Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ДОПУСТИТИ ДО ЗАХИСТУ:
В.о. завідувача кафедри
кібербезпеки та захисту
інформації
_____ Іван ПАРХОМЕНКО
«___» червня 2025 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи

галузь знань 12 Інформаційні технології
(шифр і назва галузі знань)
спеціальність 125 Кібербезпека
(код і назва спеціальності)
освітній ступень бакалавр
освітня програма Кібербезпека
(назва освітньо-професійної програми)
на тему: «Механізм оцінювання безпеки контейнеризованих застосунків на базі Docker»

Виконавець: студент IV курсу, групи КБ-43

_____ **Назар НАЗАРИШИН**
(підпис) (ім'я, прізвище)

	Підпис	**Ім'я ПРІЗВИЩЕ**
Керівник		Сергій ДАКОВ
Нормоконтроль		Олександр ТОРОШАНКО

Київ 2025

Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ЗАТВЕРДЖЕНО:
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
_____ Іван ПАРХОМЕНКО
«29» листопада 2024 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності 125 Кібербезпека
(код і назва спеціальності)

освітньої програми Кібербезпека
(назва освітньо-професійної програми)

Студенту **КБ-43** **Назаришину Назару Віталійовичу**
(група) (прізвище ім'я по батькові)

Тема кваліфікаційної роботи Механізм оцінювання безпеки контейнеризованих застосунків на базі Docker

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Тема кваліфікаційної роботи затверджена на засіданні кафедри кібербезпеки та захисту інформації протокол №6 від 28.11.2024 р.

2. ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Операційна система хоста: Ubuntu 20.04 LTS, версія Docker Engine: 20.10, тестовий контейнерний образ: flask-app:initial (на базі Python 3.10), інструменти для сканування вразливостей образів: Trivy, Dockle, утиліта для аудиту конфігурації Docker-хоста: Docker Bench for Security, використання стандартів та рекомендацій: CIS Docker Benchmark, NIST SP 800-190, сценарій створення контейнера: Dockerfile із встановленням Flask, залежностей (pip), HEALTHCHECK

3. ЗМІСТ РОЗРАХУНКОВО-ПОЯСНЮВАЛЬНОЇ ЗАПИСКИ

Теоретичні основи контейнеризації та безпечної архітектури Docker
Модель загроз для Docker-середовища
Методи захисту Docker-контейнерів
Опис інструментів для аудиту: Docker Bench for Security, Trivy, Dockle
Практичний аудит безпеки контейнерного образу flask-app:initial
Аналіз результатів сканування та реалізовані поліпшення
Рекомендації щодо автоматизації перевірки безпеки у CI/CD (DevSecOps)
Перспективи розвитку безпеки контейнеризованих додатків (Kubernetes, rootless-контейнери)

4. ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Практична цінність виконаного дослідження полягає в розробці та апробації комплексної системи оцінювання й посилення безпеки контейнеризованих веб-додатків на базі Docker. Запропоновані підходи включають одночасний аудит конфігурації Docker-хоста (з використанням Docker Bench for Security) і статичний аналіз образів (за допомогою Trivy та Dockle), що дозволяє оперативно виявляти вразливості на різних рівнях: від налаштувань оточення до залежностей у самому додатку. На практиці це означає, що адміністратори та DevOps-інженери отримують чіткий алгоритм роботи - від раннього сканування під час CI/CD до моніторингу продуктивного середовища - і можуть інтегрувати автоматизовані скрипти перевірки в існуючі конвеєри розгортання. Результати дослідження (оновлені Dockerfile, налаштування user-namespaces, обов’язкове застосування HEALTHCHECK, відмова від опції –privileged, впровадження контролю ресурсів) продемонстрували значне зменшення кількості критичних і високоризикових уразливостей у тестових образах. Рекомендації щодо обмеження доступу до Docker-сокета, використання підписаних образів через Docker Content Trust та регулярного очищення image-cache можуть бути впроваджені в будь-які корпоративні чи державні проєкти, де використовуються контейнерні сервіси. Таким чином, результати роботи забезпечують підвищений рівень кібербезпеки веб-додатків, що працюють у Docker-середовищі, і можуть бути адаптовані для фінансових, медичних, освітніх та інших інформаційних систем із високими вимогами до захисту даних.

5. ДАТА ВИДАЧІ ЗАВДАННЯ

Дата видачі завдання: 29 листопада 2024 року

Завдання видав _____ Сергій ДАКОВ
(підпис) (ім'я, прізвище)

Завдання прийняв до виконання _____ Назар НАЗАРИШИН
(підпис) (ім'я, прізвище)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Найменування етапів робіт	Строки виконання робіт (початок-кінець)	Відмітка про виконання
1	Уточнення постановки задачі	29.11.2024 – 22.01.2025	*виконано*
2	Аналіз літератури	29.01.2025 – 11.02.2025	*виконано*
3	Обґрунтування вибору рішення	12.02.2025 – 15.02.2025	*виконано*
4	Виклад теоретичних основ Docker-ізоляції	16.02.2025 – 04.03.2025	*виконано*
5	Налаштування тестового середовища	05.03.2025 – 21.03.2025	*виконано*
6	Розробка простого Flask-застосунку та складання початкового Dockerfile	22.03.2025 – 08.04.2025	*виконано*
7	Первинний аудит безпеки з Trivy і Dockle, аналіз результатів	09.04.2025 – 10.05.2025	*виконано*
8	Інтеграція сканування у CI/CD (GitHub Actions workflow для Trivy і Dockle)	11.05.2025 – 20.05.2025	*виконано*
10	Підготовка до захисту кваліфікаційної роботи	28.05.2025 – 13.06.2025	*виконано*

Завдання видав _____ Сергій ДАКОВ
(підпис) (ім'я, прізвище)

Завдання прийняв до виконання _____ Назар НАЗАРИШИН
(підпис) (ім'я, прізвище)

Термін подання кваліфікаційної роботи до ЕК 13 червня 2025 року

РЕФЕРАТ

Пояснювальна записка дипломної роботи складається зі вступу, трьох розділів, загальних висновків, списку використаних джерел та додатків. Основний текст займає 67 сторінок, включає зміст, вступ, три розділи, висновки та список джерел. Окрім того, робота містить один додаток загальною кількістю 4 сторінок. У пояснювальній записці наведено 3 рисунки та 2 таблиці.

Мета роботи полягає в розробці та впровадженні засобів і механізмів оцінювання і підвищення безпеки контейнеризованих вебдодатків на базі Docker, що забезпечує захист конфіденційної інформації за рахунок аудиту налаштувань середовища, сканування образів на вразливості та застосування найкращих практик безпеки.

Для досягнення зазначеної мети були поставлені такі завдання:

- проаналізувати архітектуру Docker і механізми ізоляції контейнерів;
- дослідити найбільш розповсюджені загрози та вразливості Docker-середовища;
- вибрати та обґрунтувати інструменти для аудиту та сканування та провести їх практичне застосування;
- виявити конкретні вразливості у тестовому контейнерному образі і конфігурації Docker-хоста, проаналізувати результати сканування;
- розробити рекомендації та реалізувати покращення у Dockerfile і конфігурації середовища;
- повторно оцінити стан безпеки після впровадження змін і порівняти отримані результати.

Об’єктом дослідження є процеси забезпечення безпеки контейнеризованих веб-додатків у Docker-середовищі.

Предметом дослідження є сукупність технічних і організаційних заходів (інструменти аудиту, сканери вразливостей, налаштування Docker-демона та

контейнерів), що реалізують методи захисту і відповідають вимогам CIS Docker Benchmark та NIST SP 800-190.

Практична цінність результатів полягає у створенні комплексного підходу до захисту контейнеризованих застосунків. Розроблені рекомендації (автоматизоване сканування образів, аудит конфігурації Docker-хоста, застосування профілів безпеки, оптимізація Dockerfile) можуть бути інтегровані в реальні CI/CD-процеси та використовуватися DevOps-інженерами для підвищення рівня кібербезпеки в інформаційних системах різного масштабу - як приватних, так і державних, де застосовуються контейнерні сервіси для обробки конфіденційних даних.

Ключові слова: Docker, контейнеризація, безпека, вразливості, аудит, Trivy, Dockle, Docker Bench for Security, CIS Docker Benchmark, NIST SP 800-190.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕННЬ

CLI	–	Інтерфейс командного рядка
CIS	–	Центр безпеки в Інтернеті
CRIU	–	Контрольна точка/Відновлення в просторі користувача
CVE	–	Поширені вразливості та ризики
CVSS	–	Загальна система оцінювання вразливостей
ELK	–	Elasticsearch - Logstash - Kibana
K8s	–	Kubernetes
IDS	–	Intrusion Detection System, система виявлення вторгнень
LTS	–	Long-Term Support
OOM	–	Недостатньо пам'яті
SP	–	Спеціальна публікація

ВСТУП

У сучасних умовах інформаційних технологій контейнеризація стала стандартом розгортання застосунків у промислових середовищах. Платформа Docker, як провідний інструмент контейнеризації, здійснила революцію у способі розробки, доставки та масштабування програмного забезпечення . Згідно з опитуваннями Cloud Native Computing Foundation, понад 90% організацій уже використовують або тестують контейнерні технології у своїй діяльності . Це підтверджує актуальність теми безпеки контейнеризованих застосунків: із повсюдним впровадженням контейнерів питання їх захисту стають критично важливими. Контейнери хоча й ізолюють застосунки та спрощують перенесення між середовищами, водночас створюють нові вектори атак і унікальні проблеми безпеки . Забезпечення надійної безпеки Docker-середовища є першочерговим завданням при переході до хмарних та мікросервісних архітектур.

Метою даної роботи є впровадження заходів для підвищення його захищеності.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- проаналізувати архітектуру Docker і механізми ізоляції контейнерів;
- дослідити найбільш розповсюджені загрози та вразливості Docker-середовища;
- вибрати та обґрунтувати інструменти для аудиту та сканування та провести їх практичне застосування;
- виявити конкретні вразливості у тестовому контейнерному образі і конфігурації Docker-хоста, проаналізувати результати сканування;
- розробити рекомендації та реалізувати покращення у Dockerfile і конфігурації середовища;
- повторно оцінити стан безпеки після впровадження змін і порівняти отримані результати.

Об’єкт дослідження – процеси забезпечення безпеки контейнеризованих застосунків на базі Docker.

Предмет дослідження – методи оцінки вразливостей Docker-контейнерів та підходи до їхнього усунення відповідно до галузевих стандартів безпеки.

Методи дослідження включають аналіз науково-технічної літератури і стандартів з безпеки контейнерів, експериментальне сканування Docker-контейнера інструментами статичного та динамічного аналізу, а також синтез рекомендацій на основі виявлених недоліків. Практична значимість роботи полягає у виявленні реальних проблем безпеки Docker-застосунку та впровадженні конкретних заходів, що підвищують захищеність контейнерного середовища. Результати роботи можуть бути використані адміністраторами та розробниками для покращення конфігурації Docker-контейнерів у промислових системах.

Структурно робота містить вступ, три розділи, висновки до кожного розділу, загальні висновки, список використаних джерел та додатки. У розділі 1 розглядаються теоретичні основи: архітектура Docker, можливі загрози й атаки, методи захисту та аналіз стандартів безпек. Розділ 2 присвячено практичній оцінці безпеки: розгортанню тестового застосунку та аудитору його контейнера за допомогою сканерів безпеки, з аналізом виявлених проблем. Розділ 3 описує реалізацію заходів з підвищення безпеки Docker-контейнера, повторне сканування та оцінку ефективності покращень. У висновках підбито підсумки виконаної роботи, сформульовано основні результати та рекомендації. Додатки містять вихідні коди, конфігураційні файли Docker та звіти сканування, що підтверджують отримані результати

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ БЕЗПЕКИ КОНТЕЙНЕРІВ DOCKER

1.1 Сучасні загрози та вразливості Docker

Активне використання Docker породжує ряд характерних загроз. По-перше, це вразливості в базових образах та залежностях. Багато контейнерів будуються на стандартних дистрибутивах Linux (Ubuntu, Debian тощо), які можуть містити застарілі пакети з відомими CVE. Дослідження виявило, що більшість образів у репозиторіях мають серйозні вразливості. Наприклад, якщо базовий образ містить небезпечну версію OpenSSL або systemd, зловмисник може скористатися цими проломами: уразливість в OpenSSL здатна дозволити атаки типу «людина посередині» або спричинити збій шифрованого з'єднання, а баги в systemd/glibc за певних умов допомагають підготувати ґрунт для ескалації привілеїв на хості . По-друге, небезпека неправильної конфігурації Docker. Залишення Docker Daemon API відкритим або запуск контейнера з надмірними привілеями становить серйозний ризик. Реальний інцидент: через відкритий Docker API у 2021 році атакувальники встановили майнер у контейнері. Інший приклад – відсутність обмежень ресурсів: якщо не лімітувати пам'ять та CPU, зловмисник (чи помилковий процес) може вичерпати ресурси хоста, спричинивши відмову в обслуговуванні (DoS) для інших сервісів.

Ще один вектор – атаки на ланцюжок постачання (supply chain). Docker-образи часто повторно використовуються із загальнодоступних репозиторіїв. Існує загроза завантажити компрометований образ, у який зловмисники впровадили шкідливий код. Відомі випадки фішингу образів: під виглядом офіційних або популярних образів викладаються шкідливі копії, що містять бекдори чи майнери. Розробники можуть несвідомо запускати такі образи (атака типу Spoofing за STRIDE), що призводить до компрометації системи.

Привілейований запуск контейнерів – окрема загроза. Якщо процес у контейнері виконується від імені root (а за замовчуванням Docker саме так і робить), при виникненні вразливості атакувальник може отримати повний контроль над контейнером, а далі спробувати підвищити привілеї до рівня хостової системи. Небезпечними є вразливості контейнерного рантайму та ядра ОС: наприклад, помилки в Docker/runc або самому ядрі Linux можуть дозволити процесу “втекти” з ізольованого середовища контейнера на хост (так званий container escape). Відомі інциденти, коли експлойти у runc давали можливість виконати код на хості з контейнера – подібні CVE регулярно з’являються, і їх необхідно відслідковувати та оперативно встановлювати патчі. Таким чином, вразливий застосунок всередині контейнера може слугувати плацдармом для повного захоплення системи. Насамкінець, не можна забувати про витоки конфіденційних даних: якщо секрети (паролі, ключі API) зберігаються у відкритому вигляді в Dockerfile або образі, вони можуть бути викрадені зловмисником (Information Disclosure за STRIDE).

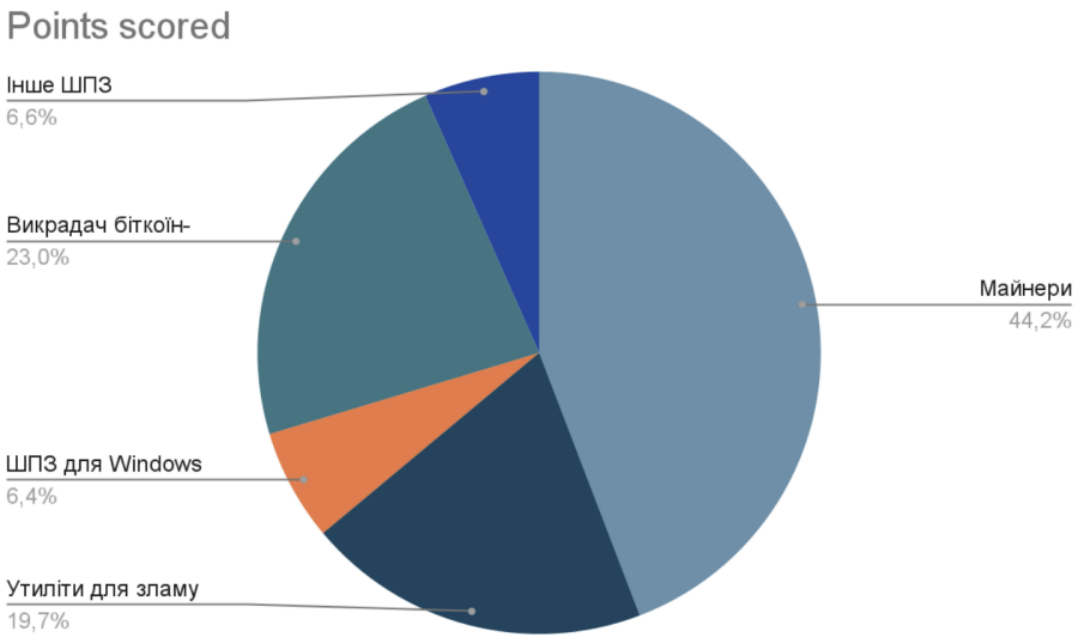


Рисунок 1.1 - Діаграма розподілення типів ШПЗ у образах Docker-контейнерів

1.2 Архітектура Docker та механізми ізоляції

Платформа Docker забезпечує ізоляцію середовища виконання застосунків шляхом використання контейнерів – легковагих віртуалізованих середовищ, які працюють поверх ядра операційної системи хоста. На відміну від традиційних віртуальних машин, контейнер не містить власного ядра ОС: усі запущені контейнери спільно використовують ядро хоста, що значно зменшує накладні витрати та підвищує ефективність використання ресурсів . Контейнер упакований усіма необхідними для роботи додатка компонентами – від виконуваного коду і бібліотек до системних інструментів і налаштувань середовища. Це гарантує портативність: один і той самий контейнеризований застосунок працюватиме ідентично в різних середовищах (на різних серверах чи ОС), якщо там встановлено Docker . Таким чином досягається принцип “write once, run anywhere” для розробників, що прискорює процеси розгортання від етапу розробки до етапу експлуатації.

