Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп'ютерних наук

«Допущено до захисту»
Завідувач кафедри
комп'ютерних наук
доктор технічних наук, професор
(науковий ступінь, наукове звання)
БОНДАРЧУК А.П. _____
(прізвище, ініціали) (підпис)
« ___» _____2025р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня «Магістр»
Спеціальність 122 Комп'ютерні науки
Освітня програма 122.00.02 Інформаційно-аналітичні системи

Тема роботи «Інструменти та технології управління уразливостями у процесі розробки програмного забезпечення»

Виконав
студент групи ІАСм-1-24-1.4д
(шифр академічної групи)
Грабар Максим Володимирович
(прізвище, ім'я, по батькові)

_____
(підпис)
Науковий керівник
кандидат технічних наук, доцент
(науковий ступінь, наукове звання)
РЗАЄВА С.Л.
(прізвище, ініціали)

_____
(підпис)

Київ - 2025

Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп'ютерних наук

«Затверджую»
Завідувач кафедри
комп'ютерних наук
кандидат технічних наук, доцент
(науковий ступінь, наукове звання)
Машкіна І.В. _____
(прізвище, ініціали) (підпис)
«____» _____ 2024 р

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Виконавець: студент групи ІАСм-1-24-1.4д
Грабар Максим Володимирович
(прізвище, ім'я, по батькові)

1. **Вихідні дані:** матеріали з теорії SDLC/SSDLC, сучасні методи виявлення вразливостей, вимоги до безпеки ПЗ, вихідний код для тестування розроблених інструментів.
2. **Основні завдання:** дослідити сучасні підходи до управління вразливостями ПЗ. Розробити власні інструменти для SAST, SCA та Docker Image сканування. Протестувати та інтегрувати створені сканери у локальні процеси розробки та CI/CD.
3. **Пояснювальна записка:** містить обґрунтування вибору технологій, опис архітектури та реалізації інструментів, результати експериментів із виявлення вразливостей, а також висновки щодо ефективності створеного рішення.
4. **Графічні матеріали:** схеми SDLC і SSDLC, архітектурні діаграми ПЗ, моделі класів, діаграми взаємодії модулів, приклади роботи сканерів та схеми інтеграції у CI/CD.

5. **Додатки:** фрагменти вихідного коду, конфігураційні файли, приклади сканування, інструкції запуску, додаткові матеріали з тестування та скріншоти роботи ПЗ.

6. Строк подання роботи на кафедру « 01» грудня **2025__р.**

(прізвище, ініціали, підпис)

Науковий керівник:

Рзаєва Світлана Леонідівна

кандидат технічних наук, доцент

(науковий ступінь, наукове звання)

Виконавець:

« ___» _____20__р.

_____

(підпис студента)

Анотація кваліфікаційної роботи

Кваліфікаційна робота: 63 с., 17 рис., 0 табл., 22 посилань.

Актуальність теми. Зростання кількості атак на програмне забезпечення, масштабування IT-інфраструктур, швидке розширення відкритих бібліотек і залежностей, а також посилення регуляторних вимог у сфері кібербезпеки визначають потребу у системному підході до управління вразливостями. У межах DevSecOps безпека має бути інтегрована в кожен етап SDLC, що дозволяє суттєво скоротити час між виявленням та усуненням недоліків, забезпечити автоматизацію перевірок і підвищити загальний рівень захищеності програмного забезпечення. Саме тому дослідження методів аналізу вразливостей та розробка інструментів для автоматизованого сканування є надзвичайно актуальними.

Об’єкт дослідження: система розробки та проєктування програмного забезпечення.

Предмет дослідження: інструменти та методики виявлення, аналізу й усунення уразливостей.

Мета роботи: аналіз сучасних підходів для управління уразливостями та розробка методики інтеграції сканерів безпеки в процес розробки ПЗ.

Для досягнення мети необхідно виконати такі завдання:

1. Проаналізувати сучасні методи управління вразливостями та оцінити їх ефективність на різних етапах SDLC.
2. Розробити власні інструменти для статичного аналізу вихідного коду.
3. Інтегрувати розроблені засоби в процес DevSecOps для забезпечення автоматизованого контролю безпеки.
4. Провести експериментальне дослідження ефективності запропонованих рішень.
5. Розробити методику інтеграції інструментів безпеки в DevSecOps, яка забезпечить своєчасне виявлення та усунення вразливостей і підвищить рівень захищеності програмного забезпечення.

Методи дослідження. У роботі застосовано аналіз наукових джерел з питань кібербезпеки та DevSecOps, моделювання процесів розробки ПЗ, створення та тестування інструментів для SAST-сканування, статистичне опрацювання отриманих результатів, експериментальне дослідження роботи інтегрованих сканерів безпеки, а також методи програмного проєктування.

Наукова новизна полягає у розробці власних інструментів автоматизованого статичного аналізу коду, а також у створенні та обґрунтуванні нової методики інтеграції інструментів управління вразливостями в DevSecOps, що забезпечує підвищення ефективності процесів виявлення та усунення недоліків безпеки.

Практичне значення дослідження полягає у можливості застосування розроблених інструментів і методики в реальних умовах розробки ПЗ. Вони можуть бути використані організаціями різного масштабу для підвищення рівня захищеності програмних продуктів, оптимізації процесів автоматизованого сканування та формування культури безпечної розробки.

Ключові слова: управління уразливостями, DevSecOps, статичний аналіз, динамічний аналіз, інструменти сканування коду, SCA, SAST, Docker image, CI/CD, GitHub, автоматизація, безпека програмного забезпечення.

ЗМІСТ

ВСТУП

Сучасний ринок програмного забезпечення характеризується високою динамікою розвитку та постійним збільшенням складності IT-інфраструктур. Разом із зростанням кількості розроблюваних продуктів та їх інтеграційних сценаріїв з’являється все більше потенційних «точок входу» для зловмисників. За даними звітів із кібербезпеки, понад 80 % інцидентів трапляються через використання відомих, але не усунутих вразливостей у коді та середовищі розгортання. Це свідчить про нагальну потребу впровадження системного управління вразливостями протягом усього життєвого циклу розробки програмного забезпечення (SDLC).

Додатковим чинником, що підсилює актуальність теми, є дотримання міжнародних стандартів інформаційної безпеки, таких як ISO/IEC 27001, NIST SP 800-64, ДСТУ ISO/IEC 27002 та вимог GDPR. Ці нормативи передбачають чіткі вимоги щодо управління ризиками, контролю вразливостей та забезпечення безпеки інформаційних систем. Недотримання цих вимог може призвести до фінансових санкцій, юридичної відповідальності та суттєвих репутаційних втрат. У цьому контексті автоматизація управління вразливостями, зокрема шляхом використання інструментів SAST, DAST та SCA у DevSecOps, стає не лише технічно необхідною, а й стратегічно важливою.

Об’єктом цього дослідження є процеси розробки, проєктування та забезпечення безпеки програмного забезпечення. Предметом дослідження є методи, інструменти та технології управління вразливостями ПЗ, а також підходи до інтеграції статичного аналізу й контролю безпеки в життєвий цикл розробки. Дослідження зумовлене зростаючою потребою у побудові стійких, безпечних і прозорих процесів DevSecOps, які дають змогу організаціям оперативно виявляти недоліки безпеки, автоматизувати аналіз коду та мінімізувати ризики використання уразливих компонентів у середовищі швидкої розробки.

Завданнями дослідження є проаналізувати сучасні методи виявлення вразливостей у ПЗ, розробити власні інструменти для SAST, SCA та Docker Image аналізу,

інтегрувати їх у процеси локальної та CI/CD перевірки, а також здійснити порівняльну оцінку їхньої ефективності.

Наукова новизна роботи полягає у створенні власних інструментів автоматизованого статичного аналізу та формуванні нової методики інтеграції механізмів управління вразливостями в DevSecOps-процеси. Це забезпечує підвищення точності, швидкості та узгодженості виявлення недоліків безпеки, а також сприяє створенню адаптивної моделі взаємодії безпекових інструментів, здатної масштабуватися під потреби різних команд і проєктів.

Практичне значення дослідження полягає у створенні програмного забезпечення, яке забезпечує повний цикл автоматизованого сканування: від аналізу залежностей і Docker-образів до виявлення помилок у вихідному коді. Розроблені інструменти можуть бути інтегровані у локальні процеси розробки та CI/CD-конвеєри, підвищуючи рівень захищеності продуктів і дозволяючи командам оперативно реагувати на ризики. Застосування запропонованої методики сприяє розвитку культури безпечної розробки та оптимізації процесів управління вразливостями в організаціях різного масштабу.

РОЗДІЛ 1. СУЧАСНІ ПІДХОДИ ДО УПРАВЛІННЯ ВРАЗЛИВОСТЯМИ ПЗ

1.1 Життєвий цикл програмного забезпечення

SDLC - це процес, який використовується для створення програмного забезпечення в організації-розробнику програмного забезпечення. SDLC складається з точного плану, який описує, як розробляти, підтримувати, замінювати та вдосконалювати конкретне програмне забезпечення. Життєвий цикл визначає метод покращення якості програмного забезпечення та загального процесу розробки [1].

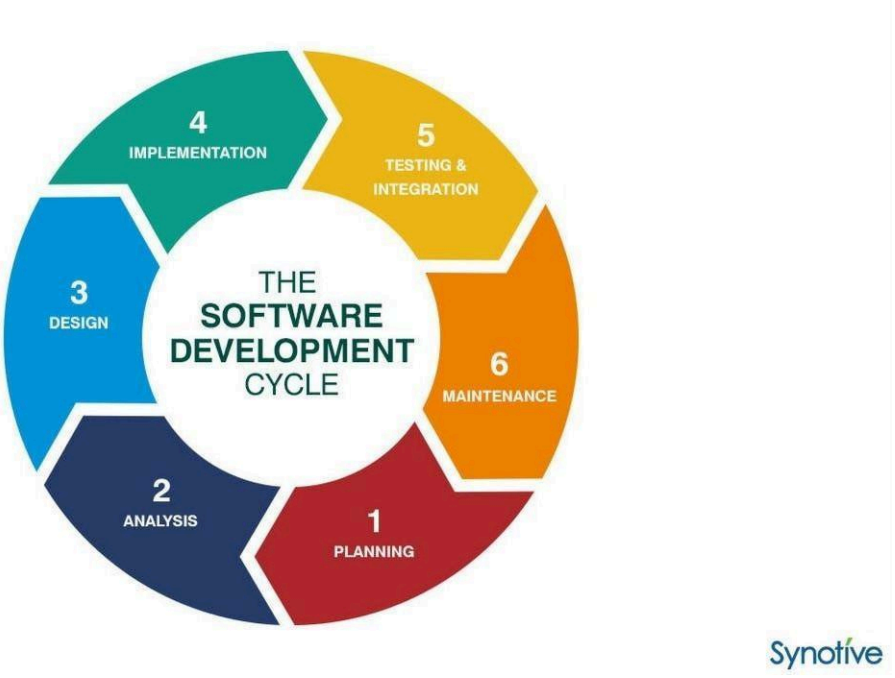

Рисунок 1.1 - Життєвий цикл програмного забезпечення (рисунок з Інтернету)

SDLC визначає завдання, які має виконувати інженер-програміст або розробник на різних етапах. Він гарантує, що кінцевий продукт здатний відповідати очікуванням замовника та вписуватися в загальний бюджет. Отже, для розробника програмного забезпечення життєво важливо мати попередні знання про цей процес розробки програмного забезпечення. SDLC – це сукупність цих шести етапів, і етапи SDLC такі [2]:

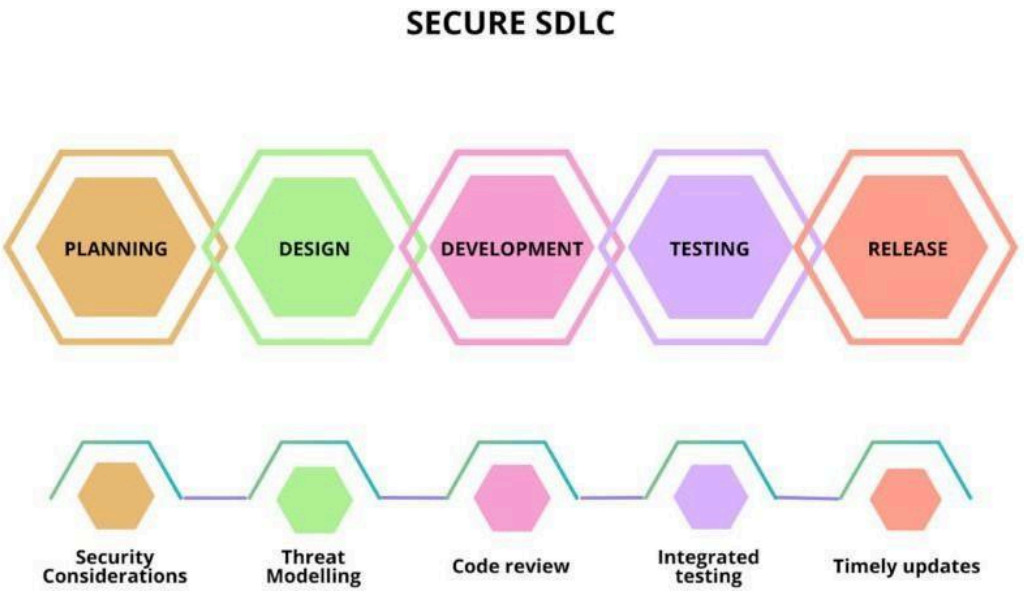

Рисунок 1.2 - Етапи SDLC (рисунок з Інтернету)

Етап 1: Планування та аналіз вимог. Планування є вирішальним кроком у всьому, як і в розробці програмного забезпечення. На цьому ж етапі розробники організації виконують аналіз вимог. Він отримується з даних клієнтів та досліджень відділу продажів/ринку. Інформація з цього аналізу формує структурні блоки базового проекту. Якість проекту є результатом планування. Таким чином, на цьому етапі базовий проект розробляється з урахуванням усієї доступної інформації.

Етап 2: Визначення вимог. На цьому етапі визначаються всі вимоги до цільового програмного забезпечення. Ці вимоги отримують схвалення від клієнтів, ринкових аналітиків та зацікавлених сторін. Це виконується за допомогою SRS (Специфікації вимог до програмного забезпечення). Це свого роду документ, який визначає всі ті речі, які необхідно визначити та створити протягом усього циклу проекту.

Етап 3: Проектування архітектури. SRS є орієнтиром для розробників програмного забезпечення, щоб створити найкращу архітектуру для програмного забезпечення. Отже, з урахуванням вимог, визначених у SRS, у Специфікації проектного документа (DDS) представлено кілька варіантів проектування

архітектури продукту. Цей DDS оцінюється ринковими аналітиками та зацікавленими сторонами. Після оцінки всіх можливих факторів для розробки обирається найбільш практичний та логічний дизайн.

Етап 4: Розробка продукту. На цьому етапі починається фундаментальна розробка продукту. Для цього розробники використовують специфічний програмний код відповідно до проекту в DDS. Отже, важливо, щоб кодери дотримувалися протоколів, встановлених асоціацією. На цьому етапі також використовуються традиційні інструменти програмування, такі як компілятори, інтерпретатори, налагоджувачі тощо. Деякі популярні мови, такі як C/C++, Python, Java тощо, використовуються відповідно до правил програмного забезпечення.

Етап 5: Тестування та інтеграція продукту. Після розробки продукту необхідне тестування програмного забезпечення для забезпечення його безперебійної роботи. Хоча на кожному етапі SDLC проводиться мінімальне тестування. Тому на цьому етапі всі ймовірні недоліки відстежуються, виправляються та повторно тестуються. Це гарантує, що продукт відповідає вимогам якості SRS.

