БАКАЛАВРСЬКА

РОБОТА

БР. ІП – 23.00.00.000 ПЗ

Група ІП-22-1

Ціник Андріан

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ціник Андріан Ігорович

(прізвище, ім'я, по батькові)

УДК 004.62
(індекс)

БАКАЛАВРСЬКА РОБОТА

Методи тестування безпеки програмного забезпечення

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121– Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня Ціник.А.І
(підпис, ініціали та прізвище здобувача)

Науковий керівник Процюк Василь Романович, доц.
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

доц. Бандура В.В.
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ ” 2026 р.

ЗАВДАННЯ

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Цінику Андріану Ігоровичу

(прізвище, ім’я, по батькові)

1. Тема проєкту (роботи) ***“Методи тестування безпеки програмного забезпечення”*** керівник проєкту (роботи) доцент. Процюк.В.Р

затверджені наказом закладу вищої освіти від “ 21 ” квітня 2026 р. № 179/7

2. Строк подання студентом проєкту (роботи) червня 2026 р.

3. Вихідні дані до проєкту (роботи) Результати і матеріали, отримані під час проходження .переддипломної практики

4. Зміст розрахунково – пояснювальної записки (перелік питань, які потрібно розробити)

1 Аналіз методів статичного аналізу безпеки коду (SAST) на етапі розробки.

2 Огляд стандартів безпеки (OWASP) та систем автоматизованого аудиту.

3 Розробка архітектури, серверного оркестратора та обчислювального ядра сканера.

4 Комплексне тестування та документування розробленої платформи.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1 Архітектурна схема взаємодії компонентів платформи (рис. 3.1, ст. 45)

2 Діаграма прецедентів (Use Case) системи статичного аналізу коду (рис. 3.2. ст. 47)

3 Інтерфейс аналітичного дашборду платформи CodeGuard SAST (рис. 4.2, ст. 56)

4 Головна сторінка веб-інтерфейсу платформи CodeGuard із панеллю моніторингу (рис. 4.4, ст. 58)

5 Структура вихідного коду обчислювального ядра сканера на базі регулярних виразів (рис. 4.5, ст. 59)

1. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	**Завдання прийняв**

2. Дата видачі завдання _____

Керівник _____ Процюк.В.Р
(підпис) (розшифровка підпису)

Завдання прийняв до виконання _____ Ціник.А.І
(підпис) (розшифровка підпису)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обгрунтування теми роботи	11.04.2026	виконано
2	Огляд існуючих трендів та рішень	19.04.2026	виконано
3	Побудова архітектурної моделі та алгоритмів власного рішення	03.05.2026	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	31.05.2026	виконано
5	Оформлення пояснювальної записки бакалаврської роботи	10.06.2026	виконано

Студент–бакалавр _____ *Ціник А.І.*
(підпис) (розшифровка підпису)

Керівник роботи _____ *Процюк В.Р.*
(підпис) (розшифровка підпису)

АНОТАЦІЯ

Бакалаврська робота містить 82 сторінки, 9 рисунків, список використаних джерел із 36 найменуваннями.

Метою дипломної роботи є розроблення вебзастосунку для автоматизованого статичного тестування безпеки вихідного коду (SAST).

Об'єкт дослідження: процеси автоматизованого виявлення архітектурних та логічних недоліків безпеки у вихідному коді програмного забезпечення.

Предметом дослідження є архітектурні, алгоритмічні та програмні засоби побудови розподіленої SAST-системи із застосуванням лексичного аналізу.

У першому розділі розглянуто теоретичні засади безпеки ПЗ, еволюцію архітектурних підходів та класифікацію методів тестування безпеки.

У другому розділі розглядаються вимоги стейкхолдерів, формальна постановка задачі, а також функціональні й нефункціональні вимоги до системи.

У третьому розділі описано проєктування stateless-архітектури, аналіз прецедентів та вибір технологічного стеку: Node.js і C# (Roslyn SDK).

У четвертому розділі описано програмну реалізацію проєкту, алгоритми пошуку вразливостей на базі абстрактних синтаксичних дерев (AST) та розробку клієнтського інтерфейсу.

У п'ятому розділі розглядаються модульне та системне тестування платформи, перевірка точності аналізатора й стійкості API.

Висновки роботи містять основні результати розроблення та тестування застосунку. Підтверджено працездатність MVP-версії платформи CodeGuard.

Отримані результати дослідження включають створення мікросервісного застосунку, що реалізує асинхронне завантаження коду, лексичний аналіз без збереження стану та виявлення вразливостей згідно з OWASP Top 10.

КЛЮЧОВІ СЛОВА: БЕЗПЕКА ПЗ, СТАТИЧНИЙ АНАЛІЗ КОДУ, SAST, DEVSECOPS, МІКРОСЕРВІСНА АРХІТЕКТУРА, NODE.JS, C#, ROSLYN SDK, OWASP TOP 10.

ANNOTATION

The bachelor's thesis contains 82 pages, 9 figures, and a list of references with 36 items.

The purpose of the thesis is to develop a web application for automated static application security testing (SAST) of source code.

Object of the research: processes of automated detection of architectural and logical security flaws in software source code.

Subject of the research: architectural, algorithmic, and software tools for building a distributed SAST system using lexical analysis.

The first section reviews software security foundations, architectural evolution, and security testing methods.

The second section examines stakeholder requirements, formal task formulation, and system specifications.

The third section describes the stateless architecture design and technology stack selection (Node.js and C# Roslyn SDK).

The fourth section covers the software implementation, AST-based vulnerability search algorithms, and client UI development.

The fifth section examines module and system testing, verifying the analyser's accuracy and API resilience.

The conclusions contain the main results of development and testing, confirming the operability of the CodeGuard MVP.

The results include a microservice application implementing asynchronous code uploading, stateless lexical analysis, and OWASP Top 10 vulnerability detection.

KEYWORDS: SOFTWARE SECURITY, STATIC CODE ANALYSIS, SAST, DEVSECOPS, MICROSERVICE ARCHITECTURE, NODE.JS, C#, ROSLYN SDK, OWASP TOP 10.

ЗМІСТ

ВСТУП

У сучасних умовах розвитку цифрової інфраструктури, поширення хмарних сервісів і переходу до мікросервісної архітектури підходи до забезпечення кібербезпеки суттєво змінилися. Традиційна модель захищеного мережевого периметра вже не є достатньою, оскільки більшість сервісів доступні через відкриті вебінтерфейси та API. Тому значна частина ризиків інформаційної безпеки зміщується на прикладний рівень, тобто безпосередньо до вихідного коду програмного забезпечення.

Водночас поширення Agile, CI/CD та DevSecOps вимагає автоматизації перевірок безпеки і їх перенесення на ранні етапи розробки. У цьому контексті важливу роль відіграє статичне тестування безпеки застосунків, яке дозволяє виявляти вразливості ще до компіляції або розгортання. Однак наявні комерційні SAST-рішення часто є дорогими, ресурсоємними та складними для інтеграції у швидкі конвеєри розробки. Тому актуальним завданням є створення легкого розподіленого вебзастосунку, який поєднує асинхронну обробку запитів із лексичним аналізом вихідного коду.

Актуальність теми зумовлена потребою сучасної IT-індустрії у доступних, гнучких і масштабованих інструментах автоматизованого тестування безпеки коду в межах парадигми DevSecOps.

Метою роботи є розроблення мікросервісної платформи автоматизованого тестування безпеки програмного забезпечення CodeGuard, яка забезпечує асинхронне завантаження файлів із вихідним кодом, їх безпечне збереження, лексичний SAST-аналіз без обов'язкової компіляції та формування структурованих звітів про виявлені вразливості.

Завданнями дослідження є: аналіз сучасного стану проблематики та еволюції підходів до безпеки ПЗ; класифікація методів тестування SAST, DAST та IAST і визначення їх архітектурних обмежень; формування функціональних і нефункціональних вимог до системи; проєктування розподіленої мікросервісної архітектури; реалізація асинхронного сервера-оркестратора, обчислювального

ядра на базі парсингу абстрактних синтаксичних дерев та клієнтського інтерфейсу; проведення функціонального й інтеграційного тестування готового програмного продукту.

Об'єктом дослідження є процеси автоматизованого виявлення архітектурних і логічних недоліків безпеки у вихідному коді програмного забезпечення в умовах безперервної інтеграції.

Предметом дослідження є архітектурні, алгоритмічні та програмні засоби побудови розподіленої платформи статичного тестування безпеки з використанням поліглотного мікросервісного підходу та лексичного аналізу.

Методи дослідження включають аналіз предметної області та міжнародних стандартів безпеки, зокрема NIST і OWASP; порівняння наявних SAST-рішень; об'єктно-орієнтоване проєктування; UML-моделювання архітектури та бізнес-процесів; програмну реалізацію з використанням Node.js, C#, .NET Compiler Platform SDK/Roslyn і PostgreSQL; а також системне тестування розробленої платформи.

Наукова новизна дослідження полягає у застосуванні поліглотного мікросервісного підходу до побудови SAST-систем, де ресурсомісткі задачі лексичного аналізу коду делегуються повністю ізольованому "stateless" ядру на базі C#, тоді як маршрутизація та логування забезпечуються асинхронним подієво-орієнтованим оркестратором на Node.js. Це дозволяє усунути проблему блокування обчислювальних ресурсів, притаманну монолітним аналогам.

Практичне значення роботи полягає у застосуванні поліглотного мікросервісного підходу до побудови SAST-систем, де ресурсомісткі задачі лексичного аналізу коду делегуються ізольованому stateless-ядру на базі C#, а маршрутизація та логування забезпечуються асинхронним подієво-орієнтованим оркестратором на Node.js. Це дозволяє зменшити ризик блокування обчислювальних ресурсів, характерний для монолітних аналогів, і підвищити пропускну здатність системи.

Бакалаврська робота містить 82 сторінки, 9 рисунків, 5 розділів, список використаних джерел із 36 найменувань, 1 додаток.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ТА ОГЛЯД ЛІТЕРАТУРИ

1.1. Основні поняття та визначення в контексті безпеки програмного забезпечення

На початковому етапі дослідження проблематики тестування безпеки програмного забезпечення було встановлено, що в сучасній ІТ-індустрії суттєво змінилося саме розуміння поняття «кібербезпека». Якщо раніше безпека програмних систем здебільшого розглядалася через призму мережевої інфраструктури, міжмережевих екранів, систем виявлення вторгнень і контролю доступу до серверів, то сьогодні акцент дедалі більше зміщується на рівень програмного коду, архітектури застосунків і процесів розробки. Класична модель «захищеного периметра» вже не є достатньою, оскільки сучасні системи активно взаємодіють із зовнішнім середовищем.

Однак сучасна практика розробки демонструє, що класична модель мережевого периметра вже не може вважатися достатньою. Це пов’язано насамперед із масовим переходом програмних систем у хмарні середовища, поширенням мікросервісної архітектури та активним використанням публічних API. Більшість сучасних вебзастосунків повинна бути доступною через інтернет, оскільки взаємодія користувачів, клієнтських застосунків, сторонніх сервісів і внутрішніх компонентів часто відбувається через REST API, GraphQL-ендпоінти або веб-інтерфейси. У такій архітектурі зовнішній доступ до окремих частин системи є не винятком, а базовою умовою її функціонування. Відповідно, зловмиснику вже не завжди потрібно долати класичний мережевий захист на рівні операційної системи або інфраструктури. Достатньо надіслати спеціально сформований HTTP-запит до легітимного інтерфейсу системи, і якщо у вихідному коді присутня вразливість, це може призвести до несанкціонованого доступу до даних або порушення логіки роботи застосунку. Отже, у сучасних умовах одна з ключових ліній кіберзахисту проходить безпосередньо через якість, безпечність і перевіреність вихідного коду.

Для коректного аналізу проблеми тестування безпеки програмного забезпечення необхідно сформувати чітку теоретичну основу. У межах цієї роботи одним із ключових джерел є стандарти Національного інституту стандартів і технологій США (NIST), зокрема документ NIST Special Publication 800-218: Secure Software Development Framework (SSDF) Version 1.1. Цей стандарт є важливим, оскільки розглядає безпечну розробку не як окремий завершальний етап перед релізом, а як сукупність практик, які мають бути інтегровані в повний життєвий цикл створення програмного забезпечення.

У процесі опрацювання стандарту NIST було визначено, що інтеграція інструментів і практик безпеки в процес розробки є необхідною умовою створення захищеного програмного продукту. Автори документа зазначають, що лише невелика кількість існуючих моделей життєвого циклу розробки програмного забезпечення явно і детально враховує питання безпеки [1, с. 1]. Тому незалежно від обраної методології, чи йдеться про Waterfall, Spiral, Agile або DevOps, практики безпечної розробки потрібно цілеспрямовано додавати до кожної моделі SDLC. Інакше безпека ризикує залишитися другорядним етапом, що розглядається лише перед релізом, коли виправлення вразливостей є значно складнішим і дорожчим [1, с. 1].

З інженерної точки зору важливо визначити, яку саме мету переслідує інтеграція практик безпечної розробки в життєвий цикл програмного забезпечення. Відповідно до стандарту NIST, дотримання таких практик має три основні цілі. По-перше, вони спрямовані на зменшення кількості вразливостей у випущеному програмному забезпеченні (released software). По-друге, вони мають зменшити потенційний вплив (potential impact) тих вразливостей, які залишилися невиявленими або невирішеними. По-третє, важливим завданням є усунення першопричин (root causes) вразливостей, щоб запобігти їх повторному виникненню в майбутніх версіях програмного продукту [1, с. 1]. Таким чином, безпечна розробка не зводиться лише до пошуку окремих помилок у коді, а передбачає системне зниження ризиків на рівні процесів, архітектури та інженерних практик.

Для ефективного аналізу цієї проблематики необхідно уточнити базову термінологію. У практиці розробки поняття «вразливість» часто помилково ототожнюють лише з помилкою в коді, тобто bug, що виникла через неправильну реалізацію певної функції. Однак у документації NIST це поняття трактується ширше: вразливості охоплюють не лише недоліки кодування, а й слабкі місця, пов'язані з неправильними налаштуваннями безпеки, некоректними припущеннями щодо довіри, недостатнім контролем доступу або застарілим аналізом ризиків [1, с. 1]. Отже, вразливість може виникати не лише у вихідному коді, а й в архітектурі, конфігурації середовища та логіці взаємодії компонентів системи.

Щоб інструменти тестування безпеки мали практичну цінність, вони повинні спиратися на концептуальну модель інформаційної безпеки, відому як тріада ЦЦД або CIA Triad. Вона охоплює три базові властивості захищеної інформаційної системи: конфіденційність, цілісність і доступність. Порушення хоча б одного з цих атрибутів може свідчити про компрометацію системи або про наявність істотного ризику для її користувачів, даних чи бізнес-процесів. Саме тому тріада ЦЦД є базовою теоретичною моделлю, через яку доцільно розглядати наслідки програмних вразливостей.

Конфіденційність (Confidentiality) означає гарантію того, що доступ до інформації мають лише авторизовані користувачі або компоненти системи. У контексті розробки програмного забезпечення це передбачає захист персональних даних, облікових записів, токенів доступу, паролів, фінансової інформації та інших чутливих ресурсів. Практична реалізація конфіденційності охоплює використання захищених протоколів передавання даних, зокрема TLS, належне хешування паролів, наприклад, із використанням bcrypt, а також недопущення збереження секретів у відкритому вигляді у вихідному коді або конфігураційних файлах.

Цілісність (Integrity) передбачає захист інформації від несанкціонованої або неконтрольованої модифікації. У програмних системах це означає, що дані мають змінюватися лише відповідно до визначеної бізнес-логіки та з урахуванням прав

доступу користувача. Наприклад, якщо зловмисник через вразливість може змінити вартість товару, підмінити параметри транзакції або несанкціоновано змінити записи в базі даних, це є прямим порушенням цілісності. Сучасні інструменти тестування безпеки спрямовані, зокрема, на виявлення таких слабких місць, які можуть призвести до некоректної зміни даних.

Доступність (Availability) означає здатність системи залишатися працездатною та доступною для легітимних користувачів. У межах вебзастосунків ця властивість є особливо важливою, оскільки навіть короткочасне порушення доступності може негативно вплинути на користувачів, бізнес-процеси або критичні сервіси. До загроз доступності належать DDoS-атаки, некоректна обробка великих обсягів запитів, зловживання ресурсоємними операціями або вразливості типу ReDoS, які можуть призводити до надмірного навантаження на процесор сервера. Таким чином, безпека програмного забезпечення охоплює не лише захист даних від витоку чи модифікації, а й забезпечення стабільної роботи системи.

Для повнішого розуміння предметної області необхідно також розмежувати кілька суміжних понять, із якими безпосередньо пов’язані автоматизовані інструменти аналізу коду. Одним із таких понять є актив (Asset). Активом вважається будь-який ресурс, що має цінність для організації та потребує захисту. Це може бути база даних із персональними або фінансовими даними користувачів, вихідний код програми, конфігураційні файли, ключі доступу до хмарних сервісів, внутрішня документація, інфраструктура, а також репутація компанії. У контексті безпеки програмного забезпечення важливо розуміти, які саме активи можуть бути скомпрометовані внаслідок експлуатації певної вразливості.

Експлойт (Exploit) - це спеціально створений фрагмент коду, сценарій або послідовність дій, які використовують вразливість для виконання несанкціонованої операції. Якщо вразливість є потенційною слабкістю системи, то експлойт демонструє практичний спосіб її використання. Наприклад, експлойт може дозволити виконати довільну команду на сервері, обійти механізм автентифікації, отримати доступ до бази даних або витягнути конфіденційну

інформацію. Саме експлойти демонструють, як теоретична слабкість системи може перетворитися на реальну атаку.

Загроза (Threat) - це будь-яка обставина, суб'єкт або подія, що має потенціал завдати шкоди системі, її користувачам або даним. Загрозою може бути зовнішній зловмисник, хакерське угруповання, шкідливе програмне забезпечення, помилка співробітника або неуважні дії розробника. У сучасних програмних системах загрози можуть мати як зовнішній, так і внутрішній характер, тому їхній аналіз повинен враховувати не лише технічні атаки, а й людський фактор, організаційні процеси та конфігураційні помилки.

Ризик (Risk) - це ймовірність того, що певна загроза скористається вразливістю та призведе до реальних негативних наслідків. Ризик формується на перетині трьох складників: наявності цінного активу, існування вразливості та можливості її експлуатації конкретною загрозою. У межах розроблення інструменту безпеки основним завданням є не досягнення абсолютної невразливості коду, оскільки для великих програмних систем це практично неможливо гарантувати, а зменшення ймовірності успішної атаки та зниження потенційного впливу виявлених слабких місць.

Окреме значення має поняття поверхні атаки, з яким безпосередньо пов'язані методи аналізу безпеки програмного забезпечення. Поверхня атаки охоплює всі точки взаємодії, через які зовнішній користувач або компонент може передати дані до системи, вплинути на її поведінку або отримати інформацію з неї. У вебзастосунках до таких точок належать поля введення, параметри URL-адрес, HTTP-заголовки, API-ендпоінти, механізми завантаження файлів, cookie та токени доступу. Чим більшою є поверхня атаки, тим більше потенційних шляхів має зловмисник для пошуку та експлуатації вразливостей.

Принцип безпечної архітектури, або Secure by Design, передбачає свідоме зменшення поверхні атаки ще на етапі проєктування системи. Це означає, що непотрібні інтерфейси мають бути закриті, вхідні дані повинні проходити сувору валідацію, права доступу мають перевірятися на кожному критичному рівні, а взаємодія між компонентами повинна відбуватися за чітко визначеними

правилами. У цьому контексті автоматизовані інструменти аналізу коду відіграють важливу роль, оскільки дозволяють виявляти потенційно небезпечні конструкції ще до їх потрапляння у production-середовище.

Підсумовуючи розглянуті теоретичні положення, можна зробити висновок, що тестування безпеки програмного забезпечення суттєво відрізняється від класичного тестування якості. Традиційне QA здебільшого перевіряє, чи виконує програма передбачені функції відповідно до вимог, тоді як security testing спрямоване на пошук небажаної, нестандартної або небезпечної поведінки системи. Іншими словами, тестування якості відповідає на питання, чи працює система так, як очікується, а тестування безпеки перевіряє, чи може система бути використана не за призначенням з метою порушення конфіденційності, цілісності або доступності.