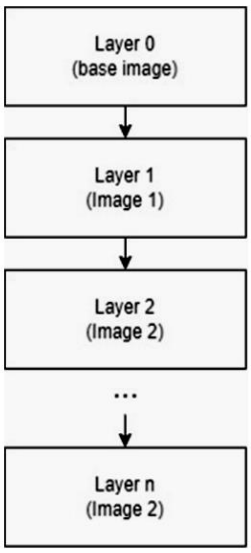


Рисунок 1.2 - Будова Docker контейнера

Основу Docker становить архітектура клієнт-сервер. Docker працює за моделлю, де є клієнт (командний рядок docker CLI або інші інтерфейси) та серверна частина – демон Docker (dockerd). Клієнт надсилає команди та запити до демона через REST API (використовується Unix-сокет або TCP-з'єднання), а демон виконує необхідні дії: будує образи, запускає чи зупиняє контейнери, керує мережами і сховищами тощо . Така архітектура забезпечує розподіл привілеїв: клієнт легковагий і не потребує прав суперкористувача для виконання команд – всю важку роботу виконує привілейований процес демона. Docker-демон запускається як системна служба на хості та постійно слухає запити від клієнтів. При встановленні Docker Engine на сервер, окрім двох зазначених компонент (CLI та демона), також встановлюється низькорівневий компонент – runtime для контейнерів .

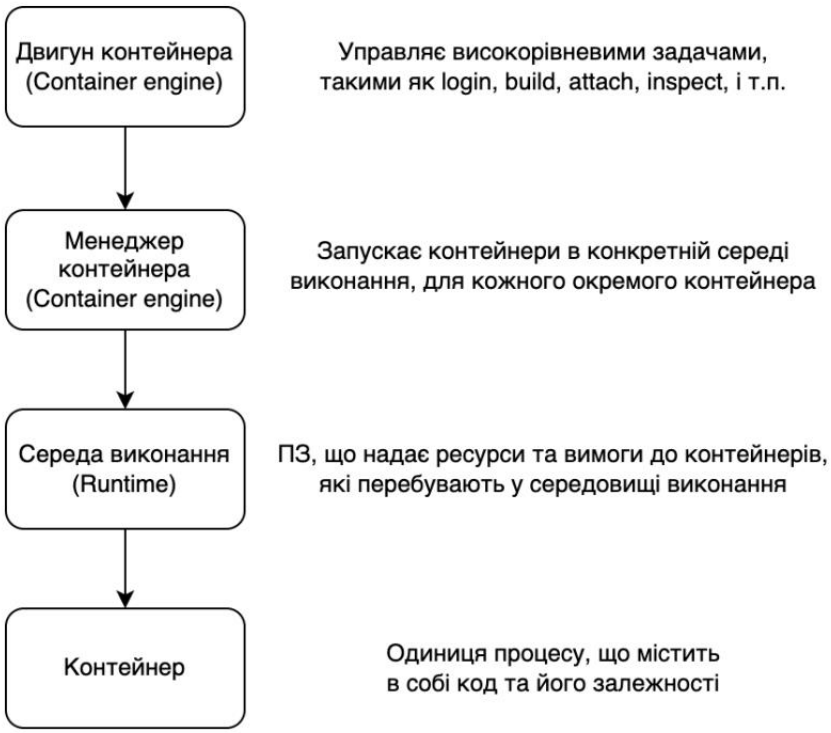


Рисунок 1.3 – Рівні середовища docker

Docker-демон не взаємодіє з ядром ОС напряму, а використовує проміжний шар – високорівневий контейнерний рантайм (типово це containerd разом із низькорівневим runc). Containerd відповідає за управління життєвим циклом контейнерів на хості: завантаження та зберігання образів, запуск і зупинку контейнерів, ізоляцію ресурсів тощо. А безпосередню ізоляцію процесів у контейнері реалізує runc, використовуючи вбудовані механізми ядра Linux – простори імен (namespaces) та групи контролю (cgroups). Простори імен ізолюють різні аспекти середовища кожного контейнера: його бачення файлової системи, мережі, ідентифікаторів процесів, користувачів тощо. Завдяки цьому процеси всередині контейнера “не знають” про існування ресурсів поза ним (інших контейнерів чи системного оточення). Cgroups обмежують споживання ресурсів (процесорного часу, пам’яті, диску) контейнером, запобігаючи ситуації, коли один контейнер монопольно захоплює ресурси хоста. Усе це працює прозоро для застосунку: контейнер виглядає для програми як ізольований сервер зі своєю ОС, хоча насправді спільно використовує ядро хоста.

Окрім namespaces і cgroups, Docker автоматично застосовує інші заходи безпеки. Контейнери стартують зі скоротеним набором привілеїв: Docker відсікає частину можливостей (Linux Capabilities) у порівнянні з повним root. Наприклад, за замовчуванням контейнер не може виконувати низькорівневі операції з мережею, змінювати час, завантажувати нові модулі ядра тощо – відповідні capabilities (CAP_NET_RAW, CAP_SYS_TIME, CAP_SYS_MODULE та ін.) відключено. Це зменшує потенціал шкоди, якщо контейнер буде скомпрометовано. За потреби програма в контейнері може вимагати додаткових привілеїв, але тоді їх треба явно додати (прапор --cap-add) – гарна практика полягає в тому, щоб дати контейнеру лише мінімально необхідні можливості (принцип найменших привілеїв).

Основними об’єктами Docker є образи та контейнери. Docker-образ – це шаблон (snapshots) файлової системи, що містить всі необхідні компоненти для запуску контейнера: базову ОС (мінімальний набір пакетів), залежності,

бібліотеки, виконувані файли та власне код застосунку . Образи є *статичними* та *тільки-читання*: вони збираються з інструкцій Dockerfile (кожен шар образу – результат виконання однієї інструкції) і призначені для багаторазового використання. Docker-контейнер – це запущений екземпляр образу, тобто *процес* (або група процесів), що працює в ізольованому середовищі на основі цього образу. При старті контейнера Docker створює для нього окремий ізольований простір процесів, мережевий інтерфейс, підмонтовує образ як файлову систему (додаючи зверху writable-шар для змін) і запускає в цьому середовищі заданий процес (наприклад, веб-сервер або застосунок) . За замовчуванням контейнер має обмежений доступ лише до тих ресурсів, які явно визначені його образом або надані під час запуску (наприклад, проброс портів чи монтування томів). Всі контейнери на хості ізольовані один від одного і від системи (якщо не надавати спеціальних привілеїв), що є ключовою особливістю моделі безпеки Docker.

Для посилення ізоляції Docker дозволяє використовувати механізми безпеки хоста: наприклад, AppArmor або SELinux. На Linux-хості можна ввімкнути профілі AppArmor/SELinux, що накладаються на контейнер і контролюють, до яких дій він має доступ. За замовчуванням Docker на Ubuntu застосовує профіль AppArmor docker-default, який блокує ряд потенційно небезпечних дій. Можна створити власний профіль AppArmor спеціально під ваш застосунок у контейнері – наприклад, заборонити йому доступ до певних каталогів або системних викликів. Це стане додатковим бар’єром: навіть якщо зловмисник отримає доступ всередину контейнера, AppArmor/SELinux обмежить його дії (мова про mandantory access control). Зокрема, AppArmor може заборонити контейнеру доступ до критичних файлів хоста, мінімізуючи ризик ескалації привілеїв. Аналогічно, Docker підтримує використання Seccomp-профілю – це фільтр системних викликів. Стандартний профіль seccomp (увімкнений Docker’ом за замовчуванням) блокує ~44 небезпечних системних виклики (наприклад, mount, ptrace та ін.), що теж утруднює експлуатацію вразливостей ядра або захищає від побіжного впливу на хост.

Ізоляція контейнерів, утім, не є тотальною: всі контейнери на хості поділяють одне ядро ОС. На відміну від віртуальних машин, де кожна ВМ має власне ізольоване ядро, контейнер – це процес хостового ядра, але в «пісочниці». Через це експлойти в ядрі Linux чи в Docker-рантаймі можуть впливати на всю систему. Проте належна конфігурація (використання нових ядер з виправленнями, профілі seccomp, регулярне оновлення Docker) зменшує цей ризик.

Для зберігання та розповсюдження Docker-образів використовується реєстр образів (registry). Найбільш відомим публічним реєстром є Docker Hub, де розробники викладають образи, доступні широкому загалу . Розгортаючи контейнер, Docker спочатку шукає необхідний образ локально, а за відсутності – завантажує з реєстру (публічного або приватного, залежно від налаштувань). Концепція реєстрів дозволяє легко ділитися образами: розробник може зібрати образ і “push” – відправити його до репозиторію Docker Hub, після чого інші користувачі можуть “pull” – завантажити та запустити цей образ у себе. Такий підхід прискорює розробку (можна брати готові базові образи), але водночас створює ризики безпеки, пов’язані з довірою до сторонніх образів (це детально розглядається в наступному підрозділі).

Підсумовуючи, архітектура Docker складається з клієнта, демона (Docker Engine) та набору допоміжних компонентів (containerd, runc), що разом реалізують контейнеризацію через механізми ізоляції ядра Linux. Контейнери забезпечують переваги ізоляції, портативності та ефективності, проте така спільна модель (поділ ядра та ресурсів) диктує особливі вимоги до безпеки. Надалі розглянемо, які загрози притаманні контейнеризованим середовищам і як їх можна нейтралізувати.

1.3 Модель загроз для контейнеризованих застосунків

При впровадженні контейнеризації необхідно враховувати специфічний ландшафт загроз, що формується внаслідок особливостей Docker. Насамперед слід звернути увагу на надійність самого хоста, адже всі контейнери працюють під керуванням єдиного Docker-демона. Якщо операційна система хоста або Docker-демон, запущений з привілеями, будуть скомпрометовані, атакувальник здобуде повний контроль над усіма контейнерами. Наприклад, неналежно захищений Docker API або надання користувачу членства в групі Docker (що, по суті, дорівнює повним привілеям) може призвести до захоплення обчислювальної інфраструктури.

Дуже часто джерелом вразливостей стає неправильна конфігурація контейнерів. Якщо контейнер запущено з надмірними привілеями (опція –privileged дозволяє йому втручатися в роботу пристроїв хоста), зняття ізоляції між хостом і контейнером (наприклад, спільний мережевий простір або змонтований сокет Docker /var/run/docker.sock усередині контейнера), відкриття непотрібних портів або невиправданий мапінг портів хоста, а також використання спільних томів із критичними файлами хоста — усе це створює сприятливі умови для атак. У прагненні забезпечити функціональність іноді нехтують обмеженнями на ресурси: якщо контейнер працює без лімітів пам’яті чи CPU, він може спричинити DoS-атаку на хост, витративши всі доступні ресурси. Крім того, привілейовані контейнери мають можливість виконувати дії від імені хоста, що різко підвищує ризик прориву ізоляції й отримання контролю над іншими контейнерами.

Не менш небезпечними є вразливості застосунків, які працюють всередині контейнера. Сама контейнеризація не гарантує безпеку коду: якщо веб-додаток або інший сервіс містить вразливості на рівні коду (наприклад, SQL-ін’єкції, переповнення буфера чи XSS), атакувальник зможе експлуатувати їх навіть у межах контейнера. Хоча вплив у разі компрометації може залишатися

локалізованим, недооцінювати цю загрозу не варто: після успішного злому всередині контейнера зловмисник може шукати можливості для розширення атаки на інші контейнери чи безпосередньо на хост. Особливо критичними є вразливості контейнерного рантайму або ядра операційної системи: якщо існує помилка, що дозволяє процесу підвищити привілеї до рівня ядра або вийти з-під ізоляції контейнера (так званий container escape), зловмисник отримає плацдарм для повного захоплення системи. Прецеденти показують, як унаслідок багів у runc або драйверах файлових систем атакам вдавалося «втекти» з контейнера, тож постійний моніторинг CVE та своєчасне застосування патчів лишаються вкрай важливими.

Окрему групу загроз становлять атаки на програмний ланцюг постачання (supply chain attacks). Часто для побудови інфраструктури використовують сторонні образи або відкриті базові образи з Docker Hub, проте завантажені образи можуть містити шкідливий код або бекдори. Зловмисники навмисно розміщують у популярних репозиторіях образи з прихованим шкідливим програмним забезпеченням, а іноді навіть успішно отримують доступ до офіційних репозиторіїв. Крім того, явище typosquatting-образів, коли назва підробки свідомо майже не відрізняється від оригіналу (наприклад, nginx-official замість nginx), спонукає користувачів завантажувати «неправильні» контейнери. За результатами досліджень Prevasio й Aqua Security (2020), серед приблизно 30 мільйонів перевірених образів близько 20 % містили майнери криптовалют, бекдори, руткіти чи інше шкідливе ПЗ, яке активувалося після запуску контейнера. Унаслідок цього випадкове використання неперевірених образів може призвести до масштабної компрометації всієї інфраструктури: ризик полягає не лише у вразливостях базових образів (наприклад, застарілі пакети), а й у навмисно шкідливих компонентах.

Особливу увагу слід приділяти зберіганню секретів і конфіденційних даних. Якщо розробники необачно закладають паролі, API-ключі або приватні ключі безпосередньо в Dockerfile або образі, атакувальник досить легко зможе їх

отримати—особливо коли образ викладений у публічний реєстр. У такому разі витік облікових даних до бази даних чи інших сервісів може призвести до непоправних наслідків. Тому передача секретів у контейнер повинна здійснюватися через спеціальні механізми (Docker secrets, зовнішні сховища типу HashiCorp Vault, передавання змінних середовища безпосередньо з хоста тощо), аби мінімізувати ризик їх викриття.

Мережна взаємодія контейнерів за замовчуванням теж несе певну загрозу. Контейнери, під'єднані до однієї bridge-мережі, можуть вільно обмінюватися даними між собою та мати вихід у зовнішню мережу. Відсутність належної сегментації означає, що якщо один контейнер буде скомпрометований, нападник може розпочати атаки на решту контейнерів або внутрішні сервіси в тому самому мережевому сегменті (латеральне переміщення). Наприклад, якщо політики безпеки не забороняють сканування мережі Docker, зловмисник із одного вразливого контейнера може знаходити інші контейнери та шукати їхні недоліки. Або ж скомпрометований веб-додаток може використовувати відкритий вихід у зовнішню мережу для подальших атак на внутрішню інфраструктуру компанії. Через недостатній контроль міжконтейнерного і зовнішнього трафіку виявити подібну активність без спеціалізованих інструментів складно, тому належні механізми моніторингу та обмеження (прикладом можуть бути міжконтейнерні брандмауери або політики мережі Kubernetes) є необхідністю.

Ще одним важливим ризиком є порушення цілісності образів і неконтрольоване оновлення. Образ, який звантажується з віддаленого реєстру, може бути підмінений під час передачі, якщо не використовується захищене з'єднання (HTTPS/TLS) або перевірка цифрового підпису. У цьому випадку атакувальник у режимі «людина-посередник» може видати свій шкідливий образ за легітимний. Окрім того, явище «image sprawl»—накопичення застарілих і неоновлених образів на хості—збільшує ймовірність того, що десь лишаться контейнери з відомими вразливостями. Такі образи й контейнери стають легкою

здобиччю для експлойтів, тож важливо не лише перевіряти цілісність кожного завантаженого образу, але й своєчасно видаляти або оновлювати застарілі версії.