Етап 6: Розгортання та обслуговування продуктів. Після детального тестування остаточний продукт випускається поетапно відповідно до стратегії організації. Потім він тестується в реальному промисловому середовищі. Важливо забезпечити його безперебійну роботу. Якщо він працює добре, організація випускає продукт в цілому. Після отримання корисних відгуків компанія випускає його як є або з додатковими покращеннями, щоб зробити його ще кориснішим для клієнтів. Однак, цього одного недостатньо. Тому, поряд з розгортанням, здійснюється також нагляд за продуктом [3].

1.2 Безпечний життєвий цикл розробки програмного забезпечення

Безпечний життєвий цикл розробки програмного забезпечення (SSDLC) гарантує, що комп'ютерні програми створюються з урахуванням безпеки з самого початку. Він включає планування, проектування, кодування, тестування,

розгортання та підтримку програмного забезпечення, одночасно постійно вирішуючи проблеми безпеки на кожному кроці. SSDLC має вирішальне значення для раннього виявлення та виправлення проблем безпеки, знижуючи ризик кіберзагроз. Інтегруючи заходи безпеки протягом усього процесу розробки, SSDLC прагне створювати безпечніші та надійніші програмні додатки. Це важлива практика в постійно мінливому ландшафті кібербезпеки. Весь процес розробки визначено фреймворком життєвого циклу розробки програмного забезпечення (SDLC). Планування, проектування, розробка, випуск, обслуговування, оновлення та виведення з експлуатації або заміна програми за необхідності - все це включено. Цей процес розширюється безпечним SDLC (SSDLC), який інтегрує безпеку протягом усього життєвого циклу. SSDLC часто використовується командами, які переходять на DevSecOps. Процедура передбачає захист середовища розробки та впровадження найкращих практик безпеки разом із функціональними аспектами розробки. Забезпечення безпеки програмного забезпечення є важливим, і саме тут на допомогу приходить Безпечний життєвий цикл розробки програмного забезпечення (SSDLC).

У минулому люди не приділяли достатньо уваги безпеці під час створення програмного забезпечення, що призводило до проблем і потенційних порушень. З SSDLC безпека враховується від самого початку до кінця створення програмного забезпечення. Це проактивний спосіб боротьби з ризиками безпеки, їх раннього виявлення та усунення, щоб мінімізувати ймовірність кібератак. Включаючи безпеку з самого початку, компанії можуть зробити своє програмне забезпечення надійнішим, краще дотримуватися правил і завоювати довіру людей, які використовують їхнє програмне забезпечення.

Отже, Безпечний SDLC є обов'язковим для створення надійного та безпечного програмного забезпечення в сучасному світі онлайн-загроз. Розробка безпечного життєвого циклу розробки програмного забезпечення (SDLC) пройшла довгий шлях, розвиваючись з часів, коли люди не приділяли багато уваги безпеці під час створення програмного забезпечення. Спочатку основна увага була зосереджена на забезпеченні належної та швидкої роботи програмного

забезпечення, а безпека часто відходила на другий план. Зі зростанням складності кіберзагроз стало зрозуміло, що вирішення питань безпеки на пізніх етапах неефективне. З часом еволюція SDLC змістилася в бік того, щоб зробити безпеку фундаментальною частиною з самого початку циклу розробки. Ця зміна була зумовлена розумінням того, що вирішення проблем безпеки на пізніх етапах не є гарною стратегією. Еволюція також включала визначення найкращих практик та принципів, акцентуючи увагу на таких речах, як безпечне кодування, оцінка ризиків та постійний контроль за безпекою. На розробку безпечного SDLC також вплинули правила та норми, а також більша усвідомленість серйозних наслідків порушень безпеки. Сьогодні це комплексна структура, яка охоплює всі аспекти розробки, гарантуючи, що безпека враховується на кожному кроці. Простіше кажучи, еволюція Secure SDLC показує, як люди навчилися думати про безпеку з самого початку, переходячи від реагування на проблеми до проактивності та забезпечення надійності та безпеки програмного забезпечення [6].

1.2.1 Ключові принципи безпечного SDLC

Ключові принципи безпечного життєвого циклу розробки програмного забезпечення (SDLC) викладають фундаментальні ідеї, що керують процесом створення безпечного програмного забезпечення. Ці принципи допомагають розробникам і командам зрозуміти, як підходити до безпеки на кожному етапі розробки.

1. Безпека за проектуванням: Цей принцип підкреслює необхідність думати про безпеку з самого початку створення програмного забезпечення. Це означає включення вимог безпеки на початкових етапах планування та проектування розробки.
2. Безперервний моніторинг: Безперервний моніторинг – це постійний процес регулярної перевірки та забезпечення безпеки програмного забезпечення на кожному етапі його розробки. Це не одноразова дія, а постійні зусилля для пошуку та виправлення проблем безпеки протягом усього процесу розробки.

3. Оцінка ризиків: Цей принцип передбачає оцінку та розуміння потенційних ризиків безпеки на ранніх етапах процесу розробки. Він включає виявлення вразливостей та визначення того, які ризики потребують найбільшої уваги та швидких дій для зменшення потенційних загроз.

4. Освіта та навчання: Освіта та навчання є важливими для того, щоб переконатися, що всі, хто бере участь у створенні програмного забезпечення, особливо розробники, знають про проблеми безпеки. Він наголошує на забезпеченні необхідного навчання, щоб люди мали навички для ефективного вирішення проблем безпеки.

5. Співпраця: Співпраця є ключовим принципом, який підкреслює важливість командної роботи. Вона передбачає заохочення співпраці між різними командами, такими як розробники, операційні команди та команди безпеки. Це гарантує, що всі працюють разом, обмінюються знаннями та координують зусилля для досягнення спільних цілей безпеки під час процесу розробки програмного забезпечення [5].

1.2.2 Фази безпечного SDLC

1. Планування: На етапі планування основна увага приділяється визначенню вимог безпеки до програмного забезпечення. Це включає виявлення можливих ризиків та створення плану того, як зробити програмне забезпечення безпечним з самого початку.

2. Проектування: На етапі проектування план безпеки впроваджується в життя. Це включає прийняття рішень щодо того, як вбудувати функції безпеки в програмне забезпечення. Мета полягає в тому, щоб проект міг впоратися з потенційними проблемами безпеки.

3. Впровадження: На етапі впровадження розробники починають створювати програмне забезпечення, використовуючи безпечні методи кодування. Це означає написання коду таким чином, щоб зменшити ймовірність проблем безпеки. Перевірки коду проводяться для виявлення та виправлення будь-яких проблем безпеки.

4. Тестування: Тестування полягає в перевірці безпеки програмного забезпечення. Проводяться різні тести, такі як спроби злому програмного забезпечення для пошуку вразливостей, сканування коду на наявність потенційних проблем та переконання, що програмне забезпечення може впоратися з різними загрозами безпеці.

5. Розгортання: Розгортання відбувається, коли програмне забезпечення випускається. Тут основна увага приділяється забезпеченню безпеки процесу випуску, вживанню запобіжних заходів, щоб уникнути будь-яких проблем безпеки на цьому етапі.

6. Технічне обслуговування: Технічне обслуговування – це безперервний процес, під час якого програмне забезпечення постійно обслуговується. Це включає стеження за безпекою та регулярне оновлення програмного забезпечення для боротьби з новими загрозами, забезпечуючи його безпеку з часом [5].

1.3 Сучасні методи виявлення вразливостей у ПЗ

Виявлення вразливостей у програмному забезпеченні є одним із ключових етапів забезпечення кібербезпеки в сучасних умовах цифрової трансформації. Зростання кількості кібератак та інцидентів, пов’язаних із компрометацією інформаційних систем, зумовлює необхідність постійного вдосконалення методів пошуку та усунення вразливостей. За даними аналітичних звітів, понад 60% вразливостей у ПЗ мають критичний рівень небезпеки, що підтверджує актуальність проблеми та потребу у використанні комплексного підходу до управління вразливостями [7]. Методи виявлення вразливостей можна поділити на дві основні групи - ручні та автоматизовані, проте сучасна практика розробки програмного забезпечення доповнює їх низкою нових підходів, зокрема інтеграцією безпеки на етапі розробки (DevSecOps), аналізом інфраструктури як коду (IaC), використанням fuzz-тестування, машинного навчання для пріоритизації ризиків та постійного моніторингу у реальному часі. Ручні методи залишаються важливою складовою процесу оцінки безпеки, оскільки дозволяють виявляти

складні логічні помилки, які автоматизовані інструменти часто не здатні розпізнати. До таких методів належать:

- Код-рев'ю (Code Review) - ретельний аналіз вихідного коду з метою виявлення небезпечних конструкцій, помилок управління пам'яттю, некоректного використання бібліотек, хардкоджених даних, SQL-ін'єкцій, XSS та інших вразливостей.
- Тестування на проникнення (Penetration Testing) - моделювання реальних атак для перевірки надійності захисту застосунку або інфраструктури.
- Аналіз конфігурацій та залежностей - перевірка конфігураційних файлів, системних налаштувань, Docker-образів, залежностей у файлах package.json, requirements.txt тощо.

Перевагами ручних методів є глибина аналізу, можливість виявлення контекстно-залежних помилок та логічних дефектів. Недоліки - значні часові витрати, потреба у високій кваліфікації фахівців та складність масштабування на великі проєкти. Автоматизовані підходи дають змогу швидко обробляти великі обсяги коду та мінімізувати людський фактор. До них належать:

- Статичний аналіз коду (SAST) - дослідження вихідного коду без його виконання, спрямоване на виявлення небезпечних конструкцій, SQL-ін'єкцій, некоректного використання функцій тощо.
- Динамічний аналіз (DAST) - перевірка безпеки додатку під час виконання, що дає змогу виявити вразливості, які проявляються лише у процесі роботи.
- Аналіз залежностей (SCA) - автоматичне виявлення використаних сторонніх бібліотек і перевірка їхніх версій на наявність відомих уразливостей (CVE).

Перевагами автоматизованих методів є швидкість, системність та можливість інтеграції у CI/CD-процеси. Недоліки - можливість хибних спрацьовувань, пропуск складних логічних вразливостей і потреба в адаптації під конкретні технології. Останніми роками у практиці безпечної розробки активно застосовуються новітні підходи, що забезпечують проактивний захист та раннє виявлення проблем.

1. Підхід “Shift-left” та DevSecOps. Безпека інтегрується на ранніх етапах життєвого циклу розробки: здійснюється моделювання загроз, впроваджуються автоматизовані перевірки у pull-request, використовуються “security gates” у CI/CD.

   Переваги: раннє виявлення помилок, зниження вартості виправлень.

   Недоліки: потреба у зміні процесів і додатковому навчанні розробників.
2. Аналіз інфраструктури як коду (IaC) та контейнерна безпека. Перевіряються конфігураційні файли Terraform, Kubernetes, Docker на наявність небезпечних параметрів, застарілих образів та надмірних прав доступу.

   Переваги: виявлення ризиків до етапу розгортання.

   Недоліки: складність у масштабних хмарних архітектурах, потреба у регулярному оновленні політик безпеки.
3. Fuzz-тестування. Автоматичне подання до програми випадкових або некоректних даних з метою виявлення неочікуваних збоїв.

   Переваги: виявлення нетипових помилок.

   Недоліки: потреба у значних обчислювальних ресурсах та складність аналізу результатів.
4. Runtime Application Self-Protection (RASP) і моніторинг у реальному часі. Забезпечується виявлення та блокування атак під час роботи додатку.

   Переваги: реагування на експлойти у реальному часі.

   Недоліки: вплив на продуктивність і складність налаштування.
5. Використання машинного навчання. Системи штучного інтелекту застосовуються для зменшення кількості хибних спрацьовувань, кластеризації вразливостей та визначення їх пріоритетності.

   Переваги: автоматизація обробки великої кількості даних.

   Недоліки: потреба у якісних навчальних вибірках і складність інтерпретації рішень.
6. Контроль ланцюга постачання програмного забезпечення (Software Supply Chain Security). Здійснюється перевірка підписів артефактів, формування та аналіз “software bill of materials” (SBOM), моніторинг сторонніх залежностей.

Переваги: прозорість і контроль походження компонентів.

Недоліки: збільшення кількості процедур перевірки.

Висновок до Розділу 1

У першому розділі проведено аналіз сучасних підходів до управління вразливостями програмного забезпечення та охарактеризовано ключові поняття, що використовуються у сфері інформаційної безпеки. Розглянуто життєвий цикл програмного забезпечення (SDLC) та його моделі, що дозволяє зрозуміти, на яких етапах можливе впровадження механізмів контролю якості та безпеки. Особливу увагу приділено безпечному життєвому циклу розробки програмного забезпечення (SSDLC), у якому виділено ключові принципи та фази, що забезпечують проактивне виявлення та усунення вразливостей ще на ранніх етапах розробки. Це підкреслює важливість інтеграції безпеки у всі етапи SDLC для зменшення ризиків експлуатації ПЗ.

Також було проаналізовано методи виявлення вразливостей, включаючи ручні та автоматизовані підходи, а також спеціалізовані сканери. Виявлено, що автоматизовані інструменти значно підвищують ефективність виявлення вразливостей, забезпечують швидку перевірку великої кількості компонентів і дозволяють інтегрувати процеси контролю безпеки у CI/CD-пайплайни. Загалом, розділ підкреслює необхідність системного підходу до управління вразливостями, де поєднуються принципи безпечного SDLC та використання сучасних методів і інструментів для ефективного забезпечення безпеки програмного забезпечення. Отримані знання створюють теоретичну основу для подальшої розробки власних сканерів безпеки, розглянутих у наступних розділах роботи.

РОЗДІЛ 2. РОЗРОБКА ВИМОГ І ПРОЕКТУВАННЯ ПЗ

Система передбачена для створення як комплексне рішення з виявлення вразливостей у процесі розроблення програмного забезпечення. Вона має об’єднувати три основні функціональні модулі: SAST-сканер для статичного аналізу коду, SCA-сканер для перевірки залежностей і зовнішніх бібліотек, а також Docker Image-сканер для аналізу безпеки контейнерних образів. Передбачається, що всі компоненти працюватимуть узгоджено, забезпечуючи комплексну оцінку безпеки програмного продукту на різних рівнях. Програмне забезпечення має бути здатним працювати як у локальному середовищі розробника, так і в інфраструктурі CI/CD. Це дозволить автоматизувати процеси перевірки безпеки під час коміту, збирання та розгортання застосунків. Для цього система повинна підтримувати інтеграцію з GitHub Actions та іншими CI/CD платформами, а її модулі мають бути розповсюджені у вигляді Docker-контейнерів для зручності розгортання й масштабування. Основними користувачами передбаченої системи є розробники програмного забезпечення, які зможуть здійснювати попередню перевірку власного коду, та інженери з інформаційної безпеки (DevSecOps), що інтегруватимуть інструмент у процеси безперервної розробки. Крім того, систему можуть використовувати аналітики з безпеки для проведення аудитів та перевірок готових проєктів.