Безпека програмного забезпечення не повинна розглядатися лише як фінальний етап перед релізом. Вразливості можуть виникати на різних рівнях: у вихідному коді, архітектурних рішеннях, конфігурації середовища, механізмах автентифікації, роботі з базою даних або логіці обробки користувацьких запитів. Саме тому їх необхідно виявляти протягом усього життєвого циклу розробки. Чим раніше виявлено проблему безпеки, тим простіше, дешевше і безпечніше її усунути. Цей підхід узгоджується з ідеєю інтеграції безпеки в SDLC та є основою для використання автоматизованих інструментів аналізу, зокрема SAST і DAST.

Отже, поняття конфіденційності, цілісності, доступності, вразливості, активу, експлойту, загрози, ризику та поверхні атаки формують теоретичну основу для подальшого аналізу методів тестування безпеки програмного забезпечення. Саме на цій базі доцільно розглядати SAST, DAST та інші підходи до автоматизованого виявлення вразливостей. Вони дозволяють перейти від загального розуміння проблем кібербезпеки до практичних механізмів їх виявлення, оцінювання та зниження в межах сучасного процесу розробки програмного забезпечення. Крім того, визначення цих понять дає змогу коректно описувати зв’язок між слабкістю ПЗ, можливістю її практичного використання зловмисником і наслідками для системи.

1.2. Аналіз сучасного стану питання та еволюція підходів до безпеки програмного забезпечення

.

У процесі дослідження еволюції методів тестування безпеки програмного забезпечення було встановлено, що їх неможливо розглядати окремо від загального розвитку підходів до розробки, розгортання та супроводу програмних систем. Інструменти пошуку вразливостей не існують ізольовано від інженерної практики, а формуються як відповідь на конкретні зміни в організації розробки, архітектурі застосунків і швидкості постачання програмного продукту. Тому аналіз сучасних засобів безпеки доцільно починати з розгляду того, як змінювалися моделі життєвого циклу програмного забезпечення.

Як зазначено в офіційному стандарті NIST SP 800-218, у традиційних моделях розробки, зокрема у каскадній моделі Waterfall, питання безпеки часто розглядалися лише на завершальних етапах створення програмного продукту [1, с. 1]. За такого підходу основна частина функціональності могла розроблятися протягом тривалого часу, тоді як перевірка безпеки виконувалася безпосередньо перед релізом. Це створювало суттєві ризики, оскільки виявлені на пізніх етапах вразливості було складніше, дорожче й довше усувати. З переходом індустрії до Agile, DevOps і швидких ітерацій розробки така модель стала недостатньо ефективною, оскільки вона не відповідала сучасній швидкості зміни програмних систем.

Саме тому в сучасній інженерній практиці сформувалася концепція «зміщення вліво» (shifting left). Її суть полягає в тому, що перевірки якості та безпеки необхідно переносити на якомога ранніші етапи життєвого циклу розробки програмного забезпечення. Такий підхід дозволяє виявляти помилки та вразливості ще під час написання або інтеграції коду, а не після розгортання готового застосунку. У стандарті NIST також підкреслюється, що значну частину проблем безпеки можна вирішувати на різних етапах SDLC, однак раннє врахування цих аспектів зменшує витрати на виправлення дефектів і знижує загальний рівень ризику [1, с. 1].

Водночас змінилися не лише методології управління проєктами, а й сама архітектура програмних систем. Для розуміння цього переходу важливим є аналіз праці Мартіна Фаулера та Джеймса Льюїса, присвяченої мікросервісній архітектурі. Автори зазначають, що раніше корпоративне програмне забезпечення здебільшого будувалося як монолітний застосунок, тобто як єдиний логічний виконуваний блок, у якому поєднувалися обробка HTTP-запитів, доменна логіка, робота з базою даних і формування інтерфейсу користувача [7, с. 2]. Зі зростанням масштабів систем такий підхід почав ускладнювати підтримку, тестування й оперативне виправлення окремих частин застосунку.

З погляду безпеки монолітна архітектура має низку суттєвих обмежень. У моноліті цикли змін окремих модулів тісно пов’язані між собою, тому навіть незначне виправлення в одному компоненті може вимагати повторного збирання, тестування та розгортання всього застосунку [7, с. 2]. Наприклад, якщо в одному модулі виявлено SQL-ін’єкцію або іншу локальну вразливість, команда часто не може швидко випустити ізольований патч лише для цього фрагмента системи. Необхідність повторного розгортання всього моноліту підвищує ризик затримки критичних виправлень.

Поступовий перехід до хмарних середовищ, автоматизованого розгортання та високої частоти змін став однією з причин поширення мікросервісного підходу. Фаулер і Льюїс визначають мікросервісну архітектуру як спосіб побудови програмного забезпечення з набору невеликих незалежних сервісів, кожен із яких виконується у власному процесі та взаємодіє з іншими компонентами через легкі механізми комунікації, зокрема HTTP-ресурсні API [7, с. 1]. Така модель дає змогу розгортати, оновлювати та масштабувати окремі сервіси незалежно один від одного, що є важливим для сучасних CI/CD-конвеєрів і швидкого випуску змін [7, с. 1-2].

Організаційний аспект цього переходу також має важливе значення для безпеки програмного забезпечення. Фаулер і Льюїс посилаються на Закон Конвея, згідно з яким архітектура системи значною мірою відображає структуру комунікації всередині організації [7, с. 4]. Мікросервісний підхід сприяв переходу

від великих централізованих команд до менших крос-функціональних груп, які відповідають за окремі сервіси протягом усього їхнього життєвого циклу. Із цим пов'язаний принцип «you build it, you run it», за яким команда не лише розробляє програмний компонент, а й відповідає за його стабільну та безпечну роботу в production-середовищі [7, с. 5].

З технічного боку мікросервісна архітектура також змінила підхід до зберігання та обробки даних. Одним із важливих принципів є поліглотна стійкість (Polyglot Persistence), яка передбачає можливість використання різних типів сховищ даних для різних сервісів [7, с. 7-8]. Це означає, що окремий мікросервіс може мати власну базу даних, оптимізовану під його функціональні потреби. З погляду безпеки такий підхід має перевагу, оскільки компрометація одного сервісу не обов'язково означає автоматичний доступ до всієї інформаційної інфраструктури компанії. Водночас він ускладнює тестування, оскільки кожен компонент може мати власну логіку роботи з даними, власні ризики та власні вимоги до захисту.

Змінився і підхід до мережевої комунікації між компонентами системи. Фаулер зазначає, що сучасна мікросервісна архітектура часто будується за принципом «розумні кінцеві точки та прості канали» (Smart endpoints and dumb pipes) [7, с. 6]. Це означає, що основна бізнес-логіка залишається всередині сервісів, тоді як канали комунікації виконують переважно транспортну функцію. Саме тому замість складних корпоративних шин ESB дедалі частіше використовуються простіші REST-запити або легкі черги повідомлень, наприклад RabbitMQ.

Однак мікросервіси не можна розглядати як універсальне вирішення всіх проблем архітектури та безпеки. Коли система складається з великої кількості сервісів, які постійно взаємодіють через мережу, зростає ймовірність збоїв, затримок, помилок інтеграції або некоректної обробки відповідей. Саме тому в мікросервісних системах важливим є принцип «проєктування для відмови» (Design for failure), відповідно до якого архітектура повинна враховувати можливість недоступності окремого сервісу або переривання мережевого виклику

[7, с. 9-10]. З погляду безпеки це також означає необхідність перевіряти не лише окремий код, а й поведінку системи в умовах збоїв, некоректних відповідей і нестабільної взаємодії між компонентами.

На цьому етапі виникає ключове питання: як забезпечити ефективне тестування безпеки в системі, де десятки мікросервісів можуть оновлюватися кілька разів на день через CI/CD-конвеєр. У такому середовищі ручна перевірка кожної зміни перед розгортанням стає практично неможливою. Фахівець із безпеки не може оперативно аналізувати кожен коміт, кожну зміну API або кожне оновлення окремого сервісу, особливо якщо розробка ведеться паралельно кількома командами. Тому традиційна модель періодичних ручних аудитів уже не відповідає швидкості сучасної розробки та не може бути єдиним механізмом контролю безпеки.

Щоб безпека не перетворювалася на вузьке місце у DevSecOps-процесі, перевірки мають бути автоматизованими та інтегрованими безпосередньо в життєвий цикл створення програмного забезпечення. Саме для цього використовуються інструменти SAST, DAST та інші засоби аналізу, які можуть перевіряти код у середовищі розробника, під час збирання проєкту, у межах CI/CD-конвеєра або перед розгортанням у production-середовище. Такі інструменти не замінюють повністю експертний аудит, однак дозволяють швидко виявляти типові вразливості, зменшувати ризик потрапляння небезпечного коду в робоче середовище та підтримувати безпеку на рівні, сумісному зі швидкістю мікросервісної розробки. У результаті автоматизоване тестування стає не окремою разовою процедурою, а постійним елементом інженерного процесу, що супроводжує програмний код від моменту його написання до етапу розгортання. Саме тому подальший аналіз інструментів автоматизованого тестування безпеки є логічним продовженням дослідження. Такий підхід забезпечує безперервний контроль безпеки без істотного уповільнення процесу розробки. Такі інструменти не замінюють повністю експертний аудит, однак дозволяють швидко виявляти типові вразливості. Завдяки цьому безпека підтримується на рівні, сумісному зі швидкістю мікросервісної розробки.

1.3. Класифікація підходів і методів тестування безпеки програмного забезпечення

Після аналізу еволюції архітектури програмних систем, зокрема переходу від монолітних застосунків до мікросервісної архітектури [7, с. 1-2], а також після розгляду прискорення процесів розгортання завдяки CI/CD-конвеєрам, логічним наступним етапом дослідження є аналіз інструментів виявлення вразливостей. Сучасні програмні системи швидко змінюються, складаються з багатьох незалежних компонентів і часто використовують різні мови програмування, фреймворки та зовнішні бібліотеки. У таких умовах неможливо покладатися лише на один універсальний засіб перевірки безпеки, оскільки різні інструменти орієнтовані на різні етапи життєвого циклу програмного забезпечення та різні класи вразливостей.

Саме тому в сучасній практиці кібербезпеки сформувався комплексний набір методів тестування безпеки застосунків, який об’єднують поняттям Application Security Testing (AST). Ці методи можуть застосовуватися на етапі написання коду, під час збирання проєкту, після розгортання тестового середовища або вже в процесі виконання застосунку. Кожен із підходів має власні переваги, обмеження та сферу доцільного використання. У межах цього дослідження доцільно розглянути п’ять основних категорій таких інструментів: SAST, DAST, IAST, SCA та RASP.

Статичне тестування безпеки застосунків

Одним із базових підходів до автоматизованого виявлення вразливостей є статичне тестування безпеки застосунків, або SAST. Його суть полягає в аналізі вихідного коду, проміжного представлення або скомпільованих артефактів без фактичного запуску програми. Саме цей підхід є особливо важливим для реалізації принципу «зміщення вліво».

Необхідність такого аналізу підтверджується у стандарті NIST SP 800-218. Зокрема, практика PW.7 «Review and/or Analyze Human-Readable Code» передбачає перевірку та аналіз зрозумілого людині коду. У документі зазначено,

що організації повинні використовувати автоматизовані інструменти для аналізу вихідного коду, скриптів та інших програмних артефактів із метою виявлення вразливостей і їх усунення ще до випуску програмного забезпечення [1, с. 14]. Це робить SAST важливим інструментом раннього контролю безпеки в межах SDLC.

Схоже визначення подає й документація OWASP. Відповідно до неї, інструменти аналізу вихідного коду, тобто SAST, призначені для перевірки вихідного коду або його скомпільованих версій з метою виявлення недоліків безпеки до того, як програма буде запущена [13, с. 1]. Отже, статичний аналіз працює за принципом «білого ящика» (white-box), оскільки має доступ до внутрішньої структури програми, її файлів, функцій, методів, залежностей і потенційно небезпечних конструкцій.

Важливою перевагою SAST є можливість інтеграції безпосередньо в середовище розробника. Такі інструменти можуть працювати як окремі сканери, частина CI/CD-конвеєра або плагіни для IDE. У результаті розробник може отримати попередження про потенційну проблему ще під час написання коду або перед відправленням змін до репозиторію. Такий миттєвий зворотний зв’язок значно зменшує витрати на виправлення дефектів, оскільки проблема усувається до того, як вона потрапить у загальну кодову базу або стане частиною готового релізу [13, с. 1].

До основних переваг SAST належить висока масштабованість. Статичний аналіз можна автоматизувати та запускати на великій кількості проєктів, наприклад під час нічних збірок, pull request-перевірок або кожного коміту в репозиторій [13, с. 1]. Це особливо важливо для команд, які працюють із великою кодовою базою або підтримують кілька сервісів одночасно. Завдяки цьому SAST може стати постійним елементом процесу розробки, а не разовою перевіркою перед релізом.

Ще однією перевагою є здатність виявляти добре формалізовані технічні вразливості. До таких проблем належать SQL-ін’єкції, небезпечна конкатенація рядків у запитах, жорстко закодовані секрети, небезпечні виклики API, переповнення буфера та інші типові дефекти, які можна знайти за допомогою

алгоритмічного аналізу [13, с. 1]. Для розробника важливим є також формат результатів: SAST-інструмент зазвичай вказує конкретний файл, рядок коду та фрагмент, у якому було виявлено потенційну проблему. Це спрощує локалізацію дефекту та прискорює його виправлення.

Водночас SAST не можна розглядати як повне вирішення всіх проблем безпеки. Одним із головних обмежень статичного аналізу є складність виявлення логічних вразливостей. Сканер може знайти небезпечну конструкцію в коді, але він не завжди здатен правильно інтерпретувати бізнес-логіку застосунку, правила доступу або складні сценарії автентифікації. Через це SAST-інструменти можуть пропускати проблеми з авторизацією, некоректною логікою ролей, небезпечним використанням криптографії або помилками в процесах обробки даних [13, с. 1-2].

Іншим суттєвим недоліком є наявність хибнопозитивних результатів (False Positives). У таких випадках сканер позначає безпечний фрагмент коду як потенційно небезпечний, що змушує розробника або фахівця з безпеки витрачати додатковий час на ручну перевірку звіту [13, с. 2]. Якщо кількість таких спрацьовувань є надмірною, практична цінність інструменту знижується, оскільки команда починає сприймати повідомлення сканера як інформаційний шум.

Окремим обмеженням є залежність деяких корпоративних SAST-інструментів від повноти проєкту та можливості його компіляції. Для аналізу може знадобитися наявність усіх сторонніх бібліотек, правильних інструкцій збірки, конфігураційних файлів і повного набору вихідного коду [13, с. 2]. Це створює додаткові труднощі для аналізу неповних репозиторіїв, окремих фрагментів коду або проєктів зі складною структурою залежностей.

Відповідно, під час вибору SAST-інструменту необхідно враховувати низку критеріїв. Насамперед важливою є підтримка цільової мови програмування та технологічного стеку. Далі слід оцінювати здатність інструменту виявляти поширені класи загроз, зокрема ті, що входять до OWASP Top 10 або інших галузевих переліків. Також критичною є точність сканера, тобто співвідношення

хибнопозитивних і хибнонегативних результатів, яке може оцінюватися за допомогою OWASP Benchmark. Додатковими факторами є можливість роботи з незібраним вихідним кодом, інтеграція з IDE та CI/CD-конвеєрами, а також вартість ліцензії, яка в комерційних продуктах може залежати від кількості рядків коду [13, с. 2-3].

Динамічне тестування безпеки застосунків

На відміну від SAST, динамічне тестування безпеки застосунків, або DAST, аналізує вже запущену програму. Цей підхід не потребує доступу до вихідного коду та працює з системою як зовнішній користувач або потенційний зловмисник. Саме тому DAST відносять до методів «чорного ящика» (black-box): сканер взаємодіє із застосунком через HTTP/HTTPS-запити, вебформи, API-ендпоінти та інші зовнішні інтерфейси, не знаючи внутрішньої реалізації програми.

Необхідність такого підходу відображена у стандарті NIST SP 800-218 у практиці PW.8 «Test Executable Code», тобто тестуванні виконуваного коду. У документі зазначено, що виконуваний код, зокрема скомпільовані компоненти або запущені вебсервіси, необхідно перевіряти для виявлення вразливостей, які неможливо було знайти під час статичного аналізу або перевірки вихідного коду [1, с. 15]. Отже, DAST доповнює SAST, оскільки дозволяє оцінити поведінку системи в умовах, наближених до реальної експлуатації.

DAST-інструментам не має значення, якою мовою програмування написано backend-рівень, оскільки вони аналізують не вихідний код, а зовнішню поведінку застосунку. Це робить їх зручними для перевірки систем зі змішаним технологічним стеком, мікросервісних архітектур або сторонніх сервісів, до коду яких команда не має прямого доступу. DAST працює з розгорнутим застосунком через стандартні протоколи та імітує дії користувача.

Зазвичай робота DAST-сканера складається з двох основних етапів. На першому етапі виконується crawling, тобто обхід застосунку з метою виявлення доступних сторінок, форм, параметрів, API-маршрутів та інших точок взаємодії. У такий спосіб сканер будує карту поверхні атаки. На другому етапі застосовуються активні перевірки, зокрема fuzzing, який також згадується в NIST

як рекомендований метод тестування виконуваного коду [1, с. 15]. У процесі фазингу сканер надсилає до застосунку велику кількість спеціально сформованих вхідних даних, наприклад XSS-навантажень, SQL-ін’єкцій або некоректних параметрів, і аналізує реакцію сервера.

Перевагою DAST є те, що його результати часто ближчі до реальних сценаріїв експлуатації. Якщо сканер зміг викликати помилку, отримати небезпечну відповідь або підтвердити виконання шкідливого сценарію, така проблема зазвичай має вищу практичну значущість. Крім того, DAST здатен виявляти проблеми конфігурації серверів, некоректні HTTP-заголовки, відкриті ендпоінти, небезпечні налаштування CORS або TLS, тобто ті аспекти, які не завжди присутні безпосередньо у вихідному коді.

Однак DAST також має низку обмежень. Найважливіше з них полягає в тому, що сканер не може прямо вказати розробнику конкретний рядок коду, у якому міститься проблема. Звіт може містити інформацію про URL, параметр запиту, тип помилки або відповідь сервера, але подальша локалізація вразливості потребує ручного аналізу. Крім того, DAST неможливо ефективно застосувати на ранніх етапах розробки, оскільки для його роботи потрібен розгорнутий і доступний тестовий стенд.

Інтерактивне тестування безпеки застосунків

Інтерактивне тестування безпеки застосунків, або IAST, виникло як спроба поєднати переваги SAST і DAST. Його часто відносять до підходів «сірого ящика» (grey-box), оскільки він поєднує аналіз внутрішньої поведінки програми з перевіркою її виконання в реальному середовищі. На відміну від SAST, IAST працює під час виконання застосунку, а на відміну від DAST, має доступ до внутрішніх даних про виконання коду.

Основою IAST є механізм інструментування (Instrumentation). Спеціальний програмний агент вбудовується в середовище виконання застосунку, наприклад у JVM або .NET CLR, і відстежує, як дані проходять через програму під час її роботи. Коли застосунок тестується вручну, автоматизованими QA-тестами або DAST-сканером, IAST-агент аналізує внутрішній шлях даних, значення змінних,

виклики функцій, звернення до бази даних та інші важливі операції. Завдяки такому підходу інструмент отримує не лише зовнішню інформацію про реакцію застосунку на запит, а й внутрішній контекст виконання, що дозволяє точніше визначати джерело потенційної вразливості та шлях її виникнення в програмному коді.

Перевага IAST полягає в тому, що він може надати більш точний контекст виявленої проблеми. Якщо несанітизовані користувацькі дані потрапляють до SQL-запиту, файлової системи або іншої критичної операції, агент може зафіксувати не лише факт небезпечної поведінки, а й шлях проходження цих даних у програмі. Таким чином, IAST може поєднувати переваги DAST, який бачить реальний запит, і SAST, який допомагає локалізувати проблему в коді.

Водночас IAST має і практичні обмеження. Для його роботи необхідно інтегрувати агент у середовище виконання застосунку, що може бути технічно складним у розподілених або контейнеризованих системах. Крім того, інструментування створює додаткові накладні витрати на продуктивність, а комерційні IAST-рішення часто є дорогими. Через це IAST доцільно використовувати не як універсальну заміну SAST або DAST, а як додатковий інструмент для глибшого аналізу складних застосунків.

