Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Факультет комп'ютерних наук
Спеціальність 125 «Кібербезпека»

Освітня програма «Кібербезпека»

«Допущено до захисту»
В. о. завідувача кафедрою БІСТ
Мелкозьорова О. М.
____
«   »   2024 р.

Пояснювальна записка

до кваліфікаційної роботи бакалавра

на тему: «Підвищення кібербезпеки в розподілених системах за допомогою передових методів контейнеризації»

оцінка «   »

Голова ЕК
Лемешко О. В. ______________

Керівник ст. викладач Шеханін К.Ю.

Рецензент доцент Карась І. В.

Виконавець: студент групи КБ-42

Гаєвський А.В.

Харків – 2024

РЕФЕРАТ

Дипломна робота складається з 52 сторінок, включачи 16 ілюстрацій, 13 лістингів, і містить 20 джерел у списку посилань.

Метою роботи є дослідження методів підвищення кібербезпеки в розподілених системах за допомогою передових методів контейнеризації. У роботі використовувалися методи аналізу та синтезу, порівняння, моделювання та експериментального дослідження.

У результаті дослідження було розроблено комплекс методів захисту розподілених систем, включаючи сканування контейнерних образів, налаштування політик безпеки в Kubernetes, а також впровадження системи моніторингу та аудиту. Новизна роботи полягає у застосуванні комплексного підходу до забезпечення безпеки контейнеризованих додатків.

Результати можуть бути використані в різних галузях, де застосовуються розподілені системи та технології контейнеризації, включаючи фінансові установи, медичні заклади, транспортні системи та інші критично важливі інфраструктури.

Робота є значущою у контексті зростаючих кіберзагроз і важливості захисту даних у розподілених системах. Основними висновками є необхідність впровадження автоматизованих засобів для сканування, моніторингу та управління безпекою контейнеризованих додатків. Подальші дослідження можуть бути спрямовані на розробку нових методів захисту від специфічних атак на контейнеризовані додатки, а також на інтеграцію засобів штучного інтелекту для прогнозування та виявлення загроз у реальному часі.

Ключові слова: DOCKER, KUBERNETES, АУДИТ, ВРАЗЛИВОСТІ, КІБЕРБЕЗПЕКА, КОНТЕЙНЕРИЗАЦІЯ, МОНІТОРИНГ, ОРКЕСТРАЦІЯ, РОЗПОДІЛЕНІ СИСТЕМИ

ABSTRACT

The thesis consists of 52 pages, includes 16 illustrations, 13 listings, and contains 20 sources in the bibliography.

The purpose of the work is to study methods of improving cybersecurity in distributed systems using advanced containerization methods. The methods of analysis and synthesis, comparison, modeling, and experimental research were used in the work.

As a result of the study, a set of methods for protecting distributed systems was developed, including scanning container images, setting up security policies in Kubernetes, and implementing a monitoring and auditing system. The novelty of the work lies in the application of an integrated approach to ensuring the security of containerized applications.

The results can be used in various industries where distributed systems and containerization technologies are used, including financial institutions, medical facilities, transportation systems, and other critical infrastructures.

The work is significant in the context of growing cyber threats and the importance of data protection in distributed systems. The main conclusions are the need to implement automated tools for scanning, monitoring, and managing the security of containerized applications. Further research can be aimed at developing new methods of protection against specific attacks on containerized applications, as well as at integrating artificial intelligence tools for real-time threat prediction and detection.

Keywords: AUDIT, CONTAINERIZATION, CYBERSECURITY, DISTRIBUTED SYSTEMS, DOCKER, KUBERNETES, MONITORING, ORCHESTRATION, VULNERABILITIES

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ 5
ВСТУП 6
1 АРХІТЕКТУРНІ СТИЛІ ПРОЄКТУВАННЯ РОЗПОДІЛЕНИХ СИСТЕМ 7
1.1 Архітектурні стилі 7
1.2 Контейнеризація та віртуальні машини 13
1.3 Оркестратори 15
2 КІБЕРБЕЗПЕКА В РОЗПОДІЛЕНИХ СИСТЕМАХ З ВИКОРИСТАННЯМ КОНТЕЙНЕРИЗАЦІЇ 18
2.1 Вступ до кібербезпеки в контейнеризованих системах 18
2.2 Порівняння безпеки традиційних монолітних та кластеризованих додатків 20
2.3 Основні ризики контейнеризації 22
2.4 Кібербезпека в оркестрації контейнерів 24
2.5 Передові методи захисту контейнеризованих додатків 26
2.6 Контроль доступу та автентифікація в контейнеризованих додатках 28
2.7 Моніторинг та аудит безпеки в контейнеризованих системах 30
3 РЕАЛІЗАЦІЯ ПІДВИЩЕННЯ РІВНЯ КІБЕРБЕЗПЕКИ В РОЗПОДІЛЕНОМУ ЗАСТОСУНКУ 32
3.1 Створення розподіленого додатка 32
3.2 Комунікація між компонентами застосунку 33
3.3 Автентифікація та авторизація 35
3.4 Контейнеризація 36
3.5 Процес кластеризації з використанням Kubernetes 42
3.6 Процес кластеризації з використанням Kubernetes 44
3.7 KubeScape та його роль в підвищенні кібербезпеки розподілених застосунків 47
ВИСНОВКИ 51
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 53