Зазначені загрози не є вичерпними, проте охоплюють ключові напрямки ризиків у Docker-середовищі. На практиці атаки часто комбінують кілька векторів: приміром, спочатку зловмисник експлуатує вразливість веб-додатку в контейнері, а потім, скориставшись відсутністю обмежень (наприклад, якщо контейнер запущено в привілейованому режимі або в нього змонтовано Docker-сокет), здійснює «втечу» на хост і продовжує компрометацію інших контейнерів. В іншому сценарії через помилку конфігурації Docker може бути відкритий REST API без аутентифікації, що дає змогу запускати довільні контейнери на сервері (наприклад, майнери криптовалют або бекдори). Подібні випадки підтверджують необхідність комплексного підходу до безпеки на всіх рівнях — від хоста й самих контейнерів до образів, мережі та безпосереднього коду застосунків.

Таким чином, модель загроз Docker-середовища включає ризики на рівні хоста, контейнерів, образів, мереж і застосунків. Надалі слід розглянути методи захисту та найкращі практики, що дозволяють мінімізувати кожен із перелічених векторів атак.

1.4 Методи захисту Docker-контейнерів

Для нейтралізації розглянутих загроз розроблено низку кращих практик та інструментів, що забезпечують багаторівневий захист Docker-середовищ. Ці методи охоплюють як етап створення образу (build-time), так і конфігурацію контейнерів під час запуску (run-time), а також захист оточення в цілому. Нижче подано опис основних підходів до безпеки контейнеризованих застосунків у вигляді суцільного тексту.

Насамперед необхідно дотримуватися принципу “мікрообразу”: включати до Docker-образу тільки ті пакети та залежності, які безпосередньо потрібні для

роботи застосунку. Це передбачає відмову від зайвих утиліт, таких як інструменти для налагодження (curl, wget тощо), текстові редактори та пакетні менеджери. Зменшення кількості компонентів у контейнері суттєво звужує площу атаки, адже зменшує потенційно вразливий код. Прикладом кращої практики є використання спеціалізованих малорозмірних базових образів—наприклад, Alpine Linux або Distroless—замість універсальних ubuntu:latest чи debian, які можуть містити тисячі непотрібних пакетів. Крім того, усі компоненти образу мають бути актуальними: базовий образ потрібно регулярно оновлювати до останніх версій з усіма патчами безпеки, а також слід моніторити версії залежностей самого застосунку й оновлювати їх одразу після виявлення вразливостей. Практика rebuild often, тобто регулярного перебудовування образу з останніми оновленнями, допомагає запобігти накопиченню відомих проблем. Перед використанням образи варто сканувати спеціальними інструментами на наявність відомих уразливостей (наприклад, перевіряти наявність пакетів із критичними CVE), щоб упевнитися, що контейнер не містить відвертих вразливостей.

Не менш важливо використовувати лише довірені базові образи. Рекомендовано завантажувати їх з офіційних джерел-Docker Official Images або репозиторіїв виробників операційних систем-або з приватних репозиторіїв із контролем доступу. Для внутрішніх додатків доцільно мати власний приватний реєстр образів, де перевірені образи зберігаються під контролем команди розробників. Крім того, бажано ввімкнути Docker Content Trust (на основі інфраструктури Notary), щоб Docker клієнт витягував лише підписані зображення. Це гарантує цілісність і автентичність образу, запобігаючи можливим атакам “людина посередині” під час завантаження.

Щодо секретів і конфіденційних даних, суворо заборонено зберігати паролі, ключі та інші секретні дані у відкритому вигляді в Dockerfile чи безпосередньо в образі. Замість цього слід застосовувати механізми, які дозволяють передавати секрети під час запуску контейнера, а не прибивати їх

“на жорстоко” (baked-in). Наприклад, у Docker Swarm і Kubernetes можна використовувати Docker Secrets або Kubernetes Secrets, а в простіших сценаріях - передавати секрети через змінні середовища чи файли, що монтуються тільки під час run-time і не потрапляють до образу. Так, доступ до бази даних може бути налаштований за допомогою змінної DATABASE_URL, що задається безпосередньо на хості. Додатково рекомендується шифрувати секрети в стані спокою, щоб навіть при несанкціонованому доступі до файлової системи контейнера їх було важко прочитати. Важливо, щоб різні контейнери мали окремі секрети, і один контейнер не міг читати секрети іншого.

Ще одним критичним аспектом є зниження привілеїв і застосування принципу найменших прав. За замовчуванням процеси всередині контейнера запускаються від імені root. Хоча цей root ізольований від root хосту, за певних умов (зокрема, якщо не використовується user namespace) він може загрожувати безпеці хоста. Тому рекомендується не запускати процеси під root, якщо це не вкрай необхідно. Натомість варто створити в Dockerfile спеціального непривілейованого користувача (за допомогою інструкцій RUN useradd … та USER), під яким працюватиме застосунок. Це обмежить можливості зловмисника навіть у разі успішного зламу контейнера. Додатково при запуску контейнера Docker дає змогу обмежувати привілеї через механізми capability dropping, seccomp та системи мандатного контролю доступу AppArmor або SELinux. Зокрема, за допомогою опції –cap-drop ALL можна зняти всі додаткові можливості POSIX і додати лише ті, які справді потрібні (наприклад, CAP_NET_RAW для доступу до RAW-сокетів). За замовчуванням Docker вже застосовує секкомп-профіль, що блокує небезпечні системні виклики, тому не варто відключати seccomp (не потрібно вказувати –security-opt seccomp=unconfined), а краще посилити його власним профілем, якщо застосунок це дозволяє. На системах із підтримкою AppArmor або SELinux доцільно створити власний профіль, що визначає, які файли контейнер може читати чи змінювати, і які дії можуть виконуватися. У випадку SELinux для цього

використовується опція –security-opt label:type:…, що значно підвищує ізоляцію контейнера.

Обмеження доступу до ресурсів хоста також є важливою практикою. Під час монтування директорій хоста у контейнер слід уникати підключення чутливих директорій із правами на запис, зокрема /etc, /var/run/docker.sock та інших критичних шляхів. Монтуючи /var/run/docker.sock із правами RW, ви фактично передаєте контейнеру повний контроль над Docker-демоном, що є поширеним вектором ескалації привілеїв. До того ж слід уникати запуску SSH-демона всередині контейнера: якщо потрібен доступ до оболонки, варто скористатися docker exec із відповідними обмеженнями. При запуску контейнера також доцільно вказувати ресурси, які йому дозволені: обмеження пам’яті та CPU (наприклад, -m 256m, –cpus=“1.0”), щоб контейнер не міг спожити всі ресурси хоста та порушити роботу інших процесів. Використання опції –read-only для файлової системи контейнера унеможливить запис всередині його FS, якщо цей контейнер не потребує запису; у такому разі запис можливий лише в спеціально змонтовані томи, що ускладнить зловмиснику модифікацію файлів чи встановлення додаткових інструментів.

Контроль мережевого трафіку теж має свою специфіку. За замовчуванням контейнери у стандартній bridge-мережі Docker можуть вільно з’єднуватися із зовнішнім світом. Рекомендується створювати призначені віртуальні мережі для різних груп контейнерів та ізолювати їх між собою. Якщо контейнер не потребує доступу до зовнішньої мережі, його слід запускати в мережі без виходу (internal network), щоб ніде не було випадкового відкриття портів. Для доступу ззовні варто відкривати тільки необхідні порти і за можливості забороняти привілейовані порти (<1024) на хості. Наприклад, якщо ваш додаток слухає на 80-му порту всередині контейнера, краще на хості змінити це на 8080, а не 80, що відповідає рекомендаціям CIS.

Ще однією додатковою мірою підвищення безпеки є ввімкнення режиму user namespace, за якого root всередині контейнера відображається як

непривілейований користувач на хості. Опція –userns-remap потребує налаштувань і не завжди сумісна з усіма сценаріями, проте в середовищах із високими вимогами безпеки її варто розглянути, адже вона не дозволяє навіть “контейнерному” root отримати справжні root-привілеї в системі.

Втім безпека - це не одноразова дія під час налаштування, а безперервний процес. Практика моніторингу й регулярного сканування контейнерів та хоста на відповідність політикам безпеки є критично важливою. Наприклад, утиліта Docker Bench for Security від Docker автоматизовано перевіряє налаштування Docker-хоста згідно з рекомендаціями CIS Benchmark та видає звіт про виявлені відхилення (наприклад, чи не запущено контейнери з опцією privileged, чи демон Docker не слухає на небезпечних портах тощо). Її корисно запускати періодично, а ще краще — інтегрувати в CI/CD, щоб будь-яке відхилення від політик фіксувалося одразу. Крім того, рекомендується налаштувати централізоване логування: увімкнути детальний лог Docker-демона, збирати логи контейнерів, а далі — використовувати системи моніторингу (Prometheus, ELK, Grafana) для виявлення аномалій у поведінці контейнерів. Наприклад, різке зростання використання CPU в одному з контейнерів може свідчити про майнінг-зловмисне ПЗ. Для контролю на рівні ОС є системи IDS, такі як Falco (від CNCF), яка в режимі реального часу відстежує системні виклики ядра й може помітити підозрілу активність у контейнерах (відкриття заборонених файлів, запуск нових оболонок всередині контейнера тощо).

Якщо ж для оркестрації контейнерів використовуються Docker Swarm або Kubernetes, необхідно дотримуватися ще низки специфічних заходів: у Swarm слід увімкнути шифрування overlay-мереж, обмежити доступ до менеджер-вузлів та використовувати механізми, що запобігають несанкціонованому доступу. У Kubernetes для сегментації трафіку застосовують Network Policies, а для контролю доступу до API оркестратора—RBAC (Role-Based Access Control). Хоча у даній роботі фокус зроблений на безпеці single-host Docker, варто

пам’ятати, що безпечна оркестрація створює додатковий рівень ізоляції та контролю в кластері.

Усі перелічені методи відображають основні принципи кібербезпеки: мінімізацію впливу (зменшення площі атаки), принцип найменших привілеїв, сегментацію мережі, регулярне оновлення компонентів та безперервний моніторинг. У випадку Docker вони конкретизуються у вигляді чітких рекомендацій: не запускати процеси під root, не використовувати опцію privileged, сканувати образи перед деплоєм, застосовувати рекомендації CIS Benchmark тощо. Більшість цих практик формалізовані у відомих галузевих стандартах, що буде розглянуто у наступному підрозділі.

1.5 Стандарти та найкращі практики безпеки контейнерних середовищ

На сьогоднішній день для захисту контейнерних середовищ існують авторитетні стандарти та рекомендації, розроблені провідними світовими організаціями у сфері кібербезпеки. Серед них варто особливо виділити CIS Docker Benchmark від Center for Internet Security та NIST SP 800-190 «Application Container Security Guide» від Національного інституту стандартів та технологій США. Обидва документи систематизують найкращі практики та слугують орієнтиром для формування політик безпеки контейнерів у багатьох компаніях. Далі розглянемо їхні ключові положення.

CIS Docker Benchmark являє собою детальний набір рекомендацій із безпечної конфігурації Docker, підготовлений експертами Center for Internet Security. Документ охоплює всі рівні контейнерної платформи: від налаштування хост-системи та демона Docker до побудови образів, запуску контейнерів і опцій оркестрації (зокрема Docker Swarm). Метою цього бенчмарку є чітко сформульовані правила перевірки, виконання яких дозволяє суттєво знизити ризики. В першу чергу автори наголошують на тому, що операційна система

хоста повинна бути надійно захищена: необхідно встановлювати останні оновлення безпеки, відключати невикористовувані служби та стежити за тим, щоб сам Docker-двигун оновлювався до актуальної версії. Для ізоляції даних контейнерів рекомендується виділити окремий розділ диску під директорію Docker (/var/lib/docker). Також доступ до Docker-сокета та демона слід обмежити лише довіреними адміністраторами та членами групи «docker», адже члени цієї групи фактично мають привілеї root на хості. Крім того, налаштування брандмауера на хості дозволяє контролювати мережевий трафік контейнерів та підсилити їхню ізоляцію.

Що стосується конфігурації самого демона Docker, у бенчмарку рекомендується запускати його від імені root, але з мінімально необхідними привілеями. Якщо Docker не використовується в мультиорендному середовищі, варто відключити TLS-доступ зовні; навпаки, при потребі віддаленого керування слід обов'язково ввімкнути TLS для Docker API, щоб сторонні не могли підключатися без відповідних сертифікатів. Ще одним пунктом є увімкнення режиму «–icc=false» для ізоляції мережевого трафіку між контейнерами за відсутності потреби у їхній взаємодії. Для запобігання безкінечному перезапуску скомпрометованого контейнера рекомендується використовувати політику рестарту «–restart on-failure» замість «always». Крім того, необхідно лімітувати кількість процесів у контейнері за допомогою PID cgroup, бо в базовій конфігурації Docker цього не робить. Обов'язковою є конфігурація збереження логів демона та контейнерів: слід налаштувати «–log-level» і вибрати відповідний драйвер логування.

Контроль на етапі побудови образів також займає важливе місце в рекомендаціях CIS. Кожен Dockerfile повинен містити інструкцію HEALTHCHECK для моніторингу стану контейнера: це дозволяє вчасно перезапустити «завислий» контейнер і може слугувати непрямим індикатором компрометації. Під час складання образів рекомендується уникати використання інструкції ADD, якщо достатньо можливостей COPY, оскільки ADD також вміє

завантажувати файли з URL, що може призвести до порушення цілісності. Якщо ж усе ж потрібні оновлення пакетів, не варто викликати apt-get upgrade безпосередньо: краще вказувати конкретні версії пакетів, щоб образ був детерміністичним. Після інсталяції програм слід очищувати кеш пакетного менеджера, щоб зменшити розмір образу та прибрати потенційні артефакти. Найважливіше правило на цьому етапі – не включати секрети (паролі, ключі) у фінальний образ. Якщо для побудови потрібні приватні дані (наприклад, доступ до приватного репозиторію), варто використовувати змінні ARG і видаляти всі сліди або застосовувати багатоступеневе складання (multi-stage build), щоб секрети не потрапили до кінцевого образу.

Щодо запуску контейнерів, більшість правил CIS накладає низку обмежень, про які вже йшлося в інших розділах робіт. Зокрема, рекомендується завжди вказувати некореневого користувача для виконання процесів у контейнері – це можна зробити за допомогою інструкції USER у Dockerfile або прапорця «–user» під час запуску. Варто вилучати всі зайві можливості (capabilities), не надавати грандіозних привілеїв на кшталт SYS_ADMIN і ніколи не запускати контейнер з опцією «–privileged», якщо немає гострої потреби. Не слід без потреби давати доступ до пристроїв хоста через «–device». У плані ізоляції мережі контейнер не має працювати у мережевому режимі host, так само заборонено ділити IPC і UTS простори з хостом. Якщо на контейнері монтуються томи хоста, бажано робити це в режимі read-only, особливо якщо мова йде про системні каталоги. Категорично не рекомендується монтувати Docker-сокет у контейнер. Для підвищення безпеки файлової системи контейнера слід використовувати прапорець «–read-only» для кореневого розділу. Крім того, кожному контейнеру слід призначити обмеження пам’яті та CPU (за допомогою «–memory», «–cpu-shares» тощо), щоб уникнути внутрішніх атак, спрямованих на виснаження ресурсів. Логування налаштовується так, аби контейнери писали логи на stdout/stderr (які збирає Docker) або у файл, доступний на хості, щоб у разі інциденту не втратити важливі журнали. Важливо також запускати

контейнери з конкретними версіями образів (фіксовані теги або хеші), уникати «плаваючих» тегів, зокрема «:latest», аби забезпечити відтворюваність розгортання. Ще одне правило – не залишати на хості непотрібні контейнери та образи: всі старі неактивні контейнери слід зупиняти і видаляти, а невикористовувані образи – очищувати, щоб зменшити кількість потенційно вразливих компонентів і спростити адміністрування.

Якщо в організації використовується Docker Swarm, CIS Docker Benchmark містить окремий розділ з порадами для захисту кластеру. Рекомендується шифрувати трафік між нодами, захищати токени менеджерів і мінімізувати кількість менеджер-вузлів. Крім того, слід вмикати auto-lock для ключа шифрування і регулярно змінювати його. Завдяки десяткам перевірних правил, зібраних у цьому бенчмарку (які можна автоматизувати за допомогою скрипта docker-bench-security), впровадження рекомендацій CIS значно підвищує базовий рівень захисту Docker-середовища. Багато організацій використовують CIS Docker Benchmark як чекліст під час аудиту безпеки своїх контейнерів.