2.1 Мета та завдання розробки ПЗ

Мета цієї розробки - створення системи управління уразливостями, яка реалізує ключові практики сучасного DevSecOps та відповідає усталеним індустріальним підходам до безпечного життєвого циклу програмного забезпечення. Завдання полягає у створенні модульного інструментарію для автоматизованого виявлення вразливостей у залежностях (SCA), статичного аналізу коду (SAST) та сканування контейнерних образів, що може бути запущений локально розробником (shift-left) і інтегрований у CI/CD пайплайни (GitHub

Actions, GitLab CI, Jenkins). Розробка базується на принципах мінімальної інфраструктурної залежності, уніфікованого машинно-зчитуваного формату звітів (JSON), а також на практиках забезпечення відтворюваності результатів (включення метаданих: версія інструменту, commit SHA, час сканування). Практичне втілення передбачає використання публічних джерел і стандартних API (наприклад, OSV) для верифікації відомих вразливостей, кешування та інкрементальних перевірок для оптимізації часу виконання, а також механізмів конфігурування порогів і обробки помилкових спрацьовувань. Очікуваний внесок роботи - демонстрація ефективного поєднання простоти локального застосування для розробника та можливості інтеграції з існуючими DevOps-процесами; надання репрезентативного прототипу сканера залежностей із зрозумілим інтерфейсом командного рядка, структурованим форматом звітів і базовими політиками triage. Робота також окреслює напрямки подальшого розвитку (адаптери форматів звітів, централізована індексація результатів, плагінна архітектура та автоматизований lifecycle suppression), що дозволять трансформувати прототип у промислово придатний інструмент.

Основні завдання, що вирішуються програмним продуктом:

1. Аналіз безпеки вихідного коду з метою виявлення помилок програмування, які можуть призвести до експлуатації вразливостей (SAST-аналіз).
2. Перевірка сторонніх бібліотек і компонентів на наявність відомих уразливостей із використанням відкритих баз даних, таких як OSV.dev (SCA- аналіз).
3. Оцінка безпеки контейнерних образів Docker, включно з перевіркою системних пакетів і залежностей, що входять до складу образу.
4. Автоматизація процесів перевірки безпеки через інтеграцію з CI/CD-конвеєрами (зокрема GitHub Actions), забезпечуючи виконання аналізу при кожному коміті або запиті на злиття (Pull Request).
5. Формування структурованих звітів про результати сканування у форматі JSON, придатному для подальшого аналізу або інтеграції з іншими системами моніторингу безпеки.

6. Забезпечення масштабованості та універсальності рішень за рахунок контейнеризації модулів і підтримки різних середовищ розробки.
7. Надання інтерфейсу командного рядка (CLI) для локального використання розробниками та зручного налаштування параметрів сканування.

Очікувані результати від впровадження системи:

- підвищення рівня безпеки програмного забезпечення завдяки систематичному контролю вразливостей у коді, бібліотеках і контейнерах;
- зменшення кількості інцидентів, пов’язаних із експлуатацією відомих уразливостей;
- забезпечення раннього виявлення помилок у процесі розроблення та скорочення витрат на їх усунення;
- підвищення прозорості процесів безпеки в рамках DevOps і формування культури безпечної розробки (Secure Software Development Lifecycle);
- створення гнучкого, відкритого та розширюваного інструменту, який може бути використаний як основа для подальших наукових досліджень або промислових рішень у сфері кібербезпеки.

2.2 Обґрунтування вибору технологій та мови програмування

При проєктуванні системи автоматизованого аналізу безпеки (SAST, SCA, Docker Image Scanner) було виконано порівняльний аналіз доступних мов програмування і супутніх технологій з урахуванням вимог до швидкості розробки, наявності бібліотек для парсингу й роботи з контейнерами, інтеграції в CI/CD, кросплатформеності та можливості подальшого розширення. Переваги мови програмування Python - багата екосистема бібліотек для роботи з текстом, AST, HTTP, YAML/JSON та Docker; велика кількість готових інструментів для безпеки (Bandit, pip-audit тощо); швидка розробка прототипів; добра сумісність з CI/CD; широка спільнота. Недоліки - нижча продуктивність у порівнянні зі статично

типізованими мовами, але це частково компенсується паралелізацією запитів і клієнт–серверною архітектурою для важких завдань. Переваги мови програмування Go - висока продуктивність, проста побудова бінарників без залежностей, зручність для CLI-утиліт і роботи з паралелізмом; сильні можливості для обробки мережевих запитів. Недоліки - менша кількість готових бібліотек для парсингу різних мов (хоча для деяких задач є потужні бібліотеки), вища вартість розробки прототипу порівняно з Python. Переваги мови програмування JavaScript / Node.js - сильна екосистема для роботи з JS/TS (Esprima, Acorn, Babel), зручність для інструментів, що тісно інтегруються із фронтендом. Недоліки - частіші проблеми з управлінням численними версіями пакетів, відсутність універсальної бібліотеки для аналізу різних мов одночасно, а також менша традиційність застосування для системного аналізу та роботи з контейнерами порівняно з Python/Go. Отже, порівняння показало, що для задач змішаного аналізу (мульти-мовний SAST + SCA + робота з Docker) найоптимальнішим компромісом між швидкістю розробки, гнучкістю і наявністю бібліотек є Python. Для взаємодії з вебресурсами та зовнішніми API застосовується бібліотека requests, що забезпечує формування HTTP-запитів (GET, POST), обробку помилок, повторні спроби та таймаути. Для підвищення швидкодії при аналізі великої кількості залежностей використовуються засоби concurrent.futures та asyncio, які дозволяють реалізувати багатопотокове або асинхронне виконання запитів. Основу статичного аналізу коду формує вбудований модуль ast, який дозволяє будувати абстрактне синтаксичне дерево та досліджувати структуру коду без його виконання. Для пошуку простих шаблонів і сигнатур вразливостей додатково застосовується модуль re, що працює з регулярними виразами. Оскільки система підтримує аналіз різних мов програмування, для універсального парсингу використовується tree-sitter — потужний інструмент побудови синтаксичних дерев для мов JavaScript, PHP, Go тощо. Для глибшого аналізу окремих мов застосовуються спеціалізовані парсери: Esprima або Acorn для JavaScript (через обгортки Node.js), а також php-parser чи phply для PHP-коду. Конфігураційні файли (наприклад, docker-compose.yml) опрацьовуються за допомогою бібліотеки PyYAML, а для коректного порівняння версій пакетів

використовуються packaging.version та semver. Для взаємодії з контейнерами у процесі сканування застосовується Docker SDK for Python (docker-py), що дозволяє отримувати метадані образів, запускати контейнери у тимчасових середовищах і збирати інформацію про встановлені пакети. У разі необхідності виклику нативних утиліт Docker чи системних пакетних менеджерів (dpkg, apk, rpm) використовується модуль subprocess із контролем часу виконання. Робота з файловою системою, тимчасовими директоріями, архівами та JSON-звітами реалізується через стандартні бібліотеки os, pathlib, tempfile, tarfile та json. Ведення журналів подій, діагностика та аудит процесу забезпечуються через систему logging, яка дозволяє зручно контролювати рівні логування й зберігати результати в окремих файлах. Для реалізації зручного командного інтерфейсу використовується argparse або click - залежно від складності CLI. Тестування функціоналу системи та перевірка стабільності реалізуються за допомогою фреймворку pytest, який підтримує як модульні, так і інтеграційні сценарії. Додатково для підвищення рівня безпеки та контролю якості коду можуть використовуватись статичні інструменти Bandit, pip- audit і safety, які допомагають виявляти потенційні вразливості у залежностях і правилах сканування. Для автоматизації процесів інтеграції та перевірки безпеки використовується GitHub Actions, що дозволяє запускати сканери при кожному коміті або Pull Request, зберігати результати у вигляді артефактів і забезпечувати прозорість процесу. Форматом обміну даними обрано JSON, оскільки він є легким, універсальним і зручним для подальшої інтеграції з іншими системами та візуалізації результатів.

Вибір Python як основної мови реалізації обґрунтований балансом між швидкістю розробки, наявністю необхідних засобів для парсингу коду та роботи з контейнерами, простотою інтеграції в CI/CD і можливістю легко розширювати систему за рахунок сторонніх парсерів або сервісів. Набір бібліотек і інструментів, перелічених вище, дає змогу реалізувати всі вимоги до SAST, SCA та Docker Image Scanner у вигляді модульного, тестованого та контейнеризованого рішення; при цьому архітектура залишається відкритою для оптимізації окремих компонентів

(перепис критичних частин на Go або розподіл обробки) у разі зростання навантаження.

2.3 Технічне завдання на розробку ПЗ

Усі три сканери проєкту реалізуються як модульні компоненти з єдиним підходом до взаємодії і виводу результатів. Кожний модуль має: інтерфейс запуску (CLI), механізм логування, модуль обробки вхідних артефактів (файли/образи), ядро перевірок (правила/запити до зовнішніх сервісів) і експортер звітів у уніфікованому JSON-форматі.

Ключові вимоги для розробки сканерів безпеки:

1. Уніфікований формат результатів - звіти у JSON, з метаданими, переліком знахідок і підсумковою статистикою; це дозволяє централізовано агрегувати дані в CI/CD або в зовнішніх системах.
2. Модульність та розширюваність - кожен сканер складається з підмодулів (парсер/детектор/обробник/експортер), що спрощує додавання нових правил або підтримку нових форматів.
3. Контейнеризація - упаковка модулів у Docker-образи для відтворюваності середовища і простої інтеграції в пайплайни.
4. Єдина політика логування та обробки помилок - стандартизоване логування (logging), таймаути для зовнішніх викликів, retry-механізми та акуратне документування помилок у звітах (наприклад, як у IMAGE Scanner при помилках витягання пакетів).
5. Безпечна робота з чутливими даними - заборона збереження секретів у відкритому вигляді у звітах, маскування знайдених секретів у режимі експорту, захищені підключення до зовнішніх API (HTTPS).

Система будується за об’єктно-орієнтованою, модульною архітектурою. Кожний сканер може бути виконаний як автономний процес/компонент або як сервіс в єдиному пайплайні. Для внутрішньої організації коду рекомендовано:

окремі класи для парсингу артефактів (файлів залежностей, Dockerfile, docker-compose, файли коду), ядро перевірок (регулярні вирази, AST-перевірки, запити до OSV), провайдер зовнішніх інтеграцій (HTTP-клієнт із retry/таймаутами) та модуль генерації звітів. Наведені приклади коду демонструють цю логіку: SASTScanner._scan_with_patterns, SCAScanner._extract_dependencies, scan_dockerfile/scan_compose у IMAGE Scanner - кожен компонент відповідає за одну чітко визначену відповідальність. Для забезпечення портативності і простого розгортання – всі модулі супроводжуються Dockerfile'ами, конфігурацією залежностей (requirements.txt) і короткими інструкціями CLI (help-опис аргументів). Архітектура допускає поділ навантаження: важкі або довготривалі завдання (наприклад, масові запити до OSV для великого дерева залежностей або скан образів) можуть виконуватися паралельно або делегуватися воркерам. ТЗ передбачає, що інструменти мають ефективно працювати на проєктах малого і середнього розміру у локальному середовищі, а також масштабуватися для великих репозиторіїв у CI. Для цього рекомендовано: асинхронні або багатопоточні запити до зовнішніх сервісів (concurrent.futures / asyncio), кешування відповідей API для повторного використання (щоб уникати дублювання запитів до OSV), і можливість горизонтального масштабування через паралельні завдання або окремі контейнери- воркери. Також потрібно контролювати ресурси при скануванні локальних образів (обмеження пам'яті/CPU контейнерів, таймаути при запуску команд всередині контейнерів). Кожний інструмент має надавати зручний CLI інтерфейс для локального використання та для виклику в CI-пайплайні. CLI повинен підтримувати вибір цілі сканування (файл/директорія/образ), налаштування шляху збереження результатів, рівня логування і параметрів паралелізації. Для інтеграції в CI важливі: коректні коди завершення (exit codes), генерація артефактів (JSON) та можливість запуску у безперервних workflow (GitHub Actions). Приклад CLI у наданих скриптах (argparse із --file, --dir, --image, --output) показує рекомендований мінімальний набір опцій. Технічне завдання передбачає набір автоматизованих тестів: юніт-тести для парсерів і правил (pytest), інтеграційні тести для взаємодії з Docker (моки або тестові образи) та приймальні тести у CI (workflow, що запускає

сканер над тестовим репозиторієм і перевіряє структуру JSON-звіту). Документація повинна містити приклади запуску, опис JSON-формату звіту та інструкції з інтеграції у CI. Вихідним артефактом є git-репозиторій з кодом, Dockerfile, набір тестів і керівництво користувача. Реалізація повинна враховувати обмеження привілеїв при роботі з Docker (не надавати зайвих прав), забезпечувати HTTPS- з'єднання при зверненні до ОСВ чи інших сервісів, не зберігати секрети у звітах та логах у явному вигляді, передбачати маскування/хешування знайдених секретів. Логування має бути конфігурованим за рівнями та, за потреби, спрямовуватись у зовнішні системи моніторингу. Для коректної роботи інструментів рекомендовано наступний мінімум: Linux (або runner з Docker), Python 3.10+, Docker Engine (для IMAGE Scanner), доступ до Інтернету для зовнішніх API, CPU 2 ядра (4 рекомендовано для CI), RAM від 4 GB (8 GB рекомендовано при аналізі великих образів), дискове місце мінімум 5–10 GB для тимчасових артефактів. У CI- середовищі бажано забезпечити можливість обмеження ресурсів container runtime та контроль над правами доступу до Docker-daemon.

2.4 Опис архітектури, моделі класів та взаємодії модулів ПЗ

Проєкт передбачає розробку трьох взаємопов'язаних, але незалежних модулів-сканерів. Кожний модуль виконує свою спеціалізовану задачу, але всі разом вони формують єдину систему контролю якості та безпеки ПЗ. Основні вимоги до архітектури: модульність, читаємість коду, тестованість, можливість контейнеризації та простота інтеграції в CI/CD. Система будується за «керованим оркестратором» (Orchestrator) — центральним компонентом, який координує роботу парсерів і аналізаторів і збирає результати. Архітектурні блоки:

- CLI / Runner - точка входу: розбір аргументів, базова валідація, виклик Orchestrator.
- Orchestrator - управляє послідовністю операцій, конфігурацією і збіркою результатів.

- Парсери - модулі, що витягують вхідні артефакти: код, файли залежностей, Dockerfile, docker-compose.yml.
- Аналізатори - SAST, SCA, IMAGE - виконують детальні перевірки і повертають знахідки у стандартному форматі.
- ExternalAdapters (OSVClient, ін.) - абстракції для зовнішніх викликів (OSV API та ін.), з підтримкою кешування, таймаутів і retry.
- WorkerPool / Executor - механізм для паралельного виконання завдань.
- Reporter / Exporter - агрегує знахідки і генерує уніфікований JSON-звіт.
- Cache / Storage - локальне зберігання відповідей API і тимчасових артефактів.
- DockerAdapter - інтерфейс взаємодії з Docker Engine для аналізу локальних образів.
- Logging / Telemetry - централізоване логування та (опційно) метрики.

Модель класів описує логічну структуру розроблюваної системи та зв’язки між її основними компонентами. Програмне забезпечення створюється за об’єктно- орієнтованим підходом, де кожен клас виконує свою чітку функцію — від обробки вхідних даних до формування звітів про знайдені вразливості. У системі можна виділити кілька основних груп класів:

1. Базові класи (ядро системи). Це основні елементи, які забезпечують спільну логіку для всіх сканерів.
    - BaseScanner - головний (базовий) клас для всіх типів сканерів. Визначає загальні методи, такі як scan() (запуск перевірки), parse_target() (обробка вхідних даних) і generate_report() (створення звіту).
    - ReportExporter - відповідає за створення звіту у єдиному форматі JSON. У ньому зберігаються метадані, знайдені вразливості та коротка статистика.
    - LoggerManager - керує журналом подій у системі. Використовується для виведення інформаційних повідомлень, попереджень та помилок під час роботи сканерів.