ПЕРЕЛІК СКОРОЧЕНЬ

API – application programming interface

CI/CD – continuous integration/continuous deployment

CIS – Center for Internet Security

CUD – Create Update Delete

EFK – Elasticsearch, Fluentd, Kibana

ESB – enterprise service bus

gRPC – Google remote procedure call

HTTP – hypertext transfer protocol

JSON – JavaScript Object Notation

JWT – JSON Web Token

K8s – Kubernetes

Kube – Kubernetes

MiTM – man in the middle

NSA – National Security Agency

OWASP – Open Web Application Security Project

RBAC – Role-Based Access Control

SMS – Short Message Servise

SOA – Service Oriented Architecture

БД – база даних

ВМ – віртуальна машина

НСД – несанкціонований доступ

ОС – операційна система

ПЗ – програмне забезпечення

ВСТУП

У сучасних умовах інформаційні системи все більше покладаються на розподілені структури та технології контейнеризації. Відповідно, питання кібербезпеки стає надзвичайно актуальним для забезпечення надійної та безпечної роботи таких систем. Ця робота присвячена дослідженню методів підвищення кібербезпеки в розподілених системах за допомогою передових методів контейнеризації.

З розвитком технологій і зростанням обсягів даних, розподілені системи стають основою для багатьох галузей. Проте, зростання кіберзагроз вимагає постійного вдосконалення методів захисту. Використання контейнеризації пропонує нові можливості для підвищення безпеки, але також створює нові виклики, які необхідно враховувати та вирішувати.

Метою цієї роботи є дослідження та розробка методів підвищення кібербезпеки в розподілених системах за допомогою передових методів контейнеризації. Особлива увага приділяється аналізу поточних загроз та розробці рекомендацій щодо їх усунення.

Завдання роботи полягає у дослідженні сучасних методів контейнеризації та їх впливу на кібербезпеку, а також в аналізі ризиків і вразливостей, пов'язаних із використанням контейнерів. З практичної точки зору необхідно розробити комплексний підхід до забезпечення безпеки контейнеризованих додатків, а також впровадити та протестувати розроблені методи у реальних умовах.

Гіпотеза цієї роботи полягає в тому, що комплексне застосування передових методів контейнеризації може значно підвищити рівень кібербезпеки розподілених систем. Такий підхід включає використання сканування контейнерних образів, налаштування політик безпеки в Kubernetes, а також впровадження системи моніторингу та аудиту.

1 АРХІТЕКТУРНІ СТИЛІ ПРОЄКТУВАННЯ РОЗПОДІЛЕНИХ СИСТЕМ

1.1 Архітектурні стилі

З розвитком прикладного програмного забезпечення (ПЗ), підходи для проєктування систем також пройшли значну еволюцію. Виділяють три основні архітектурні стилі проєктування розподілених систем: моноліти, сервісно-орієнтована архітектура, мікросервіси (рис. 1.1) [5].