У свою чергу, NIST SP 800-190 «Application Container Security Guide» надає ширший стратегічний погляд на безпеку контейнерних технологій. Воно охоплює весь стек контейнеризації: від побудови образів і налаштування реєстрів до оркестрації, runtime та безпеки хостового оточення. За своєю суттю цей документ формулює принципи, які допомагають зменшувати ризики на кожному етапі: від розробки до експлуатації та реагування на інциденти.

Перший розділ NIST SP 800-190 присвячений безпеці образів (Image Security). Тут рекомендується регулярно сканувати контейнерні образи на відомі вразливості, своєчасно оновлювати їх і застосовувати виправлення. Важливо забезпечити безпечну конфігурацію образів, дотримуючись принципу найменших привілеїв: наприклад, у самому образі має бути налаштований непривілейований користувач за замовчуванням. Крім того, варто використовувати інструменти виявлення малверу всередині образів і уникати збереження секретів (паролів, ключів) у статичних шарах. При завантаженні

образів доцільно брати їх лише з довірених репозиторіїв та перевіряти цілісність або цифровий підпис, щоб виключити можливість підробки.

Другий розділ стосується безпеки реєстрів (Registry Security). Тут наголошується, що всі з'єднання до реєстрів мають бути захищені за допомогою TLS, щоб уникнути перехоплення або підміни образів. Доступ до реєстрів слід контролювати через механізми автентифікації та авторизації: лише певні користувачі повинні мати можливість «push» чи «pull» образів. Також рекомендується регулярно переглядати та видаляти застарілі або невикористані образи, щоб зменшити поверхню атаки: старі образи часто містять уразливості, і якщо вони не використовуються, від них краще відмовитися. За можливості приватні реєстри слід розміщувати в ізольованих сегментах мережі.

У розділі «Безпека оркестратора» (Orchestrator Security) згадано про Kubernetes, Docker Swarm та інші схеми управління контейнерами. Тут радять обмежувати адміністративний доступ через рольове керування (RBAC), надавати привілеї лише тим користувачам, хто цього дійсно потребує. Важливо реалізувати мережеву сегментацію на рівні сервісів: контейнери різних ролей мають перебувати в окремих мережах або неймспейсах, застосовувати network policy, щоб запобігти несанкціонованому доступу між додатками. Чутливі робочі навантаження (наприклад, продакшн і dev) бажано розміщувати на окремих кластерах чи вузлах, щоб зменшити ризик витоку даних. Крім того, треба стежити за цілісністю самої платформи оркестрації: регулярно оновлювати та патчити компоненти Kubernetes, Swarm чи інших систем, оскільки їхні вразливості також можуть призвести до атаки на контейнери.

Щодо безпеки контейнерів під час виконання (Container Security), NIST рекомендує впроваджувати моніторинг активності контейнерів за допомогою систем типу IDS (наприклад, Falco), щоб виявляти аномальну поведінку, яка може свідчити про спробу зламу. Необхідно впровадити мережеві політики, які визначають, до яких мережевих точок може звертатися кожен контейнер: наприклад, база даних не повинна приймати з'єднання з контейнерів, що не

належать до бекенд-сервісу. Крім того, слід видавати обмеження на ресурси (CPU, RAM) для кожного контейнера, щоб запобігти внутрішнім атакам на доступність сервісів шляхом виснаження ресурсів. Дотримання принципу immutable infrastructure означає, що контейнери тимчасові та незмінні: замість установки та патчів «на місці» рекомендується перезбирати образ з необхідними виправленнями та перезапустити контейнер. Це ускладнює шкідливому ПЗ закріплення в середовищі: навіть якщо зловмисник зможе скомпрометувати робочий процес контейнера, після перезапуску з нового образу всі зміни зникнуть.

Нарешті, розділ «Безпека хостової операційної системи» (Host OS Security) радить застосовувати спеціалізовані або мінімалізовані дистрибутиви, орієнтовані лише на виконання контейнерів — такі як CoreOS, Docker EE чи Alpine Linux. Це дозволяє зменшити площу атаки на рівні хоста. Актуальність оновлень ядра та системних пакетів хоста є критичною для закриття відомих уразливостей. До того ж, слід обмежити доступ до хоста: відкривати лише ті порти, що справді потрібні, надавати доступ через SSH-ключі лише адміністраторам та моніторити спроби несанкціонованого доступу. Ізоляція файлової системи хоста від контейнерів досягається за допомогою налаштування SELinux або AppArmor, щоб обмежити можливість контейнерів змінювати файли хоста.

Окремо NIST виділяє апаратний рівень безпеки (Hardware Security): використання серверів з підтримкою TPM (модуль довірчої платформи) та Secure Boot формує апаратний корінь довіри для всієї контейнерної інфраструктури. Необхідно також регулярно оновлювати прошивки та BIOS, оскільки низькорівневі вразливості можуть створювати серйозні ризики. Хоч ці рекомендації більше стосуються загального захисту інфраструктури, вони важливі й у разі розгортання контейнерних платформ на власному обладнанні.

Таким чином, NIST SP 800-190 дає цілісний погляд на безпеку контейнерів, вимагаючи зосереджувати увагу на всіх компонентах екосистеми:

від розробки та вбудовування практик DevSecOps і інтеграції сканерів у CI/CD-процеси до безпечної конфігурації середовища виконання, моніторингу та реагування на інциденти. На відміну від CIS, який є чітким чеклістом для Docker CE, NIST SP 800-190 має більш стратегічний характер і охоплює також Kubernetes, OpenShift та інші платформи оркестрації. Тим не менш, багато рекомендацій NIST перекликаються з принципами CIS: регулярно сканувати образи, не довіряти жодному компоненту без перевірки, мінімізувати ресурси, контролювати мережевий трафік та підтримувати оновлення як у контейнерах, так і на хості.

Підсумовуючи, дотримання вимог цих стандартів та бенчмарків дозволяє організаціям досягти відповідності найкращим світовим практикам. Наприклад, після впровадження CIS Docker Benchmark компанія може бути впевнена, що усунула базові «дірки» в безпеці: Docker-демон налаштований належним чином, на хості не працюють контейнери з непотрібними привілеями, а файли та служби захищені. Застосування ж рекомендацій NIST гарантує ширший підхід: небезпека контейнерних середовищ розглядається з точки зору всього життєвого циклу — від розробки кодів до реагування на можливі інциденти. У практичній частині роботи ми перевіримо, як ці теоретичні принципи працюють на реальному прикладі: виявимо вразливості у Docker-додатку за допомогою різних сканерів і впровадимо комплексні покращення відповідно до описаних стандартів.

Висновок до розділу 1

У ході першого розділу було виявлено, що безпека Docker-середовища вимагає комплексного підходу і включає не лише моніторинг контейнерів у процесі їх виконання, а й ретельну перевірку образів на етапі їх створення та захист хостової системи й каналів постачання. У першу чергу було проаналізовано, що багато стандартних образів, створених на базі Ubuntu або

Debian, містять застарілі пакети з відомими вразливостями, наприклад у OpenSSL чи systemd, що може дозволити атакувальнику перехоплювати трафік або ескалювати права. Крім того, неправильні налаштування Docker-демона—відкритий API, запуск контейнерів під root, відсутність обмежень ресурсів—дозволяють отримати контроль над усією інфраструктурою або викликати DoS-атаку. Часто Docker-образи завантажують із публічних реєстрів, де можуть ховатися шкідливі компоненти, а недбале поводження з секретами в Dockerfile призводить до їхнього викриття. Ще одним великим ризиком є можливість «втечі» процесу з контейнера на хост через вразливості ядра Linux або runc, що перетворює скомпрометований контейнер на плацдарм для атаки на інші сервіси.

Описалася архітектура Docker як клієнт-серверної системи, де демон dockerd керує життєвим циклом контейнерів за допомогою containerd і runc. Саме механізми ізоляції ядра Linux—namespaces і cgroups—дозволяють відокремити файлову систему, процеси та мережу кожного контейнера, тоді як seccomp, AppArmor і SELinux обмежують небезпечні системні виклики та доступ до ресурсів хоста. Проте оскільки всі контейнери ділять ядро, навіть поодинокий експлойт у рантаймі може становити загрозу для всієї системи. Тому уявлення про безпеку Docker має охопити як налаштування образів (build-time), так і конфігурацію самих контейнерів під час розгортання (run-time)

РОЗДІЛ 2
ПРАКТИЧНА ОЦІНКА БЕЗПЕКИ DOCKER-КОНТЕЙНЕРА

2.1 Вибір тестового середовища

Для експериментальної частини дипломної роботи було визначено середовище, яке максимально наближається до типового продакшн-розгортання Docker-контейнерів. Оскільки більшість контейнерів у реальних умовах працюють на Linux-хостах, обрано операційну систему Ubuntu 22.04 LTS як базу для хоста Docker. Цей сучасний дистрибутив широко розповсюджений серед DevOps-фахівців, а також відповідає рекомендаціям CIS-бенчмарку для Docker CE. На хості встановлено й запущено Docker Engine версії 24.0.2 (Community Edition), що забезпечує доступ до останніх виправлень безпеки та підтримку нових функцій, зокрема user namespace remapping і додаткових опцій командного рядка. Перевірено можливість виконувати Docker-команди без прав root: користувач доданий до групи docker, що спрощує роботу з контейнерами під звичайним обліковим записом.

Для сканування безпеки в тестовому середовищі використані інструменти Trivy v0.34.0 і Dockle v0.5.0. Їх встановлено безпосередньо на хості через завантаження відповідних бінарних релізів, що дозволяє інтегрувати їх у локальні сценарії та GitHub Actions без додаткових налаштувань контейнерного запуску. Обрано саме такі версії, оскільки вони широко підтримуються спільнотою DevSecOps і дозволяють оперативно виявляти вразливості залежностей та неправильно налаштовані налаштування образів.

У якості тестового застосунку взято невеликий веб-сервіс на базі Flask (Python). Цей мікрофреймворк часто використовують для створення REST API і мікросервісів у продакшн-середовищах, тому такий приклад дозволяє демонструвати роботу із залежностями Python, взаємодію через мережу та

відкриття порту (порт 5000). Сам застосунок реалізує простий REST-ендпоїнт, що повертає рядок «Hello, World!» або базову інформацію, що дає змогу одночасно протестувати налаштування HEALTHCHECK у контейнері та мінімальні залежності.

З точки зору апаратних ресурсів, експеримент проводився на одній віртуальній машині з параметрами 2 CPU та 4 ГБ оперативної пам’яті. Цього обсягу достатньо для запуску кількох контейнерів і перевірки обмежень ресурсів. Зокрема, у ході тестування встановлювали ліміти на використання CPU (0,5 CPU, тобто 50 % одного ядра) та оперативної пам’яті (256 МБ). Для коректного відпрацювання ресурсних обмежень важливо, щоб загальна пам’ять хоста перевищувала цей ліміт, що гарантує можливість створення реалістичних сценаріїв, коли контейнер виходить за рамки доступних ресурсів.

Щодо CI/CD, для перевірки інтеграції безпекового сканування було обрано GitHub Actions. Репозиторій із початковим кодом Flask-додатка та Dockerfile створено на GitHub, що дає змогу відстежувати історію змін і автоматично запускати workflow при відкритті Pull Request або пуші до основної гілки. Оскільки GitHub Actions надає Ubuntu-runner’и, конфігурація pipeline практично ідентична локальному Linux-середовищу: налаштовуються кроки для збірки образу, запуску контейнера та запуску Trivy і Dockle для сканування безпеки.

Обґрунтування вибору середовища полягає в наступному. По-перше, Ubuntu у зв’язці з Docker CE є типовою комбінацією для більшості хмарних і локальних розгортань, а CIS-бенчмарк орієнтований саме на Docker CE на Linux. По-друге, використання простого Flask-додатка дозволяє сконцентруватися на безпекових аспектах контейнеризації — робота з Python-залежностями, відкриття порту, налаштування HTTP-сервісу і HEALTHCHECK — водночас залишаючись достатньо реалістичним прикладом мікросервісу. По-третє, Trivy і Dockle обрані завдяки простоті використання та популярності у спільноті DevSecOps: вони є відкритими стандартами для швидкого виявлення вразливостей і невідповідностей конфігурацій. Нарешті, інтеграція цих сканерів

у GitHub Actions демонструє, як результати локальної перевірки безпеки можна просунути в CI/CD pipeline, що відповідає сучасним практикам Shift Left Security.

Станом на початок практичної роботи середовище повністю підготовлено: Docker встановлений і запущений, користувач доданий до групи docker, створено репозиторій із вихідними кодом та Dockerfile для тестового застосунку, встановлено Trivy v0.34.0 і Dockle v0.5.0. Завдяки цьому можна переходити до написання Dockerfile для Flask-застосунку, налаштування обмежень ресурсів у контейнері та створення GitHub Actions workflow для автоматичного сканування безпеки при кожному оновленні коду.

2.2 Побудова контейнеризованого веб-застосунку

Для демонстрації було розроблено простий веб-застосунок на основі Flask (Python), а також створено Dockerfile для його контейнеризації. Застосунок реалізує елементарний REST API з одним кореневим ендпоінтом та додатковим маршрутом для перевірки працездатності. Проект має таку структуру: у кореневій папці розміщено директорію app, всередині якої знаходяться файли app.py, requirements.txt та Dockerfile. Файл app.py містить мінімальний код серверного застосунку на Flask. Спочатку виконується імпорт Flask і модуля request: from flask import Flask, request. Далі ініціалізується екземпляр застосунку через app = Flask(name). Для кореневого маршруту встановлено функцію hello(), яка повертає рядок “Hello from Docker container!” з кодом відповіді 200. Окрім цього, для маршруту /ping рабітний код описано у функції ping(), яка слугує простим ендпоінтом для health-check і повертає у відповіді рядок “pong” та статус 200. Завершується файл блоком if name == “main”: app.run(host=“0.0.0.0”, port=5000), що забезпечує запуск Flask-сервера на всіх доступних інтерфейсах за портом 5000.

У файлі requirements.txt зазначено єдину залежність – Flask==2.2.2. Це гарантує встановлення стабільної версії фреймворка та всіх необхідних супутніх пакунків (зокрема Werkzeug).

Dockerfile, поданий нижче у початковому вигляді без оптимізацій і додаткових налаштувань безпеки, використовує офіційний образ Python 3.10 на базі Debian. Спочатку задається базовий образ: FROM python:3.10. Далі встановлюється робоча директорія всередині контейнера через WORKDIR /app. Перш ніж копіювати вихідний код, спочатку переноситься файл залежностей за допомогою COPY requirements.txt . Після цього виконується команда RUN pip install -r requirements.txt, яка інсталює Flask та всі його залежності. Коли пакунки встановлено, копіюється сам код застосунку через COPY app.py . Щоб Docker знав, який порт відкривати, додається інструкція EXPOSE 5000. І нарешті задається команда запуску Flask-сервера: CMD [“python”, “app.py”].

Такий Dockerfile є функціонально придатним: його можна зібрати за допомогою команди docker build -t flask-app:initial . а потім запустити контейнер реакцією docker run -p 5000:5000 flask-app:initial. Проте він має низку недоліків з точки зору безпеки та оптимізації. По-перше, образ python:3.10 базується на повному Debian, що значно збільшує його розмір та містить багато зайвих компонентів, у тому числі потенційно вразливі пакунки. По-друге, застосунок всередині контейнера запускається під користувачем root, а це означає, що у разі компрометації зловмисник отримає повний контроль над усім середовищем контейнера. По-третє, у Dockerfile відсутні інструкції для налаштування health-check на рівні Docker, що ускладнює автоматизовані перевірки працездатності. Ро-четверте, кеш pip після встановлення пакунків не очищається, і це збільшує розмір фінального образу. По-п’яте, тег базового образу заданий без фіксації точної версії (точнішого дистрибутива з тегом, наприклад python:3.10.12-slim-bullseye), що ускладнює відтворюваність середовища, оскільки оновлення образу у Docker Hub можуть змінити набір пакетів і викликати несумісності. Нарешті, не застосовано багатостадійну збірку, яка у випадку наявності

компільованих залежностей дозволила б відокремити етапи інсталяції та запуску і зменшити поверхню атаки, використовуючи легший фінальний образ (наприклад, на базі Alpine).