2. Функціональні класи (сканери). Ці класи виконують основну роботу системи - пошук і аналіз вразливостей.
    - SASTScanner - проводить статичний аналіз коду, знаходить небезпечні конструкції, неправильні виклики функцій або потенційно уразливі фрагменти.
    - SCAScanner - аналізує файли залежностей (наприклад, requirements.txt, package.json) і перевіряє, чи містять вони бібліотеки з відомими вразливостями.
    - ImageScanner - перевіряє Docker-образи, аналізує встановлені пакети, конфігураційні файли (Dockerfile, docker-compose) і визначає застарілі чи небезпечні компоненти.
3. Сервісні класи. Це допоміжні елементи, які забезпечують роботу основних модулів.
    - ArtifactParser - відповідає за попередню підготовку даних до аналізу (наприклад, розбір коду чи образів).
    - VulnerabilityRecord - описує структуру знайденої вразливості, зберігаючи її ідентифікатор, рівень небезпеки, опис, розташування у файлі та рекомендацію щодо усунення.
    - ConfigManager - зберігає параметри запуску: шляхи до файлів, рівень логування, налаштування продуктивності тощо.
4. Класи взаємодії із зовнішніми сервісами. Вони дозволяють сканерам отримувати додаткові дані ззовні.
    - OSVClient - підключається до відкритої бази уразливостей (OSV), виконує пошук уразливих версій бібліотек, зберігає кеш запитів і повторює їх у разі помилки.
    - DockerClient - забезпечує зв’язок із Docker Engine для отримання інформації про образи, встановлені пакети та конфігурацію середовища.

Базовий клас BaseScanner є спільним предком для трьох типів сканерів: SASTScanner, SCAScanner і ImageScanner. Усі вони користуються допоміжними модулями - ReportExporter для створення звітів, LoggerManager для ведення логів, ArtifactParser для підготовки даних і ConfigManager для отримання налаштувань. Додатково SCAScanner використовує OSVClient для звернення до бази уразливостей, а ImageScanner - DockerClient для роботи з контейнерами.

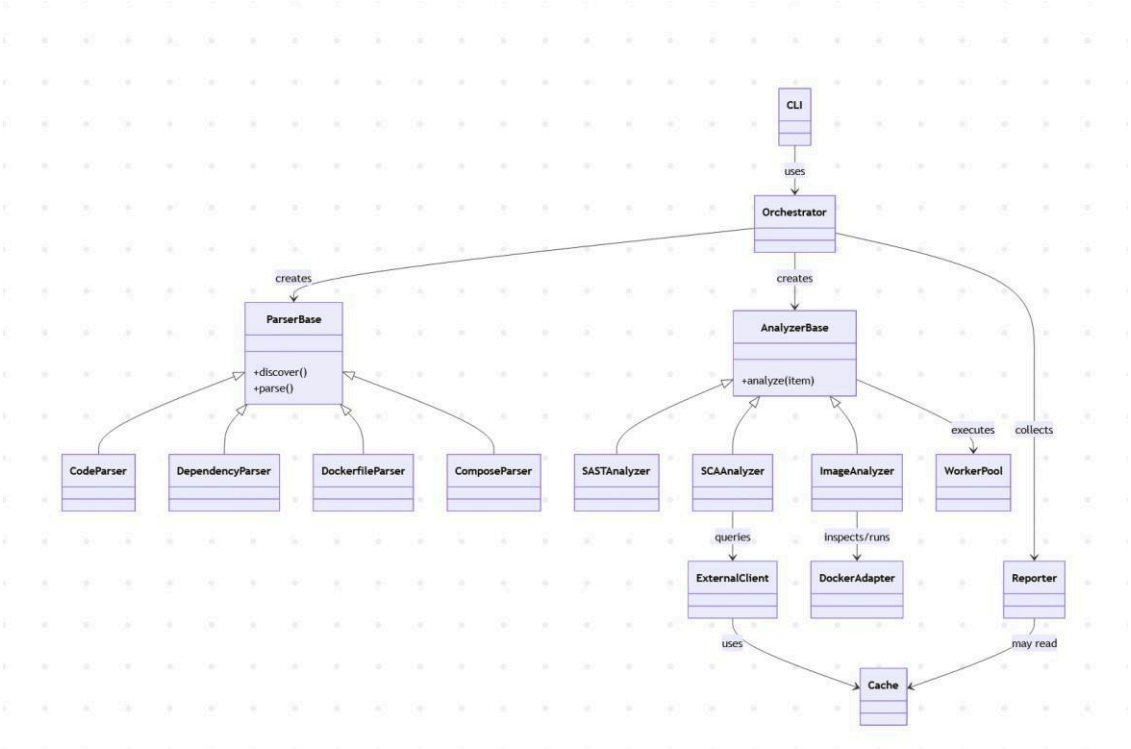

Рисунок 2.1 - Схема моделі класів системи

Узагальнена діаграма послідовності роботи сканерів відображає логіку взаємодії основних компонентів системи під час виконання аналізу. Вона показує, як користувач ініціює процес сканування, як сканери взаємодіють із модулями обробки даних, зовнішніми сервісами (OSV, Docker), системою логування та генерації звітів. Діаграма допомагає наочно продемонструвати послідовність викликів між класами та порядок виконання основних етапів роботи програмного комплексу.

Узагальнена діаграма послідовності роботи SCA сканера має такий вигляд:

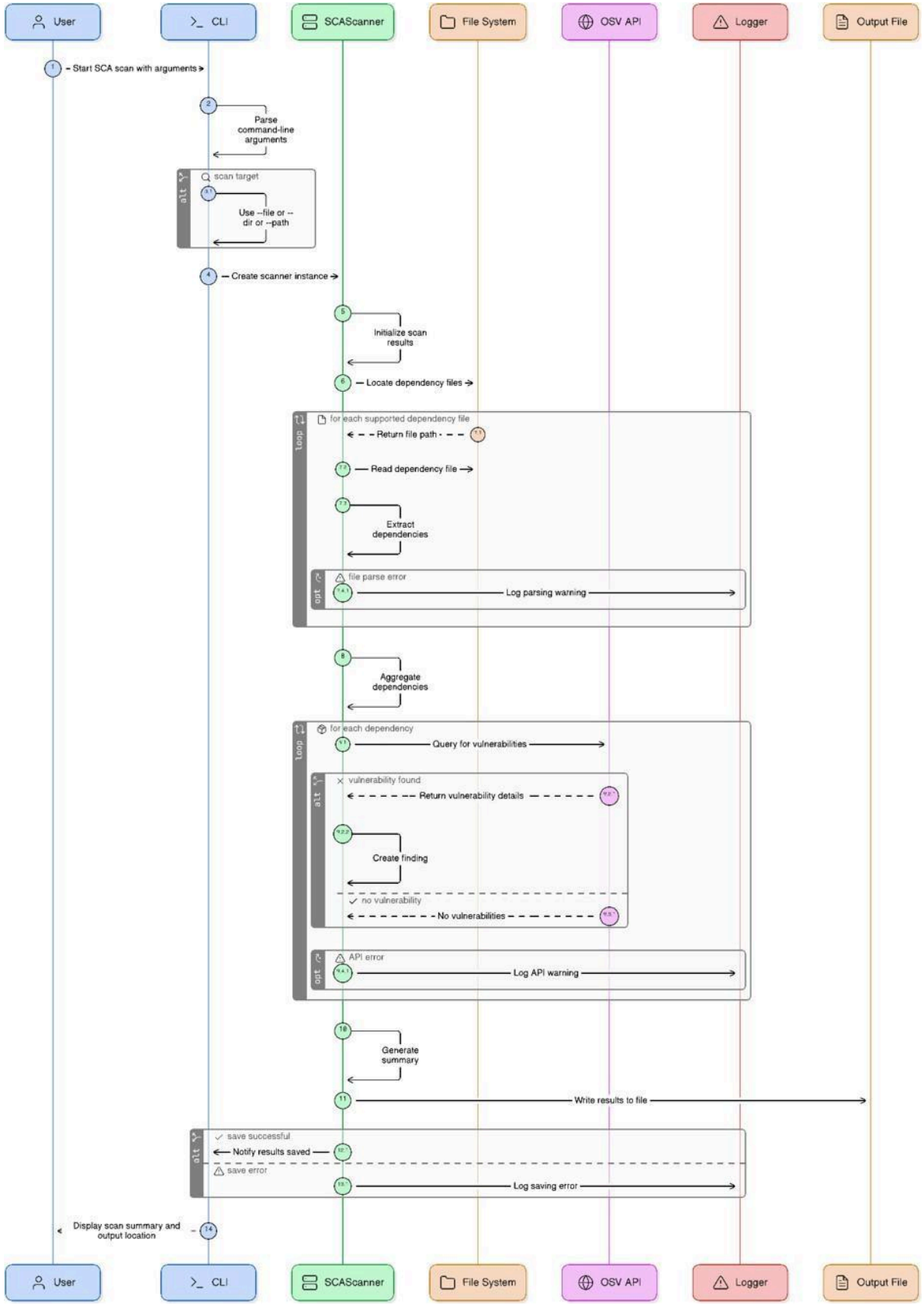

Рисунок 2.2 - Архітектура роботи інструмента SCA сканера

Узагальнена діаграма послідовності роботи Image сканера має такий вигляд:

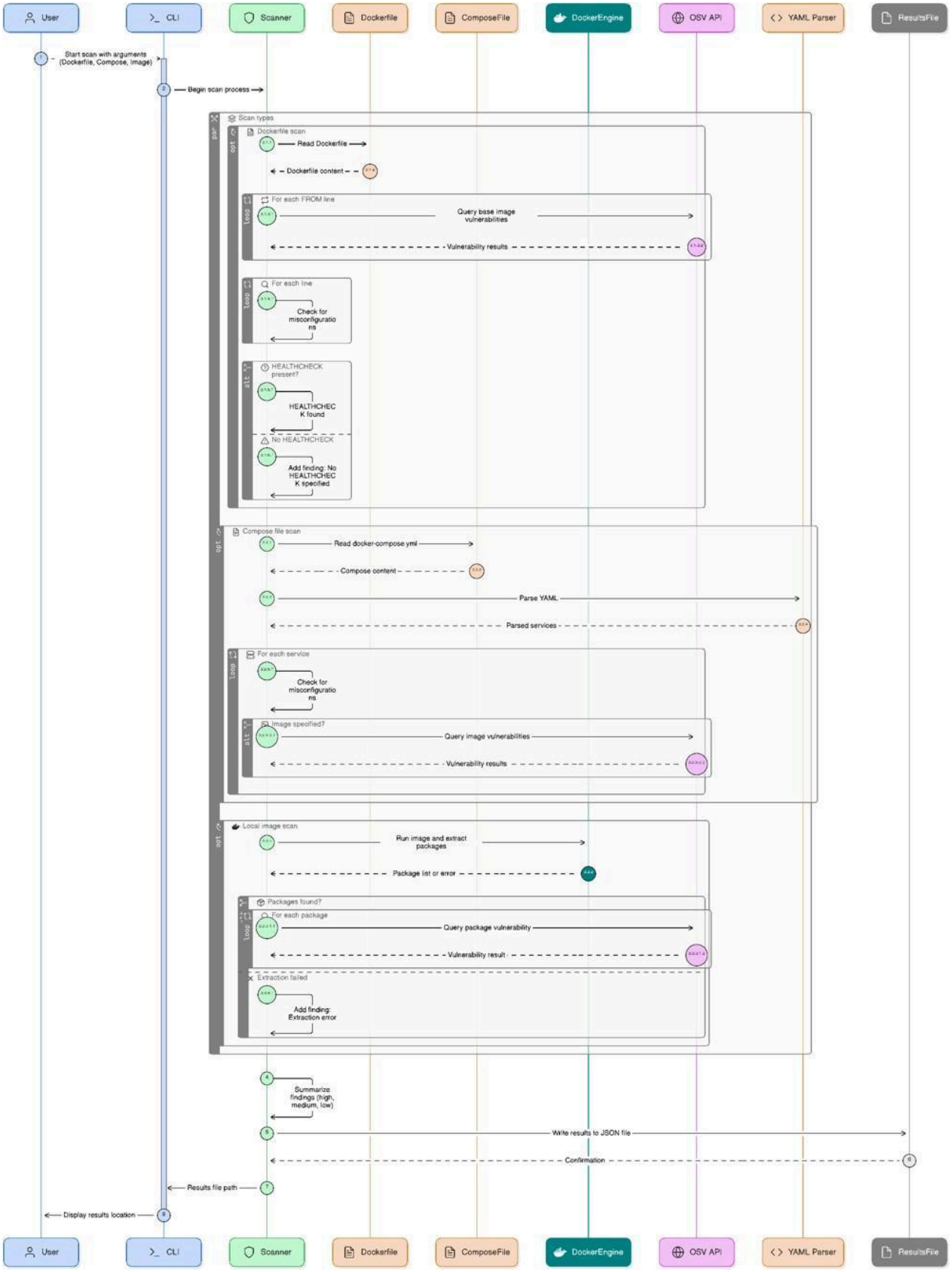

Рисунок 2.3 - Архітектура роботи інструмента Docker Image сканера

Узагальнена діаграма послідовності роботи SAST сканера має такий вигляд:

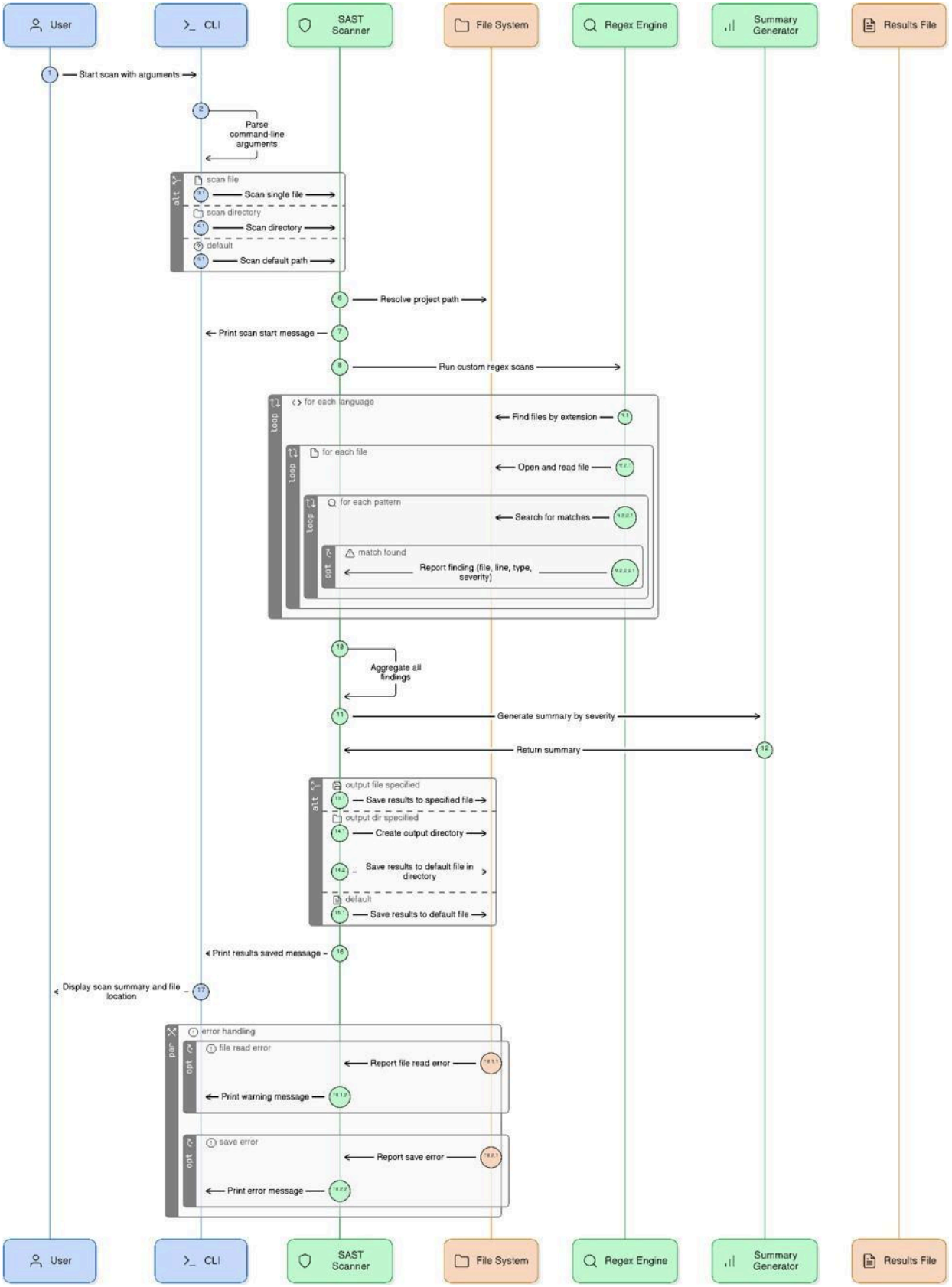

Рисунок 2.4 - Архітектура роботи інструмента SAST сканера

Проєкт має чітку модульну архітектуру, яка забезпечує гнучкість, тестованість і простоту інтеграції у DevOps-процеси. Надана модель класів і схема взаємодії дозволяють розпочати поетапну реалізацію: спочатку — мінімально працездатний прототип (CLI, Orchestrator, прості парсери й Reporter), потім — розширення правил аналізу, впровадження кешування, паралелізації і докеризації.