Аналіз композиції програмного забезпечення

Сучасні програмні системи значною мірою складаються не лише з власного коду, а й зі сторонніх бібліотек, фреймворків і пакетів із відкритим кодом. Такі залежності підключаються через менеджери пакетів, наприклад npm, NuGet, Maven або pip, і можуть становити значну частину функціональності застосунку. З одного боку, повторне використання готових компонентів прискорює розробку та зменшує дублювання коду. З іншого боку, кожна зовнішня залежність може містити власні вразливості або створювати ризики для ланцюга постачання програмного забезпечення.

У стандарті NIST SP 800-218 практика PW.4 рекомендує повторно використовувати вже наявне, добре захищене програмне забезпечення замість повторної реалізації функціоналу, оскільки це може зменшити витрати й ризик

появи нових помилок. Водночас такі сторонні компоненти обов'язково мають перевірятися на відповідність вимогам безпеки [1, с. 12-13]. Це особливо актуально в умовах зростання кількості атак на ланцюги постачання, коли зловмисники можуть використовувати вразливі або скомпрометовані пакети як шлях проникнення в систему.

Для вирішення цієї проблеми застосовуються інструменти SCA. Вони аналізують файли конфігурації проєкту, наприклад package.json, package-lock.json, .csproj або інші маніфести залежностей, будують дерево використаних бібліотек і порівнюють їхні версії з базами відомих вразливостей, зокрема CVE та NVD. Якщо виявлено пакет із відомою вразливістю, інструмент повідомляє про ризик, вказує небезпечну версію та може запропонувати оновлення до безпечнішої версії.

SCA є важливим доповненням до SAST, оскільки статичний аналіз власного коду не завжди дозволяє виявити ризики, пов'язані зі сторонніми компонентами. Навіть якщо власний код написаний коректно, використання застарілої бібліотеки може створити критичну вразливість у всьому застосунку. Саме тому перевірка залежностей є необхідною частиною сучасного DevSecOps-процесу.

Самозахист застосунків у середовищі виконання

Runtime Application Self-Protection, або RASP, є підходом, який частково розвиває ідеї IAST, але має іншу основну мету. Якщо IAST орієнтований насамперед на тестування та виявлення проблем, то RASP призначений для активного захисту застосунку під час його роботи в production-середовищі. Такий інструмент інтегрується в середовище виконання програми та аналізує небезпечні дії безпосередньо під час обробки запитів.

На відміну від класичного Web Application Firewall, який аналізує трафік ззовні, RASP має доступ до внутрішнього контексту виконання застосунку. Він може бачити, які саме дані передаються до бази даних, файлової системи або інших компонентів, і оцінювати небезпеку дії безпосередньо в момент її виконання. Наприклад, якщо застосунок формує SQL-запит із шкідливим вхідним параметром, RASP може його заблокувати.

Перевагою RASP є здатність реагувати на атаку в реальному часі. Це робить його корисним як додатковий рівень захисту для критичних систем, які вже працюють у production-середовищі. Водночас RASP не можна розглядати як повну заміну іншим методам тестування безпеки, оскільки він виявляє проблему вже під час виконання застосунку. Тому його доцільно використовувати як частину багаторівневої стратегії захисту, а не як основний інструмент пошуку вразливостей на етапі розробки.

Отже, універсального методу тестування або захисту програмного забезпечення не існує. Кожен із розглянутих підходів має власне призначення: SAST дозволяє аналізувати код на ранніх етапах, DAST перевіряє поведінку розгорнутого застосунку, IAST поєднує внутрішній і зовнішній контекст виконання, SCA контролює ризики сторонніх залежностей, а RASP забезпечує додатковий рівень захисту під час роботи системи. На практиці найефективнішим є багаторівневий підхід Defense-in-Depth, за якого кілька методів взаємно доповнюють один одного. Водночас для щоденної перевірки коду в межах сучасного процесу розробки базовим інструментом залишається SAST, оскільки він найкраще відповідає принципу раннього виявлення вразливостей.

Таким чином, аналіз основних категорій інструментів Application Security Testing показує, що ефективне забезпечення безпеки програмного забезпечення потребує поєднання кількох взаємодоповнювальних підходів. Жоден із розглянутих методів не здатний самостійно охопити всі можливі типи вразливостей, етапи життєвого циклу розробки та сценарії експлуатації системи. Саме тому вибір конкретного інструменту має залежати від архітектури застосунку, технологічного стеку, вимог до швидкості розробки та рівня критичності програмного продукту. У межах цієї роботи особливу увагу доцільно приділити SAST, оскільки цей підхід найкраще узгоджується з метою раннього виявлення дефектів безпеки у вихідному коді та може бути ефективно інтегрований у сучасний DevSecOps-процес. Це визначає подальший напрям практичної частини дослідження. Це визначає подальший напрям практичної частини дослідження.

1.4. Огляд існуючих рішень та архітектурне обґрунтування власної розробки

Після розгляду теоретичної бази та класифікації методів тестування безпеки постає практичне завдання вибору інструментальної основи для реалізації системи. У межах бакалаврської роботи необхідно було визначити, чи доцільно використовувати готовий інструмент аналізу безпеки, чи обґрунтованішим буде проєктування власного рішення. Для цього було проведено критичний аналіз наявних засобів Application Security Testing (AST), зокрема SAST-інструментів, які дають змогу виявляти вразливості ще на етапі роботи з вихідним кодом.

Аналіз комерційних та Enterprise-рішень

Під час дослідження було використано офіційний перелік інструментів аналізу вихідного коду, підготовлений спільнотою OWASP. Основну увагу було зосереджено саме на SAST-рішеннях, оскільки вони найкраще відповідають концепції «зміщення вліво» (Shift-Left) і дозволяють виявляти потенційні дефекти безпеки ще до етапу розгортання програмного забезпечення.

Аналіз переліку OWASP показує, що наявні інструменти умовно можна поділити на дві основні групи. До першої належать безкоштовні open-source лінтери та утиліти, орієнтовані на одну конкретну мову програмування або фреймворк. Наприклад, Bandit призначений для пошуку вразливостей у Python-коді [13, с. 2], тоді як Brakeman використовується переважно для аналізу проєктів на Ruby on Rails [13, с. 2]. Такі інструменти можуть бути ефективними в межах окремого технологічного стеку, однак їхнє застосування у складних багатомовних системах має обмеження.

Це обмеження є особливо помітним у контексті мікросервісної архітектури, для якої характерною є поліглотність технологічного середовища [7, с. 7]. В одному проєкті можуть одночасно використовуватися C#, Go, TypeScript та інші мови програмування. У такому випадку застосування кількох вузькоспеціалізованих інструментів ускладнює налаштування, підтримку та уніфікацію результатів перевірки. Команді безпеки доводиться об’єднувати звіти

з різних джерел, що ускладнює інтеграцію таких засобів у CI/CD-конвеєр і підвищує загальну складність супроводу.

До другої групи належать великі комерційні Enterprise-платформи. У переліку OWASP згадуються такі рішення, як Micro Focus Fortify, Checkmarx (CxSAST) та SonarQube [13, с. 4]. Вони підтримують значну кількість мов програмування, мають розвинені механізми звітності та можуть використовуватися у великих корпоративних середовищах. Водночас такі рішення мають низку архітектурних і практичних обмежень, які можуть бути критичними для невеликих команд або гнучких Agile-проєктів.

По-перше, значна частина Enterprise-рішень має складну та ресурсомістку архітектуру. Їх розгортання може потребувати окремого серверного середовища, значного обсягу оперативної пам’яті та додаткових служб для індексації й збереження результатів. Це ускладнює використання таких інструментів у невеликих або гнучких DevSecOps-проєктах.

По-друге, суттєвою проблемою є залежність багатьох SAST-інструментів від можливості компіляції проєкту. У документації OWASP зазначено, що окремі інструменти мають труднощі з аналізом коду, який неможливо зібрати [13, с. 2]. Через це перевірка окремих фрагментів коду або неповних проєктів стає проблематичною.

По-третє, важливим обмеженням є фінансовий бар’єр і залежність від постачальника. Вартість Enterprise-рішень часто залежить від кількості користувачів, обсягу проєкту або кількості рядків коду [13, с. 3]. Крім того, закритість таких продуктів обмежує можливість адаптації правил аналізу до потреб конкретного проєкту.

Отже, аналіз ринку показує певну поляризацію наявних рішень. З одного боку, існують легкі безкоштовні інструменти, які ефективно працюють у межах окремої мови, але погано масштабуються. З іншого боку, є потужні корпоративні платформи, що підтримують багато мов, однак потребують ресурсів та складного налаштування. Це створює підстави для розроблення проміжного рішення, яке поєднує легкість, швидкість і можливість розширення.

Концепція власного рішення та вибір мікросервісної архітектури

З урахуванням виявлених обмежень існуючих платформ у межах бакалаврської роботи було прийнято рішення розробити власний прототип автоматизованого SAST-сканера. Основна ідея полягає в застосуванні до інструменту безпеки мікросервісних принципів, які використовуються під час побудови сучасних вебзастосунків. Щоб уникнути створення важкого монолітного рішення, систему концептуально поділено на дві незалежні частини, реалізовані різними мовами програмування, що узгоджується з ідеєю поліглотності, описаною Фаулером [7, с. 7].

Першою частиною є серверний оркестратор, або Backend / API Gateway, реалізований на Node.js. Його завдання полягає у прийомі файлів від користувача, обробці HTTP-запитів, валідації вхідних даних і передаванні команди на сканування. Асинхронна подієво-орієнтована модель Node.js добре підходить для таких операцій, оскільки дозволяє координувати роботу компонентів без блокування основного потоку виконання.

Другою частиною є ядро аналізатора, реалізоване мовою C# на базі .NET Compiler Platform SDK, відомої як Roslyn. Використання Roslyn дозволяє не створювати власний парсер з нуля, а працювати з внутрішньою моделлю компілятора. Згідно з документацією Microsoft, компілятор формує детальне представлення коду, перевіряючи його синтаксис і семантику, а SDK надає розробникам програмний доступ до цієї моделі [20, с. 1].

Завдяки цьому сканер може аналізувати не лише текстові збіги або регулярні вирази, а й структуру вихідного коду у вигляді абстрактного синтаксичного дерева. Аналізатор проходить по відповідних вузлах AST і виявляє патерни, які можуть свідчити про SQL-ін’єкції, XSS або інші типові вразливості. Отже, запропонована архітектура поєдн ує швидкий Node.js-оркестратор із спеціалізованим C#-Roslyn-ядром, що добре узгоджується з принципами сучасних CI/CD-конвеєрів. Крім того, такий поділ компонентів створює передумови для подальшого розширення сканера, оскільки нові правила аналізу або додаткові мови програмування можуть додаватися на рівні ядра.

1.5. Висновки по розділу

Підсумовуючи результати дослідження, можна виокремити кілька положень, що стали основою подальшого проєктування системи.

По-перше, зміна парадигми кібербезпеки зумовила перенесення значної частини ризиків на прикладний рівень. Оскільки сучасні хмарні та мікросервісні застосунки часто доступні через відкриті API, захист конфіденційності, цілісності та доступності даних має враховуватися вже на рівні вихідного коду системи [1, с. 1].

По-друге, поширення Agile, DevOps і CI/CD обмежує ефективність традиційних ручних перевірок перед релізом. Натомість актуальним підходом є Shift-Left, тобто інтеграція автоматизованих перевірок безпеки на ранніх етапах життєвого циклу розробки, що зменшує ризики, фінансові втрати та технічний борг [1, с. 1].

По-третє, порівняння DAST, IAST, SCA, RASP і SAST показало, що статичний аналіз залишається базовим інструментом раннього виявлення дефектів безпеки. Його перевага полягає в можливості перевіряти вихідний код без очікування компіляції чи розгортання та вказувати конкретні ділянки, де потенційно міститься вразливість [13, с. 1].

По-четверте, аналіз наявних Enterprise-рішень виявив низку обмежень, зокрема високу вартість, vendor lock-in, складність розгортання та монолітну архітектуру. Такі властивості можуть ускладнювати інтеграцію SAST-інструментів у швидкі мікросервісні процеси розробки [13, с. 2-3].

Для подолання цих обмежень було обґрунтовано розроблення власної розподіленої SAST-платформи на основі мікросервісних принципів М. Фаулера [7, с. 1]. Запропонована архітектура використовує поліглотний підхід: асинхронний сервер-оркестратор реалізується на Node.js, а обчислювальне ядро сканера, що виконує аналіз вихідного коду, будується на C# із використанням Roslyn, тобто .NET Compiler Platform SDK [20, с. 1]. Такий поділ забезпечує гнучкість і незалежність основних компонентів системи.

РОЗДІЛ 2. ПРОЄКТУВАННЯ СИСТЕМИ

2.1. Збір та аналіз вимог користувачів

Коли я почав проєктувати CodeGuard (мою платформу для автоматизованого тестування безпеки), стало абсолютно очевидно, що без чіткого дотримання етапів життєвого циклу розробки (SDLC) нормальної інженерної системи не вийде. Я звернувся до класичного керівництва з бізнес-аналізу BABOK від Інституту IIBA. Там дуже влучно зазначено, що еліцитація (тобто виявлення) вимог – це найкритичніший процес, від якого залежить, чи взагалі вирішуватиме система реальні проблеми бізнесу [31, с. 54]. У випадку з інструментами класу SAST найбільший виклик полягає в тому, щоб знайти золоту середину: сканер має перевіряти код максимально глибоко, але при цьому не "гальмувати" роботу розробника.Як пише Ієн Соммервіль, фундаментальним кроком у цій справі є ідентифікація акторів – тобто людей і систем, які безпосередньо взаємодіятимуть із моїм продуктом [32, с. 118]. Аналізуючи сучасну парадигму DevSecOps, я виділив три ключові категорії стейкхолдерів, чиї вимоги я мав задовольнити в архітектурі:

Software Engineers (Розробники ПЗ) Контекст: Це мої первинні користувачі. Саме вони ініціюють перевірку свого коду.Проблематика: Класичні інструменти безпеки часто грішать тим, що переривають процес розробки ("context switching"). Програмісту доводиться лізти в інші складні системи, щоб розібратися з помилками [33, с. 45].Специфічні вимоги: Їм потрібен мінімалістичний Web UI. Розробнику не потрібні складні налаштування; йому потрібен Fast Feedback Loop – швидкий результат перевірки та чітка вказівка на місце вразливості в коді. Це дозволить швидко пофіксити баг, не смикаючи фахівців із кібербезпеки.

Application Security Engineers (Фахівці з безпеки додатків)Контекст: Це люди, які відповідають за те, щоб корпоративний код був захищеним і відповідав стандартам (Compliance).Проблематика: Вони фізично не можуть вручну

переглядати кожен коміт, який заливають розробники [34, с. 88]. Їм потрібен надійний інструмент для автоматичного аудиту.Специфічні вимоги: Для цієї групи головне – точність рушія. Я маю мінімізувати False Negatives (пропущені загрози). Сканер зобов'язаний розуміти патерни з методології OWASP: XSS, SQL-ін'єкції та жорстко закодовані паролі. Ну і, звісно, їм потрібні дашборди з метриками для аналітики.

Infrastructure / DevOps Architects (Архітектори інфраструктури)Контекст: Це ті специ, які будуть розгортати мій аналізатор на серверах і підтримувати його життя.

Проблематика: Традиційні SAST-моноліти споживають просто купу процесорного часу та оперативки. У сучасних хмарних середовищах це виливається у величезні рахунки за сервери [35, с. 112].

Специфічні вимоги: Вони вимагають слабозв'язної мікросервісної архітектури (Loosely Coupled). Обчислювальний модуль сканера має масштабуватися горизонтально і абсолютно незалежно від веб-сервера. Також їм потрібен стандартизований REST API для автоматизації своїх CI/CD пайплайнів.

Очевидно, що між цими групами виникають суперечності. Фахівці з безпеки хочуть максимально глибокого (а отже, довгого) аналізу, тоді як розробники вимагають результатів за кілька секунд. Щоб розв'язати цей конфлікт, я застосував метод компромісного аналізу (Trade-off Analysis), як це рекомендує стандарт ISO/IEC/IEEE 29148:2018 [36, с. 22]. Моє інженерне рішення: асинхронна обробка запитів. Користувач клікає кнопку в легкому веб-інтерфейсі, сервер-оркестратор кидає задачу в чергу, а все важке навантаження бере на себе ізольоване ядро аналізатора десь у фоні.

Результатом цього етапу стала чітка концепція CodeGuard: це сучасна розподілена клієнт-серверна платформа, де модульність забезпечує швидкість, а спеціалізоване ядро гарантує точність пошуку дефектів за міжнародними стандартами.

Коли я зібрав усі побажання від стейкхолдерів (розробників, безпекових інженерів та DevOps), стало абсолютно зрозуміло, що проєктувати архітектуру

просто «на словах» – це шлях у нікуди. Будь-який серйозний інженерний проект вимагає чіткої формалізації. Звернувшись до міжнародного стандарту IEEE Std 1233-1998 (який згодом еволюціонував у сучасний ISO/IEC/IEEE 29148), я структурував усі вимоги до системи на дві фундаментальні категорії: функціональні (FR) та нефункціональні (NFR) [27, с. 14].

Як дуже влучно і просто пояснює Карл Вігерс у своїй книзі з розробки вимог, функціональні вимоги описують безпосередню поведінку системи – тобто конкретні дії, які програма зобов'язана виконати в заданих умовах [28, с. 112]. Щоб не розпилятися на написання непотрібних фіч і зосередитися на головному, я використав методологію прецедентів (Use Case) у поєднанні з підходом MoSCoW [19, с. 76]. Це дозволило мені жорстко відфільтрувати критичний функціонал (Must have).

До переліку обов'язкових функціональних вимог для SAST-аналізатора було віднесено ті можливості, без яких система не могла б виконувати своє основне призначення. Ці вимоги визначають базову поведінку платформи, порядок взаємодії користувача з інтерфейсом, логіку обробки файлів та формат повернення результатів аналізу.

FR-1. Завантаження та передача артефактів. Користувач повинен мати можливість завантажувати файли з вихідним кодом через веб-інтерфейс системи. Серверна частина має приймати файли у форматі multipart/form-data, виконувати їхню первинну перевірку та зберігати у директорії для аналізу.

FR-2. Ініціалізація скануючого ядра. Після успішного завантаження файлу серверний оркестратор повинен автоматично запускати ізольований обчислювальний модуль. Для цього ядру аналізатора передається абсолютний шлях до збереженого файлу як аргумент командного рядка.

FR-3. Лексичний аналіз та пошук вразливостей. Основною функцією системи є аналіз вихідного коду та пошук потенційно небезпечних конструкцій. У межах MVP-версії сканер повинен виявляти щонайменше три класи поширених загроз: SQL-ін'єкції, міжсайтовий скриптинг XSS та жорстко закодовані паролі або ключі доступу.

FR-4. Формування структурованого звіту. За результатами аналізу система має формувати звіт про знайдені вразливості із зазначенням їхнього типу, короткого опису та рівня критичності. Результати повинні серіалізуватися у формат JSON для подальшої передачі між серверною та клієнтською частинами.

FR-5. Моніторинг та аналітика використання. Окрім аналізу коду, система повинна збирати базові статистичні дані про свою роботу, зокрема кількість запусків сканера та переглядів результатів. Надання цих даних через REST API дозволяє клієнтській частині відображати їх у вигляді діаграм.

Для узагальнення сформованих функціональних вимог доцільно подати їх у вигляді схеми трасування. Вона показує, які групи стейкхолдерів найбільше впливають на формування окремих функцій системи та як ці потреби перетворюються на конкретні вимоги до SAST-платформи.

