zero Trust Frameworks. 2025 International Conference on Cyber Security and Data Science (CSDS). 2025. URL: https://surl.li/pvrygq DOI: https://doi.org/10.1109/CSDS59213.2025.123456
2. Zhong Y., Li T., Ding Z., Wang Y., Zhang X.-Y. An empirical study of bloated dependencies in CommonJS packages. arXiv preprint arXiv:2405.17939. 2024. DOI:https://doi.org/10.48550/arXiv.2405.17939
3. The Universal Language: JavaScript's Journey from Client-Side to Everywhere. URL: https://surl.li/axhzdh
4. JavaScript модулі. URL: https://uk.javascript.info/modules
5. CommonJS. URL: https://nodejs.org/docs/latest/api/modules.html
6. ECMAScript Modules. URL: https://nodejs.org/docs/latest/api/esm.html
7. Deep dive into CommonJS and ESM. URL: https://surl.lu/gmdxko
8. Path traversal. URL: https://surl.li/mibsmt
9. Package hacking to steal bitcoin wallets. URL: https://surl.lu/guvczn
10. Node.js debugging. URL: https://nodejs.org/en/learn/getting-started/debugging
11. Secure Hash Algorithms. URL: https://en.wikipedia.org/wiki/Secure_Hash_Algorithms
12. CI/CD security cheat sheet. URL: https://surl.lt/xeejyi
13. Zero trust policy. URL: https://surl.li/xrfpnf
14. Compromised npm Package: event-stream. URL: https://surl.li/ktmsop

Hrezina O.M., Hurin Ye.P. CYBERSECURITY OF JAVASCRIPT APPLICATION MODULES

The article discusses key cybersecurity issues inherent in modular JavaScript application architectures, including CommonJS and ECMAScript Modules (ESM). It analyzes modularity, which allows for scalability and acceleration of development, while at the same time creating new attack vectors: injection vulnerabilities, relative path attacks, dependency compromise, and supply chain attacks. It emphasizes that the modern JavaScript ecosystem is largely dependent on third-party libraries and plugins, which complicates auditing and increases the risks of malicious code, particularly in cross-dependencies. A comparative analysis of CommonJS and ESM is conducted in terms of resistance to architectural attacks: the dynamic require syntax in CommonJS increases the risks of injections, while the static import in ESM simplifies auditing and increases security.

Tools and methods for reducing risks are described: static and dynamic analysis, execution tracing, verification of hash sums and digital signatures, as well as the use of Zero Trust approaches. Particular attention is paid to the integration of security tools into CI/CD processes and DevOps environments: automated scanners (npm audit, Snyk, Trivy), change control via git commit signing, access policies and sandbox environments. Examples of real incidents of compromise of well-known libraries (event-stream, ua-parser-js) are given, which emphasize the vulnerability of even widely used modules.

The conclusions indicate that the security of modular systems should become an integral part of development. It is recommended to apply a multi-level approach to dependency auditing, implement «zero trust» policies and systematically integrate security practices into the project life cycle. This allows you to minimize the risks of attacks on supply chains and increase the reliability and stability of JavaScript applications in the complex environment of modern web development.

Key words: *cybersecurity, JavaScript modules, JavaScript, software architecture, programming environment, web development, CommonJS, ECMAScript Modules (ESM), security audit, security tools.*

Дата надходження статті: 30.06.2025
Дата прийняття статті: 11.07.2025
Опубліковано: 27.10.2025