Висновок до Розділу 2

У цьому розділі було сформовано основні вимоги до програмного забезпечення, визначено його призначення, цільову аудиторію та обґрунтовано вибір технологій для подальшої реалізації. Проведено аналіз можливих мов програмування та інструментів, у результаті якого для розробки системи обрано Python завдяки його гнучкості, великій кількості бібліотек для аналізу коду й взаємодії з Docker, а також зручності інтеграції у DevSecOps-середовище. Було визначено архітектурні принципи побудови системи - модульність, розширюваність і кросплатформеність, що забезпечують простоту підтримки та масштабування. Розроблено технічне завдання, яке описує функціональні, нефункціональні та безпекові вимоги до програмного продукту. Окрему увагу приділено вимогам до сумісності з інструментами CI/CD, продуктивності, надійності та безпечного оброблення даних. Проведено проектування програмного забезпечення: описано загальну архітектуру системи, модель класів і взаємодію основних модулів. У межах проектування також розроблено єдину архітектуру інструментів безпеки, яка становить наукову новизну роботи. Створено уніфікований підхід до реалізації SAST-, SCA- та Docker Image-сканерів, що використовують спільне ядро, однакові принципи оброблення результатів і єдиний формат звітності. Це дозволило сформувати цілісну екосистему інструментів, які працюють узгоджено, взаємодіють через спільні абстракції та можуть легко розширюватися новими модулями.

Розроблена структура дозволяє реалізувати три окремі, але повністю узгоджені між собою сканери - SAST, SCA та Docker Image, які взаємодіють через

спільне ядро та уніфікований формат звітів. Таким чином, у цьому розділі закладено концептуальну основу майбутньої реалізації програмного забезпечення для аналізу безпеки коду, що стане базою для практичної розробки, тестування й інтеграції системи, розглянутих у наступному розділі.

РОЗДІЛ 3. РОЗРОБКА ТА ВИКОРИСТАННЯ ПЗ ДЛЯ СКАНУВАННЯ ВИХІДНОГО КОДУ

3.1 Розробка власного ПЗ

У цьому розділі представлено результати розробки комплексного інструментарію для аналізу безпеки програмних систем, що включає три окремі, але взаємодоповнюючі модулі: SCA Scanner, IMAGE Scanner та SAST Scanner. Кожен із них орієнтований на вирішення окремого класу завдань, пов’язаних із виявленням вразливостей у сучасному програмному забезпеченні, що дозволяє сформувати цілісний підхід до забезпечення безпеки на різних рівнях життєвого циклу розробки (SDLC). На відміну від монолітних рішень, у яких весь функціонал об’єднаний в одному продукті, реалізовані інструменти розділені між собою. Такий підхід має низку переваг:

- менший розмір кожного модуля;
- спрощене оновлення та внесення змін без ризику порушення роботи інших компонентів;
- можливість вибіркового використання (наприклад, застосовувати лише SCA або лише IMAGE Scanner у конкретному процесі CI/CD).

Також, на відміну від більшості сторонніх продуктів, створене рішення є відкритим, незалежним від комерційних сервісів та максимально гнучким. Завдяки модульній архітектурі кожен компонент може використовуватися як окремо, так і в комплексі, що робить інструментарій придатним для інтеграції у різні CI/CD процеси. Важливою перевагою є підтримка універсального формату збереження результатів у вигляді JSON-звітів. Це забезпечує можливість подальшої обробки даних, їх візуалізації або використання в автоматизованих системах контролю якості [13-14].

3.1.1 Розробка SCA сканера на вразливості

Основною метою розробки SCA сканера стала підвищення рівня безпеки програмного забезпечення за рахунок автоматизованого виявлення вразливих бібліотек і компонентів, що використовуються у процесі розробки. Розроблення інструмента відбувалося послідовно у кілька основних етапів. На початковому етапі проведено дослідження поширених форматів файлів залежностей, які використовуються у сучасних мовах програмування. Було визначено та детально проаналізовано такі конфігураційні файли, як requirements.txt (Python), composer.json (PHP), package.json (JavaScript/Node.js) та go.mod (Go). Кожен із них має власну структуру та правила опису бібліотек, тому для забезпечення універсальності роботи інструмента реалізовано окремі алгоритми обробки для кожного формату. Далі розроблено модуль парсингу залежностей, який автоматично зчитує зазначені файли та виділяє з них назви бібліотек і відповідні версії. Отримані дані зберігаються у структурованому вигляді для подальшої перевірки. На наступному етапі реалізовано модуль перевірки безпеки, який забезпечує взаємодію з відкритим сервісом OSV.dev API. Цей сервіс є загальнодоступною базою даних, що містить інформацію про відомі вразливості програмних компонентів із різних мов програмування. Для кожної знайденої залежності інструмент формує HTTP-запит до API та отримує детальні відомості про виявлені вразливості - зокрема, їхні ідентифікатори (CVE або OSV ID), рівень критичності (high, medium, low), короткий опис проблеми та потенційний вплив на систему. Після отримання результатів виконується їх уніфікація та обробка. Модуль обробки результатів формує зведений звіт, який містить список знайдених вразливостей, розподілених за рівнем небезпеки, а також загальну статистику: кількість високих, середніх і низьких ризиків. Для зручності інтеграції з іншими системами результати експортуються у форматі JSON, що дозволяє легко використовувати їх у CI/CD конвеєрах, автоматизованих процесах безпеки або сторонніх аналітичних інструментах. Інструмент побудований за модульною архітектурою з використанням об’єктно-орієнтованого підходу. Центральним

елементом системи є клас SCAScanner, який реалізує основну логіку пошуку, перевірки та обробки залежностей. Такий підхід забезпечує масштабованість і гнучкість системи, дозволяючи в подальшому додавати підтримку нових форматів файлів або альтернативних джерел даних про вразливості. Архітектура інструмента охоплює кілька ключових компонентів:

- Модуль пошуку файлів залежностей - автоматично визначає наявність підтримуваних конфігураційних файлів (requirements.txt, composer.json, package.json, go.mod) у заданому каталозі;
- Модуль парсингу - зчитує знайдені файли, аналізує їх вміст і формує перелік бібліотек із зазначенням версій;
- Модуль перевірки залежностей - здійснює обмін даними з OSV.dev API та отримує інформацію про відомі вразливості;
- Модуль обробки результатів - нормалізує отримані дані, визначає рівень критичності, створює підсумкову статистику та формує структуру звіту;
- Модуль збереження даних - відповідає за експорт результатів у форматі JSON;
- Інтерфейс командного рядка (CLI) - забезпечує зручний запуск програми з різними параметрами, такими як перевірка окремого файлу, сканування всієї директорії чи збереження результатів у визначену папку.

Загальна логіка роботи програми побудована за принципом послідовної обробки даних. Після запуску користувач у CLI вказує шлях до каталогу або файлу залежностей. Програма виконує пошук підтримуваних форматів, зчитує їхній вміст, а далі для кожної знайденої бібліотеки формує запит до сервісу OSV.dev. Отримані результати аналізуються, структуруються та зберігаються у JSON-файлі (sca_scan_results.json або іншому, визначеному користувачем).

Для зручності використання інструмент підтримує основні параметри командного рядка:

- --file - сканування конкретного файлу залежностей;
- --dir - сканування всього каталогу проєкту;

- --output - визначення імені файлу для збереження результатів;
- --output-dir - директорія для збереження JSON-файлу;
- --path - шлях до проєкту або каталогу за замовчуванням.

Система реалізує логування через стандартний модуль logging, що дозволяє відстежувати хід виконання сканування, діагностувати помилки й контролювати процес перевірки. З метою забезпечення універсальності, стабільності та простоти розгортання інструмента розроблено Dockerfile, який дозволяє запускати програму у контейнеризованому середовищі. Такий підхід забезпечує повторюваність результатів незалежно від операційної системи користувача, спрощує інтеграцію в DevOps-процеси та дає змогу легко вбудовувати інструмент у CI/CD пайплайни. Docker-середовище усуває потребу в ручному налаштуванні Python-залежностей, гарантуючи коректну роботу програми в будь-якому середовищі. Завдяки цьому інструмент може бути використаний як локально розробниками, так і на етапах автоматизованого тестування та розгортання в організаціях, що практикують підхід DevSecOps. Весь вихідний код інструмента опублікований у відкритому доступі на платформі GitHub за посиланням: https://github.com/mvhrabarfitm24m-bit/SCA_Scanner. У репозиторії представлено реалізацію основних модулів, алгоритми парсингу залежностей, приклади взаємодії з API та структуру сформованих звітів. Відкритий код дає змогу іншим дослідникам або розробникам використовувати програму у власних проєктах, модифікувати її під специфічні потреби чи інтегрувати в інші системи управління вразливостями.

3.1.2 Розробка Docker Image сканера на вразливості

Необхідність створення Docker image сканера зумовлена зростанням популярності контейнеризації у сучасних DevOps-процесах, що, своєю чергою, створює нові вектори атак через неправильно налаштовані або застарілі базові образи. Для досягнення поставленої мети було виконано низку завдань, спрямованих на побудову комплексного рішення для безпечного аналізу

контейнерних компонентів. На початковому етапі було проаналізовано типові підходи до забезпечення безпеки контейнерів, існуючі методи сканування образів (Trivy, Clair, Grype) та визначено недоліки, які впливають на гнучкість і швидкість інтеграції таких інструментів у DevSecOps-процеси. На основі проведеного аналізу сформульовано вимоги до власного рішення: незалежність від зовнішніх платформ, можливість використання відкритого API (OSV.dev), універсальність щодо типів файлів та контейнерних систем. Далі розроблено модуль аналізу Dockerfile, який виконує перевірку базових образів через OSV.dev API, а також виявляє поширені помилки конфігурації - використання користувача root, тегу latest, відсутність HEALTHCHECK, копіювання файлів у /root, незахищені RUN-команди тощо. Для цього реалізовано набір регулярних виразів та функцій валідації, що дозволяють ефективно виявляти потенційно небезпечні практики. На наступному етапі створено модуль перевірки docker-compose.yml, який виконує аналіз конфігурацій сервісів на предмет використання привілейованого режиму (privileged: true), автоматичного перезапуску (restart: always), некоректних тегів образів (latest) та небезпечних параметрів мережевої конфігурації. Кожен сервіс, виявлений у файлі, додатково перевіряється на наявність вразливостей базових образів через OSV.dev API. Окремо реалізовано модуль для перевірки локальних контейнерних образів, який дозволяє аналізувати встановлені всередині них пакети. Інструмент автоматично визначає тип дистрибутиву (Debian, Alpine, RedHat), витягує список інстальованих пакетів і перевіряє кожен із них на наявність вразливостей через запит до OSV.dev API. Такий підхід забезпечує повноцінний аналіз навіть тих контейнерів, які вже побудовані локально та не мають прямого зв’язку з Docker Hub. Після збору результатів усі знайдені проблеми проходять уніфікацію та класифікацію за рівнем критичності (high, medium, low). Модуль обробки результатів формує структурований звіт, який містить опис кожної знайденої проблеми, її джерело, рівень ризику та рекомендацію. Підсумковий звіт зберігається у форматі JSON, що дає змогу легко інтегрувати результати в CI/CD процеси, системи моніторингу безпеки або інші автоматизовані інструменти DevSecOps. Інструмент побудований за модульною архітектурою з використанням

об’єктно-орієнтованого підходу. Центральним елементом системи є клас SCAScanner, який реалізує основну логіку пошуку, перевірки та обробки залежностей. Такий підхід забезпечує масштабованість і гнучкість системи, дозволяючи в подальшому додавати підтримку нових форматів файлів або альтернативних джерел даних про вразливості.

Архітектура IMAGE Scanner побудована за модульним принципом і включає такі основні компоненти:

- Модуль аналізу Dockerfile - перевіряє базові образи через OSV.dev API та виконує виявлення помилок конфігурації;
- Модуль аналізу docker-compose.yml - здійснює перевірку налаштувань сервісів і базових образів;
- Модуль перевірки локальних образів - визначає дистрибутив, витягує встановлені пакети та перевіряє їх на вразливості;
- Модуль обробки результатів - формує структуру звіту та підраховує статистику ризиків;
- Модуль збереження даних - експортує результати у форматі JSON;
- Інтерфейс командного рядка (CLI) – забезпечує гнучке використання інструмента завдяки підтримці параметрів:
    - --dockerfile - аналіз Dockerfile;
    - --compose - аналіз docker-compose.yml;
    - --image - перевірка локального контейнерного образу;
    - --output - ім’я файлу для збереження результатів;
    - --output-dir - каталог для виводу звітів.

Загальна логіка роботи програми побудована за принципом послідовної обробки даних. Після запуску користувач через CLI задає тип об’єкта для сканування. Програма виконує вибірку відповідного файлу або образу, проводить аналіз, формує перелік виявлених проблем та зберігає результати у JSON-звіт. Для кожної перевірки зазначається час виконання, кількість знайдених проблем та

рівень їх критичності. Для забезпечення стабільності, універсальності та простоти розгортання інструмента розроблено Dockerfile, який дозволяє запускати IMAGE Scanner у контейнеризованому середовищі. Це забезпечує повторюваність результатів, незалежність від операційної системи користувача та спрощує інтеграцію у DevOps-конвеєри. Завдяки цьому інструмент може бути використаний як у локальному середовищі розробника, так і в автоматизованих процесах CI/CD. Весь вихідний код програми опубліковано у відкритому доступі на GitHub за посиланням: https://github.com/mvhrabarfitm24m-bit/IMAGE_Scanner. У репозиторії представлено вихідний код усіх модулів, алгоритми аналізу Dockerfile, docker- compose.yml і локальних образів, а також приклади звітів. Відкрита структура коду дозволяє адаптувати інструмент під специфічні вимоги, розширювати функціонал або інтегрувати його з іншими системами управління вразливостями.