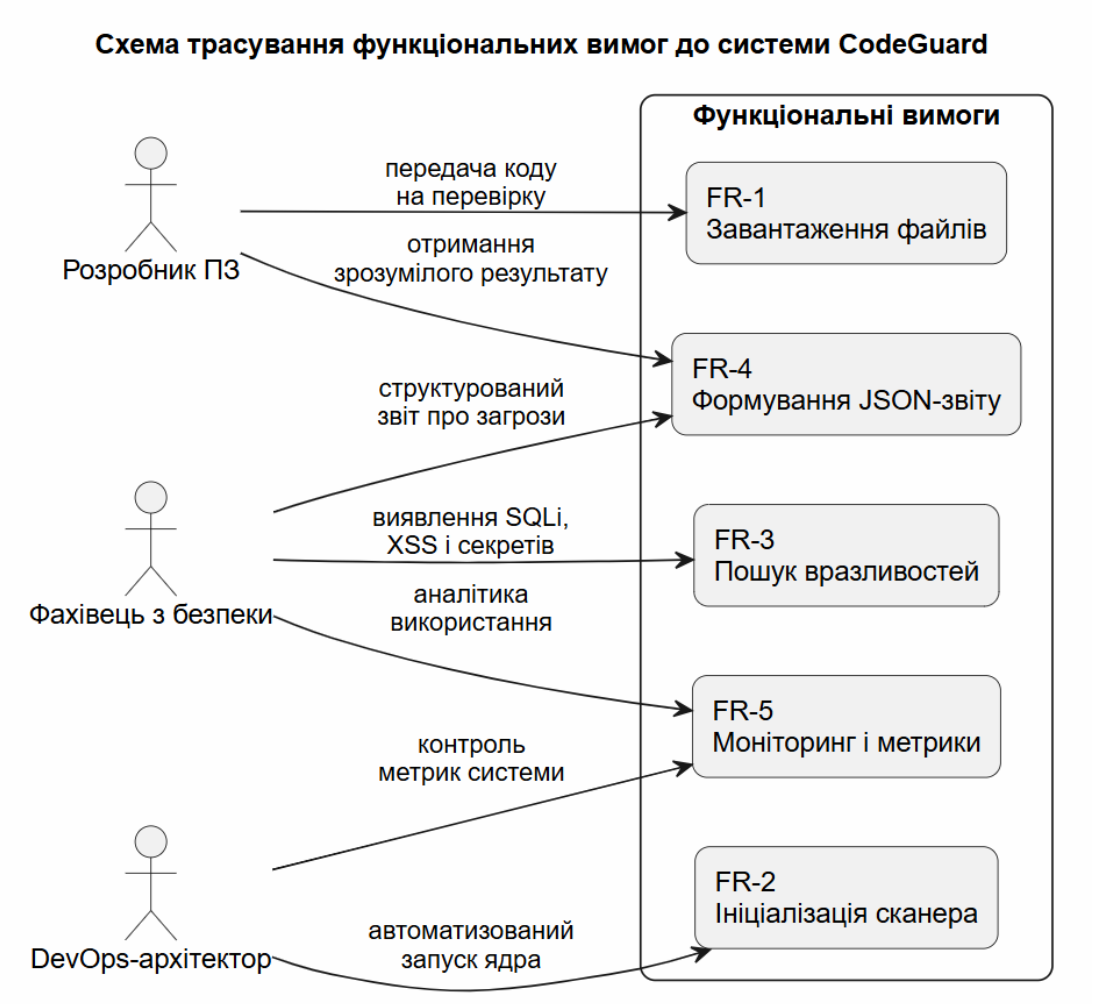


Рисунок 2.1 – Схема трасування функціональних вимог до системи

Але, чесно кажучи, написати функції – це лише половина справи. Є ще нефункціональні вимоги (атрибути якості). Згідно з моделлю якості ISO/IEC 25010:2011, вони визначають не те, що робить система, а те, як добре вона це робить [30, с. 8]. З власного досвіду розробки я переконався, що NFR часто впливають на успіх продукту навіть більше, ніж сам функціонал, бо вони формують досвід розробника (Developer Experience). Якщо сканер працюватиме повільно, ним просто ніхто не буде користуватися.Основні нефункціональні вимоги я сформулював так:

NFR-1. Продуктивність та ефективність (Performance Efficiency). Оскільки CodeGuard має вписуватися у швидкі спринти (Agile/Scrum), час обробки стандартного файлу не повинен перевищувати 3-5 секунд. Саме тому для написання ядра я вирішив обрати мову зі строгою типізацією та швидкою компіляцією.

NFR-2. Масштабованість та архітектурна гнучкість (Maintainability & Scalability). Я категорично відмовився від моноліту. Система проєктується за мікросервісним патерном (окремо оркестратор, окремо ядро сканера). Така слабка зв'язність (Loose Coupling) гарантує, що я зможу оновлювати правила пошуку вразливостей, взагалі не чіпаючи веб-сервер. До того ж, грамотно спроєктований REST API дозволяє легко підключати до системи не лише веб-інтерфейс, а й мобільні додатки.

NFR-3. Надійність та відмовостійкість (Reliability). Програма має "виживати" в будь-яких умовах. Якщо юзер закине битий файл або під час парсингу виникне помилка, ядро не має права аварійно завершувати роботу (Crash). Система повинна акуратно перехопити Exception і повернути коректний JSON зі статусом "error".

NFR-4. Зручність використання (Usability). Інтерфейс має бути ергономічним. Я обов'язково заклав вимогу на розробку темної теми (Dark Mode). Розробники годинами дивляться на монітор на темні IDE, тому білий інтерфейс сканера просто виїдав би очі. Ну і, звісно, наявність індикаторів завантаження (спінерів) для асинхронних операцій.

Окремо доцільно подати схему трасування нефункціональних вимог, оскільки саме вони визначають якісні характеристики майбутньої системи. На відміну від функціональних вимог, які описують конкретні дії платформи, нефункціональні вимоги показують, наскільки швидко, стабільно, зручно та гнучко система повинна виконувати свої функції.

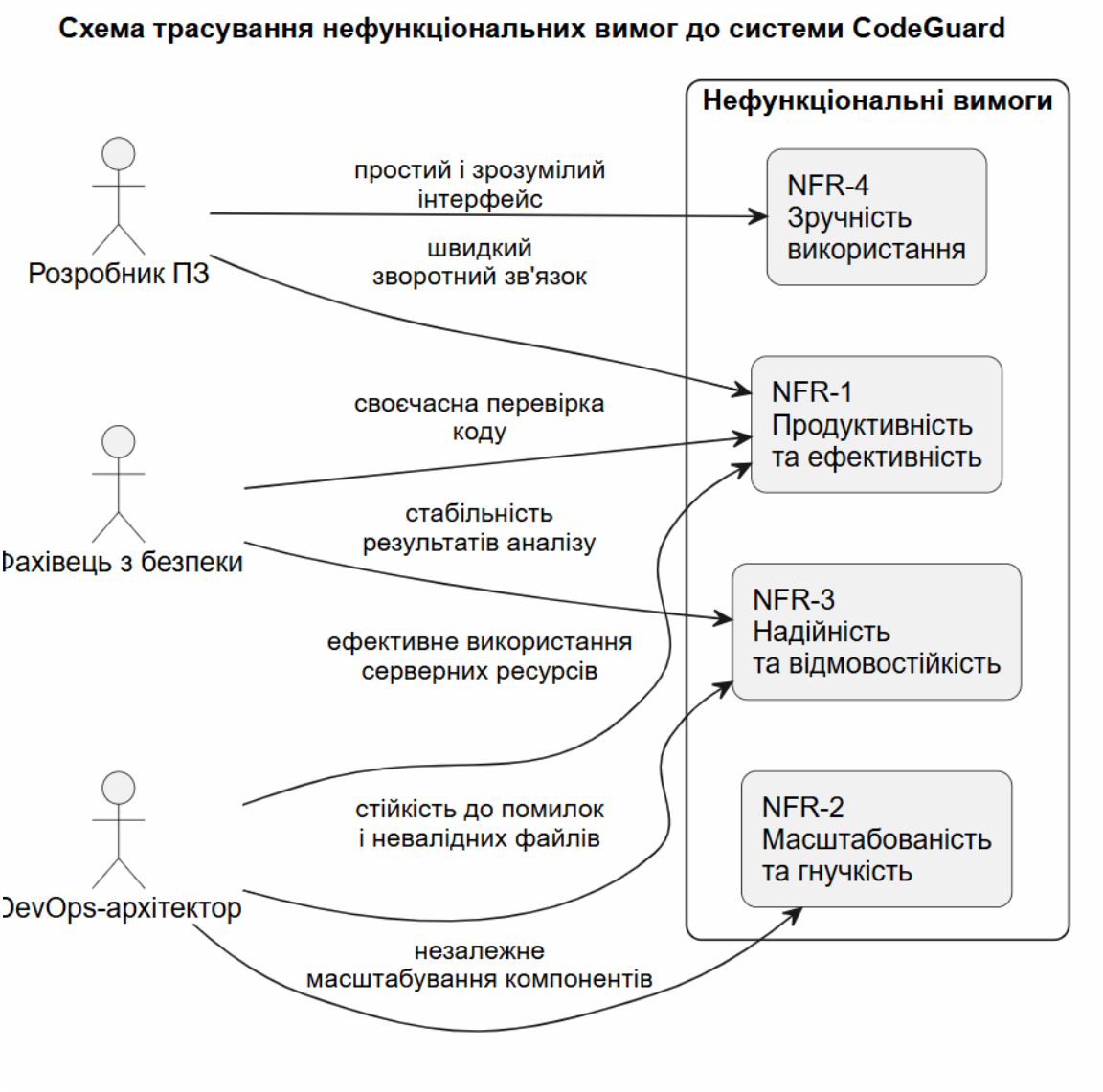


Рисунок 2.2 – Схема трасування нефункціональних вимог до системи

Таким чином, детально розписавши всі FR та NFR, я отримав не просто абстрактну ідею, а залізобетонний інженерний фундамент. Відповідність цим вимогам гарантує мені, що прототип SAST-системи буде відповідати реальним викликам DevSecOps, а не залишиться просто красивою теорією.

2.2. Постановка задачі проєктування системи

Після формування функціональних і нефункціональних вимог виникла необхідність перейти від загального опису очікуваної поведінки системи до конкретної постановки задачі її проєктування. На цьому етапі важливо було не лише визначити функції, які має виконувати платформа, а й встановити принципи розподілу відповідальності між її основними компонентами. Отже, основним завданням стало перетворення сформованих вимог у цілісну програмну архітектуру, придатну для подальшої реалізації, тестування та масштабування.

Ключовою технічною проблемою, яку необхідно було розв’язати під час проєктування, стали обмеження традиційних монолітних SAST-систем. Такі рішення часто характеризуються високим споживанням ресурсів, складністю розгортання та недостатньою гнучкістю під час інтеграції в сучасні CI/CD-процеси. Для розроблюваної платформи важливо було забезпечити швидку обробку файлів, ізоляцію обчислювального навантаження, можливість подальшого розширення функціоналу та зручну взаємодію з клієнтськими застосунками.

З огляду на зазначені вимоги було обрано розподілений підхід до проєктування системи. Відповідно до принципів побудови слабозв’язаних програмних компонентів, описаних у працях Мартіна Фаулера, використання мікросервісної або сервіс-орієнтованої архітектури дозволяє зменшити залежність між модулями та підвищити масштабованість системи [14, с. 104]. На основі цього підходу задачу проєктування платформи було декомпоновано на три основні підсистеми: ядро статичного аналізатора, серверний оркестратор та кросплатформний клієнтський рівень.

1. Проєктування ядра статичного аналізатора (Scanner Engine)

Задача полягає у створенні високоефективного обчислювального модуля, здатного виконувати лексичний аналіз вихідного коду без необхідності його повної компіляції або встановлення всіх залежностей проєкту. Такий модуль має забезпечувати швидку перевірку окремих файлів і виявлення типових

небезпечних конструкцій на основі попередньо визначених правил аналізу. Це є особливо важливим для MVP-версії системи, оскільки на початковому етапі розробки доцільно зосередитися на базовій функціональності, мінімізувати складність розгортання та зменшити залежність від зовнішнього середовища виконання.

З погляду реалізації ядро статичного аналізатора має бути розроблене як консольний застосунок. Воно отримує абсолютний шлях до цільового файлу як аргумент командного рядка, виконує аналіз і завершує роботу після формування результату. Важливою архітектурною вимогою є відсутність внутрішнього стану між окремими запусками, тобто stateless-модель. Це дозволяє запускати кілька екземплярів сканера паралельно та зменшує ризик взаємного впливу між окремими процесами аналізу. Для забезпечення автоматизованої взаємодії з іншими компонентами системи ядро повинно повертати результати сканування у строго структурованому JSON-форматі.

2. Проєктування серверного оркестратора (API Gateway / Backend)

Задача серверного оркестратора полягає у створенні проміжного програмного шару, який виконує роль єдиного фасаду між користувацькими застосунками та обчислювальним ядром. Такий компонент необхідний для централізованої обробки запитів, контролю вхідних даних, запуску процесу сканування та передавання результатів клієнтській частині.

Серверний оркестратор повинен реалізовувати архітектурний стиль REST (Representational State Transfer), який забезпечує уніфіковану взаємодію між клієнтським рівнем і backend-компонентами системи [22, с. 56]. До його основних функцій належать прийом файлів, валідація вхідних даних, обмеження типів завантажуваних файлів, асинхронний запуск ядра аналізатора через дочірні процеси операційної системи та обробка результатів сканування. Також передбачається логування транзакцій і збирання базових метрик із використанням PostgreSQL. Наявність оркестратора підвищує безпеку платформи, ізолює клієнтський рівень від скануючого ядра та спрощує подальшу модифікацію внутрішньої логіки без зміни API системи.

3. Проєктування кросплатформного клієнтського рівня (Client Layer)

Задача клієнтського рівня полягає у створенні зручних та інтуїтивно зрозумілих інтерфейсів для користувачів системи з урахуванням сучасного підходу API-First [33, с. 89]. Оскільки серверна частина надає універсальний REST API, клієнтський рівень може бути реалізований у кількох формах без необхідності зміни базової логіки backend-компонентів.

У межах проєктування передбачено два основні напрями розвитку клієнтського рівня. Першим є створення веб-інтерфейсу, який забезпечує швидкий доступ до функцій сканування через браузер і дозволяє відображати результати аналізу та статистичні дані у вигляді діаграм. Другим напрямом є закладення архітектурної основи для потенційного мобільного клієнта на Android із використанням патерну MVVM та реактивних потоків даних StateFlow. Такий підхід підтверджує можливість використання системи в різних клієнтських середовищах.

Після визначення основних підсистем було сформовано технологічний стек реалізації. Для обчислювального ядра обрано платформу .NET і мову C#, оскільки вони забезпечують високу продуктивність під час обробки текстових даних, підтримують надійну типізацію та мають зручні засоби для реалізації алгоритмів аналізу коду. Для серверного оркестратора було використано Node.js, що є доцільним рішенням для асинхронної обробки файлових потоків, HTTP-запитів та I/O-операцій [24, с. 112]. Такий розподіл технологій дозволяє поєднати продуктивне обчислювальне ядро з гнучким серверним рівнем.

Таким чином, у межах постановки задачі було визначено загальну архітектурну структуру майбутньої системи та обґрунтовано доцільність її поділу на кілька незалежних компонентів: ядро статичного аналізу, серверний оркестратор і клієнтський рівень. Така декомпозиція зменшує зв’язаність компонентів, спрощує подальшу підтримку та створює основу для практичної реалізації програмного рішення, призначеного для автоматизованого тестування безпеки вихідного коду.

2.3. Висновки по розділу

У другому розділі дипломної роботи було проведено системний аналіз предметної області та сформовано вимоги до платформи автоматизованого тестування безпеки програмного забезпечення.

У межах аналізу було визначено основні групи стейкхолдерів системи: розробників програмного забезпечення, фахівців з безпеки додатків та інфраструктурних архітекторів. Для кожної групи було окреслено ключові потреби, зокрема швидке отримання результатів перевірки, точність виявлення вразливостей, можливість автоматизації процесів та гнучкість розгортання системи.

На основі проведеного аналізу було сформовано функціональні та нефункціональні вимоги до системи. До основних функціональних вимог належать завантаження файлів з вихідним кодом, запуск процесу аналізу, виявлення типових вразливостей, формування структурованого звіту та відображення статистичних даних. Нефункціональні вимоги охоплюють продуктивність, надійність, зручність використання, масштабованість та можливість інтеграції з іншими програмними середовищами.

Окрему увагу було приділено постановці задачі проєктування. У результаті було обґрунтовано доцільність використання розподіленої архітектури, у якій окремо виділено ядро статичного аналізу, серверний оркестратор та клієнтський рівень. Такий підхід дозволяє зменшити зв'язаність компонентів, підвищити гнучкість системи та спростити її подальший розвиток.

Таким чином, у другому розділі було сформовано інженерну основу для подальшого проєктування та реалізації системи. Отримані вимоги і визначена архітектурна структура забезпечують логічний перехід до етапу розроблення програмного рішення. Також результати аналізу дали змогу уточнити технічні межі майбутньої системи та визначити пріоритети її практичної реалізації. Це забезпечує узгодженість між потребами користувачів, архітектурними рішеннями та обраним технологічним стеком.

РОЗДІЛ 3. ПРОЄКТУВАННЯ СИСТЕМИ

3.1. Розробка архітектури програмного забезпечення

Коли я закінчив з вимогами і перейшов до третього розділу своєї дипломної роботи, я чітко усвідомив: це найважливіший рубіж. Написати гарні вимоги на папері – це одне, а от перетворити їх на реальну, працюючу і стабільну архітектуру – це зовсім інший рівень складності. Я розумів, що якщо зараз, на етапі закладання фундаменту системи CodeGuard, я оберу неправильний архітектурний патерн, то через кілька місяців мені доведеться переписувати весь код з нуля, бо система просто не витримає навантаження.

Аналізуючи технічну літературу, я звернув увагу на фундаментальну проблему всіх аналізаторів коду. Як детально описують Браян Чесс та Джейкоб Вест у своїй книзі «Secure Programming with Static Analysis», процес розбору синтаксису (парсинг) та побудова абстрактних синтаксичних дерев (AST) – це надзвичайно ресурсоємна і процесорозалежна (CPU-bound) операція, яка здатна швидко вичерпати ліміти оперативної пам'яті [14, с. 154].

Тобто, якби я пішов найпростішим шляхом і спроєктував свій сканер як класичний моноліт (де і веб-сервер, який приймає запити, і сам двигунець сканера живуть в одному процесі), це закінчилося б катастрофою. Уявіть ситуацію: користувач завантажує для перевірки великий файл на кілька тисяч рядків. Монолітний сервер починає його парсити, його головний потік блокується, і в цей момент сервер просто "зависає", перестаючи відповідати на запити інших користувачів. Це абсолютно неприпустимо.

Щоб уникнути цього, я звернувся до принципів проєктування, викладених у класичній праці «Head First Design Patterns». Автори наголошують на критичній важливості застосування принципу слабкої зв'язності (Loose Coupling) при розробці складних систем. Вони зазначають, що слабкозв'язані системи набагато гнучкіші і стійкіші до змін, оскільки об'єкти (або модулі) мінімально залежать один від одного [24, с. 92]. Крім того, такий підхід ідеально лягає в сучасну

парадигму DevOps, описану Джином Кімом: архітектура має дозволяти компонентам працювати та розгортатися незалежно, щоб забезпечити стійкість усієї системи до локальних збоїв [10, с. 145].

Спираючись на ці дослідження, я прийняв категоричне рішення: архітектура мого SAST-аналізатора CodeGuard буде розділена на два ізольованих рівні (мікросервіси), які спілкуються між собою за чітким та легким протоколом.

Перший рівень – це Сервер-оркестратор (API Gateway). Я спроєктував його як "мозок" системи, що відповідає виключно за комунікацію та логістику. Його головна роль полягає в тому, щоб приймати HTTP-запити від клієнтів (веб-браузера), проводити жорстку попередню валідацію завантажених файлів (ми ж тестуємо безпеку, тому маємо відсіяти потенційно шкідливі або надто великі файли ще на вході), безпечно зберігати їх у тимчасових директоріях і керувати чергою задач. Оркестратор нічого не знає про те, як шукати вразливості – він лише керує процесом.

Другий рівень – це Обчислювальне ядро (Scanner Engine). Це "м'язи" моєї системи. Я задумав його як повністю ізольований консольний виконуваний модуль. Це ядро взагалі не має поняття про існування інтернету, HTTP-запитів чи мережевих протоколів. Його єдина, вузькоспеціалізована задача: отримати від оркестратора локальний шлях до файлу, побудувати його AST, перевірити дерево за сигнатурами вразливостей і вивести результат у структурованому форматі JSON.

Найцікавіше в цій архітектурі – механізм їхньої взаємодії (Inter-Process Communication). Я вирішив, що оркестратор буде викликати обчислювальне ядро як окремий дочірній процес операційної системи (Child Process), перехоплюючи його стандартний потік виводу (stdout).

Чому саме так? Тому що це дає мені колосальну перевагу у відмовостійкості. У статичному аналізі часто трапляється так, що через якусь специфічну рекурсію в коді чи переповнення пам'яті процес сканування "крашиться". Якби це був моноліт – "впав" би весь сервер. А в моїй архітектурі, якщо ядро сканера "впаде" під час аналізу специфічного файлу, оркестратор

навіть не похитнеться. Він просто зафіксує код помилки дочірнього процесу (Exit Code), акуратно запише це в системний лог бази даних і віддасть користувачу повідомлення про невдале сканування (HTTP 500 або 422), продовжуючи спокійно обслуговувати всіх інших клієнтів.