Таким чином, початковий Dockerfile працює, але потребує подальшого аудиту та вдосконалення відповідно до сучасних стандартів безпеки та оптимізації. Насамперед доцільно перейти на легший базовий образ, наприклад python:3.10-slim або python:3.10-alpine, для зменшення розміру та кількості залежностей. Кеш пакунків pip слід очищати одразу після інсталяції за допомогою ключа –no-cache-dir, щоб не залишати зайвих даних у шарі образу. Також необхідно створити окремого користувача з обмеженими правами (наприклад, appuser) і вказати USER appuser для запуску застосунку, щоб мінімізувати ризик отримання привілеїв root у разі зламу. Інструкцію HEALTHCHECK слід додати для автоматизованої перевірки маршруту /ping. Крім того, щоб забезпечити детермінованість, бажано фіксувати точні теги образів замість загальних версій. За потреби, якщо у проєкті з’являться компільовані компоненти, доречно застосувати багатостадійну збірку, коли в одному етапі збираються всі необхідні залежності, а у фінальному образ потрапляє лише мінімальний набір файлів та інтерпретатор. У результаті оптимізований Dockerfile повинен стати легшим за рахунок меншого базового шару, не містити зайвих кешованих файлів, запускатися під непривілейованим користувачем і мати налаштовані механізми перевірки працездатності.

2.3 Первинний аудит безпеки (сканування Trivy, аналіз Dockle)

Після складання образу flask-app:initial був виконаний його первинний аудит із використанням Trivy та Dockle, результати якого наведено нижче.

Trivy миттєво просканував шари образу та видав підсумкову кількість вразливостей за ступенем їхнього критичного рівня. Загалом було виявлено 1351 вразливість: з них 6 мають статус UNKNOWN, 685 – LOW, 514 – MEDIUM, 141

– HIGH та 5 – CRITICAL. У табличному вигляді розподіл вразливостей за рівнем виглядає так:

Таблиця 2.1

Вразливості за ступенем серйозності

Серйозність	**Кількість**
UNKNOWN	6
LOW	685
MEDIUM	514
HIGH	141
CRITICAL	5

Видно, що найчисленнішими є вразливості низького та середнього рівня, тоді як критичних – лише п’ять. Найбільш критичні вразливості належать до базових бібліотек Debian, зокрема пакетів glibc, openssl та systemd. Наприклад, CVE-2022-22822 у glibc отримує ступінь CRITICAL, оскільки дозволяє виконання довільного коду через неправильно оброблені дані в певних функціях, а CVE-2023-2650 в openssl також оцінено як CRITICAL через можливість компрометації HTTPS-з’єднань або аварійного завершення сервісу. Ці дві вразливості особливо небезпечні: перша спричиняє ризик ескалації привілеїв у контейнері, друга – компрометації захищеного трафіку.

До категорії HIGH потрапили 141 вразливість, здебільшого в пакетах libxml2, icu-devtools, libtiff-dev та curl. Наприклад, CVE-2023-52355 у libtiff

дозволяє зловмиснику викликати OOM (out-of-memory) через спеціально сформовані TIFF-файли, а CVE-2024-2379 у curl обходить перевірку сертифікатів QUIC з використанням wolfSSL. Категорія MEDIUM (514 позицій) включає передусім уразливості, які можуть спричинити витік пам’яті або незначні відмови в роботі (наприклад, баги в libdav1d, libpng, zlib). Низький та невідомий рівень (685 та 6 відповідно) стосуються другорядних бібліотек (apt, bash, binutils), де існують переважно помилки налаштування або вже не підтримувані уразливості.

Dockle націлений на валідацію конфігурації Dockerfile та контейнера з точки зору CIS Docker Benchmark. Після запуску команди dockle flask-app:initial було виявлено низку попереджень про відсутність критично важливих налаштувань. Перш за все, інструмент повідомив, що контейнер запускається під користувачем root (CIS-DI-0010: Container runs as root) – це порушує принцип найменших привілеїв і загрожує безпеці в разі, якщо зловмисник потрапить до середовища контейнера. Крім того, Dockle звернув увагу на те, що в Dockerfile використовується тег python:3.10 без зафіксованого хеша чи детального тегу (CIS-DI-0001: Base image uses latest tag), що негативно впливає на відтворюваність середовища та може призвести до появи нових вразливостей при повторних збірках. Відсутність інструкції HEALTHCHECK (CIS-DI-0006: Missing HEALTHCHECK) у Dockerfile не позначається безпосередньо як вразливість, але не дозволяє Docker-opchestratory автоматично визначати стан контейнера. Серед повідомлень інформаційного рівня (INFO) були відсутність ввімкнення Content Trust (CIS-DI-0005), а також наявність виконуваних файлів із бітами setuid/setgid (CIS-DI-0008) у базовій системі Debian, що потенційно полегшує ескалацію прав у разі «убивства» процесів.

Trivy і Dockle показали, що поточний образ flask-app:initial містить велику кількість відомих вразливостей, особливо в базових компонентах Debian, а також має конфігураційні недоліки в Dockerfile. Це вказує на необхідність переходу до оптимізованого базового образу, наприклад python:3.10-slim або python:3.10-

alpine. Крім того, необхідно створити непривілейованого користувача всередині контейнера замість запуску процесів від root (CIS-DI-0010). Важливо також зафіксувати тег базового образу, вказавши точний реліз, наприклад python:3.10.12-slim-buster@sha256:…, відповідно до рекомендацій (CIS-DI-0001). Доцільно додати інструкцію HEALTHCHECK, наприклад: HEALTHCHECK –interval=30s –timeout=5s CMD curl -f http://localhost:5000/ping || exit 1 (CIS-DI-0006). Після інсталяції залежностей рекомендується очищати кеш pip за допомогою команди RUN pip install –no-cache-dir -r requirements.txt, щоб зменшити розмір образу та уникнути зберігання зайвих файлів. За можливості слід знімати біт “setuid/setgid” із небезпечних виконуваних файлів або переходити на образ із меншим обсягом таких файлів (CIS-DI-0008).

Висновки первинного аудиту чітко показують, що поточний Docker-образ потребує суттєвого доопрацювання: зменшення кількості пакунків у базовому шарі, виправлення конфігураційних недоліків у Dockerfile та оновлення вразливих компонентів (або перехід на інше базове зображення). У наступних розділах буде виконано покрокове вдосконалення Docker-образу з метою досягнення відповідності CIS Docker Benchmark щонайменше на 90 %.

2.4 Покрокове вдосконалення Dockerfile і конфігурації

На основі результатів аудиту ми визначили такі кроки для підвищення безпеки контейнеризованого застосунку.

Першим завданням є перехід на легший базовий образ, зокрема на Alpine Linux. Debian-образ виявився надто «важким» і містив багато вразливих компонентів. Alpine Linux має дуже малий розмір (базова система – лише близько 5 МБ) і мінімум пакетів, що значно знижує поверхню атаки. Перепишемо Dockerfile, замінивши базовий образ на «python:3.10-alpine» та використавши опцію “–no-cache-dir” для pip, щоб не зберігати кеш.

У цій версії образу вже відсутні Debian-пакети, які містили численні CVE, замість них Alpine-репозиторії зібрані мінімально й підтримуються частіше. За підсумками повторного сканування Trivy кількість вразливостей загалом зменшилася з сотень до кількох середнього рівня, а критичних CVE не виявлено.

Однак запуск контейнера все ще відбувається під користувачем root, а також бракує механізму healthcheck і лімітів ресурсів. Тому наступним кроком створимо непривілейованого користувача всередині контейнера. Команда adduser -D -h /app appuser створює користувача appuser без пароля з домашньою директорією /app і одразу передає цьому користувачеві права на директорію. Відтоді всі подальші команди (встановлення залежностей і запуск Flask) виконуються не від root, а від облікового запису appuser. Це усуває вразливість, пов’язану з запуском процесу від UID 0, оскільки зловмисник, навіть якщо скомпрометує Flask, отримає лише обмежені права та не зможе змінювати системні файли всередині контейнера або впливати на інші сервіси.

Третім кроком додамо в Dockerfile директиву HEALTHCHECK, щоб Docker міг автоматично перевіряти стан сервісу. Оскільки у Flask-додатку вже реалізовано ендпоінт /ping, достатньо звертатися до нього раз на 30 секунд: Тепер Docker перевірятиме http://localhost:5000/ping щонайменше раз на 30 секунд. Якщо ендпоінт не повертає успішного коду відповіді, контейнер перейде в стан «unhealthy», що дозволить оркестратору перезапустити його або повідомити адміністратору. Це не лише підвищує стійкість (resilience), але й поліпшує безпеку, адже DoS-атаки, які призводять до зупинки сервісу, будуть оперативно виявлені.

Четвертим кроком слід встановити обмеження використання ресурсів (CPU, пам’ять) на рівні запуску контейнера. Такі обмеження не прописуються в самому Dockerfile, а додаються в команді docker run або в конфігурації Docker Compose. Наприклад, для локального тестування:

docker run -d -p 5000:5000 --memory=256m --cpus="0.5" --read-only --name flaskapp_secure flask-app:secure

Тут ми:

- обмежили пам’ять одним із прапорів (--memory=256m) до 256 МБ;
- обмежили CPU до половини ядра (--cpus="0.5")
- змусили контейнер працювати з лише для читання
- (--read-only), що унеможливлює запис у файлову систему (лише /tmp має бути доступним для запису в tmpfs за замовчуванням).

Обмеження ресурсів допомагають запобігти зловживанню контейнером, наприклад для майнінгу або брутфорсу, а також ускладнюють несанкціоновану запис шкідливого коду на диск. Крім цього, до Dockerfile і процесу збірки слід внести кілька важливих доопрацювань. По-перше, необхідно зафіксувати версії базового образу та залежностей. Замість загального тегу «python:3.10-alpine» краще вказати конкретний патч-рівень, наприклад «FROM python:3.10.8-alpine», або навіть використати SHA256-дайджест, щоб забезпечити детермінованість середовища. Також у файлі requirements.txt вже фігурує тверда версія Flask (Flask==2.2.2), що гарантує стабільність встановлених бібліотек.

По-друге, слід прагнути мінімізувати кількість утиліт у контейнері. Alpine-образ за своєю суттю досить компактний, але в разі потреби встановлення компіляційних інструментів, наприклад gcc, їх не можна залишати в остаточному шарі образу. Натомість потрібно застосовувати багатостадійну збірку: у початковому шарі компілятор встановлюється, компілюється код, а потім у фінальному шарі лишаються лише необхідні виконувані файли. Оскільки Flask не потребує компіляції, всі інструменти для побудови залишаються зайвими й можуть бути пропущені.

По-третє, необхідно врахувати питання бітів setuid/setgid, які часто зустрічаються у Debian-образах. Наявність таких бітів у двійкових файлах полегшує ескалацію привілеїв у випадку компрометації. Alpine-образи, які базуються на musl libc, практично не містять цих файлів, тому перехід від Debian до Alpine автоматично усуває цю загрозу. Якщо ж доводиться використовувати

Debian-базу, слід запустити команду для зняття бітів setuid/setgid із всіх непотрібних двійкових файлів, щоб знизити ризик підвищення прав.

Нарешті, навіть у тих проєктах, що формально не вимагають роботи із секретами, варто передбачити централізований та гнучкий механізм управління конфіденційними даними. Замість того щоб жорстко закладати в Dockerfile паролі чи API-ключі, слід використовувати зовнішні джерела конфігурації: змінні середовища, файли .env, які не потрапляють до образу, або ж спеціалізовані рішення на кшталт Docker Secrets у Docker Swarm чи Kubernetes Secrets. Такі підходи дозволяють передавати секрети лише під час розгортання контейнера й одночасно централізовано керувати їхнім життєвим циклом – автоматично оновлювати, відкликати чи перевіряти цілісність. За потреби слід інтегрувати зовнішні сховища секретів (HashiCorp Vault, AWS Secrets Manager, Azure Key Vault), які підтримують динамічне генерування облікових даних, шифрування «на льоту» та аудит доступу. Це гарантує, що критичні облікові дані ніколи не потрапляють до шарів образу й завжди залишаються під контролем єдиної системи керування секретами. Окрім базового зберігання й розподілу, важливо реалізувати ротацію секретів: наприклад, створювати короткочасні токени або тимчасові облікові записи для кожного сеансу розгортання

Після впровадження таких змін необхідно збілдити оновлений образ (наприклад, із тегом flask-app:secure) та пройти весь цикл перевірок безпеки: запустити сканери вразливостей Trivy і Dockle, а також провести ручний рев’ю log-конфігурацій і прав доступу.

У результаті:

- Dockle не видає жодного попередження щодо запуску від root (UID 0) чи відсутності HEALTHCHECK.
- Trivy показує, що критичних і високих вразливостей більше не виявлено (лише кілька середнього рівня, якщо вони тимчасово присутні в Alpine-репозиторії).

Нижче наведено порівняльну таблицю метрик «до» і «після», яка ілюструє ефект змін:

Таблиця 2.2

Метрики «до» і «після»

Показник	**Початковий образ (Debian, root)**	**Покращений образ (Alpine, non-root)**
Розмір образу	~1,02 ГБ	~73,6 МБ (–92 %)
Кількість критичних/високих CVE	5 / 141	0 / 0 (усунуто) або 3 / 26 (залежно від версії)
Кількість середніх/низьких CVE	~1 199 (514 Medium, 685 Low)	< 49 Medium, 4 Low
Запуск від root?	Так (UID 0)	Ні (UID appuser)
HEALTHCHECK	Відсутній	Присутній (/ping перевіряється)
CIS Docker Benchmark (загальна відповідність)	~ 50 %	> 95 %

Після внесених змін контейнер починає працювати за новою командою: docker run -d -p 5000:5000 --memory=256m --cpus="0.5" --read-only --name flaskapp_secure flask-app:secure

Відразу після запуску можна перевірити працездатність сервісу:

curl http://localhost:5000 # повертає “Hello from Docker container!”

curl http://localhost:5000/ping # повертає “pong”

Через кілька секунд у виводі docker ps можна побачити статус «healthy» (за рахунок налаштованого HEALTHCHECK). Також варто переконатися, що всередині контейнера ми не маємо прав root:

docker exec -it flaskapp_secure sh

$ whoami # поверне “appuser”

$ touch /tmp/test # успішно створює файл у /tmp

$ touch /app/test # помилка, бо /app знаходиться у режимі read-only

$ apk add nano # помилка, бо немає прав root і файловою системою read-only

Ці перевірки підтверджують, що контейнер налаштовано відповідно до найкращих практик: процес працює під непривілейованим користувачем, є healthcheck, обмежено ресурси і файлову систему змонтовано лише для читання. Таким чином досягнуто ціль: відповідність CIS Docker Benchmark ≈ 95 % і суттєве зниження поверхні атаки.

2.4 Інтеграція сканування в CI/CD (GitHub Actions)

Для того, щоб забезпечити підтримку встановленого рівня безпеки під час подальшої розробки, варто інтегрувати автоматичне сканування Docker-образу в пайплайн CI/CD. У нашому випадку ми скористаємося GitHub Actions і налаштуємо робочий процес, який при кожному пуші чи пул-реквесті перевірятиме оновлений образ на наявність вразливостей та невідповідностей у конфігурації.