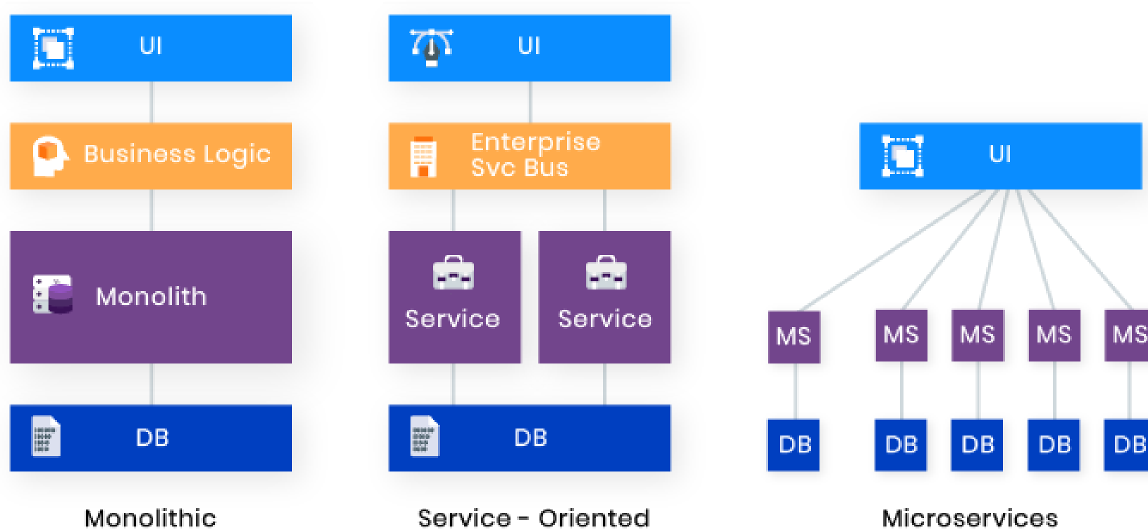

Рисунок 1.1 – Архітектурні стилі проєктування

Кожен із цих стилів був розроблений на певному етапі еволюції технологій для вирішення різних проблем, що виникали під час створення складних систем [2]. Всі вони продовжують використовуватися не тільки через підтримку застарілого успадкованого ПЗ, але й завдяки їх перевагам, які часто визначають вибір архітектурного підходу на стадії планування. Перед початком огляду архітектурних стилів проєктування, варто ознайомитися із поняттям та особливостями розподілених систем.

Розподілена система – це сукупність комп'ютерних програм, які використовують обчислювальні ресурси декількох окремих обчислювальних вузлів для досягнення спільної мети. Також відома як розподілені обчислення або розподілені бази даних, вона покладається на окремі вузли для зв'язку та синхронізації через спільну мережу. Ці вузли зазвичай представляють окремі

фізичні апаратні пристрої, але можуть також представляти окремі програмні процеси або інші рекурсивні інкапсульовані системи. Розподілені системи спрямовані на усунення вузьких місць або центральних точок відмови в системі.

Розподілені обчислювальні системи мають низку характеристик. По-перше, вони забезпечують спільне використання ресурсів, що означає можливість розподіленої системи використовувати апаратне та програмне забезпечення або дані разом. По-друге, такі системи дозволяють одночасну обробку, коли декілька машин можуть одночасно обробляти одну і ту ж функцію. Масштабованість також є важливою характеристикою, що дозволяє розширювати обчислювальні потужності за необхідності, додаючи додаткові машини. Також вони забезпечують прозорість і спрощують виявлення помилок, що дозволяє вузлу отримати доступ до інших вузлів системи та запобігати виявленням помилок.

Моноліт - традиційна система для проєктування систем, оскільки представляє найпростіший підхід з погляду планування, реалізації, когнітивного навантаження розробників та розгортання. Він є самостійною, повністю автономною програмною одиницею, яка містить всю кодову базу бізнес-логіки та інфраструктурної логіки. Це означає, що всі компоненти, такі як взаємодії з базою даних і зовнішні API від третіх сторін, наприклад платіжні системи, SMS та електронні листи, інтегровані в один нероздільний блок. Схема архітектури типового моноліту представлена на рис. 1.2. [3].

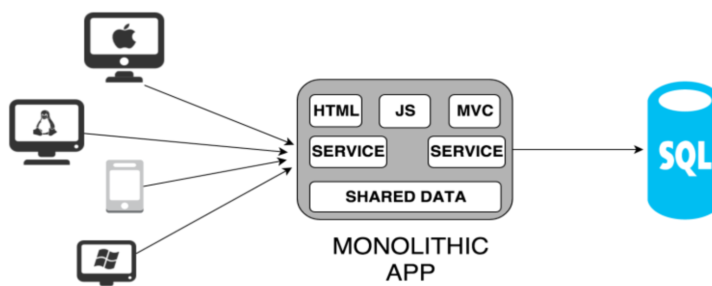

Рисунок 1.2 - Схема монолітного додатка

До переваг монолітів можна віднести простоту і швидкість розробки, оскільки всі частини програми розробляються в єдиній кодовій базі, що спрощує

розуміння і керування проєктом, особливо на початкових стадіях роботи. Моноліти також вирізняються простотою розгортання, оскільки зазвичай розгортаються у вигляді одного виконуваного пакета, що спрощує введення додатка в експлуатацію. Крім того, вони мають низьку затримку комунікації та високу швидкодію, оскільки компоненти моноліта взаємодіють без мережевих затримок, що може забезпечувати вищу продуктивність.