Цей архітектурний поділ ідеально вирішив мою головну проблему і заклав потужний фундамент для написання коду.

На рисунку 3.1 наведено загальну архітектуру мікросервісної платформи CodeGuard. Схема відображає розподіл системи на клієнтську частину, сервер-оркестратор (Node.js) та ізольоване обчислювальне ядро сканера (C#), а також демонструє потоки даних під час асинхронної обробки файлів без збереження стану.

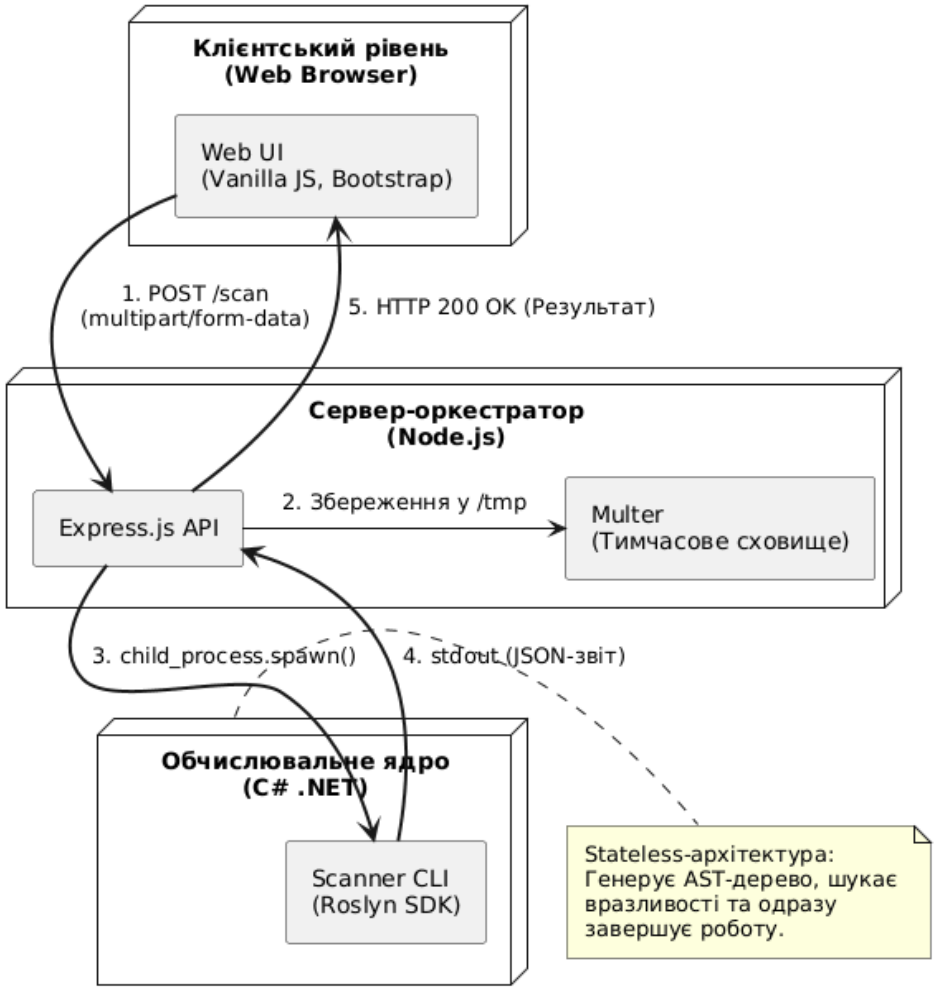


Рисунок 3.1 – Загальна архітектура мікросервісної платформи CodeGuard

3.2. Моделювання бізнес-процесів та системних компонентів

Щоб реалізація проєкту не перетворилася на хаотичне написання коду, я застосував підхід візуального моделювання за допомогою мови UML. Як справедливо зазначають у своїй праці Браян Чесс та Джейкоб Вест, на етапі проєктування системи безпеки вкрай важливо розуміти не лише технічні деталі, а й логіку взаємодії всіх компонентів, щоб уникнути фундаментальних архітектурних помилок [14, с. 198]. Візуалізація дозволила мені побачити слабкі місця системи ще до того, як я написав перший рядок коду.

Для свого проєкту я розробив три типи діаграм, які є обов'язковими для будь-якої технічної документації за стандартом інженерії ПЗ.

1. Діаграма прецедентів (Use Case Diagram) Це була моя відправна точка. Вона дозволяє розмежувати, що саме може робити користувач, а що залишається "під капотом". Основні актори в моїй системі – це «Розробник» (Developer) та «Адміністратор безпеки» (Security Admin). Розробник має обмежений набір прав: завантаження файлу, запуск сканування та отримання фінального звіту. Його мета – швидкий фідбек. Адміністратор безпеки має розширені права: він може не лише переглядати звіти інших розробників, а й керувати базою сигнатур, додаючи нові правила для виявлення специфічних для компанії вразливостей.

На рисунку 3.2 наведено діаграму прецедентів (Use Case Diagram), що визначає архітектурні межі платформи CodeGuard та її функціональні можливості з точки зору кінцевих користувачів. Модель системи передбачає розподіл повноважень між двома основними акторами:

Розробник (Developer): ініціює ключовий бізнес-процес SAST-аналізу. Базовий прецедент запуску сканування (UC-2) є комплексним і через відношення <<includes>> автоматично охоплює етапи завантаження артефактів вихідного коду (UC-1) та генерації структурованого звіту про виявлені вразливості (UC-3).

Адміністратор (Admin): виконує сервісні функції, відповідаючи за налаштування системних конфігурацій (UC-4) та моніторинг використання апаратних ресурсів сервера-оркестратора (UC-5).

Рисунок 3.2 – Діаграма прецедентів (Use Case Diagram) взаємодії користувачів із системою CodeGuard

Такий розподіл прецедентів дозволяє чітко розмежувати процеси експлуатації обчислювального ядра (Roslyn SDK) та адміністрування інфраструктури (Node.js).

Як вказують автори стандарту NIST SP 800-115, процес ідентифікації ролей є критичним для забезпечення цілісності системи – ми повинні чітко знати, хто саме ініціював сканування та хто мав доступ до отриманих результатів, щоб уникнути витоку конфіденційної інформації [5, с. 24].

2. Діаграма діяльності (Activity Diagram) Ця діаграма описує алгоритмічний процес сканування. Це, по суті, "скелет" логіки системи. Процес виглядає так:

Процес обробки завантаженого файлу в системі складається з кількох послідовних етапів, кожен із яких виконує окрему роль у загальному алгоритмі аналізу вихідного коду.

Спочатку користувач завантажує файл через веб-інтерфейс системи. Після отримання файлу бекенд-оркестратор виконує первинну перевірку вхідних даних. На цьому етапі система визначає, чи не перевищує файл допустимий розмір, чи має він коректний формат і чи справді може розглядатися як файл із вихідним кодом, а не як сторонній об'єкт, наприклад зображення або виконуваний файл. Якщо файл не відповідає встановленим вимогам, сервер припиняє подальшу обробку запиту та повертає користувачу відповідь 400 Bad Request.

Якщо файл успішно проходить перевірку, оркестратор зберігає його у тимчасовому сховищі Storage і ставить відповідний запит у чергу на подальшу обробку. Такий підхід дозволяє уникнути блокування основного потоку виконання веб-сервера та забезпечує стабільну роботу системи навіть у разі одночасного надходження кількох запитів від користувачів.

Після цього бекенд викликає ядро сканера Scanner Engine, яке безпосередньо виконує аналіз переданого файлу. На цьому етапі відбувається побудова та обхід AST-дерева вихідного коду, що дає змогу виявляти потенційно небезпечні конструкції, підозрілі шаблони та фрагменти, які можуть свідчити про наявність вразливостей. Після завершення аналізу ядро формує результат і передає його назад серверному оркестратору.

Отримавши результат від скануючого ядра, оркестратор записує дані аналізу до бази PostgreSQL, після чого формує фінальну JSON-відповідь для користувача. У цій відповіді міститься структурована інформація про знайдені вразливості, їхній тип, рівень критичності та інші дані, необхідні для подальшого аналізу або виправлення коду.

Кожен етап цієї діяльності супроводжується логуванням. Це дає змогу відстежувати повний шлях обробки запиту, фіксувати помилки, аналізувати нестандартні ситуації та накопичувати дані для подальшого вдосконалення системи. Згідно з рекомендаціями NIST SP 800-30, належне логування всіх етапів

обробки даних є важливою умовою для проведення якісного аналізу ризиків у майбутньому [6, с. 38].

На рисунку 3.3 зображено діаграму діяльності Activity Diagram, яка деталізує алгоритм обробки завантаженого вихідного коду системою CodeGuard. Діаграма демонструє повний життєвий цикл запиту: від первинної валідації файлу на рівні Node.js-оркестратора до побудови AST-дерева обчислювальним ядром на основі Roslyn SDK. Завершальними етапами є формування JSON-звіту, збереження результатів аналізу та видалення тимчасових файлів, що відповідає stateless-підходу до організації роботи скануючого модуля.

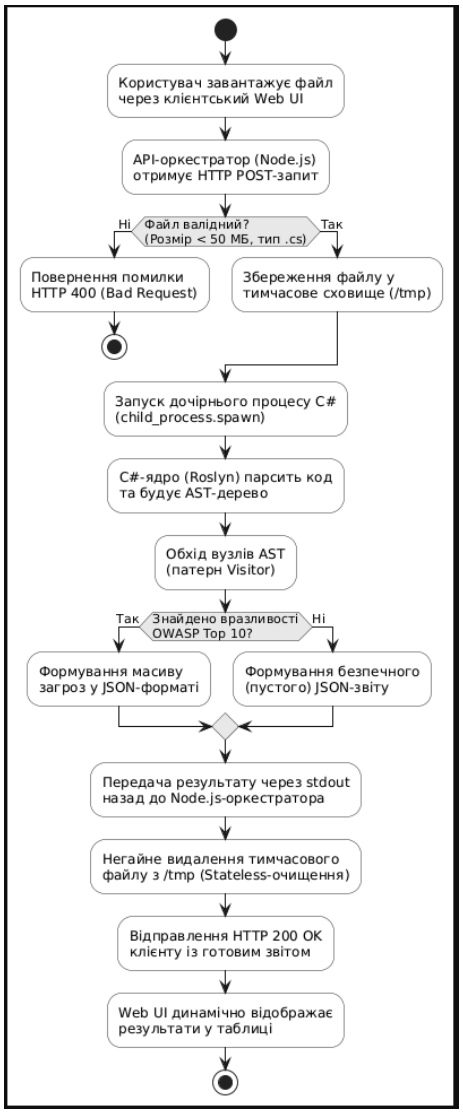


Рисунок 3.3 – Діаграма діяльності (Activity Diagram) процесу лексичного аналізу вихідного коду

3. Діаграма послідовності (Sequence Diagram) показує часову взаємодію клієнта, API Gateway, скануючого ядра та бази даних під час перевірки коду. Клієнт надсилає POST-запит, після чого API Gateway приймає файл, виконує його первинну обробку, запускає ізольований процес C#-аналізатора та очікує результат сканування.

Після завершення аналізу сканер повертає результат у форматі JSON, а API Gateway оновлює відповідний запис у базі даних і формує фінальну HTTP-відповідь. Така взаємодія демонструє асинхронний характер роботи системи та відповідає шаблону Callback, описаному у Head First Design Patterns [24, с. 145].

Часову взаємодію компонентів системи під час сканування коду наведено на рисунку 3.4.

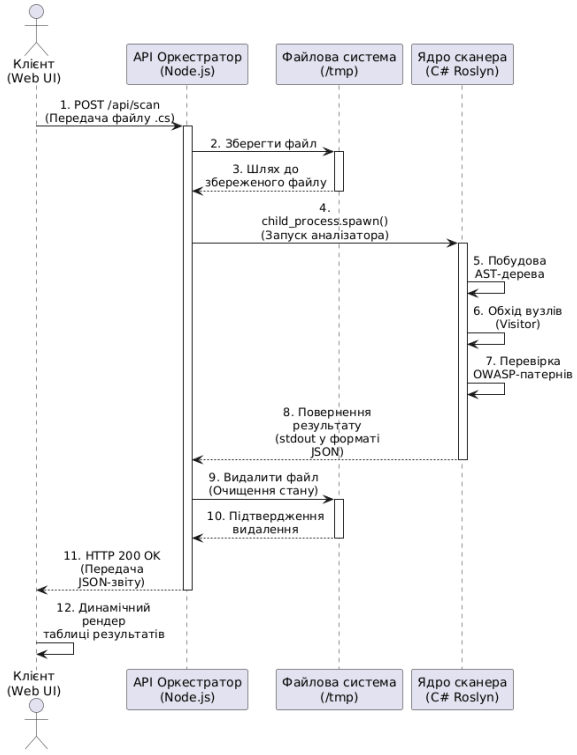


Рисунок 3.4 – Діаграма послідовності (Sequence Diagram) часової взаємодії компонентів системи під час сканування коду

Схема також демонструє роль Node.js-оркестратора як проміжної ланки, що координує роботу клієнта, файлової системи, C#-аналізатора та бази даних, а також забезпечує очищення тимчасових даних відповідно до stateless-підходу.

3.3. Вибір та обґрунтування технологій та інструментів розробки

Коли прийшов час обирати стек, я відкинув усі "модні" тренди і зосередився на тому, що дасть мені максимальну продуктивність і надійність при мінімальних витратах ресурсів.

Вибір технологічного стеку здійснювався з урахуванням архітектурної моделі, визначеної у попередніх підрозділах. Оскільки система побудована як розподілена платформа, її окремі компоненти виконують різні за характером задачі. Серверний оркестратор відповідає переважно за мережеву взаємодію, прийом файлів, маршрутизацію запитів і передавання команд скануючому ядру. Натомість обчислювальний модуль виконує більш ресурсоємну роботу, пов'язану з аналізом вихідного коду та пошуком потенційно небезпечних конструкцій.

Тому під час вибору технологій було важливо не лише оцінити їхню популярність або зручність використання, а й визначити, наскільки вони відповідають конкретним ролям у межах архітектури системи. Основними критеріями вибору стали продуктивність, підтримка асинхронної обробки, стабільність роботи, простота інтеграції, доступність бібліотек і можливість подальшого масштабування.

Серверна частина (Оркестратор): Node.js та Express. Як я вже зазначав, я обирав інструменти з розумом. Згідно з практичним посібником "Node.js in Action", головна фішка Node.js – це неблокуючий ввід-вивід (Event Loop) [25, с. 41]. Оркестратор постійно працює з файловою системою та мережевими запитами. Node.js дозволяє мені обслуговувати сотні одночасних користувачів, які завантажують файли, використовуючи лише один потік виконання, що неймовірно економно з точки зору оперативної пам'яті сервера. Окремою причиною вибору Express стала його простота та достатня гнучкість для побудови REST API.

Ядро сканера (Scanner Engine): .NET / C# та Roslyn SDK. Тут я був безкомпромісний. Мені потрібна була швидкість. Я обрав .NET, через високий рівень оптимізації, а також **Roslyn SDK**. В офіційній Microsoft сказано, що Roslyn

відкриває доступ до внутрішньої моделі компілятора, дозволяючи програмно аналізувати лексеми та будувати синтаксичні дерева [20, с. 1]. Це вищий рівень, ніж порівняння тексту регулярними виразами. Аналіз коду не як набір символів, а як об'єктів. Якщо в коді є виклик функції ExecuteQuery(), Roslyn точно каже мені: це виклик методу, Це дає мені змогу реалізувати Taint-аналіз, про який пишуть у книзі «Software Security» Гаррі Макгроу як про золотий стандарт статичного аналізу [1, с. 195].

База даних: PostgreSQL. Коли мова заходить про надійність даних, компромісів бути не може. Я обрав **PostgreSQL**, оскільки, як описано в посібнику "PostgreSQL: Introduction and Concepts", ця СУБД є стандартом де-факто для надійних ACID-транзакцій [26, с. 82]. Оскільки мій сканер працює асинхронно, потрібно зберігати стан кожної задачі, щоб нічого не втратити, якщо раптом сервер перезавантажиться. PostgreSQL дозволяє мені структурувати звіти у форматі JSONB, що дає гнучкість документоорієнтованих БД разом із надійністю реляційних.

Окрему роль у вибраному технологічному стеку відіграють засоби інтеграції між компонентами системи. Оскільки платформа побудована як розподілена структура, її окремі частини повинні мати уніфікований і передбачуваний спосіб обміну даними. Передавання результатів у форматі JSON забезпечує стандартизовану взаємодію між C#-ядром, Node.js-оркестратором і клієнтським інтерфейсом. Використання JSON також спрощує подальшу обробку результатів сканування на різних рівнях системи.

Таким чином, вибраний технологічний стек відповідає архітектурній логіці системи CodeGuard. Node.js та Express забезпечують асинхронну обробку запитів і координацію компонентів, C# та Roslyn SDK відповідають за аналіз вихідного коду, PostgreSQL використовується для збереження службових даних і метрик, а JSON і REST API забезпечують стандартизовану взаємодію між рівнями системи. Це створює збалансовану основу для реалізації легкої, розподіленої та придатної до подальшого розвитку SAST-платформи. Такий стек також спрощує подальше масштабування й підтримку системи.

3.4. Висновки по розділу

У третьому розділі було здійснено проєктування архітектури системи автоматизованого тестування безпеки програмного забезпечення. На основі сформованих раніше вимог було визначено загальну структуру системи, основні програмні компоненти та характер їхньої взаємодії.

Обґрунтовано доцільність використання розподіленої мікросервісної архітектури, у межах якої серверний оркестратор на Node.js відповідає за прийом запитів, обробку файлів та взаємодію з клієнтським рівнем, а ядро аналізатора на C# виконує безпосередню перевірку вихідного коду. Такий підхід дозволяє розділити мережеву та обчислювальну логіку, зменшити зв'язаність компонентів і підвищити гнучкість подальшого розвитку системи.

У межах UML-моделювання було побудовано діаграми прецедентів, діяльності та послідовності. Вони дали змогу формалізувати основні сценарії використання системи, описати алгоритм обробки завантаженого файлу та відобразити часову взаємодію між клієнтом, серверним оркестратором, скануючим ядром і базою даних.

Також було обгрунтовано вибір технологічного стеку, який відповідає розподіленій архітектурі системи. Використання Roslyn SDK забезпечує можливість аналізу структури вихідного коду через AST, що дозволяє працювати не лише з текстом програми, а й з її синтаксичною будовою. Node.js, своєю чергою, є доцільним рішенням для побудови асинхронного серверного рівня, оскільки ефективно обробляє I/O-операції, зокрема прийом файлів, HTTP-запити та взаємодію з дочірніми процесами. REST API забезпечує уніфіковану взаємодію між компонентами системи та створює основу для подальшої інтеграції платформи.

Отже, у третьому розділі було сформовано архітектурну основу системи, визначено її ключові компоненти, описано їхню взаємодію та обґрунтовано вибір основних технологій реалізації. Отримані результати створюють необхідну базу для переходу до практичної реалізації програмного рішення.

РОЗДІЛ 4. РЕАЛІЗАЦІЯ ПРОЄКТУ

4.1. Опис розробки програмного коду

Перехід від проєктування безпосередньо до написання коду - це завжди той момент, коли архітектурні рішення перевіряються практичною реалізацією. Маючи чітку архітектурну схему, розроблену в попередньому розділі, я розпочав фізичну реалізацію платформи CodeGuard. Оскільки для системи було обрано мікросервісний та поліглотний підхід, розробка велася паралельно у двох різних технологічних стеках: JavaScript на базі Node.js для серверної логіки та C# на платформі .NET для реалізації скануючого ядра.

Першим кроком стала організація робочого простору. Для цього було створено єдиний репозиторій на GitHub, у межах якого проєкт поділено на дві незалежні директорії: api-gateway та scanner-core. Така структура дозволила розмежувати залежності Node.js, що встановлюються через NPM, і залежності .NET, які керуються через NuGet. Загальну структуру каталогів проєкту CodeGuard у середовищі розробки наведено на рисунку 4.1.