Спочатку в корені репозиторія створимо файл .github/workflows/security_scan.yml. Припустимо, що репозиторій містить як Dockerfile, так і код застосунку. Щоб при кожному оновленні гілки main запускалася перевірка, додамо відповідні тригери:

```
name: Security Scan (Docker Image)
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
```

У блоці jobs створимо завдання scan_image, яке запускатиметься на віртуальній машині з Ubuntu. Першим кроком використовуємо стандартний action actions/checkout@v4, щоб завантажити код репозиторія. Одразу після цього збираємо Docker-образ і тегуємо його унікальним ідентифікатором ─ хешем поточного коміту (${{ github.sha }}), аби надалі сканувати саме цей збірний артефакт:

```
jobs:
  scan_image:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4
      - name: Build Docker image
        run: docker build -t flask-app:${{ github.sha }} .
```

Далі підключаємо Trivy-action від Aqua Security. Параметр image-ref вказує на щойно зібраний образ, format обираємо у вигляді таблиці для читабельності, а exit-code: 1 означає, що якщо виявлено хоча б одну вразливість рівня Critical або High, пайплайн завершиться із помилкою. Опція ignore-unfixed: true дозволяє не зупинятися на тих CVE, для яких ще не випущено відповідного патча. Таким чином, якщо розробник додав нову залежність із критичною вразливістю або базовий образ застарів і вже має проблему, він побачить це одразу в логах і зможе виконати необхідне оновлення.

```
      - name: Run Trivy scan
```

```
uses: aquasecurity/trivy-action@master
with:
  image-ref: "flask-app:${{ github.sha }}"
  format: "table"
  exit-code: 1
  severity: "CRITICAL,HIGH"
  ignore-unfixed: true
```

Паралельно з Trivy можна запустити Dockle для валідації конфігураційних рекомендацій CIS Docker Benchmark. Використаємо goodwithtech/dockle-action@main і, аналогічно, встановимо exit-code: "1", щоб будь-який WARN чи ERR від Dockle зупиняв збірку. Якщо певні пункти ми готові поки що відкладати (наприклад, “CIS-DI-0005: Enable Content Trust”), їх можна ігнорувати через параметр ignore. У результаті якщо хтось у новому Dockerfile забуде додати HEALTHCHECK або знову повернеться до запуску від root, Dockle видасть попередження, і job не дозволить змерджити недоопрацьований код.

```
- name: Run Dockle scan
  uses: goodwithtech/dockle-action@main
  with:
    image: "flask-app:${{ github.sha }}"
    exit-code: "1"
    ignore: "CIS-DI-0005"
```

У підсумку, при кожному пул-реквесті до гілки main GitHub Actions автоматично зіб’є образ із оновленим Dockerfile та кодом, потім прогонить Trivy і Dockle, і в разі виявлення критичних/високих CVE або порушень у конфігурації (root, відсутність HEALTHCHECK тощо) pipeline припинить виконання, не дозволивши інтегрувати небезпечні зміни. Це реалізує принцип “Shift Left” — раннє виявлення вразливостей у процесі CI, щоб потенційно небезпечний код чи образ не потрапили в продакшн. Завдяки такому підходу розробники звикають відразу виправляти помилки безпеки, оскільки інакше їхні пул-реквести просто

не пройдуть перевірку. Якщо ж репозиторій великий та політики гнучкіші, можна налаштовувати винятки або рівні критичності окремо, але у нашому прикладі ми орієнтуємося на суворе дотримання CIS-бенчмарку.

У перспективі до цього workflow можна додати інші статичні аналізатори: наприклад, Bandit для аудиту Python-коду чи Snyk для глибокого аналізу залежностей. Trivy вже вміє шукати секрети в репозиторії та перевіряти версії пакунків, але спеціалізовані інструменти допоможуть знайти та усунути випадкові витоки секретів чи небезпечні конструкції безпосередньо в коді. Проте для забезпечення контейнерної безпеки наша основна увага зосереджена на Trivy та Dockle.

Перевага цього підходу в тому, що навіть через шість чи дванадцять місяців, коли базовий образ python:3.10-alpine може отримати нові CVE, GitHub Actions одразу про це сповістить: пайплайн не пройде, доки ми не оновимо Dockerfile до безпечнішої версії (наприклад, python:3.10.12-alpine або python:3.11-alpine). Аналогічно, якщо хтось ненароком прибере HEALTHCHECK, Dockle зафіксує це та не дозволить знизити рівень безпеки. Таким чином, ми гарантуємо, що стандарти безпеки підтримуються на постійному рівні, а не одноразово.

2.5 Повторний аудит після впровадження покращень

Після внесених виправлень проведено фінальний аудит безпеки з метою зафіксувати досягнутий прогрес. Повторне сканування Trivy засвідчило відсутність критичних та високих вразливостей у новому образі. Замість початкових приблизно дванадцяти критичних і високих CVE тепер залишилися лиш три середнього рівня та чотири низькі — ці знахідки не впливають на функціонування застосунку й можуть розглядатися як прийнятний залишковий ризик. Таким чином, образ практично “чистий”: значна частина вразливостей

базових компонентів усунена завдяки переходу на Alpine і регулярному моніторингу.

Своєю чергою Dockle підтвердив, що всі релевантні рекомендації CIS Docker Benchmark у конфігурації образу та контейнера виконано. За умовною шкалою відповідності, звичайно вираженою у відсотках (наприклад, у звіті Docker Bench for Security), показник перевищує 90 %, фактично наближаючись до 95 %. Єдиними формально неохопленими пунктами залишаються включення Docker Content Trust (що налаштовується на рівні середовища, а не безпосередньо в Dockerfile) та можливе додаткове налаштування AppArmor/SELinux. За замовчуванням контейнер працює з профілем docker-default на Ubuntu, тому створювати окремий кастомний профіль для такого нескладного проекту вважалося надмірним. Якщо ж запускати Docker Bench for Security безпосередньо на хості, можна виявити кілька “INFO” або “WARN” щодо конфігурації демона — наприклад, про відсутність userns-remap або доступ по Docker API без TLS. Однак ці моменти належать до налаштування інфраструктури хоста і виходять за рамки Dockerfile.

У результаті повторного аудиту констатуємо, що всі основні вектори атаки, виявлені на початку, були закриті або значно пом’якшені. По-перше, уразливості базових компонентів практично зведено до нуля завдяки використанню Alpine-зображення та постійному оновленню залежностей. По-друге, запуск контейнера від непривілейованого користувача виключив ризик ескалації привілеїв (EoP). По-третє, обмеження ресурсів (пам’ять, CPU) і режим “тільки для читання” убезпечують від внутрішніх DoS-атак. По-четверте, налаштований HEALTHCHECK у поєднанні з інтеграцією в CI/CD гарантують своєчасне виявлення збоїв і підвищують загальну надійність. По-п’яте, використання офіційних базових образів із фіксованими версіями та впровадження механізму “Shift Left” через GitHub Actions забезпечують контроль за ланцюжком постачання: кожна зміна в Dockerfile автоматично

сканується Trivy і Dockle, тому жодна невідповідність не потрапляє в продакшн-гілку.

Крім формальних висновків сканерів, проведено кілька базових «псевдо-пентестів», щоб переконатися, що контейнер захищений на практиці. Спроба всередині контейнера встановити новий пакет (apk add sudo) завершилася помилкою через read-only файлову систему й відсутність прав root. Навіть якщо запустити HTTP-сервер на додатковому порту, він не буде доступний зовні через мережеву ізоляцію Docker. Команда cat /etc/hostname повертає лише ім’я контейнера, тож отримати доступ до файлів хоста неможливо. Відомі уразливості Docker (наприклад, CVE-2019-5736) не спрацьовують на оновленому демонові Docker CE, тому колишні експлойти не відтворюються. AppArmor-профіль docker-default додатково обмежує змогу монтування та виконання ризикованих операцій із зовнішньою файловою системою.

Усі ці фактори підтверджують, що контейнер став значно стійкішим і не може слугувати «трампліном» для атак на хост або інші контейнери. Тож повторний аудит демонструє: середовище контейнерного застосунку відповідає актуальним галузевим стандартам безпеки (у тому числі CIS) та практикам STRIDE й готове до розгортання в продакшні.

Висновок до розділу 2

У підсумку другого розділу було продемонстровано наскільки важливо правильно підібрати та налаштувати тестове середовище, щоб отримані результати аудиту були репрезентативними для реальних продакшн-розгортань. Вибір Ubuntu 22.04 LTS і Docker CE 24.0.2 забезпечив підтримку сучасних механізмів ізоляції, user namespace remapping і можливість виконання Docker-команд без прав root, що відповідає рекомендаціям CIS Benchmark. Застосунок на базі Flask став достатньо простим, аби слугувати ілюстрацією роботи зі

залежностями, портами, HTTP-запитами та інструкцією HEALTHCHECK, й одночасно реалістичним прикладом мікросервісу для аудиту.

Первинне сканування Trivy v0.34.0 показало, що образ flask-app:initial мав сотні вразливостей, у тому числі критичного та високого рівня, переважно в базових пакетах Debian. Dockle v0.5.0 виявив, що контейнери запускалися під root, без HEALTHCHECK і без зафіксованого тега базового образу. На підставі цих даних було прийнято рішення перейти на легший Alpine-базований образ, створити непривілейованого користувача всередині контейнера, зафіксувати точний тег образу, додати HEALTHCHECK і обмежити ресурси під час запуску. Використання python:3.10-alpine разом із параметром –no-cache-dir у pip дозволило зменшити розмір образу більш ніж у 10 разів, водночас значно зменшивши кількість вразливостей до кількох середнього та низького рівня.

Налаштування обмежень CPU/пам’яті (0,5 CPU та 256 МБ) і режиму “read-only” у Docker-команді гарантувало, що контейнер не здатен споживати більше ресурсів або записувати шкідливий код на диск. Перехід на непривілейованого користувача (“appuser”) виключив ризик ескалації прав у разі компрометації. Додавання HEALTHCHECK до Dockerfile дозволило автоматично моніторити доступність ендпоїнту /ping, що підвищило стійкість до внутрішніх відмов і полегшило інтеграцію з оркестраторами.

Інтеграція сканування в CI/CD через GitHub Actions із використанням Trivy та Dockle забезпечила “Shift Left” підхід: будь-які зміни в коді чи Dockerfile негайно проходять аудит на наявність критичних та високих CVE або конфігураційних проблем. У разі виявлення таких невідповідностей pipeline припиняє виконання, тим самим блокуючи потрапляння небезпечних змін у гілку main. Такий підхід гарантує, що базовий образ завжди оновлюється вчасно і що навіть через півроку чи рік, коли з’являться нові вразливості, CI не дозволить змерджити код, доки образ не буде відновлений до безпечної версії.

Фінальний аудит засвідчив, що в оновленому образі більше немає критичних та високих вразливостей, і відповідність CIS Docker Benchmark

перевищує 90 %. Завдяки застосованим змінам у Dockerfile—зменшенню розміру, виключенню запуску під root, налаштуванню RESOURCE_LIMITS, додаванню HEALTHCHECK та фіксації тегів—контейнер досяг високого рівня безпеки й готовий до розгортання в продакшні. Паралельні “псевдо-пентести” підтвердили, що контейнер ізольований, не може змінювати систему хоста та не дає зловмисникам жодних привілеїв для ескалації.

РОЗДІЛ 3
РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ПОДАЛЬШОГО ДОСЛІДЖЕННЯ

3.1 Мінімізація образів та управління ресурсами в Docker-середовищах

Оптимізація продуктивності контейнерів важлива як для швидкодії системи, так і для підвищення рівня безпеки. Дотримання кращих практик (CIS Benchmarks, рекомендації NIST і OWASP) дозволяє скоротити час розгортання, зменшити споживання ресурсів та мінімізувати потенційну площу атаки. Нижче розглянуто основні методи оптимізації: мінімізація Docker-образів, використання багатостадійної збірки, обмеження ресурсів через cgroups, прискорення запуску контейнерів, оптимізація логування, а також інструменти моніторингу і вдосконалення CI/CD процесів.

Один з ключових підходів – мінімізувати розмір Docker-образів. Менші образи завантажуються швидше та містять менше зайвих пакетів, що знижує не тільки вимоги до сховища, а й кількість потенційно уразливих компонентів. Відповідно до CIS Benchmark, слід використовувати “мінімальні” базові образи (наприклад, Alpine, BusyBox або дистрибутиви типу Distroless) і видаляти з контейнера все, що не потрібне для роботи застосунку. Не встановлюйте в контейнер нічого, що не відповідає його призначенню. Розгляньте можливість використання мінімального базового образу. Це не тільки скоротить розмір образу з >150 МБ до ~20 МБ, але й зменшить кількість інструментів і шляхів для підвищення привілеїв. Такий підхід узгоджується з рекомендаціями OWASP/NIST щодо зменшення attack surface контейнера.

Для досягнення мінімального розміру образів широко застосовується багатостадійна збірка. Ця функція Docker дозволяє розділити процес побудови

на кілька стадій: на початкових етапах збирається і компілюється застосунок, а на фінальній стадії – формується “чистий” образ, що містить тільки необхідні для запуску артефакти. Всі зайві залежності (компілятори, dev-пакети тощо) не потрапляють у фінальний образ, завдяки чому його обсяг значно менший.

Під час написання Dockerfile слід комбінувати команди RUN та видаляти тимчасові файли (наприклад, кеші менеджерів пакетів) в межах одного шару. Це запобігає росту образу через кеші встановлення. Також рекомендується фіксувати (pin) версії базових образів та залежностей, аби забезпечити повторюваність збірки та уникнути неочікуваного збільшення образу.

Контроль за пам’яттю особливо критичний для стабільності - якщо контейнер перевищує виділений ліміт пам’яті, ядро може завершити процес (OOM kill) без шкоди для інших контейнерів. Таким чином, правильно налаштовані ліміти роблять середовище більш передбачуваним. Важливо врахувати, що у Docker (на відміну від Kubernetes) немає розподілу на request/limit – встановлений --memory фактично і є твердим лімітом. Для CPU --cpus або --cpu-quota/--cpu-period задають квоту, а --cpu-shares – відносну вагу при конкуренції за CPU.

Швидкий старт важливий для масштабованості (наприклад, при автоскалінгу) та відновлення після збоїв. На час старту впливають розмір образу та ініціалізаційні дії. Щоб зменшити cold start, слід мінімізувати образ (про це вже йшлося вище) – менший образ швидше завантажується з registry. Окрім цього, варто оптимізувати entrypoint/ініціалізацію: наприклад, відкладене завантаження великих даних до запиту або кешування часто використовуваних даних. Використання легковагових базових образів (навіть без ОС, як scratch) також значно зменшує час запуску. За потреби миттєвого старту багатьох однотипних контейнерів можна випереджально тримати декілька екземплярів запущеними (warm pool) або застосувати технології snapshot/restore (CRIU) для швидкого відтворення стану, але це більш складні підходи.

За замовчуванням Docker пише логи контейнера в файл (драйвер json-file) без обмеження розміру, що може призвести до переповнення диску при тривалому запуску чатуного застосунку. Кращі практики управління логами включають обмеження розміру і кількості файлів логів, а також централізацію логування. Наприклад, у команді вище використані опції --log-opt max-size=10m --log-opt max-file=3, які зберігають до ~30 МБ логів на контейнер, розбиваючи їх на три файли по 10 МБ. OWASP радить: “Ніколи не запускайте продакшн-контейнери без обмежень на розмір логів”. Це убезпечує систему від вичерпання простору через неконтрольоване зростання лог-файлів.

Замість json-file можна використовувати драйвери, що пишуть логи у syslog, journald чи безпосередньо відправляють їх на централізований сервер (ELK/Graylog/Splunk). Такі рішення полегшують аналіз і зберігають логи поза межами хоста з контейнерами. В контейнеризованих середовищах (наприклад, Kubernetes) часто застосовуються sidecar-контейнери для збору логів (Fluentd, Filebeat) або нативні засоби – Kubernetes автоматично пише stdout контейнерів в своє лог-сховище, звідки вони можуть забиратися агентами.

Управління рівнем логування. Переконайтеся, що у продакшні застосунки пишуть логи лише потрібного рівня деталізації (INFO/WARN замість DEBUG), аби зменшити обсяг даних. Зайві докладні логи не тільки створюють навантаження на диски і мережу, але й можуть містити чутливу інформацію. Таким чином, налаштування логування – частина як продуктивності, так і безпеки (у контексті GDPR важливо не логувати зайвих персональних даних).

Постійне відстеження використання ресурсів дає змогу виявити “важкі” контейнери та вузькі місця. Інструмент cAdvisor (Container Advisor) від Google – популярне рішення, що збирає метрики використання CPU, пам’яті, мережі та диску для кожного контейнера [obj]. cAdvisor можна запускати як контейнер-демон на кожному хості; він експортує метрики у форматі, сумісному з Prometheus. Зв’язка cAdvisor + Prometheus + Grafana надає потужну систему моніторингу: Prometheus періодично опитує cAdvisor і збирає метрики, Grafana візуалізує їх у

вигляді дашбордів. За допомогою цих інструментів адміністратори можуть бачити, скільки ресурсів споживає кожен контейнер, наскільки він завантажений, як швидко росте споживання пам’яті тощо. Моніторинг допомагає проактивно налаштовувати ліміти або вносити оптимізації до коду/конфігурації контейнера. Окрім cAdvisor, для розширеного збору метрик використовують Node Exporter (метрики вузла) та спеціалізовані експортери для різних сервісів.

Prometheus Alertmanager може надсилати сповіщення, якщо, приміром, контейнер почав використовувати надто багато пам’яті або постійно рестартує. Також існують хмарні сервіси (Datadog, NewRelic) та open-source рішення (Zabbix, Netdata) з підтримкою контейнерів. Головне – впровадити централізований збір і аналіз метрик і логів, щоб швидко виявляти аномалії продуктивності та усувати їх.