3.1.3 Розробка SAST сканера на вразливості

Основною метою розробки SAST Scanner стало підвищення рівня безпеки програмного забезпечення шляхом автоматизованого виявлення вразливостей безпосередньо у вихідному коді. На відміну від інструментів аналізу залежностей, цей сканер орієнтований на виявлення небезпечних конструкцій, логічних помилок та потенційних ризиків, що виникають у процесі написання коду. Розроблення інструмента здійснювалося послідовно у кілька етапів. На початковому етапі було проведено аналіз підходів до статичного аналізу коду (SAST) і визначено найпоширеніші типи вразливостей, притаманні різним мовам програмування - зокрема SQL-ін’єкції, XSS (міжсайтове виконання скриптів), небезпечне виконання системних команд, використання ненадійних функцій, витік конфіденційних даних та помилки конфігурації. На основі цього аналізу сформовано вимоги до архітектури майбутнього сканера та визначено набір мов, що підтримуються (.py,
.js, .php, .go та ін.). На наступному етапі реалізовано модуль пошуку вихідного коду, який рекурсивно сканує заданий каталог і визначає файли з підтримуваними

розширеннями. При цьому ігноруються службові директорії, такі як .git,

node_modules, venv, щоб підвищити продуктивність аналізу. Центральним компонентом інструмента є модуль перевірки на вразливості, який реалізує набір правил і сигнатур для кожної підтримуваної мови програмування. Наприклад, для Python виявляються небезпечні виклики eval(), exec(), subprocess.Popen() без валідації даних; для PHP - використання функцій mysql_query() або eval() з неперевіреними параметрами; для JavaScript - виклики innerHTML, document.write() без очищення вхідних даних. Кожна знайдена потенційна проблема фіксується у вигляді структурованого запису із зазначенням файлу, номера рядка, типу вразливості, рівня критичності (high/medium/low) та короткого опису проблеми. Після завершення перевірки результати проходять етап обробки та уніфікації. Модуль обробки результатів формує зведений звіт, який містить повну інформацію про знайдені вразливості, а також статистичні показники — кількість високих, середніх та низьких ризиків. Для зручності інтеграції у DevSecOps процеси результати експортуються у форматі JSON, що дозволяє легко передавати їх у CI/CD конвеєри, системи управління вразливостями або інші аналітичні інструменти. Архітектура інструмента побудована на основі об’єктно-орієнтованого підходу. Основним елементом є клас SASTScanner, який об’єднує основну логіку пошуку, парсингу, аналізу та формування звітів. Така модульна структура забезпечує гнучкість і розширюваність системи - у майбутньому можна легко додати нові правила перевірки або підтримку інших мов програмування.

Структурно SAST Scanner складається з кількох ключових компонентів:

- Модуль пошуку вихідного коду - здійснює рекурсивний пошук файлів із підтримуваними розширеннями (.py, .js, .php, .go тощо);
- Модуль парсингу - будує AST або застосовує регулярні вирази для аналізу синтаксичних конструкцій;
- Модуль перевірки на вразливості - виконує пошук небезпечних шаблонів та сигнатур відповідно до набору правил безпеки;

- Модуль обробки результатів - формує структуровані дані про знайдені вразливості, класифікує їх за рівнем критичності та створює зведену статистику;
- Модуль збереження результатів - експортує результати у форматі JSON для подальшої інтеграції;
- Інтерфейс командного рядка (CLI) - надає можливість запуску сканування окремих файлів, каталогів або всього проєкту та задає параметри збереження результатів.

Загальна логіка роботи програми побудована за принципом послідовної обробки даних. Після запуску користувач через CLI задає шлях до файлу або каталогу, що необхідно перевірити. Програма виконує пошук відповідних файлів, парсить їх, застосовує набір правил перевірки та формує підсумковий звіт. Результати зберігаються у файл sast_scan_results.json або інший, визначений користувачем. Інструмент підтримує такі основні параметри командного рядка:

- --file - сканування конкретного файлу вихідного коду;
- --dir - сканування всього каталогу проєкту;
- --output - визначення імені файлу для збереження результатів;
- --output-dir - вибір директорії для збереження JSON-звіту;
- --path - шлях до каталогу або файлу за замовчуванням.

Для забезпечення стабільності та незалежності від операційної системи створено Dockerfile, який дозволяє запускати сканер у контейнеризованому середовищі. Це рішення гарантує повторюваність результатів, усуває потребу у встановленні зовнішніх бібліотек та забезпечує легку інтеграцію інструмента у CI/CD пайплайни. Docker також дозволяє використовувати сканер як локально під час розробки, так і у хмарних або корпоративних середовищах, що практикують підхід DevSecOps. Усі вихідні коди проєкту опубліковані у відкритому доступі на GitHub за посиланням: https://github.com/mvhrabarfitm24m-bit/SAST_Scanner. У репозиторії представлено реалізацію основних модулів, приклади правил виявлення вразливостей, алгоритми побудови AST та структуру сформованих

звітів. Відкритий код дозволяє дослідникам і розробникам розширювати функціонал, інтегрувати сканер у власні DevSecOps рішення або використовувати його в освітніх цілях для практичного вивчення методів статичного аналізу безпеки.

3.2 Використання розробленого ПЗ під час локальної перевірки коду

У рамках симуляції було створено прикладову ситуацію, коли розробник працює над власним програмним забезпеченням та має локальну робочу директорію з готовим проєктом на своєму комп'ютері. Для прикладу використано операційну систему Windows 11, однак цей аспект не є принциповим, оскільки структура та логіка роботи коду залишаються універсальними. Розробка здійснювалася мовою Python, що повністю підтримується створеним сканером у процесі виявлення вразливостей. Саме тому обраний стек технологій є зручним варіантом для демонстрації функціоналу інструменту. Крім основного коду додатку, проєкт містить:

- Файл залежностей (requirements.txt), у якому визначені зовнішні бібліотеки, необхідні для роботи додатку. Деякі з них мають відомі уразливості у старих версіях, що дозволяє продемонструвати можливості SCA (Software Composition Analysis).
- Docker-імедж із конфігурацією для запуску веб-додатку у контейнеризованому середовищі. Його специфікація (docker-compose.yml) містить параметри, що створюють потенційні ризики безпеки (наприклад, використання застарілого образу або запуск контейнера у привілейованому режимі).

Таким чином, тестове середовище відтворює реалістичну ситуацію з типовим сучасним проєктом: вихідний код, залежності та контейнеризація. Це дозволяє перевірити роботу сканера одразу у трьох напрямках — SAST (аналіз коду), SCA (аналіз залежностей) та Container Scanning (перевірка конфігурацій контейнерів).

3.2.1 Приклад роботи власного SAST сканера локально

У якості тестового прикладу було реалізовано невеликий веб-додаток на Flask, який за своїм виглядом і структурою імітує типовий сервіс, що міг би використовуватися у реальних умовах. Водночас він навмисно містить низку класичних уразливостей безпеки, характерних для багатьох проєктів на ранніх стадіях розробки. Завдяки цьому такий код може слугувати навчальною базою та платформою для тестування як статичних, так і динамічних інструментів аналізу безпеки.

Детальний код додатку:

sast_test.py > ...

```python
from flask import Flask, request
import os
import pickle

app = Flask(__name__)

# Hardcoded secret key
app.config['SECRET_KEY'] = "supersecret123"


@app.route("/")
def index():
    return "Welcome to vulnerable app!"


# SQL Injection
@app.route("/login")
def login():
    username = request.args.get("user")
    query = "SELECT * FROM users WHERE name = '" + username + "'"
    # Імітація виконання SQL-запиту
    return f"Executing query: {query}"


# Command Injection
@app.route("/run")
def run_command():
    cmd = request.args.get("cmd")
    return os.popen(cmd).read()


# Insecure deserialization
@app.route("/load", methods=["POST"])
def load_data():
    data = request.data
    return pickle.loads(data)  # Небезпечно


# Path Traversal
@app.route("/file")
def read_file():
    filename = request.args.get("name")
    with open("/tmp/" + filename) as f:
        return f.read()


if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

Рисунок 3.1 - Вразливий Python код

Для демонстрації можливостей інструменту було здійснено перевірку створеного тестового проєкту за допомогою SAST-сканера, який запускається у середовищі розробника у форматі CLI-команди: docker run --rm -v "${PWD}\SAST_Test:/data" sast-scanner --dir /data/sast_test.py --output-dir /data --output sast_results_docker.json.

```
PS C:\Users\makso\OneDrive\Desktop\Магістерська робота\Code\SAST> docker run --rm -v "${PWD}\SAST_Test:/data" sast-scanner --dir /data/sast_t
est.py --output-dir /data --output sast_results_docker.json
🔍 Starting SAST & Secrets scan for: /data/sast_test.py
💾 Results saved to: /data/sast_results_docker.json
```

Рисунок 3.2 - Робота власного SAST сканера

У процесі аналізу сканер пройшовся по вихідному коду та застосував набір правил для пошуку потенційно небезпечних конструкцій. Після завершення роботи було сформовано структурований звіт про виявлені проблеми:


```json
{
  "scan_timestamp": "2025-09-20T05:32:11.579972",
  "project_path": "/data/sast_test.py",
  "findings": [
    {
      "id": "b3a339a81bcfc49f1a5f06fb2de91cbb",
      "type": "code_vulnerability",
      "severity": "medium",
      "file": "/data/sast_test.py",
      "line": 29,
      "message": "Use of os.popen(): os.popen(...",
      "source": "regex-python"
    },
    {
      "id": "3790eea7588e49a592b168e321e74146",
      "type": "sql_injection",
      "severity": "high",
      "file": "/data/sast_test.py",
      "line": 20,
      "message": "SQL query with string concatenation (possible SQL injection): query = \"SELECT * FROM users WHERE name = '\" + use...",
      "source": "regex-python"
    },
    {
      "id": "8ce3256af20a9cd6a6b14fba5efa184a",
      "type": "deserialization",
      "severity": "high",
      "file": "/data/sast_test.py",
      "line": 36,
      "message": "Use of pickle.loads() (unsafe deserialization): pickle.loads(...",
      "source": "regex-python"
    },
    {
      "id": "1d17db32a946002a930069a853a1c6c9",
      "type": "path_traversal",
      "severity": "medium",
      "file": "/data/sast_test.py",
      "line": 43,
      "message": "File path manipulation (possible path traversal): open(\"/tmp/\" + filename)...",
      "source": "regex-python"
    }
  ],
  "summary": {
    "total_findings": 4,
    "high": 2,
    "medium": 2,
    "low": 0,
    "scan_methods": [
      "regex"
    ]
  },
  "scan_methods": [
    "regex"
  ]
}
```


Рисунок 3.3 - Результат роботи SAST сканера

Результати перевірки підтвердили коректність роботи інструменту - усі закладені уразливості були успішно знайдені. Це включає як використання небезпечних функцій (eval, exec, os.system), так і типові приклади ін’єкцій (SQL, Path Traversal), роботу з небезпечними модулями (pickle.loads), а також інші ризиковані практики на кшталт хардкоджених паролів. Таким чином, експеримент показав, що навіть на базовому рівні реалізації сканер здатний виявляти ключові класи вразливостей та надавати результати у зручному для подальшої обробки форматі. Нажаль, під час експерименту мені не вдалося коректно запустити сторонні SAST-сканери (Semgrep та Bearer) локально через некоректні помилки в їх роботі на Windows. Однак це не вплинуло на результати, оскільки я свідомо додав у код низку вразливостей, і мій власний сканер успішно виявив майже всі з них. Тому експеримент із порівняння якості виявлення вразливостей вважаю успішним.

3.2.2 Приклад роботи власного SCA сканера локально

У якості тестового прикладу для перевірки роботи інструменту було створено файл requirements.txt, який містить перелік сторонніх бібліотек, необхідних для функціонування програмного забезпечення. Для симуляції реальної ситуації до цього списку були свідомо включені версії пакетів із відомими уразливостями.

Вміст файлу з залежностями (requirements.txt):

```
django==1.11
requests==2.19.0
flask==0.12
```

Для демонстрації можливостей інструменту було здійснено перевірку створеного тестового проєкту за допомогою SCA-сканера, який запускається у середовищі розробника у форматі CLI-команди: docker run --rm -v "${PWD}/SCA_Test:/data" sca-scanner --dir /data --output-dir /data --output sca_results_docker.json.

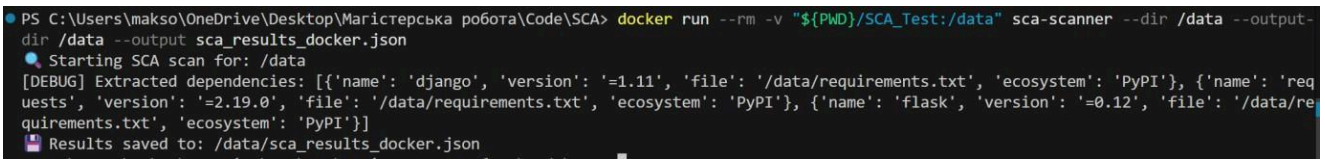

Рисунок 3.4 - Робота власного SCA сканера

У процесі аналізу модуль SCA здійснив парсинг файлу requirements.txt, розпізнав усі вказані залежності та визначив їхні версії. Далі кожна з бібліотек була перевірена через відкриті бази даних відомих уразливостей, що дало можливість автоматично виявити відповідні CVE-записи. Після завершення роботи було сформовано структурований звіт про виявлені проблеми:

Рисунок 3.5 - Результат роботи SCA сканера

Результати перевірки підтвердили коректність роботи інструменту - усі закладені уразливості у залежностях були успішно виявлені. Це включає бібліотеки з відомими вразливостями (CVE), застарілі або небезпечні версії пакунків (Flask,

Django, Requests) та інші проблемні компоненти. Це демонструє ефективність інструменту у швидкому виявленні ризикових компонентів та підвищує загальну безпеку програмного забезпечення.

Для порівняння якості роботи свого сканеру я запустив Trivy на цей же файл із залежностями:

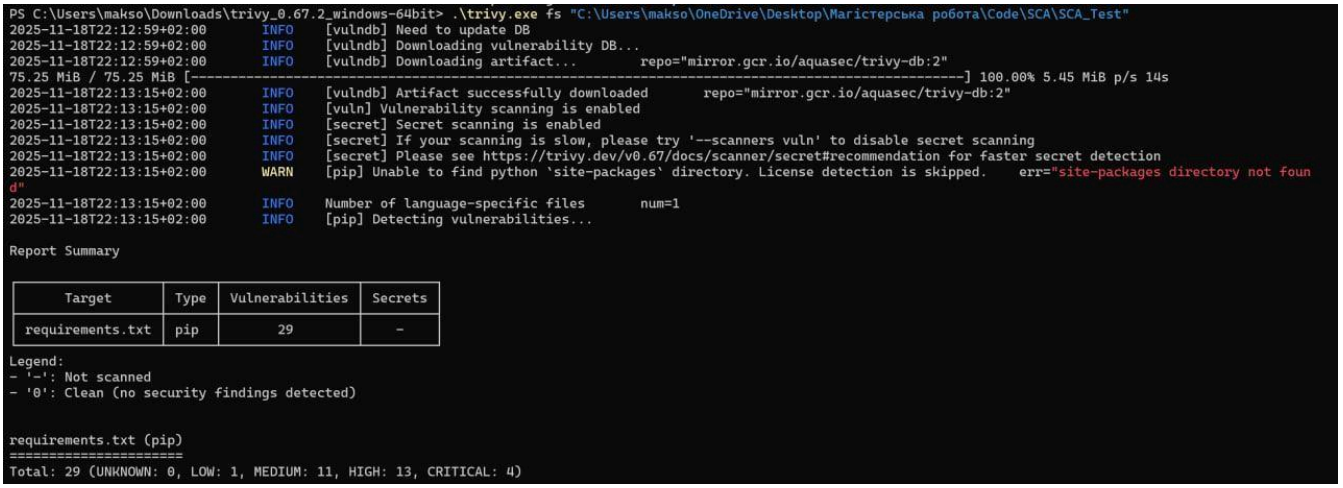

Рисунок 3.6 - Результат роботи trivy сканера

Він виявив більше вразливостей у залежностях, однак мій сканер спеціально уникає дублювання та показує лише одну вразливість на бібліотеку. Це робить результати компактнішими й легшими для сприйняття, тому я вважаю, що мій сканер впорався точно не гірше.

3.2.3 Приклад роботи власного Docker Image сканера локально

У якості тестового прикладу для перевірки роботи інструменту було створено файл docker-compose.yml, який описує запуск тестового додатка у контейнеризованому середовищі. Для симуляції реальної ситуації до конфігурації свідомо додані потенційно небезпечні налаштування, такі як використання USER root та WORKDIR /root.

Вміст файлу Dockerfile:

```
FROM drupal:8.7.0
USER root
ADD . /app
WORKDIR /root
RUN apt-get update && apt-get install -y curl
EXPOSE 80
```

Для демонстрації можливостей інструменту було здійснено перевірку створеного Docker-проєкту за допомогою SCA/Docker-сканера, який запускається у середовищі розробника у форматі CLI-команди: docker run --rm -v "${PWD}/IMAGE_Test:/data" image-scanner --dockerfile /data/Dockerfile --output-dir /data --output dockerfile_results_docker.json.

Рисунок 3.7 - Робота власного Docker Image сканера

У процесі аналізу модуль сканера здійснив парсинг файлу docker-compose.yml, ідентифікував усі вказані сервіси, базові образи, порти та налаштування контейнерів. Кожен компонент було перевірено на наявність відомих уразливостей через відкриті бази даних CVE, а також на потенційно ризикові конфігурації, такі як privileged або restart: always. Після завершення роботи було сформовано структурований звіт про виявлені проблеми:

IMAGE_Test > {} dockerfile_results_docker.json > [ ] findings > { } 1 > abc source


```json
{
  "scan_timestamp": "2025-11-18T19:22:25.762108",
  "findings": [
    {
      "type": "misconfiguration",
      "severity": "high",
      "line": 2,
      "message": "Container runs as root user",
      "source": "custom_rule"
    },
    {
      "type": "misconfiguration",
      "severity": "medium",
      "line": 3,
      "message": "Use of ADD instead of COPY",
      "source": "custom_rule"
    },
    {
      "type": "misconfiguration",
      "severity": "low",
      "line": 4,
      "message": "Working directory is /root (avoid running as root)",
      "source": "custom_rule"
    },
    {
      "type": "misconfiguration",
      "severity": "low",
      "line": 6,
      "message": "Exposes HTTP port 80 (consider HTTPS)",
      "source": "custom_rule"
    },
    {
      "type": "misconfiguration",
      "severity": "low",
      "line": null,
      "message": "No HEALTHCHECK specified",
      "source": "custom_rule"
    }
  ],
  "summary": {
    "total_findings": 5,
    "high": 1,
    "medium": 1,
    "low": 3,
    "scan_methods": [
      "dockerfile"
    ]
  },
  "scan_methods": [
    "dockerfile"
```


Рисунок 3.8 - Результат роботи Docker Image сканера

Результати перевірки підтвердили коректність роботи інструменту — усі закладені уразливості були успішно виявлені. Це включає небезпечні налаштування контейнерів, застарілі або вразливі версії базових образів та інші проблемні компоненти. Це демонструє ефективність інструменту у швидкому виявленні ризикових компонентів у контейнерних середовищах та підвищує загальну безпеку розгортання додатків.

Для порівняння якості роботи свого сканеру я запустив Trivy на цей же файл із місконфігураціями:

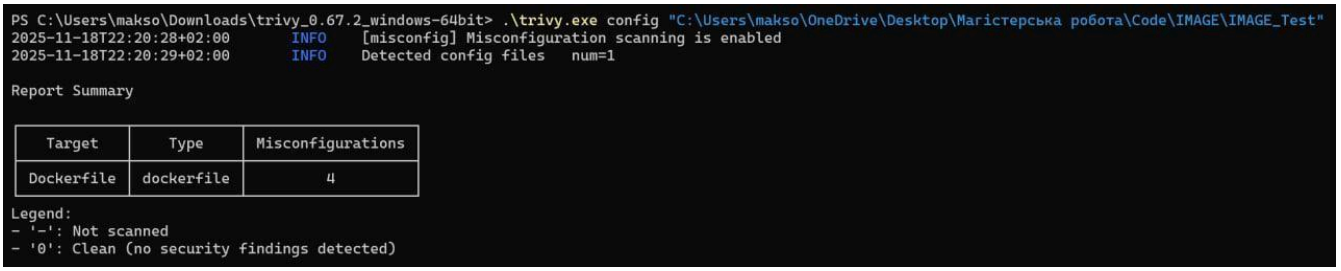

Рисунок 3.9 - Результат роботи trivy сканера

Мій сканер виявив більше місконфігураційних помилок у коді, в саме вказав на те, що в мене все відбувається в директорії root, що є погано. Це робить результати більш інформативним, тому я вважаю, що мій сканер впорався краще.

3.3 Використання розробленого ПЗ під час перевірки коду у CI/CD

У попередньому розділі я продемонстрував запуск власноруч створених контейнерних сканерів локально. Такий підхід дає змогу перевірити працездатність інструментів і відлагодити процеси аналізу безпосередньо на робочій машині. Однак у реальних умовах розробки цього недостатньо: код змінюється постійно, і ручний запуск сканування стає малоефективним. Щоб гарантувати безпеку програмного забезпечення на всіх етапах його життєвого циклу, необхідно інтегрувати сканери у конвеєри безперервної інтеграції та доставки (CI/CD). Це дозволяє:

- автоматично виконувати аналіз під час кожного коміту чи створення нової збірки;
- виявляти вразливості на ранніх стадіях розробки;
- забезпечувати однакові умови запуску завдяки використанню контейнерних образів;
- централізовано керувати інструментами без додаткового налаштування на кожній машині [19].

3.3.1 Розповсюдження власного ПЗ через публічний Docker-реєстр

Для реалізації цього підходу я підготував заздалегідь створені образи сканерів та завантажив їх у віддалений Docker-реєстр. Це дозволяє підключати їх у пайплайни незалежно від середовища виконання та робити процес сканування коду відтворюваним, прозорим і масштабованим [20]. Кроки, необхідні для виконання при завантаженні локальних імеджів у віддалений Docker-реєстр:

1. Для початку необхідно було здійснити автентифікацію в Docker Hub за допомогою команди: docker login
2. Щоб завантажити локальні образи у віддалений реєстр, потрібно спершу прив’язати їх до репозиторіїв у форматі <username>/<repo>:<version>. Це здійснюється за допомогою команди docker tag. Наприклад, для SAST-сканера: docker tag sast-scanner:latest testik1111/sast-scanner:1.0 Аналогічним чином були протеговані образи SCA-сканера та image-сканера.
3. Наступним кроком стало завантаження образів у Docker Hub. Для цього використовується команда: docker push <username>/<repo>:<version>. Приклад для SAST-сканера: docker push testik1111/sast-scanner:1.0 У результаті всі шари образу були успішно передані у віддалений реєстр.
4. Після завершення всіх операцій у моєму обліковому записі на Docker Hub з’явилися три публічні репозиторії з тегом 1.0:
    - testik1111/sast-scanner
    - testik1111/sca-scanner
    - testik1111/image-scanner

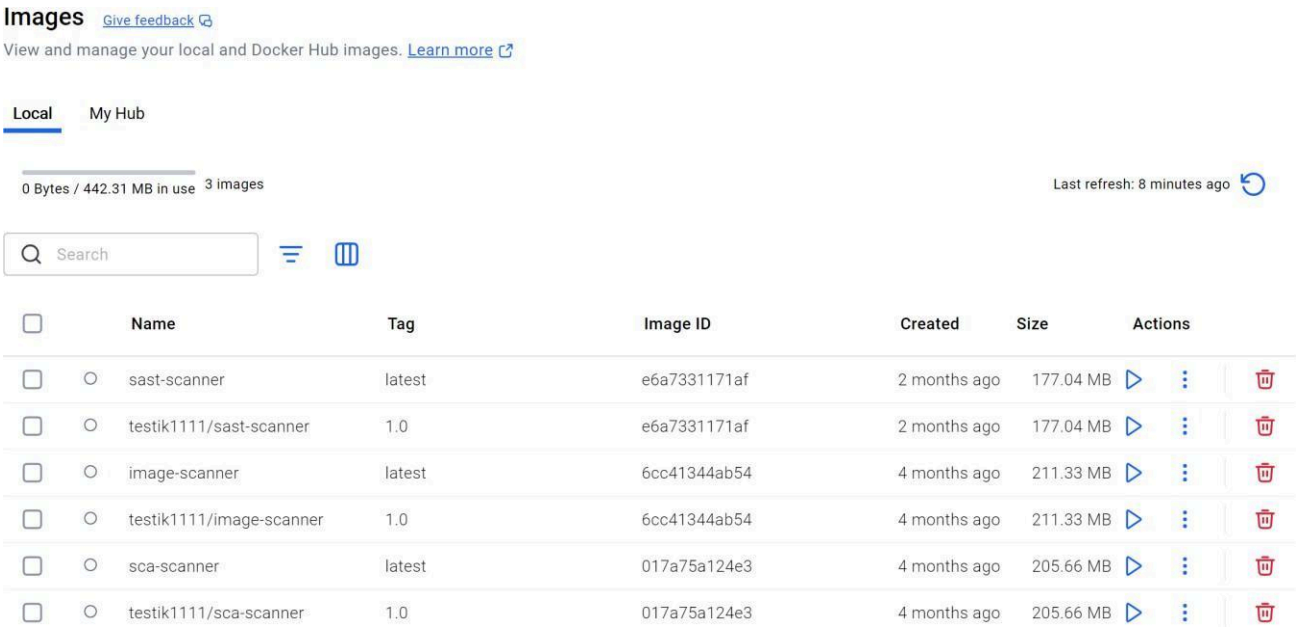

Рисунок 3.10 - Наявність кастомних docker імеджів на Docker Hub

Таким чином, створені образи сканерів були централізовано завантажені у віддалений Docker-реєстр. Це дало змогу відокремити процес їх підготовки від процесу використання, а також забезпечило незалежність від конкретного середовища виконання. Надалі ці образи можна інтегрувати у пайплайни CI/CD, що гарантує автоматичне та відтворюване сканування коду при кожній збірці [20].

3.3.2 Використання власних сканерів безпеки у процесі CI/CD

У процесі розробки програмного забезпечення часто виникає потреба у ранньому виявленні помилок та потенційних уразливостей. Для цього було інтегровано автоматизовані сканери, які дозволяють перевіряти код та залежності на кожному коміті, не чекаючи завершення розробки або ручного тестування. Для демонстрації автоматизованого контролю якості та безпеки коду був створений репозиторій Test_Security_Scanners на платформі GitHub. У цьому репозиторії відбувається активна розробка проекту, який імітує реальний програмний продукт. Розробка ведеться у централізованому середовищі GitHub, що забезпечує:

- Контроль версій коду - відстеження всіх змін та історії комітів;

- Спільну роботу команди - розробники можуть паралельно працювати над різними частинами проекту та подавати зміни через Pull Request;
- Чітку організацію структури проекту, включно з основним кодом (app.py), файлами залежностей (requirements.txt) та конфігурацією Docker середовищ (docker-compose.yml);
- Інтеграцію прикладів тестових даних для демонстрації SAST, SCA та Image сканерів [21].

Щоб забезпечити автоматичне та постійне сканування коду, у репозиторії було використано GitHub Actions - інструмент CI/CD, який дозволяє:

- Автоматично виконувати перевірки на кожному коміті або при створенні Pull Request;
- Гарантувати повторюваність процесів незалежно від середовища розробника;
- Забезпечити прозорість результатів завдяки збереженню звітів як артефактів;
- Масштабувати процес – нові сканери та перевірки можна легко додати у workflow.

Для перевірки використання моїх сканерів безпеки було створено власний workflow у репозиторії Test_Security_Scanners. Він організований таким чином, що на кожному коміті або Pull Request автоматично запускаються три паралельні jobs: SAST, SCA та Image scan (для прикладу сканери запускаються разом, але якщо є необхідність - можна розділити їх роботу).

Загальний алгоритм workflow

1. Три jobs запускаються паралельно, що дозволяє скоротити час перевірки коду.
2. Монтування директорії проекту у контейнер гарантує доступ сканера до всього коду та конфігурацій.
3. Використання публічних Docker-образів дозволяє не виконувати логін у Docker Hub, спрощує інтеграцію.
4. Збереження результатів у директорії reports/ забезпечує централізоване накопичення звітів.

5. Завантаження JSON-файлів як артефактів дозволяє розробникам та аналітикам переглядати результати сканування прямо в GitHub, без необхідності локального запуску сканерів [22].

Детальний код workflow можна знайти за посиланням: https://github.com/mvhrabarfitm24m-bit/Test_Security_Scanners/blob/main/.github/workflows/test.yml

Результат роботи workflow:

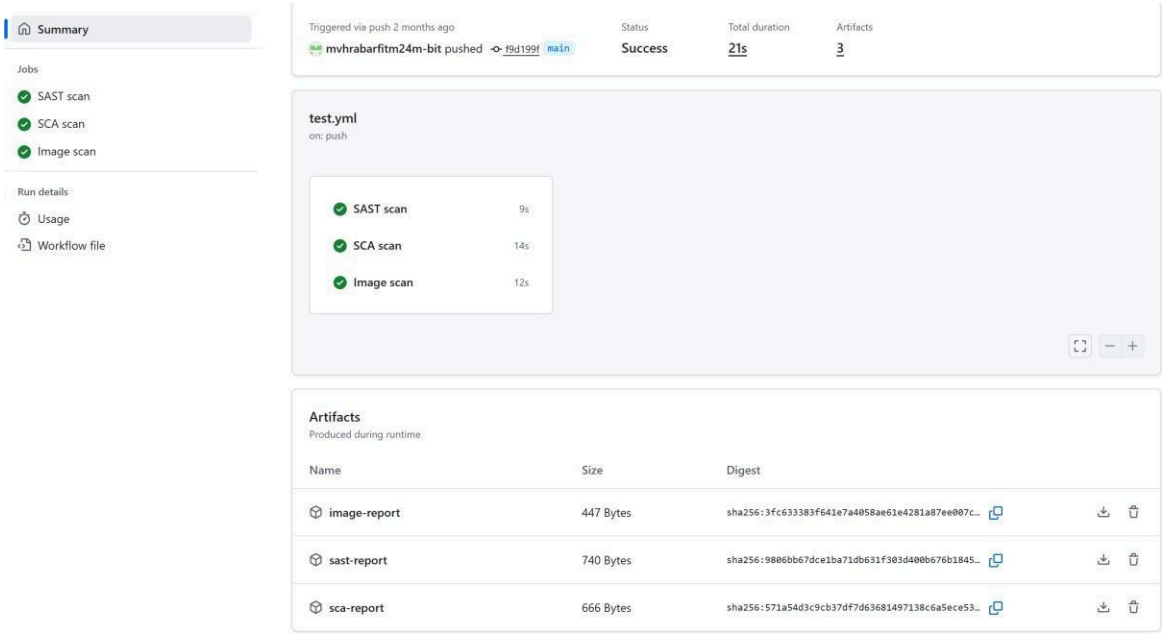

Рисунок 3.11 - Успішне сканування коду в GitHub CI/CD

У результаті ми отримали 3 файли з результататати сканувань, які автоматично завантажуються як артефакти GitHub Actions. Переглянути їх можна у вкладці Actions репозиторію, обравши конкретний запуск workflow та відповідний job. Завантаження цих файлів відбувається в архів, де міститься json файл з результатми сканувань. Отже, у рамках реалізації автоматизованого контролю якості та безпеки коду ми створили репозиторій Test_Security_Scanners та інтегрували у нього процес сканування за допомогою GitHub Actions. Було підготовлено три паралельні jobs: SAST, SCA та Image scan, які автоматично виконуються на кожному коміті або Pull Request. Таким чином, інтеграція сканерів у CI/CD workflow підвищує безпеку

проекту, економить час на ручні перевірки та формує основу для подальшого масштабування процесів автоматизації безпеки.

Висновок до Розділу 3

У цьому розділі було продемонстровано практичне застосування розробленого програмного забезпечення для забезпечення безпеки коду та середовища розроблення. Було розроблено та показано роботу інструментів, які дозволяють виконувати аналіз вихідного коду, перевірку залежностей і контейнерних образів на наявність помилок та відомих уразливостей. Запропоновані рішення забезпечують можливість виявлення проблем ще до етапу інтеграції у CI/CD, що сприяє підвищенню безпеки та якості програмного продукту. Розроблене програмне забезпечення також було інтегровано в автоматизовані процеси CI/CD, що дало змогу зробити процес сканування:

- автоматизованим - із виконанням перевірки на кожному коміті або Pull Request;
- повторюваним - незалежним від середовища розробника;
- прозорим - із можливістю зберігати результати у вигляді артефактів і надавати до них спільний доступ;
- масштабованим - із можливістю розширення набору перевірок та додавання нових сканерів.

Таким чином, у межах цього розділу було розроблено, реалізовано та протестовано програмне забезпечення, що включає власні інструменти для автоматизованого аналізу безпеки програмного забезпечення. Отримані результати підтверджують ефективність запропонованих підходів у підвищенні рівня захищеності коду, якості розроблення та впровадженні принципів DevSecOps у практичні процеси створення програмних продуктів

ВИСНОВОК

У кваліфікаційній роботі проведено комплексне дослідження сучасних підходів до управління вразливостями програмного забезпечення, описано та змодельовано процеси аналізу безпеки, розроблено та реалізовано власні інструменти для автоматизованого аналізу вихідного коду, а також показано їх практичне застосування у процесах розробки та CI/CD.