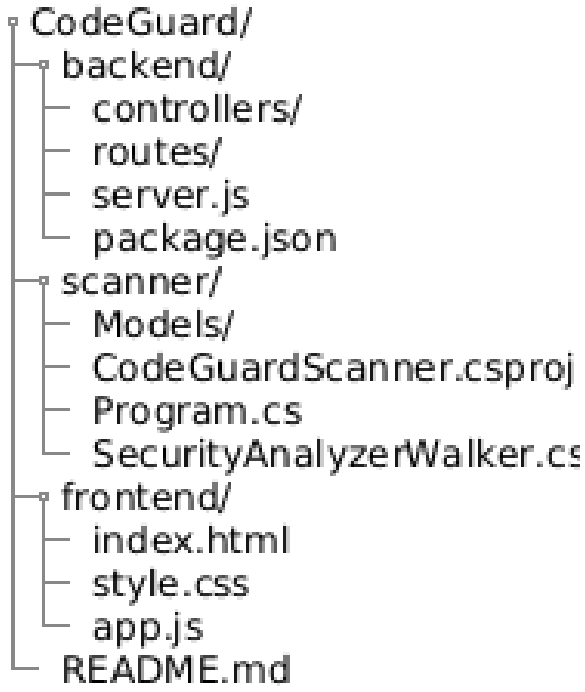


Рисунок 4.1 – Дерево каталогів проєкту CodeGuard у середовищі розробки

Згідно з обраною мікросервісною архітектурою, кодова база логічно розділена на три основні директорії: backend (містить логіку API-оркестратора на Node.js), scanner (містить обчислювальне ядро на C# та алгоритми обходу AST) та frontend (містить статичні файли клієнтського веб-інтерфейсу). Такий розподіл забезпечує високу модульність та спрощує подальше масштабування системи.

Реалізація API-оркестратора (Node.js) Розробка бекенду розпочалася зі створення базового HTTP-сервера на основі Express. Для обробки завантажених файлів було використано бібліотеку multer, яка забезпечує роботу із запитами multipart/form-data. Додатково реалізовано middleware для перевірки розширення файлу та його збереження у тимчасову папку uploads/ з унікальним іменем.

Після збереження файлу бекенд запускає скануюче ядро за допомогою модуля child_process і функції spawn. До ScannerCore.exe передається абсолютний шлях до файлу, після чого Node.js асинхронно отримує результат із stdout. Отриманий JSON парситься, записується до PostgreSQL і повертається клієнту у вигляді HTTP-відповіді.

Реалізація обчислювального ядра (C# .NET) Паралельно я взявся за розробку самого "серця" системи – аналізатора. Я створив консольний застосунок на платформі .NET. Щоб він міг розуміти вихідний код, я підключив бібліотеки **Microsoft.CodeAnalysis (Roslyn SDK)**.

Код сканера має лінійну структуру виконання. Спочатку метод Main зчитує шлях до файлу з аргументів командного рядка, після чого вміст файлу завантажується в пам'ять за допомогою File.ReadAllTextAsync. Далі текст передається до CSharpSyntaxTree.ParseText(), що дозволяє побудувати абстрактне синтаксичне дерево AST. Основна логіка пошуку вразливостей винесена в окремі класи-візитори, успадковані від CSharpSyntaxWalker. Після завершення аналізу результати серіалізуються у формат JSON за допомогою System.Text.Json і виводяться через Console.WriteLine.

Реалізація клієнтського інтерфейсу (Web UI) Для зручної взаємодії з користувачем було реалізовано веб-інтерфейс. У межах MVP-версії використано чистий JavaScript, CSS та Bootstrap без залучення складних frontend-фреймворків.

Головна сторінка містить форму Drag-and-Drop для завантаження файлу, після чого Fetch API надсилає POST-запит на бекенд. Під час обробки користувач бачить індикатор завантаження, а після завершення сканування система відображає таблицю з результатами або повідомлення про відсутність вразливостей.

Головну сторінку веб-інтерфейсу платформи CodeGuard із панеллю моніторингу наведено на рисунку 4.2.

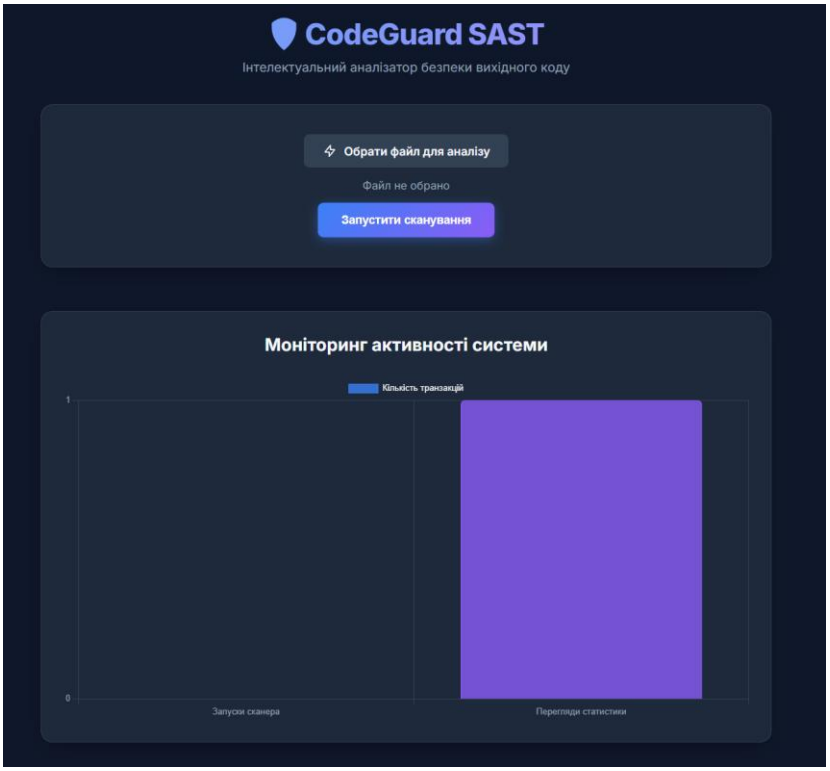


Рисунок 4.2 – Головна сторінка веб-інтерфейсу платформи CodeGuard із панеллю моніторингу.

Панель управління скануванням забезпечує основну взаємодію користувача із системою. Вона містить форму вибору файлу з вихідним кодом, кнопку запуску SAST-аналізу та динамічне відображення статусу обраного файлу.

Аналітичний дашборд призначений для візуалізації метрик використання платформи. За допомогою стовпчастої діаграми система відображає кількість запусків сканера та переглядів результатів, що дає змогу здійснювати базовий моніторинг активності бекенд-сервера.

Відображення результатів сканування та знайдених вразливостей у веб-інтерфейсі CodeGuard наведено на рисунку 4.3.

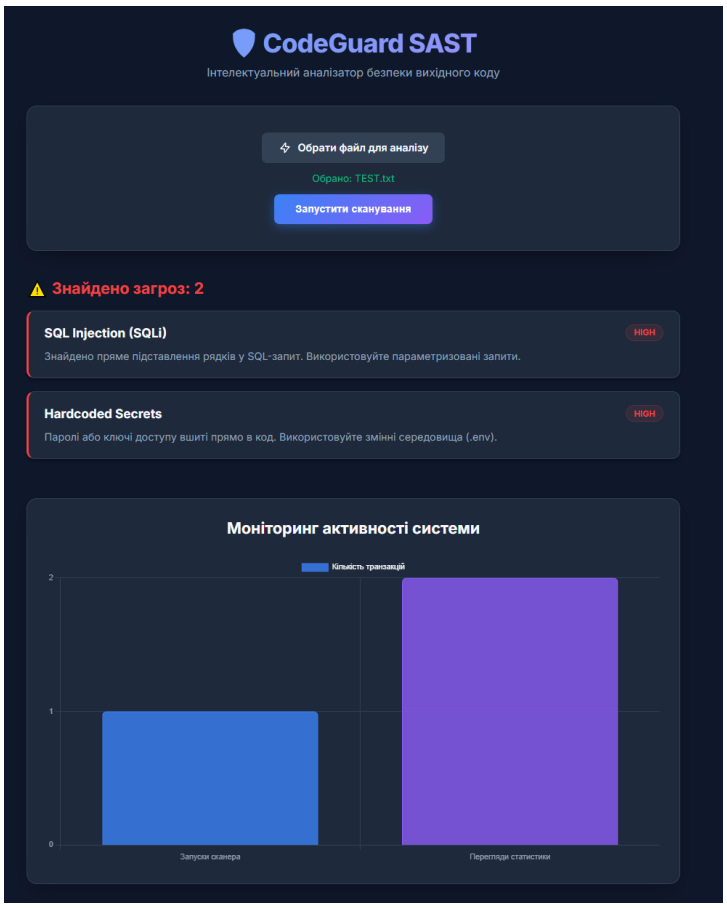


Рисунок 4.3 – Відображення результатів сканування та виявлених вразливостей у веб-інтерфейсі CodeGuard

Функціональною особливістю модуля візуалізації є подання не лише типу знайденої вразливості, а й практичної рекомендації щодо її усунення, наприклад використання параметризованих запитів або винесення секретів у змінні середовища. Це спрощує подальше виправлення коду розробником. Додатково дашборд оновлює статистику після кожного запуску сканування. Така організація інтерфейсу забезпечує зручну взаємодію з користувачем, наочне подання результатів аналізу та контроль основних метрик роботи системи. Крім того, дане поєднання дозволяє використовувати інтерфейс не лише як засіб перегляду результатів сканування, а й як інструмент оперативного аналізу загального стану безпеки вихідного коду.

4.2. Використані алгоритми та структури даних

Під час реалізації механізму пошуку вразливостей було розглянуто два можливі підходи: текстовий пошук за допомогою регулярних виразів та структурний аналіз коду. Простий пошук за ключовими словами, наприклад SELECT * FROM або password =, є недостатньо точним, оскільки не враховує контекст виконання програми і може створювати значну кількість хибнопозитивних результатів [14, с. 182].

Тому ядро сканера було реалізовано на основі абстрактного синтаксичного дерева AST. За допомогою Roslyn SDK вихідний код не обробляється як суцільний текст, а проходить лексичний і синтаксичний аналіз. У результаті формується деревоподібна структура, де окремими вузлами представлені файл, класи, методи, вирази, інструкції та лексеми [23, с. 845].

Приклад структури абстрактного синтаксичного дерева для фрагмента коду з жорстко закодованим секретом наведено на рисунку 4.4.

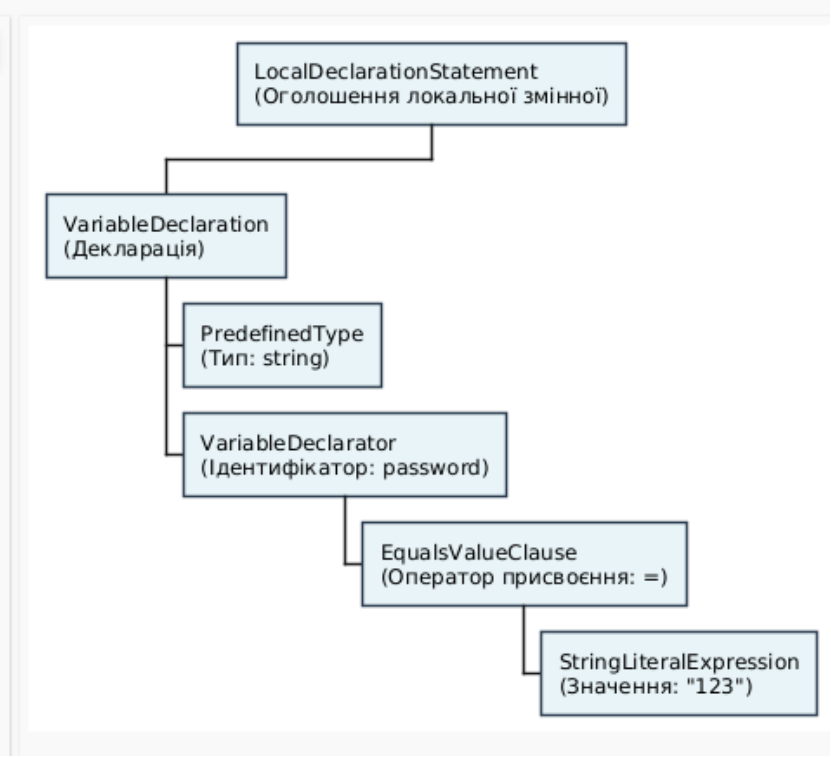


Рисунок 4.4 - Структура абстрактного синтаксичного дерева AST для фрагмента коду з жорстко закодованим секретом [23, с. 845]

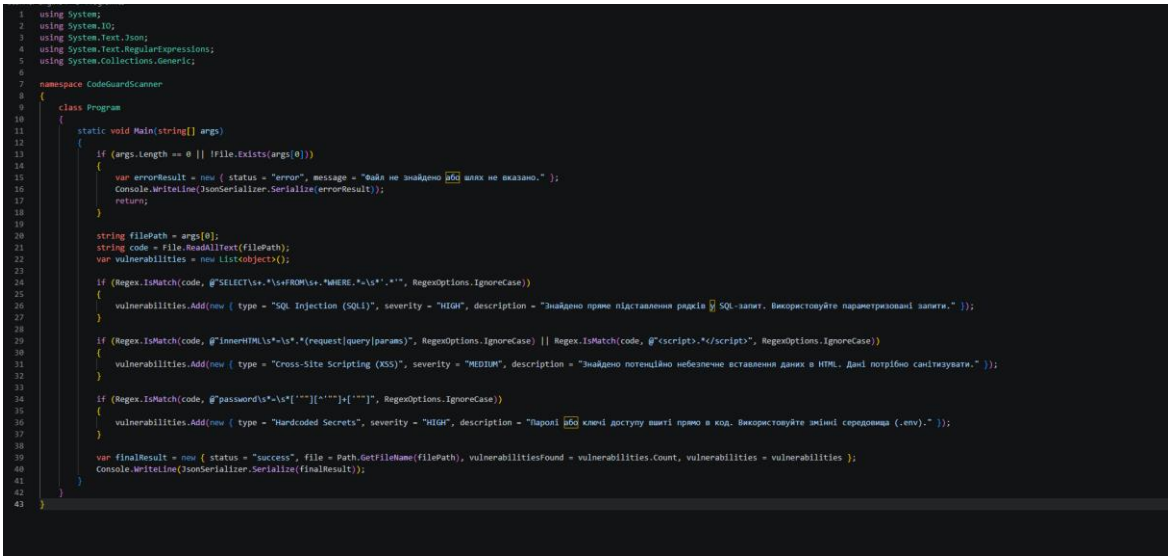
Маючи таку структуру даних, мені потрібен був ефективний алгоритм для її обходу. Написати рекурсивний обхідник складного дерева з нуля – задача нетривіальна. Тому я використав класичний шаблон проєктування «Відвідувач» (Visitor), який детально описаний у "Head First Design Patterns" [24, с. 320]. У світі Roslyn цей патерн реалізований у вигляді базового класу CSharpSyntaxWalker.

Я створив власний клас SecurityAnalyzerWalker, який успадковується від цього базового класу. Алгоритм його роботи наступний:

- Оркестратор ініціалізує Walker і передає йому корінь AST.
- Walker автоматично починає спускатися по дереву зверху вниз.
- Я не пишу код для обходу кожного типу вузла. Я просто перевизначаю (override) конкретні методи для тих типів вузлів, які мене цікавлять з точки зору безпеки.

Для реалізації MVP-версії обчислювального ядра було використано лексичний аналіз на основі патерн-матчингу із застосуванням регулярних виразів. Такий підхід забезпечує швидку обробку вхідних файлів, простоту реалізації та мінімальне споживання оперативної пам'яті.

Структуру вихідного коду обчислювального ядра сканера CodeGuardScanner наведено на рисунку 4.5.



```
using System;
using System.IO;
using System.Text.Json;
using System.Text.RegularExpressions;
using System.Collections.Generic;

namespace CodeGuardScanner
{
    class Program
    {
        static void Main(string[] args)
        {
            if (args.Length == 0 || !File.Exists(args[0]))
            {
                var errorResult = new { status = "error", message = "Файл не знайдено або шлях не вказано." };
                Console.WriteLine(JsonSerializer.Serialize(errorResult));
                return;
            }

            string filePath = args[0];
            string code = File.ReadAllText(filePath);
            var vulnerabilities = new List<object>();

            if (Regex.IsMatch(code, @"SELECT\s+.*\s+FROM\s+.*WHERE.*=\s*'.*'", RegexOptions.IgnoreCase))
            {
                vulnerabilities.Add(new { type = "SQL Injection (SQLi)", severity = "HIGH", description = "Знайдено пряме підставлення рядків у SQL-запит. Використовуйте параметризовані запити." });
            }

            if (Regex.IsMatch(code, @"innerHTML\s*=\s*.*(request|query|params)", RegexOptions.IgnoreCase) || Regex.IsMatch(code, @"<script>.*</script>", RegexOptions.IgnoreCase))
            {
                vulnerabilities.Add(new { type = "Cross-Site Scripting (XSS)", severity = "MEDIUM", description = "Знайдено потенційно небезпечне вставлення даних в HTML. Дані потрібно санітизувати." });
            }

            if (Regex.IsMatch(code, @"password\s*=\s*['""][^'""]+['""]", RegexOptions.IgnoreCase))
            {
                vulnerabilities.Add(new { type = "Hardcoded Secrets", severity = "HIGH", description = "Паролі або ключі доступу вшиті прямо в код. Використовуйте змінні середовища (.env)." });
            }

            var finalResult = new { status = "success", file = Path.GetFileName(filePath), vulnerabilitiesFound = vulnerabilities.Count, vulnerabilities = vulnerabilities };
            Console.WriteLine(JsonSerializer.Serialize(finalResult));
        }
    }
}
```

Рисунок 4.5 - Структура вихідного коду обчислювального ядра сканера на базі регулярних виразів

Для виявлення різних типів загроз (OWASP Top 10) я реалізував різні алгоритмічні патерни аналізу:

Алгоритм пошуку SQL-ін'єкцій (SQLi): Головна ознака класичної SQL-ін'єкції – це формування SQL-запиту до бази даних шляхом конкатенації (склеювання) рядків із неперевіреними користувацькими даними. Мій алгоритм працює так: Я перевизначаю метод VisitInvocationExpression, який спрацьовує щоразу, коли в дереві зустрічається виклик будь-якої функції. Алгоритм перевіряє, чи належить ця функція до списку "небезпечних раковин" (Vulnerable Sinks) – наприклад, ExecuteQuery, SqlCommand.ExecuteReader або FromSqlRaw (якщо використовується Entity Framework). Якщо це виклик до БД, алгоритм починає аналізувати аргументи, передані в цю функцію. Якщо аргумент є чистим текстовим літералом (LiteralExpression) або параметризованим запитом – все добре. Але якщо алгоритм бачить, що аргументом є BinaryExpression (операція склеювання +) або InterpolatedStringExpression (рядкова інтерполяція $"{var}"), і при цьому одна з частин є змінною, – алгоритм фіксує критичну вразливість SQL Injection.

Алгоритм виявлення жорстко закодованих секретів (Hardcoded Secrets): Цей алгоритм працює на рівні присвоєння змінних. Я перевизначаю метод VisitVariableDeclarator (коли створюється змінна) та VisitAssignmentExpression (коли їй присвоюється значення). Алгоритм отримує ім'я змінної. Далі він використовує набір лексичних сигнатур (масив ключових слів: "password", "secret", "api_key", "token", "connection_string"). Якщо ім'я змінної містить щось із цього списку, алгоритм перевіряє праву частину виразу. Якщо правій частині присвоюється не виклик функції доступу до змінних середовища (наприклад, Environment.GetEnvironmentVariable), а звичайний, зашитий у код рядок тексту (StringLiteral) – сканер миттєво генерує попередження про Hardcoded Secret.

Базовий Taint-аналіз (Аналіз потоку забруднених даних): Звичайно, для повноцінної роботи я спробував закласти основи Taint-аналізу. Як пише Гаррі Макгроу у фундаментальній праці «Software Security», Taint-аналіз – це метод, що

дозволяє відстежити шлях даних від неперевіреного джерела (Source) до критичної точки виконання (Sink) [1, с. 195]. У межах бакалаврської роботи я реалізував спрощений внутрішньопроцедурний (intraprocedural) Taint-аналіз. Мій алгоритм зберігає імена змінних, які отримують дані з HTTP-запитів (наприклад, з Request.Query або Request.Form), у спеціальну структуру даних HashSet<string> (набір забруднених змінних). Під час подальшого обходу дерева, якщо алгоритм зустрічає використання змінної з цього набору всередині небезпечної функції (наприклад, виведення її в HTML без екранування, що призводить до XSS), він фіксує вразливість.