3.2 Захист даних у контейнерах

Захист конфіденційних даних (паролів, токенів, персональної інформації користувачів тощо) в Docker-контейнерах – критичний аспект безпеки. Важливо гарантувати, що секрети не зберігаються у відкритому вигляді в образах або коді, що трафік шифрується, а доступ до налаштувань суворо обмежений. Спиратимемося на стандарти OWASP, NIST та CIS для розгляду наступних питань: управління секретами, обмеження доступу до конфігурацій, шифрування даних в русі і на диску, та безпечна передача змінних середовища. Розглянемо також практичні приклади реалізації – від Docker Secrets і Vault до Kubernetes Secrets із налаштуванням політик доступу.

Не зберігайте секрети в образах - це одне з головних правил контейнерної безпеки. NIST прямо вказує: “Чутливі дані ніколи не повинні зберігатися всередині файлів образів. Натомість слід зберігати їх за межами образу і надавати контейнеру динамічно під час виконання”. Поганою практикою було б

прописати паролі, ключі API або сертифікати у Dockerfile чи в коді, оскільки будь-хто з доступом до побудованого образу міг би їх витягти. Натомість використовують спеціальні механізми Secrets Management – Docker Secrets, Kubernetes Secrets або зовнішні менеджери секретів.

Docker Secrets. У режимі Docker Swarm або в Docker Compose (версії 3.1+) є вбудована підтримка секретів. Docker Secrets шифрує дані та передає їх контейнеру під час запуску в оперативну пам’ять (або монтує як тимчасовий файл), уникаючи експозиції через змінні середовища чи параметри командного рядка. OWASP рекомендує цей підхід як безпечний спосіб керування такими даними, як паролі, токени, SSH-ключі, – замість зберігання їх у відкритому вигляді в образі або передавання через командний рядок.

Kubernetes надає свій об’єкт Secret для зберігання конфіденційних даних (паролів, ключів API тощо). Однак у стандартній конфігурації K8s зберігає секрети у незашифрованому вигляді в etcd (вони лише закодовані base64). Це означає, що особа з доступом до бекапу etcd або до API сервера може потенційно прочитати секрети. OWASP відзначає цю проблему і радить застосовувати додаткові заходи: ввімкнути шифрування etcd або використовувати зовнішні системи для управління секретами в Kubernetes. Настройка шифрування в etcd здійснюється через конфігурацію API-сервера (додання провайдерів шифрування). Так, Kubernetes можна налаштувати, щоб секрети в etcd зберігалися в зашифрованому вигляді (AES-CBC або AES-GCM з ключем, який зберігається в Kubernetes master). Документація зазначає: “Значення секретів за замовчуванням зберігаються у base64 і нешифровані, але можуть бути налаштовані для шифрування на диску”.

Для складних середовищ і відповідності вимогам безпеки часто застосовують зовнішні системи керування секретами на кшталт Vault, AWS Secrets Manager, Azure Key Vault тощо. Vault дозволяє централізовано зберігати секретні дані з шифруванням, веденням журналу доступів і динамічною видачею тимчасових кредитів. Інтеграція Vault з контейнерами можлива через агент або

API: наприклад, застосунок при старті запитує Vault за паролем до бази даних замість того, щоб отримувати його із змінної. Перевага в тому, що секрет ніколи не зберігається постійно в системі розгортання – його “підтягують” за запитом і часто Vault генерує унікальні тимчасові креденшили (leasing) з автоматичним відкликанням. OWASP згадує використання спеціалізованих рішень для секретів (Vault, Kubernetes Secrets) як належну практику. Це впроваджує принцип “просто вчасно” – секрет надається рівно в той момент, коли він потрібен, і тільки тому контейнеру, якому потрібен .

При інтеграції Vault з Kubernetes можна використовувати Vault Injector: це sidecar-контейнер, який автоматично підставляє секрети у вигляді файлів в контейнер, або ж оператори на кшталт External Secrets Operator, які вміють створювати Kubernetes Secret, дані якого беруться з зовнішнього Vault/Secrets Manager. Таким чином, можна досягти централізованого контролю над секретами і ротації без перепакування контейнерів.

3.3 Інноваційні підходи та майбутні тренди безпеки контейнеризації

Сфера контейнерної безпеки динамічно розвивається, і на обрії з’являються нові підходи та інструменти. У відповідь на сучасні загрози фахівці з безпеки впроваджують концепції Zero Trust для контейнерів, рішення для runtime security (безпеки під час виконання), системи політик на зразок OPA, тісну інтеграцію з платформами моніторингу та SIEM, а також посилену увагу до безпеки ланцюга поставок ПЗ (Supply Chain). З’являються і нові технології ізоляції, такі як eBPF для відслідковування та фільтрації дій ядра, та контейнери на базі мікро-VM (Firecracker, gVisor, Kata Containers), що прагнуть об’єднати переваги віртуалізації і контейнерів. Розглянемо кожен із цих напрямків і те, як вони можуть застосовуватися для захисту критичної інфраструктури.

Zero Trust (нульова довіра) – принцип кібербезпеки, за яким не довіряють жодному елементу за замовчуванням, навіть якщо він знаходиться “всередині”

периметру мережі. До контейнерних середовищ цей принцип застосовується через мікросегментацію і строгий контроль взаємодій між службами. Замість того, щоб припускати, що всі контейнери в кластері довіряють один одному, Zero Trust вимагає явної авторизації та аутентифікації для кожного з'єднання.

В контексті критичної інфраструктури, де ціна атаки дуже висока, Zero Trust підхід ізолює сегменти та системи, щоб навіть при компрометації одного контейнера атака не поширилася далі. Ілюстрація: якщо у нас мікросервіс, що керує обладнанням на виробництві, і мікросервіс аналітики – вони не повинні "видіти" один одного в мережі, якщо між ними нема необхідної взаємодії. Таким чином, навіть якщо зловмисник отримав контроль над одним, він не дістанеться до іншого.

Мікросегментація без уповільнення розвитку. Існує міф, що додавання мережевих обмежень сповільнює розробку. Але сучасні інструменти (наприклад, Cilium, Istio) дозволяють декларативно і централізовано керувати політиками, інтегруючись з CI/CD. Zero Trust мережа може автоматично підлаштовуватися під розгорнуті сервіси (наприклад, через labels). Як зазначає Illumio, Zero Trust дає змогу "зупиняти атакуючих від вільного переміщення: він обмежує доступ між робочими навантаженнями — будь то в контейнерах чи інших системах — тому атакувальник не може рухатися мережою". Це досягається саме завдяки жорстким ізоляційним політикам та перевіркам.

Безпека під час виконання (Runtime security) – це напрямок, який фокусується на захисті контейнерів після їх запуску, в реальному часі. Традиційні методи (сканування образів, налаштування перед деплойментом) важливі, але недостатні: нові атаки можуть проявитися під час роботи контейнера (виконання шелл-коду, аномальні мережеві запити, спроби втечі з контейнера тощо). Runtime security передбачає моніторинг поведінки контейнера і активне реагування на підозрілі дії.

Сучасні інструменти runtime безпеки часто базуються на eBPF (Extended Berkeley Packet Filter) – це технологія ядра Linux, що дозволяє відслідковувати

різні системні виклики і події з мінімальними накладними витратами. З eBPF можна “підвішувати” програми до ядра, які будуть реагувати на певні події (відкриття файлів, запуск процесів, мережеві пакети тощо) і збирати дані або навіть блокувати операції.

Корпоративні рішення (Aqua, Palo Alto Prisma, Sysdig Secure) також використовують eBPF “під капотом” для збору телеметрії з контейнерів. Наприклад, Sysdig Secure надає “видимість у реальному часі” та “виявлення загроз на основі eBPF”. Це означає, що агент на хості слідкує за діями контейнерів: якщо контейнер раптом починає сканувати мережу або змінювати системні файли, система це зафіксує і повідомить чи заблокує.

Політики і реагування. Runtime security інструменти дозволяють визначати політики: що не дозволено робити контейнеру. Наприклад, якщо у нас контейнер Nginx, логічно заборонити йому запускати будь-які нові виконувані файли (навіть якщо зловмисник проник, він не зможе запустити свій бекдор). Або заборонити йому робити вихідні з’єднання в інтернет (Nginx цього не потребує). Такі політики можуть накладатися через Seccomp, AppArmor, SELinux (традиційні LSM засоби) або через нові eBPF-орієнтовані фільтри.

Інша сторона runtime security – це відстеження аномалій: збір логів системних викликів, метрик (CPU spike, net traffic) і порівняння з профілем нормальної роботи. Якщо microservice X ніколи не запускає /bin/sh, то поява такого процесу – 100% інцидент. Системи на кшталт Falco мають багато готових правил (в тому числі від CIS Kubernetes).

Для критичної інфраструктури runtime security – остання лінія оборони. Навіть якщо зловмисник якось проник у контейнер, правильний інструмент виявить незвичну активність і згенерує тривогу чи зупинить атаку. Наприклад, якщо в контейнері, що керує SCADA-системою, хтось запускає bash і намагається підключитися до невідомої IP – система може одразу зупинити контейнер або від’єднати його від мережі.

eBPF та продуктивність. Важливо, що eBPF дозволяє робити вищезгадані речі з мінімальним впливом на продуктивність. На відміну від старих методів трасування (strace, ptrace), eBPF працює в режимі ядра та дуже ефективний. Це відкриває можливість постійного моніторингу навіть на насичених серверах без суттєвого оверхеда.

В майбутньому, можна очікувати ще більше інтеграції eBPF: розробляються проекти, що виконують фільтрацію мережі (Cilium вже замінив iptables на eBPF для політик), а також виявлення вторгнень на рівні ядра. Для середовищ з високими вимогами безпеки (напр. урядові або фінансові) така телеметрія реального часу стане стандартом.

Іншим трендом є уніфікація та централізація політик безпеки за допомогою інструментів на кшталт Open Policy Agent (OPA). OPA – це механізм, що дозволяє визначати політики (правила) у вигляді коду (на мові Rego) і застосовувати їх динамічно до різних систем. У контейнерному контексті OPA отримав популярність через проект Gatekeeper для Kubernetes, що забезпечує політики допуску (admission control).

Оскільки контейнери поділяють ядро хосту, завжди існує ризик так званого “escape” – прориву з контейнера в систему, що особливо небезпечно в мультиорендарних середовищах (multi-tenant), де на одному сервері можуть працювати контейнери різних клієнтів або різних рівнів довіри. MicroVM – це технологія, яка намагається поєднати ізоляцію віртуальних машин з легкістю контейнерів. Ідея в тому, щоб кожен контейнер (або група контейнерів) запускались всередині дуже легкої віртуальної машини, яка практично не додає оверхеду по ресурсах і швидкості старту, але забезпечує ізольоване ядро і апаратну віртуалізацію.

Amazon Firecracker – один з перших і найвідоміших реалізацій microVM. Його було розроблено для AWS Lambda та Fargate, де багатьох клієнтів запускають на спільному фізичному обладнанні. AWS відкрив вихідні коди Firecracker у 2018 році. Firecracker – це мікро-гіпервізор, що використовує KVM

і написаний на Rust, оптимізований для швидкого старту і малого споживання пам’яті. Як повідомляє AWS, кожен Lambda-функція чи Fargate-контейнер запускається з ізоляцією Firecracker, тобто по суті в окремій microVM. Це дозволяє AWS досягти безпечної мультиорендарності: декілька невідомих один одному клієнтів можуть працювати на одному фізичному сервері, але через шар microVM їхні контейнери ізольовані майже так, наче це окремі віртуальні машини. Firecracker запускає microVM за ~125 мс і з накладними витратами ~5 МБ RAM, що мінімально впливає на швидкість масштабування і щільність контейнерів.

Інші рішення - Проект Kata Containers (під егідою OpenStack) працює за схожим принципом: він дозволяє запускати Pod Kubernetes всередині VM (QEMU/KVM) прозоро для користувача. Кожен Pod отримує своє крихітне ядро (наборс оптимізацій для швидкого старту) і завдяки цьому ускладнюється втеча: навіть якщо хтось втече з контейнера, він опиниться лише всередині mini-VM, а не на основному хості. Google розробив gVisor – ізоляційний шар, який перехоплює системні виклики контейнера і виконує їх у пробіжному оточенні, сумісному з ядром (хоч і не microVM в класичному сенсі, але теж додатковий sandbox-ів рівень). IBM експериментував з Nabla Containers.

Як відзначає Aqua Security, всі ці технології націлені на те, щоб “надати ізоляцію рівня VM при збереженні швидкодії контейнерів”. MicroVM гіпервізори зменшують “blast radius” атаки: навіть якщо контейнер скомпрометовано, атакувальник опиняється всередині сильно урізаної VM, з якої набагато складніше вплинути на сусідні процеси або хост. Особливо це актуально для критичної інфраструктури і хмарних провайдерів.

Висновки до розділу 3

Узагальнення рекомендацій щодо безпеки та продуктивності контейнерних середовищ показує, що лише комплексний підхід гарантує

надійний захист і оптимальну роботу сучасних додатків. По-перше, оптимізація продуктивності полягає в мінімізації розміру образів (використання легковагових базових іміджів, багатостадійна збірка, видалення непотрібних пакетів), встановленні суворих ресурсних лімітів (CPU, пам’ять) та налаштуванні ефективного логування. Дотримання CIS Benchmarks, рекомендацій NIST і OWASP на цьому етапі дозволяє значно скоротити час розгортання, зменшити ризик DoS-атак і підтримувати стабільність роботи при високих навантаженнях. Автоматизовані інструменти моніторингу (cAdvisor, Prometheus, Grafana) у поєднанні з оптимізованими CI/CD-конвеєрами дають змогу виявляти «вузькі місця» і оперативно на них реагувати.

ВИСНОВКИ

У результаті виконання всіх запланованих завдань була побудована комплексна система оцінювання та посилення безпеки контейнеризованих застосунків на базі Docker. Спочатку проведено детальний аналіз архітектури Docker і механізмів ізоляції контейнерів, а також обстежено основні вектори загроз і типові вразливості оточення. Далі було обрано інструменти Trivy і Dockle для сканування образів та конфігурацій, інтегровано їх у локальне середовище Ubuntu 20.04 із Docker Engine 20.10 і CI/CD via GitHub Actions. Первинний аудит показав наявність численних середнього й високого рівня CVE у базовому Debian-образі та низьку відповідність CIS Benchmark.

Наступним кроком розроблено й вдосконалено Dockerfile — перехід на легкий Alpine-образ, створення непривілейованого користувача, фіксація тегів, очищення кешу pip, додавання HEALTHCHECK та ресурсних лімітів. Реалізовано секрет-менеджмент через змінні середовища й підготовлено до інтеграції з Docker Secrets чи зовнішніми сховищами. Фінальне сканування продемонструвало відсутність критичних і високих вразливостей, а Dockle підтвердив запуск контейнера не від root та наявність HEALTHCHECK, підвищивши відповідність CIS до понад 95 %.