У першому розділі досліджено теоретичні основи управління вразливостями в контексті життєвого циклу програмного забезпечення (SDLC) та його безпечної модифікації (SSDLC). Описано та систематизовано ключові фази безпечного життєвого циклу, принципи побудови захищених систем і проаналізовано сучасні методи виявлення вразливостей у коді, залежностях та середовищі виконання. На основі аналізу наукових джерел і практик DevSecOps визначено та узагальнено актуальні тенденції у сфері автоматизації безпеки, що слугували теоретичною основою для створення власного рішення.

У другому розділі сформовано та обґрунтовано вимоги до програмного забезпечення, визначено та описано його мету й характеристики. Досліджено можливі технології, проведено обґрунтування вибору мови програмування Python та допоміжних бібліотек і інструментів, що забезпечують ефективну взаємодію з Docker та зовнішніми базами вразливостей. Сформульовано та деталізовано технічне завдання, спроєктовано архітектуру програмного продукту, змодельовано структуру класів та описано логіку взаємодії між основними модулями. У результаті створено та концептуально обґрунтовано основу для реалізації трьох незалежних, але взаємопов’язаних сканерів - SAST, SCA та Docker Image, що забезпечують комплексний аналіз безпеки.

У третьому розділі детально описано процес розробки програмного забезпечення, структуру та функції його модулів і продемонстровано практичні сценарії застосування. Реалізовані сканери було протестовано на прикладах вихідного коду, файлів залежностей та контейнерних образів. Показано можливість інтеграції інструментів як у локальному середовищі, так і в процесах CI/CD.

Розроблені рішення підтримують автоматизоване сканування при кожному коміті чи Pull Request, формують уніфіковані звіти у форматі JSON і забезпечують сумісність із типовими DevSecOps-пайплайнами. Отримані результати підтвердили ефективність та коректність роботи реалізованих інструментів у виявленні поширених уразливостей та їх відповідність вимогам сучасних підходів до безпеки програмного забезпечення. Запропоноване рішення підвищує рівень захищеності на ранніх етапах життєвого циклу розробки, сприяє впровадженню принципів
«security by design» та створенню більш надійного і стійкого до атак програмного продукту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. The Open Web Application Security Project. URL: https://owasp.org/www-community/vulnerabilities/ (дата звернення: 10.06.2025) [1]
2. Jonathan Singer “Understanding the Differences Between NVD and CVE”. URL: https://checkmarx.com/learn/appsec/understanding-the-differences-between-nvd-and-cve/ (дата звернення: 10.06.2025) [2]
3. “Software Development Life Cycle (SDLC)”. URL: https://www.geeksforgeeks.org/software-engineering/software-development-life-cycle-sdlc/ (дата звернення: 11.06.2025) [3]
4. “What is SDLC (Software Development Lifecycle)?” URL: https://aws.amazon.com/what-is/sdlc/ (дата звернення: 11.06.2025) [4]
5. “What is Secure Software Development Life Cycle (SSDLC)?” URL: https://www.geeksforgeeks.org/ethical-hacking/what-is-secure-software-development-life-cycle-ssdlc/ (дата звернення: 11.06.2025) [5]
6. “Secure Software Development Lifecycle (SSDLC)”. URL: https://snyk.io/articles/secure-sdlc/ (дата звернення: 11.06.2025) [6]
7. “What is a Software Vulnerability?” URL: https://jfrog.com/devops-tools/article/software-vulnerability/ (дата звернення: 12.06.2025) [7]
8. “How to Find Software Vulnerabilities?” URL: https://www.nexusgroup.com/how-to-find-software-vulnerabilities/ (дата звернення: 12.06.2025) [8]
9. “Vulnerability Assessment Types & Methodologies Explained”. URL: https://www.securityium.com/vulnerability-assessment-types-methodologies-explained/ (дата звернення: 12.06.2025) [9]
10. “Integrate external security scanners into your DevSecOps workflow”. URL: https://about.gitlab.com/blog/integrate-external-security-scanners-into-your-devsecops-workflow/ (дата звернення: 14.09.2025) [10]

11. “CI/CD Security Scanning: Types & Best Practices”. URL: https://www.sentinelone.com/cybersecurity-101/cloud-security/ci-cd-security-scanning/ (дата звернення: 14.09.2025) [11]
12. “Software supply chain security: How to audit a security bill of material”. URL: https://mattermost.com/blog/how-to-audit-a-security-bill-of-material-sbom/ (дата звернення: 14.09.2025) [12]
13. “Google OSV-Scanner: Open-Source Vulnerability Scanner”. URL: https://medium.com/nerd-for-tech/google-osv-scanner-open-source-vulnerability-scanner-35dee7b00509 (дата звернення: 14.09.2025) [13]
14. “Prisma Container Scanning: Comprehensive Vulnerability Management”. URL: https://seqops.io/prisma-container-scanning-comprehensive-vulnerability-management (дата звернення: 14.09.2025) [14]
15. “Rule Over Your Dependencies and Scan at Your Own Open Source Risk”. URL: https://www.sonatype.com/blog/rule-over-your-dependencies-and-scan-at-your-own-open-source-risk (дата звернення: 14.09.2025) [15]
16. “Using osv-scanner – Software Composition Analysis from Google”. URL: https://devsec-blog.com/2024/03/using-osv-scanner-software-composition-analysis-from-google/ (дата звернення: 14.09.2025) [16]
17. “Container Security Scanning: Vulnerabilities, Risks and Tooling”. URL: https://blog.gitguardian.com/container-security-scanning-vulnerabilities-risks-and-tooling/ (дата звернення: 14.09.2025) [17]
18. “SAST vs DAST: What they are and when to use them”. URL: https://circleci.com/blog/sast-vs-dast-when-to-use-them/ (дата звернення: 14.09.2025) [18]
19. “GitHub Actions documentation”. URL: https://docs.github.com/en/actions (дата звернення: 21.09.2025) [19]
20. “Getting Started with SAST: Detecting Vulnerabilities Early with GitHub Actions”. URL: https://medium.com/@yoshiyuki.watanabe/getting-started-with-sast-detecting-vulnerabilities-early-with-github-actions-57ecd54e8ccb (дата звернення: 21.09.2025) [20]

21. “Understanding Security Threats in Open-Source Software CI/CD Pipelines”. URL: https://arxiv.org/abs/2401.17606 (дата звернення: 21.09.2025) [21]
22. “Automated Security Testing in CI/CD Pipelines Using GitHub Actions”. URL: https://medium.com/edts/automated-security-testing-in-ci-cd-pipelines-using-github-actions-7e974804a92c (дата звернення: 21.09.2025) [22]
23. Алексіна, Л. Т., & Бондарчук, А. П. (2024). Оптимізація гіперпараметрів для машинного навчання. Зв’язок, (2), 18-22.
24. Bondarchuk, A., Dibrivniy, O., Grebenyk, V., & Onyshchenko, V. (2021, October). Motion Vector Search Algorithm for Motion Compensation in Video Encoding. In 2021 IEEE 8th International Conference on Problems of Infocommunications, Science and Technology (PIC S&T) (pp. 345-348). IEEE.
25. Сторчак, К. П., Тушич, А. М., & Бондарчук, А. П. (2018). Кластерний аналіз даних із використанням штучних нейронних мереж. Зв'язок, (6), 36-38.
26. Shantyr, A., Zinchenko, O., Storchak, K., Bondarchuk, A., & Pepa, Y. (2025). Prediction of quality software quality indicators with applied modifications of integrated gradiates methods. Informatyka, Automatyka, Pomiary w Gospodarce i Ochronie Środowiska, 15(2), 139-146.
27. Вембер, В. П., Машкіна, І. В., Носенко, Т. І., & Яскевич, В. О. (2025). Можливості та виклики використання штучного інтелекту у навчанні фахових дисциплін студентів спеціальностей «Комп’ютерні науки» та «Інженерія програмного забезпечення». Open educational e-environment of modern University, (19), 1-16.
28. Тушич, А. М., Сторчак, К. П., & Бондарчук, А. П. (2019). Вимоги до інтелектуальних систем аналізу даних та їх класифікацій. Телекомунікаційні та інформаційні технології, (1), 31-36.
29. Бондарчук, А., Жебка, В., Корецька, В., & Шилкіна, А. (2024). Порівняльна характеристика web-орієнтованих інструментів автоматизації освітнього процесу в умовах цифрової трансформації. Публічно-управлінські та цифрові практики, (1), 13-21.
30. Бондарчук, А. П., Корнага, Я. І., Базалій, М. Ю., Сергієнко, П. А., & Ільїн, О. Ю. (2020). Метод захисту програмного коду від аналізу засобами

обфускації. Телекомунікаційні та інформаційні технології, (4), 140-148.

31. Співак, С., Бондарчук, А., & Черевик, О. (2025). AI-система для професійної орієнтації у сфері 3D-графіки. Електронне фахове наукове видання «Кібербезпека: освіта, наука, техніка», 3(31), 698-709.
32. Співак, С. М., Білоус, В. В., Горбатовський, Д. В., & Бондарчук, А. П. (2025). Adapting education to the 3D graphics market using AI. Телекомунікаційні та інформаційні технології, 89(4), 215-221.
33. Бондарчук, А. П., & Глушак, О. М. (2025). Предиктивне управління оновленнями програмного забезпечення в інтернеті речей. Зв'язок, (5), 13-17.
34. Бушма, Олександр Володимирович та Турукало, Андрій Валерійович (2022) Оцінка параметрів програмної реалізації шкального відображення даних Cybersecurity: Education, Science, Technique, 4 (16). с. 142-158. ISSN 2663-4023 https://doi.org/10.28925/2663-4023
35. Бушма, Олександр Володимирович та Турукало, Андрій Валерійович (2021) Багатоелементні шкальні індикаторні пристрої у вбудованих системах Кібербезпека: освіта, наука, техніка, 3 (11). с. 43-60. ISSN 2663-4023 https://doi.org/10.28925/2663-4023
36. Бушма, Олександр Володимирович та Абрамов, Вадим Олексійович (2022) Підвищення достовірності визначення концентрації газів в середовищі моніторингу In: III Міжнародна науково-практична конференція: "Інформаційні технології та цифрова економіка" III International scientific and practical conference: "Information technologies and digital economy", 19-20 april 2022, Kyiv. https://elibrary.kubg.edu.ua/id/eprint/41648/
37. Білоус, Владислав та Бодненко, Дмитро та Локазюк, Олександра та Складаний, Павло та Абрамов, Вадим (2025) Програмне забезпечення для кібердоказів як інструмент цифрової криміналістики у розслідуванні кіберзлочинів. Забезпечення кібербезпеки в інформаційно-телекомунікаційних системах. 2025 (3991). С. 26-37. ISSN 1613-0073 https://ceur-ws.org/Vol-3991/
38. Білоус, Владислав Володимирович та Бодненко, Дмитро Миколайович та Хохлов, Олексій Костянтинович та Локазюк, Олександра Вікторівна та Стаднік, Ірина Петрівна (2024) Open Source Intelligence for War Crime

Documentation Workshop Cybersecurity Providing in Information and Telecommunication Systems (CPITS 2024) (3654). с. 368-375. ISSN 1613-0073 : https://ceur-ws.org/Vol-3654/short3.pdf

39. Андрій Петрович Бондарчук, Ірина Юріївна Мельник, Євгеній Іванович Суханевич, Вадим Олексійович Абрамов Підвищення ефективності систем відеоспостереження за рахунок гібридного методу відбору ключових кадрів і інтерпретації рішень Телекомунікаційні та інформаційні технології 2025 , №3 с101-105 https://doi.org/10.31673/2412-4338.2025.038710
40. Abramov, V., Astafieva, M., Boiko, M., Bodnenko, D., Bushma, A., Vember, V., Hlushak, O., Zhyltsov, O., Ilich, L., Kobets, N., Kovaliuk, T., Kuchakovska, H., Lytvyn, O., Lytvyn, P., Mashkina, I., Morze, N., Nosenko, T., Proshkin, V., Radchenko, S., … Yaskevych, V. (2021). Theoretical and practical aspects of the use of mathematical methods and information technology in education and science. https://doi.org/10.28925/9720213284km.
41. Бушма, Олександр Володимирович та Машкіна, Ірина Вікторівна та Носенко, Тетяна Іванівна та Яскевич, Владислав Олександрович (2024) Кваліфікаційна робота магістра: Навчально-методичний посібник для спеціальності «Комп’ютерні науки» Київський столичний університет імені Бориса Грінченка, Україна.https://elibrary.kubg.edu.ua/id/eprint/50205/