Усі знайдені дефекти алгоритм упаковує у список об'єктів власного класу VulnerabilityReport, який наприкінці роботи серіалізується у формат JSON і повертається оркестратору. Використання структури AST та патерна Visitor дозволило зробити мій алгоритм надзвичайно швидким – він аналізує тисячі рядків коду за мілісекунди без необхідності його компіляції.

Таким чином, у підрозділі було розглянуто основні алгоритмічні рішення, використані для реалізації обчислювального ядра системи. Основу аналізу становить обробка вихідного коду з подальшим пошуком потенційно небезпечних конструкцій, які можуть свідчити про наявність типових вразливостей. Такий підхід дозволяє виконувати первинну автоматизовану перевірку коду без необхідності повного розгортання або запуску застосунку.

Використання структурованого аналізу коду та механізмів зіставлення з визначеними шаблонами забезпечує достатню швидкість роботи MVP-версії платформи. При цьому результати аналізу подаються у форматі, придатному для подальшої обробки серверним оркестратором і відображення у веб-інтерфейсі користувача.

Отже, обрані алгоритми та структури даних відповідають поставленим вимогам до продуктивності, простоти інтеграції та практичної придатності системи. Вони забезпечують функціональну основу для виявлення базових класів вразливостей і створюють можливість для подальшого розширення аналізатора новими правилами перевірки.

4.3. Модульне тестування та налагодження

Після завершення реалізації основного коду серверного оркестратора та обчислювального ядра було виконано перехід до етапу тестування і налагодження системи. Цей етап є критично важливим, оскільки саме він дозволяє перевірити не лише працездатність окремих модулів, а й коректність їхньої взаємодії в межах загального процесу сканування.

Особливу увагу було приділено стабільності роботи системи та передбачуваності отриманих результатів. Як зазначають Лорі Вільямс та Гаррі Макгроу у дослідженні, присвяченому статичному аналізу безпеки, інструмент, який працює нестабільно, видає суперечливі результати або завершується помилкою під час перевірки, швидко втрачає довіру розробників і перестає використовуватися на практиці [13, с. 78].

Оскільки розроблена система належить до класу SAST-інструментів, довіра до результатів аналізу є одним із ключових показників її якості. Саме тому тестування було спрямоване не лише на пошук технічних помилок, а й на перевірку того, чи здатна система стабільно обробляти вхідні файли, коректно виявляти задані типи вразливостей і повертати результати у зрозумілому структурованому форматі.

Процес тестування я розділив на дві паралельні гілки, відповідно до моєї мікросервісної архітектури.

Тестування обчислювального ядра (Scanner Engine на C#)

Для тестування обчислювального ядра було використано фреймворк xUnit. Оскільки ядро працює за принципом чистої функції, тобто отримує на вхід текст коду і повертає масив знайдених вразливостей, воно добре підходить для модульного тестування.

У Visual Studio було створено окремий тестовий проєкт із набором спеціалізованих mock-фрагментів коду. Кожен із них містив окремий сценарій перевірки, наприклад небезпечну SQL-конкатенацію або безпечний параметризований запит.

Під час тестування було виявлено проблему хибнопозитивних спрацьовувань, що є типовим недоліком SAST-систем [12, с. 89]. Початковий Regex-алгоритм помилково визначав як SQL-ін'єкцію будь-яку конкатенацію рядків із ключовим словом SELECT, навіть якщо вона використовувалася лише для покращення читабельності коду.

Для усунення проблеми було використано дебагер Visual Studio. У результаті регулярний вираз було уточнено: крім факту конкатенації, він почав враховувати наявність динамічних змінних у SQL-запиті. Це дозволило зменшити кількість хибних спрацьовувань і підвищити точність аналізу.

Тестування API-оркестратора (Node.js)

Тестування серверної частини платформи базувалося на інших принципах порівняно з обчислювальним ядром. Основний фокус було спрямовано на перевірку мережевої стійкості, безпеку операцій вводу/виводу (I/O) та обробку крайових випадків (Edge Cases).

З метою перевірки відмовостійкості системи було розроблено та проведено серію негативних тестів, що імітують нетипову або зловмисну поведінку клієнта:

- **Завантаження невалідних форматів:** здійснено спробу передачі виконуваних файлів (наприклад, .exe) або документів (.pdf) замість цільових файлів із вихідним кодом (.cs). API-оркестратор успішно відхиляв подібні запити на рівні проміжного програмного забезпечення (middleware), повертаючи клієнту статус-код HTTP 400 Bad Request.
- **Тестування на відмову від обслуговування (File Upload DoS):** проведено симуляцію завантаження цільового файлу об'ємом понад 500 МБ. Для запобігання вичерпанню оперативної пам'яті (Out of Memory) на сервері було сконфігуровано жорсткі ліміти в бібліотеці multer. Це дозволило системі миттєво розривати мережеве з'єднання при перевищенні встановленого порогу в 50 МБ.
- **Емуляція "зависання" дочірнього процесу:** перевірено сценарій, за якого консольний C#-сканер потрапляє у нескінченний цикл або блокується. Для захисту оркестратора на рівні Node.js було імплементовано програмний механізм таймаутів. Якщо дочірній процес (child_process.spawn) не повертає результат

протягом 30 секунд, оркестратор примусово завершує його роботу (надсилаючи системний сигнал SIGTERM) та повертає помилку Timeout Exceeded. Це гарантує безперебійну доступність ресурсів сервера та унеможливлює їх вічне блокування.

Такий комплексний підхід до тестування (як алгоритмів лексичного аналізу на базі регулярних виразів, так і мережевої взаємодії на рівні API) дозволив стабілізувати платформу та повністю підготувати її до фінальних системних випробувань.

Таким чином, у підрозділі було розглянуто процес модульного тестування та налагодження основних компонентів системи. Особливу увагу було приділено перевірці обчислювального ядра, оскільки саме воно відповідає за виявлення потенційних вразливостей у вихідному коді. Використання підготовлених тестових фрагментів дозволило перевірити роботу алгоритмів у контрольованих умовах.

Під час тестування було виявлено окремі недоліки в логіці аналізу, зокрема хибнопозитивні спрацьовування для безпечних фрагментів коду. Їх усунення дало змогу уточнити правила пошуку вразливостей і підвищити точність роботи сканера. Це є важливим для SAST-систем, оскільки надмірна кількість помилкових попереджень знижує практичну цінність інструменту для розробника.

Отже, проведене тестування та налагодження підтвердили працездатність реалізованих алгоритмів і стабільність основних сценаріїв роботи системи. У процесі перевірки було встановлено, що скануюче ядро коректно обробляє підготовлені тестові файли, виявляє передбачені класи вразливостей і формує результати у структурованому вигляді. Окремі недоліки, виявлені під час тестування, були усунені на етапі налагодження, що дозволило підвищити точність і передбачуваність роботи аналізатора. Отримані результати створили основу для подальшого комплексного тестування платформи, зокрема перевірки клієнтського інтерфейсу, backend API та поведінки системи під значним навантаженням. Це підтвердило готовність системи до наступного етапу перевірки.

4.4.. Висновки по розділу

У четвертому розділі було описано практичну реалізацію платформи автоматизованого тестування безпеки програмного забезпечення. На основі спроєктованої архітектури було реалізовано серверний оркестратор, обчислювальне ядро сканера та клієнтський веб-інтерфейс.

Серверну частину створено на базі Node.js та Express. Вона забезпечує прийом і валідацію файлів, запуск скануючого ядра, обробку результатів та повернення відповіді клієнту у форматі JSON. Обчислювальне ядро реалізовано мовою C# на платформі .NET і спрямовано на виявлення базових типів вразливостей, зокрема SQL-ін'єкцій.

Для взаємодії з користувачем було розроблено веб-інтерфейс, який дозволяє завантажувати файли, запускати сканування та переглядати результати аналізу. Окремо було проведено тестування системи за допомогою xUnit і Postman, що підтвердило коректність роботи основних компонентів.

Таким чином, у четвертому розділі було реалізовано MVP-версію системи, яка виконує основні функції статичного аналізу коду та створює основу для подальшого оцінювання її ефективності..

Важливим результатом реалізації стало забезпечення узгодженої взаємодії між усіма компонентами платформи. Серверний оркестратор, скануюче ядро та клієнтський інтерфейс були об'єднані в єдиний цикл обробки запиту: від завантаження файлу користувачем до отримання структурованого звіту про знайдені вразливості.

Окремо слід зазначити, що реалізована MVP-версія зберігає потенціал для подальшого розвитку системи. Архітектура системи дозволяє розширювати набір правил аналізу, додавати нові типи вразливостей, удосконалювати механізми фільтрації хибнопозитивних спрацьовувань та інтегрувати платформу з іншими інструментами розробки. Це робить створене рішення не лише завершеним прототипом, а й основою для подальшого вдосконалення SAST-системи в межах майбутніх етапів розвитку.

РОЗДІЛ 5. ТЕСТУВАННЯ ТА ОЦІНКА ЯКОСТІ ПРОГРАМНОЇ СИСТЕМИ

5.1. Організація системного тестування

Тестування є одним із ключових етапів життєвого циклу розробки програмного забезпечення, оскільки саме на цьому етапі здійснюється перевірка відповідності реалізованої системи визначеним функціональним і нефункціональним вимогам. Для розробленої SAST-платформи CodeGuard процес тестування мав особливе значення, оскільки система складається з кількох взаємопов'язаних компонентів: серверного оркестратора на базі Node.js, ізольованого обчислювального ядра на C# та клієнтського веб-інтерфейсу.

Архітектурна особливість системи полягає в тому, що обчислювальне ядро не є частиною основного серверного процесу, а запускається як окремий дочірній процес. Взаємодія між Node.js-оркестратором і C#-сканером здійснюється через стандартні потоки вводу та виводу. Саме тому тестування мало охоплювати не лише коректність роботи окремих модулів, а й стабільність обміну даними між ними.

Основною метою системного тестування було підтвердження того, що всі компоненти платформи працюють узгоджено та забезпечують повний цикл обробки запиту. Система повинна коректно приймати файл від користувача, виконувати його первинну валідацію, передавати файл до скануючого ядра, отримувати результати аналізу, зберігати службові дані та повертати користувачу структуровану відповідь. Окремо перевірялася здатність сканера виявляти задекларовані типи вразливостей, зокрема SQL-ін'єкції, XSS та жорстко закодовані секрети.

Стратегія тестування будувалася за принципом поступового ускладнення: від перевірки окремих модулів до оцінювання роботи системи в цілому. На першому етапі було виконано модульне тестування обчислювального ядра за допомогою фреймворку xUnit. Воно дозволило перевірити правильність роботи

алгоритмів виявлення вразливостей на ізольованих фрагментах коду. На другому етапі проводилося функціональне тестування, спрямоване на перевірку роботи MVP-версії з позиції кінцевого користувача, зокрема завантаження файлів, відображення результатів і обробки помилкових сценаріїв.

Наступним етапом стало інтеграційне тестування. Його метою була перевірка коректності взаємодії між клієнтською частиною, backend-API та скануючим ядром. Для цього використовувався інструмент Postman, за допомогою якого перевірялися HTTP-запити, коди відповідей сервера, структура JSON-відповідей та поведінка системи у випадку некоректних вхідних даних. Окремо аналізувалася взаємодія оркестратора з дочірнім процесом сканера, оскільки помилки на цьому рівні могли вплинути на стабільність усієї платформи.

Також було проведено навантажувальне тестування, метою якого була оцінка здатності Node.js-сервера обробляти кілька паралельних запитів без критичного зростання використання оперативної пам'яті або зависання процесів. Особлива увага приділялася перевірці роботи з файлами різного розміру, повторним запуском сканування та коректному завершенню дочірніх процесів після виконання аналізу.

Для забезпечення об'єктивності тестування було підготовлено спеціальний набір тестових артефактів - mock-файлів із фрагментами вихідного коду мовою C#. Частина цих файлів містила безпечні конструкції, а частина - навмисно додані вразливості. Такий підхід дозволив перевірити не лише здатність сканера знаходити небезпечні фрагменти, а й його вміння не реагувати на безпечний код. Це було важливо для зменшення кількості хибнопозитивних спрацьовувань.

Таким чином, організована стратегія тестування дозволила комплексно оцінити роботу платформи CodeGuard на різних рівнях: від окремих алгоритмів аналізу до повного сценарію взаємодії користувача із системою. Отримані результати стали основою для подальшого оцінювання ефективності, стабільності та практичної придатності розробленого застосунку. Це забезпечило послідовність і повноту перевірки системи, а також підвищило достовірність отриманих результатів.

5.2. Функціональне та ручне тестування MVP

Функціональне тестування клієнтської частини було спрямоване на перевірку коректної взаємодії системи з кінцевим користувачем. Оскільки платформа CodeGuard реалізована як MVP-версія, основну увагу було приділено ключовим користувацьким сценаріям та стійкості інтерфейсу до нестандартних дій.

Для перевірки веб-інтерфейсу було використано задокументовані тест-кейси та елементи дослідницького тестування. Це дозволило змоделювати поведінку користувача під час роботи із системою, зокрема завантаження неправильних файлів, повторні натискання кнопок або спроби перервати процес сканування.

Функціональне тестування проводилося з позиції кінцевого користувача, який не має доступу до внутрішньої логіки серверної частини або скануючого ядра. Основним завданням було перевірити, чи здатна система виконати повний користувацький сценарій: вибір файлу, запуск сканування, очікування результату, отримання відповіді та перегляд знайдених вразливостей у зрозумілому вигляді. Такий підхід дозволив оцінити не лише технічну працездатність окремих функцій, а й загальну зручність використання MVP-версії.

Для підготовки тестових сценаріїв було виділено кілька типових груп вхідних даних. До першої групи належали коректні C#-файли без вразливостей, які використовувалися для перевірки того, чи система не генерує зайві попередження. До другої групи входили файли з навмисно доданими небезпечними конструкціями, зокрема SQL-конкатенаціями, XSS-фрагментами та жорстко закодованими секретами. Третю групу становили некоректні або нестандартні файли, які дозволяли перевірити стійкість інтерфейсу та backend-логіки до помилкових дій користувача. Такий поділ тестових даних дозволив перевірити не лише стандартний сценарій успішного сканування, а й поведінку системи в умовах помилкових або нестандартних вхідних даних. Це забезпечило ширше покриття можливих сценаріїв використання системи.

Окремо перевірялися позитивні та негативні сценарії використання. Позитивні сценарії передбачали завантаження валідного файлу, успішний запуск сканування, отримання JSON-відповіді від сервера та коректне відображення результатів у веб-інтерфейсі. Негативні сценарії охоплювали спроби завантаження файлів неправильного формату, повторне натискання кнопки сканування, запуск перевірки без вибраного файлу та зміну вибраного файлу перед повторним скануванням. Це дозволило переконатися, що система не переходить у некоректний стан і надає користувачу зрозумілий зворотний зв'язок.

Важливим критерієм успішності було не лише правильне виконання основної функції, а й передбачувана поведінка інтерфейсу під час нестандартних ситуацій. Наприклад, якщо користувач не вибрав файл, система повинна повідомити про необхідність його завантаження, а не надсилати порожній запит на сервер. Якщо файл не відповідає очікуваному формату, користувач має отримати повідомлення про помилку без зависання сторінки або втрати попереднього стану інтерфейсу.

Також перевірялася послідовність зміни станів клієнтської частини. Після вибору файлу інтерфейс повинен відобразити його назву або інший візуальний індикатор готовності до сканування. Після натискання кнопки запуску має з'явитися стан очікування, який демонструє, що запит обробляється. Після завершення аналізу цей стан повинен зникнути, а на його місці мають бути відображені результати сканування або повідомлення про відсутність знайдених вразливостей.

Особливу увагу було приділено повторюваності результатів. Один і той самий тестовий файл запускався кілька разів поспіль, щоб перевірити, чи не накопичуються попередні результати в DOM-структурі сторінки та чи не виникають дублікати повідомлень. Це важливо для практичного використання системи, оскільки розробник може багаторазово змінювати код і повторно запускати перевірку після кожного виправлення. Це дозволило переконатися в коректному оновленні результатів після запуску сканування. Це підтвердило стабільність механізму повторного відображення результатів.

Функціональне тестування також дало змогу оцінити зрозумілість результатів для користувача. Для кожної знайденої проблеми інтерфейс має відображати не лише сам факт виявлення вразливості, а й її тип, рівень критичності та короткий опис. Такий формат подання результатів є важливим для SAST-систем, оскільки розробник повинен швидко зрозуміти, яку саме проблему потрібно виправити і наскільки вона є небезпечною.

Отже, функціональне та ручне тестування MVP-версії було спрямоване на перевірку повного користувацького сценарію роботи з платформою. Отримані результати підтвердили, що веб-інтерфейс коректно реагує на типові та нестандартні дії користувача, забезпечує зрозуміле відображення результатів і не порушує логіку роботи системи під час повторних запусків сканування.

У межах тестування було перевірено основні компоненти веб-інтерфейсу:

1. Валідація модуля завантаження файлів (File Upload Module)

Перевірка на рівні MIME-типів та розширень: Окрім базового завантаження валідних файлів .cs, було проведено тестування спроб обходу клієнтської валідації. Наприклад, було створено звичайний текстовий документ .txt із довільним текстом, якому вручну змінили розширення на .cs. Система успішно перехопила цей файл і, хоча фронтенд пропустив його до API, бекенд повернув помилку валідації коду, яку інтерфейс коректно відобразив користувачеві.

Поведінка при скасуванні вибору: Перевірявся сценарій, коли користувач відкриває вікно вибору файлу, але натискає «Скасувати». Інтерфейс успішно зберігав попередній стан і не видавав помилок типу NullReferenceException на клієнті.

2. Управління станом очікування (Loading State Management). Оскільки SAST-сканування може займати певний час залежно від розміру файлу, критично важливим було протестувати поведінку UI під час транзакції.

Захист від дублювання запитів (Double-Submit Prevention): Перевірялося, чи блокується кнопка «Запустити сканування» після першого натискання. Це гарантує, що нетерплячий користувач не відправить 10 однакових файлів на

сервер-оркестратор одночасно, спричинивши штучне навантаження (клієнтський DoS).

Візуальний зворотний зв'язок: Інтерфейс коректно відображав анімацію завантаження, сигналізуючи розробнику, що аналіз триває.

3. Динамічний рендеринг результатів сканування (DOM Rendering)

Обробка безпечного коду: Сценарій, за якого користувач завантажує ідеально чистий код без жодної вразливості. Інтерфейс успішно обробляв порожній масив загроз від сервера та виводив зелений банер із повідомленням «Вразливостей не виявлено», замість того, щоб показувати порожню таблицю.

Стрес-тест візуалізації: Було завантажено спеціальний «mock-файл», у якому було штучно згенеровано 50+ різних вразливостей. Перевірялося, чи не "ламається" верстка сторінки. Контейнер із результатами коректно застосував внутрішній скрол (overflow-y: auto), зберігши читабельність сторінки.

Маркування критичності: Ручна перевірка того, що бейджі (Badges) правильно змінюють колір (червоний для HIGH, жовтий для MEDIUM) залежно від рівня загрози, який надійшов у JSON-відповіді.

4. Кросбраузерна сумісність (Cross-Browser Compatibility) Кросбраузерну сумісність MVP-версії було перевірено у Google Chrome, Mozilla Firefox та Microsoft Edge. У всіх браузерах інтерфейс, Flexbox-контейнери дашборду та темна тема відображалися коректно, без зміщення елементів і проблем із читабельністю.

Під час дослідницького тестування було виявлено дефект управління станом клієнтської частини: при повторному завантаженні файлу результати попереднього сканування залишалися в інтерфейсі, а нові дані додавалися до вже наявного списку. Через це користувач міг бачити змішані результати кількох різних перевірок, що створювало ризик помилкового сприйняття результатів аналізу. Проблему було усунуто шляхом примусового очищення контейнера результатів перед кожним новим fetch-запитом. Після внесення змін кожне сканування почало відображатися як окрема завершена операція, що забезпечило коректне оновлення даних під час послідовних перевірок файлів.

5.3. Інтеграційне тестування backend API

Наступним логічним етапом після функціональної перевірки ізольованих компонентів клієнтської частини стало інтеграційне тестування (Integration Testing). У сучасній компонентно-орієнтованій архітектурі цей вид перевірки є критично важливим, оскільки більшість програмних збоїв та вразливостей виникає саме на стику взаємодії різних модулів. У контексті розробленої платформи CodeGuard головним об'єктом інтеграційного тестування виступав сервер-оркестратор на базі середовища Node.js (із використанням фреймворку Express). Цей вузол виконує роль сполучної ланки між клієнтським веб-інтерфейсом та обчислювальним ядром на C#.