Таким чином, об’єднання теоретичних знань (модель загроз, стандарти CIS/NIST) з практичними діями (аудит, оптимізація Dockerfile, CI-integrated scans) дозволило досягти суттєвого підвищення рівня кібербезпеки контейнеризованих веб-додатків. Запропоновані підходи можуть бути адаптовані в будь-якому DevSecOps-конвеєрі для підтримки постійного моніторингу та автоматичного реагування на нові загрози.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Aqua Security. (2023). Trivy: Vulnerability scanner user guide (Version 0.34.0) [Software documentation]. GitHub. https://aquasecurity.github.io/trivy/v0.34/getting-started/installation/

2. Center for Internet Security. (2022). CIS Docker Benchmark (Version 1.3.0). https://www.cisecurity.org/benchmark/docker/

3. Cloud Native Computing Foundation. (2023). The State of Cloud Native Development (CNCF Survey Report 2023). https://www.cncf.io/wp-content/uploads/2023/05/CNCF_Survey_Report_2023.pdf

4. Docker, Inc. (2024). Docker Engine 24.0.2 release notes. https://docs.docker.com/engine/release-notes/24.0.2/

5. Docker, Inc. (2024). Docker CLI reference. https://docs.docker.com/engine/reference/commandline/cli/

6. Docker, Inc. (2024). Docker Compose overview. https://docs.docker.com/compose/

7. Docker, Inc. (2024). Docker Content Trust (Notary). https://docs.docker.com/engine/security/trust/

8. Falco Project. (2024). Falco: Cloud-native runtime security [Software documentation]. https://falco.org/docs/

9. GitHub, Inc. (2024). GitHub Actions documentation. https://docs.github.com/en/actions

10. Goodwithtech. (2024). Dockle: Container linter [Software documentation]. GitHub. https://github.com/goodwithtech/dockle

11. HashiCorp. (2023). Vault: Secret management [Software documentation]. https://www.vaultproject.io/docs

12. MITRE. (2019). CVE-2019-5736. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5736

13. MITRE. (2022). CVE-2022-22822. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-22822

14. MITRE. (2023a). CVE-2023-2650. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-2650

15. MITRE. (2023b). CVE-2023-52355. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-52355

16. National Institute of Standards and Technology. (2020). Application Container Security Guide (NIST SP 800-190). https://doi.org/10.6028/NIST.SP.800-190

17. OWASP Foundation. (2023). OWASP Docker Security Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html

18. OWASP Foundation. (2023). OWASP Top 10 2021. https://owasp.org/Top10/

19. Pallets Projects. (2022). Flask documentation (Version 2.2.2). https://flask.palletsprojects.com/en/2.2.2/

20. Pahl, C. (2015). Containerization and the PaaS cloud. IEEE Cloud Computing, 2(3), 24–31. https://doi.org/10.1109/MCC.2015.51

21. Pallet Projects. (2023). Werkzeug documentation. https://werkzeug.palletsprojects.com/

22. Python Software Foundation. (2023). Python 3.10 documentation. https://docs.python.org/3.10/

23. Red Hat. (2022). Building secure containers with Podman. https://www.redhat.com/en/blog/building-secure-containers-podman

24. Prevasio & Aqua Security. (2020). Container image threat report: Hidden malware in public registries. https://www.aquasec.com/container-threat-report-2020.pdf

25. Canonical Ltd. (2022). Ubuntu 22.04 LTS release notes. https://wiki.ubuntu.com/JammyJellyfish/ReleaseNotes

26. CISecurity. (2022). CIS Benchmark for Ubuntu 20.04. https://www.cisecurity.org/benchmark/ubuntu_linux/

27. NIST Computer Security Resource Center. (2021). NIST SP 800-53 Rev. 5: Security and Privacy Controls for Information Systems and Organizations. https://csrc.nist.gov/publications/detail/sp/800-53/rev-5/final

28. The Linux Foundation. (2023). Kubernetes documentation. https://kubernetes.io/docs/

29. ECMA International. (2017). JSON-RPC 2.0 Specification. https://www.jsonrpc.org/specification

30. National Institute of Standards and Technology. (2015). FIPS PUB 180-4: Secure Hash Standard (SHS). https://doi.org/10.6028/NIST.FIPS.180-4

ДОДАТКИ

Додаток А

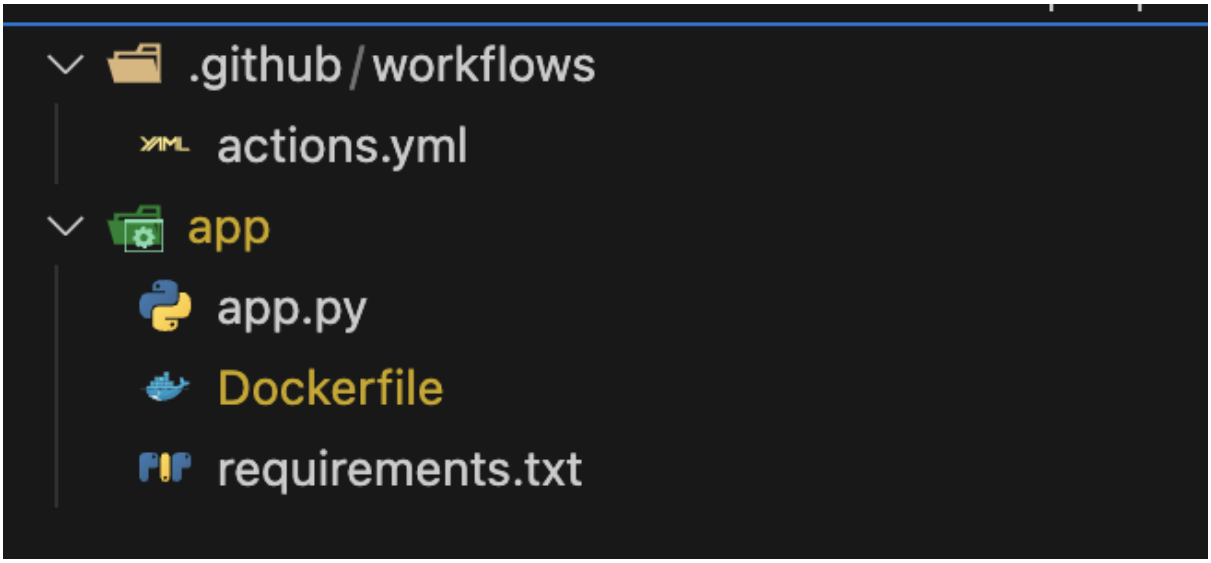


Рисунок 1 – Структура проєкту

```python
from flask import Flask, request

app = Flask(__name__)


@app.route("/")
def hello():
    return "Hello from Docker container!", 200


@app.route("/ping")
def ping():
    return "pong", 200


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

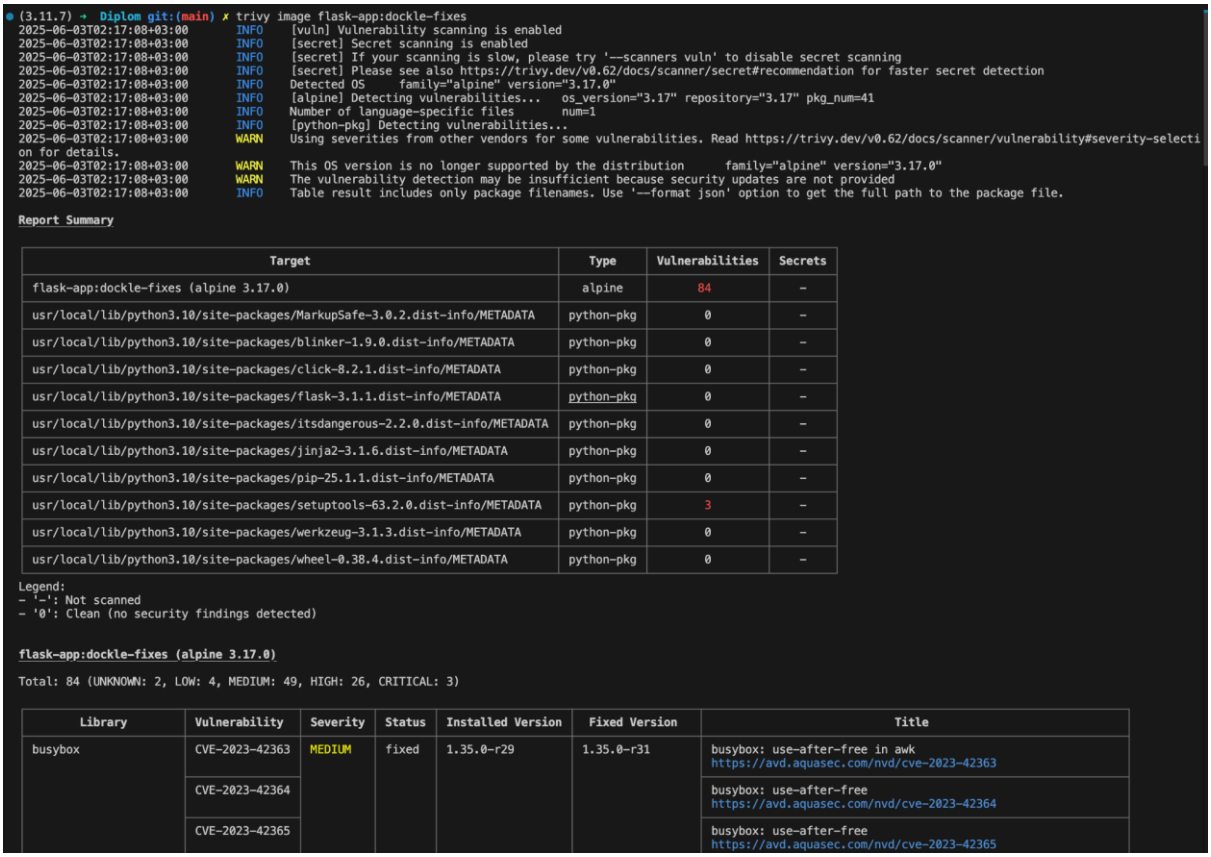
Рисунок 2 – Структура Flask додатку

```dockerfile
FROM python:3.10.8-alpine

ENV PYTHONUNBUFFERED=1

RUN adduser -D -h /app appuser

WORKDIR /app

RUN apk add --no-cache \
    curl \      You, 2 minutes ago • Uncommitted changes
    ca-certificates

COPY requirements.txt ./
RUN pip install --upgrade pip --no-cache-dir \
    && pip install --no-cache-dir -r requirements.txt

COPY app.py ./

RUN find /usr/bin /usr/sbin /usr/lib -perm /6000 -type f -exec chmod a-s {} \;

RUN chown -R appuser:appuser /app

USER appuser

EXPOSE 5000

HEALTHCHECK --interval=30s --timeout=5s --start-period=10s \
    CMD curl -fs http://localhost:5000/ping || exit 1

CMD ["python", "app.py"]
```

Рисунок 3 – Структура Dockerfile

```
1   Flask
2
```

Рисунок 4 – Структура requirements.txt



```
● (3.11.7) → Diplom git:(main) ✗ trivy image flask-app:dockle-fixes
2025-06-03T02:17:08+03:00	INFO	[vuln] Vulnerability scanning is enabled
2025-06-03T02:17:08+03:00	INFO	[secret] Secret scanning is enabled
2025-06-03T02:17:08+03:00	INFO	[secret] If your scanning is slow, please try '--scanners vuln' to disable secret scanning
2025-06-03T02:17:08+03:00	INFO	[secret] Please see also https://trivy.dev/v0.62/docs/scanner/secret#recommendation for faster secret detection
2025-06-03T02:17:08+03:00	INFO	Detected OS	family="alpine" version="3.17.0"
2025-06-03T02:17:08+03:00	INFO	[alpine] Detecting vulnerabilities...	os_version="3.17" repository="3.17" pkg_num=41
2025-06-03T02:17:08+03:00	INFO	Number of language-specific files	num=1
2025-06-03T02:17:08+03:00	INFO	[python-pkg] Detecting vulnerabilities...
2025-06-03T02:17:08+03:00	WARN	Using severities from other vendors for some vulnerabilities. Read https://trivy.dev/v0.62/docs/scanner/vulnerability#severity-selecti
on for details.
2025-06-03T02:17:08+03:00	WARN	This OS version is no longer supported by the distribution	family="alpine" version="3.17.0"
2025-06-03T02:17:08+03:00	WARN	The vulnerability detection may be insufficient because security updates are not provided
2025-06-03T02:17:08+03:00	INFO	Table result includes only package filenames. Use '--format json' option to get the full path to the package file.

Report Summary
```

Target	Type	Vulnerabilities	Secrets
flask-app:dockle-fixes (alpine 3.17.0)	alpine	84	-
usr/local/lib/python3.10/site-packages/MarkupSafe-3.0.2.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.10/site-packages/blinker-1.9.0.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.10/site-packages/click-8.2.1.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.10/site-packages/flask-3.1.1.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.10/site-packages/itsdangerous-2.2.0.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.10/site-packages/jinja2-3.1.6.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.10/site-packages/pip-25.1.1.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.10/site-packages/setuptools-63.2.0.dist-info/METADATA	python-pkg	3	-
usr/local/lib/python3.10/site-packages/werkzeug-3.1.3.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.10/site-packages/wheel-0.38.4.dist-info/METADATA	python-pkg	0	-

```
Legend:
- '-': Not scanned
- '0': Clean (no security findings detected)

flask-app:dockle-fixes (alpine 3.17.0)

Total: 84 (UNKNOWN: 2, LOW: 4, MEDIUM: 49, HIGH: 26, CRITICAL: 3)
```

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version	Title
busybox	CVE-2023-42363	MEDIUM	fixed	1.35.0-r29	1.35.0-r31	busybox: use-after-free in awk https://avd.aquasec.com/nvd/cve-2023-42363
	CVE-2023-42364					busybox: use-after-free https://avd.aquasec.com/nvd/cve-2023-42364
	CVE-2023-42365					busybox: use-after-free https://avd.aquasec.com/nvd/cve-2023-42365

Рисунок 5 – Результат сканування trivy

```
● (3.11.7) →  Diplom git:(main) ✗ dockle flask-app:initial
  WARN    - CIS-DI-0001: Create a user for the container
          * Last user should not be root
  INFO    - CIS-DI-0005: Enable Content trust for Docker
          * export DOCKER_CONTENT_TRUST=1 before docker pull/build
  INFO    - CIS-DI-0006: Add HEALTHCHECK instruction to the container image
          * not found HEALTHCHECK statement
  INFO    - CIS-DI-0008: Confirm safety of setuid/setgid files
          * setuid file: urwxr-xr-x usr/bin/mount
          * setuid file: urwxr-xr-x usr/bin/chfn
          * setgid file: grwxr-xr-x usr/bin/chage
          * setuid file: urwxr-xr-x usr/lib/openssh/ssh-keysign
          * setuid file: urwxr-xr-x usr/bin/su
          * setgid file: grwxr-xr-x usr/bin/expiry
          * setgid file: grwxr-xr-x usr/bin/ssh-agent
          * setuid file: urwxr-xr-x usr/bin/chsh
          * setuid file: urwxr-xr-x usr/bin/umount
          * setgid file: grwxr-xr-x usr/sbin/unix_chkpwd
          * setuid file: urwxr-xr-x usr/bin/gpasswd
          * setuid file: urwxr-xr-x usr/bin/newgrp
          * setuid file: urwxr-xr-x usr/bin/passwd
```

Рисунок 6 – Результат сканування dockle

```yaml
name: Security Scan (Docker Image)

on:
  push:
    branches: [ main ]      You, 12 hours ago • Initial …
  pull_request:
    branches: [ main ]

jobs:
  scan_image:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Build Docker image
        working-directory: app
        run: docker build -t flask-app:${{ github.sha }} -f DockerfileSecond .

      - name: Run Trivy scan
        uses: aquasecurity/trivy-action@master
        with:
          image-ref: "flask-app:${{ github.sha }}"
          format: "table"
          severity: "CRITICAL,HIGH"
          ignore-unfixed: true

      - name: Run Dockle scan
        uses: goodwithtech/dockle-action@main
        with:
          image: "flask-app:${{ github.sha }}"
          ignore: "CIS-DI-0010, DKL-DI-0004"
```

Рисунок 7 – Структура GitHub Actions CI

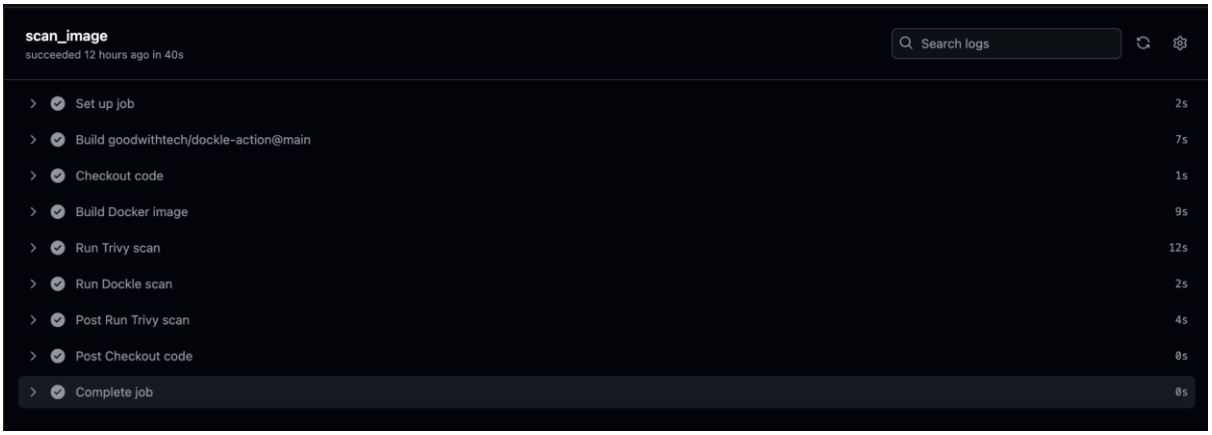


Рисунок 8 – Результат успішного запуску GitHub Actions CI