Головна мета даного етапу полягала у валідації API-контрактів, перевірці коректності передачі даних через HTTP-протокол та оцінці стабільності міжпроцесної взаємодії (Inter-Process Communication, IPC). Для стандартизації та автоматизації процесу було застосовано інструментальне середовище Postman. За його допомогою автором розроблено комплексну тестову колекцію, що покриває як позитивні (Happy Path), так і негативні (Edge Cases) сценарії роботи сервера.

Процес інтеграційного тестування було розділено на три основні вектори:

1. Валідація API-контрактів та формату даних (Contract Testing) Було перевірено дотримання архітектурних принципів REST під час обробки запитів на головний маршрут POST /api/scan. Оскільки обмін даними між фронтендом і сервером відбувається у форматі multipart/form-data (через необхідність передачі файлу вихідного коду), а повернення результатів вимагається у форматі application/json, тестувалася правильність серіалізації та десеріалізації об'єктів. За допомогою скриптів автоматизації Postman (Assertions, написаних на JavaScript) було підтверджено, що при успішному скануванні бекенд гарантовано повертає стандартизований JSON-об'єкт. Тести перевіряли наявність обов'язкових полів: status, file та масиву vulnerabilities. Це підтвердило узгодженість API-відповіді з очікуваною структурою даних. Це підтвердило коректність реалізації API-контракту.

2. Тестування обробки винятків та HTTP статус-кодів (Error Handling) Надійний сервер-оркестратор повинен передбачувано реагувати на некоректні дії клієнта. Для перевірки стійкості API було проведено серію негативних тестів:

Відправка неповного запиту: імітовано відправку HTTP-запиту без прикріпленого артефакту. Тести підтвердили, що сервер не завершує роботу аварійно (Crash), а перехоплює помилку на рівні маршрутизатора і повертає статус 400 Bad Request із відповідним повідомленням.

Перевірка обмежень middleware: тестувалася бібліотека Multer, яка відповідає за обробку файлів. При імітації завантаження файлу, об'єм якого перевищує встановлений ліміт (50 МБ), сервер коректно обривав мережеве з'єднання та повертав клієнту код помилки 413 Payload Too Large, що є базовим захистом від клієнтських атак типу DoS (Denial of Service).

3. Тестування міжпроцесної взаємодії (IPC) з обчислювальним ядром Оскільки Node.js викликає сканер коду на C# як ізольований дочірній процес через вбудований модуль child_process.spawn, тестування фокусувалося на перевірці асинхронного зчитування даних зі стандартного потоку виведення (stdout). Особливу увагу було приділено крайовим сценаріям: було штучно імітовано "зависання" C#-ядра у нескінченному циклі. Інтеграційні тести довели ефективність реалізованого на бекенді механізму таймаутів. Було підтверджено, що якщо дочірній процес не повертає результат протягом 30 секунд, оркестратор примусово знищує його системним сигналом SIGTERM і повертає клієнту статус 504 Gateway Timeout. Це повністю унеможливлює витік системних ресурсів (Memory/CPU Leaks) через "завислі" процеси аналізатора.

Проведене інтеграційне тестування дозволило виявити та усунути слабкі місця на рівні маршрутизації, обробки HTTP-запитів і взаємодії між серверним оркестратором та скануючим ядром. Усунення виявлених недоліків підвищило надійність backend-рівня та забезпечило передбачувану поведінку системи під час типових і помилкових сценаріїв. Це створило необхідну основу для переходу до наступного етапу перевірки – навантажувального тестування. Тож було підтверджено готовність системи до перевірки при навантаженні.

5.4. Навантажувальне тестування та оцінка готовності до впровадження

Навантажувальне тестування є важливим етапом валідації програмного забезпечення, оскільки воно дозволяє оцінити поведінку системи в умовах підвищеного або пікового навантаження. На відміну від функціонального тестування, яке перевіряє правильність виконання окремих сценаріїв, навантажувальне тестування спрямоване на визначення того, наскільки стабільно система працює при одночасному використанні кількома користувачами. Для платформи CodeGuard цей етап мав особливе значення, оскільки система обробляє файли з вихідним кодом, запускає окремі процеси сканування та повинна повертати результат без критичних затримок.

Архітектура розробленої платформи також зумовила необхідність окремої перевірки продуктивності. Серверний оркестратор реалізовано на базі Node.js, який використовує однопотокову модель виконання та механізм Event Loop. Такий підхід є ефективним для обробки I/O операцій, однак потребує перевірки в умовах паралельних запитів. Додатковим фактором навантаження є запуск обчислювального ядра на C# як дочірнього процесу через child_process.spawn. Кожен такий запуск потребує системних ресурсів, тому важливо було перевірити, чи не виникають затримки, витоки пам'яті або накопичення незавершених процесів.

Головною метою навантажувального тестування було визначення максимальної кількості одночасних запитів, які система здатна обробити без критичного зниження швидкодії. Також перевірялося, чи зберігає сервер стабільність під час тривалої роботи, чи коректно завершує дочірні процеси сканера та чи не відбувається поступове зростання використання оперативної пам'яті. Окремо оцінювався час відповіді системи, оскільки для SAST-інструменту важливо забезпечити швидкий зворотний зв'язок для розробника.Це дозволило оцінити не лише продуктивність, а й загальну стійкість роботи системи.

Для генерації тестового навантаження було використано інструмент Apache

JMeter. У межах тестового сценарію імітувалася робота групи користувачів, які одночасно надсилають файли з вихідним кодом на перевірку. Такий підхід дозволив наблизити умови тестування до реального сценарію використання платформи в команді розробників.

Як тестовий артефакт використовувався файл коду мовою C# розміром близько 50 КБ, що містив приблизно 2000 рядків. У цьому файлі були представлені типові синтаксичні конструкції, а також навмисно додані вразливості, які мав виявити сканер. Використання такого файлу дозволило перевірити не лише здатність сервера приймати паралельні запити, а й роботу повного циклу обробки: завантаження файлу, запуск сканера, аналіз коду, формування JSON-відповіді та повернення результату клієнту. Тестування проводилося за двома основними профілями:

- **Базове навантаження (Baseline Testing):** від 1 до 5 одночасних користувачів. Мета – фіксація еталонного часу відгуку системи в ідеальних умовах без конкуренції за ресурси.
- **Стресове навантаження (Stress Testing):** поступове збільшення кількості паралельних потоків (Ramp-up) від 10 до 50 одночасних підключень протягом короткого проміжку часу.

Аналіз результатів та метрик продуктивності Під час виконання тестових сценаріїв здійснювався безперервний моніторинг апаратних ресурсів сервера. Отримані результати дозволили сформувати наступну картину продуктивності:

- **Пропускна здатність та час відгуку (Response Time):** При базовому навантаженні (до 15-20 конкурентних запитів) середній час повної обробки одного файлу складав 1.2–2.5 секунди. Асинхронна архітектура Node.js довела свою ефективність: головний потік успішно делегував виконання важких операцій лексичного аналізу операційній системі, не блокуючи при цьому прийом нових HTTP-підключень від інших клієнтів. Це підтвердило здатність системи обробляти паралельні запити без критичного блокування.
- **Використання оперативної пам'яті**: Під час стресового

навантаження споживання RAM зростало пропорційно кількості активних C#-процесів. Після завершення транзакцій пам'ять коректно вивільнялася механізмами Garbage Collector, тому витоків пам'яті зафіксовано не було.

- **Завантаження процесора та межа деградації:** Основним обмежувальним фактором виявився процесорний час, оскільки Regex-аналіз є CPU-bound задачею. При 35-40 одночасних запитах CPU наближався до 100%, що спричиняло черги на рівні OS Scheduler і збільшення часу відгуку до 10-15 секунд.

Оцінка готовності до впровадження Проведені системні випробування засвідчили, що створена MVP-версія платформи CodeGuard є повністю працездатною, стабільною і безпечною в межах закладених архітектурних рішень. Система успішно обробляє помилки мережевого рівня, має вбудований захист від перевантажень (ліміти на розмір файлів та таймаути життєвого циклу процесів) і здатна безвідмовно обслуговувати щоденні потреби невеликої команди розробників (Small-to-Medium Team).

Водночас, результати навантажувального тестування чітко окреслюють вектори для подальшого масштабування. Для переходу від MVP до рівня високомасштабованого корпоративного рішення (Enterprise) поточна архітектура потребуватиме модернізації: відмови від створення нових процесів (child_process.spawn) для кожного запиту на користь пулу постійно активних обчислювальних воркерів (Worker Pool) або мікросервісної черги (наприклад, RabbitMQ чи Kafka), а також контейнеризації через Docker. Незважаючи на визначені напрямки майбутнього вдосконалення продуктивності, поточний стан розробленого програмного продукту на 100% відповідає функціональним вимогам технічного завдання. Система готова до етапу дослідної експлуатації та поетапної інтеграції в локальні процеси життєвого циклу розробки програмного забезпечення (SDLC). Це свідчить про те, що обрана модель взаємодії дозволяє зберігати прийнятний час відповіді навіть за умови одночасної обробки кількох запитів. Крім того, результати перевірки демонструють, що обрана архітектура забезпечує достатню стабільність роботи системи для її подальшого використання в реальних сценаріях аналізу вихідного коду.

5.5. Висновки по розділу

У п'ятому розділі було проведено комплексне тестування розробленої SAST-платформи CodeGuard. Перевірка охоплювала основні рівні системи: обчислювальне ядро, серверний оркестратор, клієнтський веб-інтерфейс та їхню інтегровану взаємодію.

Модульне тестування за допомогою xUnit підтвердило коректність роботи алгоритмів аналізу коду та дало змогу зменшити кількість хибнопозитивних спрацьовувань. Функціональне тестування веб-інтерфейсу дозволило перевірити основні користувацькі сценарії, зокрема завантаження файлів, запуск сканування та відображення результатів.

Інтеграційне тестування за допомогою Postman підтвердило правильність роботи REST API, обробки помилок і взаємодії Node.js-оркестратора з C#-ядром. Навантажувальне тестування показало, що система здатна стабільно обробляти одночасні запити невеликої команди розробників без витоків пам'яті та критичної деградації продуктивності.

Отже, результати тестування підтвердили відповідність MVP-версії системи основним функціональним і нефункціональним вимогам. Розроблена платформа є працездатною, стабільною та готовою до дослідної експлуатації і подальшого розвитку.

Окремо слід зазначити, що результати тестування підтвердили правильність обраного підходу до розподілу відповідальності між компонентами системи. Ізольоване скануюче ядро коректно виконувало аналіз коду, тоді як серверний оркестратор забезпечував прийом файлів, запуск процесу сканування, обробку відповідей і передачу результатів клієнтській частині.

Також було підтверджено, що MVP-версія платформи здатна стабільно працювати в умовах типового користувацького навантаження. Виявлені під час тестування недоліки не мали критичного характеру та були усунені в процесі налагодження, що підвищило надійність системи й підготувало її до подальшого розвитку.

ВИСНОВКИ

Створено програмний продукт – MVP-версію вебзастосунку для автоматизованого статичного тестування безпеки вихідного коду (SAST) CodeGuard. Система забезпечує асинхронне завантаження файлів з кодом, їх безпечну тимчасову обробку, глибокий лексичний аналіз на основі синтаксичних дерев та миттєву передачу структурованих звітів про знайдені вразливості на клієнтську частину.

Backend-частину системи реалізовано засобами Node.js та фреймворком Express як сервер-оркестратор, що відповідає за маршрутизацію HTTP-запитів, валідацію вхідних файлів, керування чергою завдань і запуск дочірніх процесів. Для прийому артефактів використано middleware multer, а завантажений код ізолюється в тимчасових директоріях. Також передбачено таймаути виконання, що дозволяють контролювати життєвий цикл сканування та запобігати зависанню сервера.

У системі реалізовано підхід Stateless API, який дозволив відмовитися від традиційної бази даних. Запити обробляються «на льоту»: після завершення сканування, повернення JSON-результату клієнту та очищення тимчасових файлів сервер не зберігає стан виконання. Це зменшує використання пам’яті й дискового простору та усуває затримки, пов’язані з транзакціями до сховища даних.

Ізольоване обчислювальне ядро сканера створено засобами C# та .NET Compiler Platform SDK (Roslyn). Використання програмного доступу до моделі компілятора дозволило відмовитися від ненадійного пошуку регулярними виразами. Натомість ядро будує абстрактні синтаксичні дерева та застосовує патерн Visitor на базі CSharpSyntaxWalker для глибшого аналізу структури коду.структурного аналізу коду без необхідності його попередньої компіляції.

У системі реалізовано алгоритми виявлення критичних вразливостей згідно з методологією OWASP Top 10. Це включає модулі для пошуку SQL-ін'єкцій (аналіз небезпечної конкатенації та інтерполяції рядків у запитах до БД), виявлення жорстко закодованих секретів (Hardcoded Secrets) у змінних та

впровадження базових елементів Taint-аналізу для відстеження потоку неперевірених користувацьких даних до небезпечних функцій-приймачів (Sinks).

Засобами базових вебтехнологій (HTML, CSS, Vanilla JavaScript, Bootstrap) реалізовано клієнтську частину вебзастосунку, яка надає користувачеві зручний робочий простір для завантаження коду. У системі реалізовано підтримку механізму Drag-and-Drop, візуалізацію статусів асинхронної обробки за допомогою анімованих індикаторів (спінерів) та динамічну генерацію результативних таблиць із детальним описом типу загрози та вказівкою на конкретний проблемний рядок.

Проведено модульне тестування обчислювального ядра за допомогою фреймворку xUnit, яке підтвердило здатність сканера розпізнавати вразливий код на спеціально підготовлених mock-файлах. Завдяки детальному налагодженню алгоритмів обходу AST вдалося суттєво знизити відсоток хибнопозитивних спрацьовувань (False Positives), навчивши систему точно відрізняти реальну загрозу від безпечного форматування рядків.

Проведено системне тестування API-оркестратора за допомогою середовища Postman. Перевірено стійкість сервера до завантаження невалідних форматів файлів, коректне відпрацювання таймаутів при імітації "зависання" обчислювального ядра та захист від перевантаження шляхом встановлення лімітів на розмір файлу до 50 МБ. Усі тестові сценарії були виконані успішно, API-ендпоінти відпрацювали стабільно без збоїв основного процесу Node.js.

Подальше вдосконалення системи може передбачати розробку плагінів для популярних інтегрованих середовищ розробки (наприклад, Visual Studio Code), розширення бази сигнатур безпеки для аналізу інших мов програмування (JavaScript, Python), впровадження повноцінного міжпроцедурного Taint-аналізу та нативну інтеграцію розробленого сканера у конвеєри безперервної інтеграції (GitLab CI/CD, GitHub Actions) для автоматичного блокування небезпечних комітів. Такі напрями розвитку дозволять поступово перетворити MVP-версію з окремого прототипу для локального аналізу коду на більш повноцінну DevSecOps-платформу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Souppaya M., Scarfone K., Dodson D. Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities. NIST Special Publication 800-218. Gaithersburg: National Institute of Standards and Technology, 2022. DOI: 10.6028/NIST.SP.800-218. Дата звернення: 16.05.2026.
2. McGraw G. Software Security: Building Security In. Addison-Wesley Professional, 2021. 448 p.
3. Stuttard D., Pinto M. The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws. 2nd ed. Wiley, 2011. 912 p.
4. Ross R. et al. Developing Cyber-Resilient Systems: A Systems Security Engineering Approach (NIST SP 800-160 Vol. 1). National Institute of Standards and Technology. 2022. 164 p.
5. Anderson R. Security Engineering: A Guide to Building Dependable Distributed Systems. 3rd ed. Wiley, 2020. 1232 p.
6. Scarfone K., Mell P. Technical Guide to Information Security Testing and Assessment (NIST SP 800-115). National Institute of Standards and Technology. 2022. 80 p.
7. Fowler M., Lewis J. Microservices. MartinFowler.com, 2014. Дата звернення: 12.05.2026.
8. Joint Task Force. Guide for Conducting Risk Assessments (NIST SP 800-30 Rev. 1). National Institute of Standards and Technology. 2012. 94 p.
9. Meucci M., Muller A. OWASP Software Assurance Maturity Model (SAMM) v2.0. Open Web Application Security Project. 2020. URL: https://owasp.org/www-project-samm/ (дата звернення: 20.05.2026).
10. OWASP Top 10:2021 – The Ten Most Critical Web Application Security Risks. OWASP Foundation, 2021. URL: https://owasp.org/Top10/ (дата звернення: 21.05.2026).

11. Vegh J., Vidal J. DevSecOps: Integrating Security into Your CI/CD Pipeline. Apress, 2023. 210 p.

12. Kim G., Humble J., Debois P., Willis J. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution Press, 2021. 480 p.

13. OWASP Foundation. Source Code Analysis Tools. OWASP Web Security Testing Guide / OWASP Community Pages. Дата звернення: 26.05.2026.

14. Bazzell M. Open Source Intelligence Techniques. 9th ed. Independently published, 2023. 580 p.

15. O'Hare B., Rivas P. Machine Learning for Application Security: Reducing False Positives in SAST Tools. IEEE Security & Privacy, 2023. Vol. 21(3). P. 88-95.

16. Williams L., McGraw G. Static Analysis for Security. IEEE Computer, 2005. Vol. 38(11). P. 76-83.

17. Chess B., West J. Secure Programming with Static Analysis. Addison-Wesley Professional, 2007. 624 p.

18. Antunes N., Vieira M. Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services. 14th IEEE Pacific Rim International Symposium on Dependable Computing, 2008. P. 301-308.

19. Arkin B., Stender S., McGraw G. Software Penetration Testing. IEEE Security & Privacy, 2005. Vol. 3(1). P. 84-87.

20. Microsoft. The .NET Compiler Platform SDK. Microsoft Learn, 2024. Дата звернення: 23.05.2026.

21. Howard M., LeBlanc D. Writing Secure Code. 2nd ed. Microsoft Press, 2002. 768 p.

22. Sutton M., Greene A., Amini P. Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley Professional, 2007. 576 p.

23. Tappenden A. F., Miller J. A Novel Approach to Interactive Application Security Testing. Proceedings of the 2014 IEEE International Conference on Software Testing, Verification and Validation, 2014. P. 133-142.

24. Plate H., Ponta S. E., Sabetta A. Impact Assessment for Vulnerabilities in Open-Source Software Libraries. 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015. P. 411-420.

25. Campbell G. A., Papapetrou P. SonarQube in Action. Manning Publications, 2013. 386 p.

26. IBM Security. Cost of a Data Breach Report 2023. IBM Corporation, 2023. URL: https://www.ibm.com/reports/data-breach (дата звернення: 22.05.2026).

27. Albahari J., Albahari B. C# 10 in a Nutshell: The Definitive Reference. O'Reilly Media, 2022. 1088 p.

28. Freeman E., Robson E., Bates B., Sierra K. Head First Design Patterns. 2nd ed. O'Reilly Media, 2020. 672 p.

29. Cantelon M., Harter M., Holowaychuk T. J., Rajlich N. Node.js in Action. 2nd ed. Manning Publications, 2017. 384 p.

30. Friedl J. E. F. Mastering Regular Expressions. 3rd ed. O'Reilly Media, 2006. 544 p.

31. Osherove R. The Art of Unit Testing: with examples in C#. 2nd ed. Manning Publications, 2013. 296 p.

32. Itkonen J., Mäntylä M. V. Test Better by Exploring: Harnessing Human Skills and Knowledge. IEEE Software, 2014. Vol. 31(4). P. 90-96.

33. Nikfard P. Functional testing on web applications. Proceedings of Postgraduate Annual Research on Informatics Seminar (PARIS), 2012.

34. Postman Learning Center. Writing tests in Postman. URL: https://learning.postman.com/docs/writing-scripts/test-scripts/ (дата звернення: 25.05.2026).

35. Express.js Official Documentation. Production Best Practices: Security. URL: https://expressjs.com/en/advanced/best-practice-security.html (дата звернення: 25.05.2026).

36. Apache JMeter. User's Manual: Building a Web Test Plan. URL: https://jmeter.apache.org/usermanual/build-web-test-plan.html (дата звернення: 25.05.2026).

ДОДАТКИ

ДОДАТОК А

Основний код застосунку наведено за наступним посиланням – https://github.com/Neobject/CodeGuard-

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: "Методи тестування безпеки ПЗ"

Обсяг пояснювальної записки: 82 аркуш

Дата закінчення дипломної роботи 10 червня 2026р.

Підпис студента _____

-