Недоліки монолітів включають кілька значних проблем. По-перше, вони мають дуже погану горизонтальну масштабованість. Часто масштабування потребують лише окремі компоненти, але монолітна архітектура вимагає масштабування всього додатку. У деяких випадках це може бути неможливо без виокремлення модулів, які зберігають спільний стан. Ще однією важливою проблемою є складність внесення змін. Зі збільшенням кількості бізнес-правил зростає розмір кодової бази та кількість тісних зв’язків у ній, що ускладнює внесення змін. Зміна однієї частини логіки може викликати неочікувані зміни в інших частинах системи, що провокує некоректну роботу всього додатку. Моноліти також складні для підтримки та розробки великими командами. Зі зростанням проекту збільшується і розмір команди, яка працює над програмним забезпеченням. У великих командах внесення змін ускладнюється, оскільки постійно виникають конфлікти в кодовій базі, і значна частина часу витрачається на їх вирішення. Складність також полягає в контролі інфраструктурних залежностей. У багатокористувацьких середовищах розгортання та функціонування моноліту можуть потребувати спеціалізованих утиліт і конфігурацій, що ускладнює процес. Монолітні додатки вимагають великого "релізного вікна" і значних ресурсів для планування релізів. Через свою масивність, введення в експлуатацію може займати багато часу. Останнє, проте не менш важливе вони мають низьку відмовостійкість. Відмова одного сервісу призводить до недоступності всієї системи, що робить монолітні архітектури менш надійними у порівнянні з іншими підходами.

Загалом, монолітна архітектура відмінно підходить для малих застосунків, особливо на ранніх етапах проєкту, коли бізнес-вимоги та деталі реалізації ще не повністю визначені. Також, моноліти можуть бути вдалим варіантом у

специфічних випадках, де критично важливі висока швидкість і ефективність програми

Першим великим кроком до розподілених систем став сервісно-орієнтований підхід (рис. 1.3) [4], який передбачає розбиття кодової бази на менші й більш гнучкі модулі. Цей підхід спрямований на подолання ряду проблем, характерних для монолітних систем, зокрема перевикористання логіки, розширення можливостей для горизонтального масштабування, підвищення відмовостійкості шляхом слабких зв'язків між компонентами, а також дозволяє кільком командам розробляти продукт паралельно.

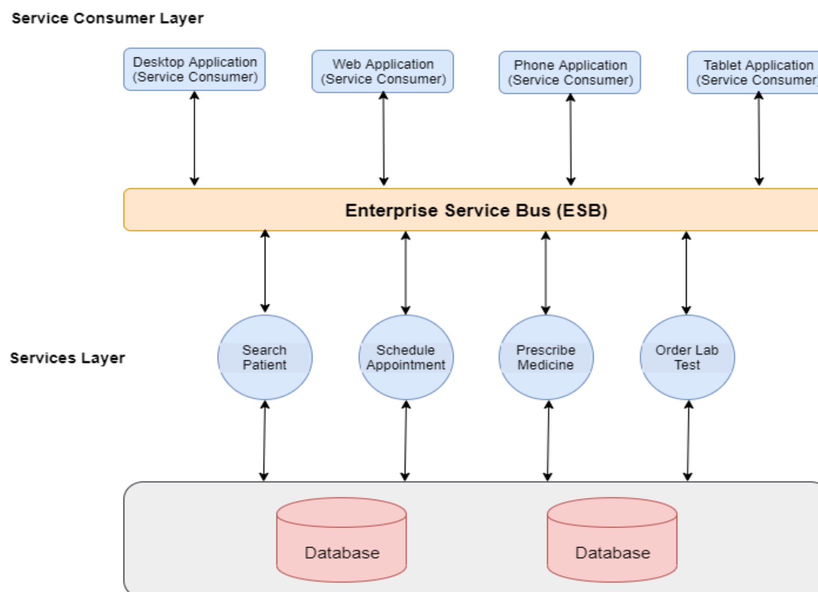

Рисунок 1.3 – Архітектура сервісно-орієнтованого додатку

При проектуванні системи за стилем SOA логіка розбивається за призначенням. Наприклад, один сервіс виконує операції доступу до бази даних (БД), інший містить бізнес-правила, третій виконує конкретні операції. Головним каналом комунікації між сервісами виступає ESB, який використовується для пересилання повідомлень. Шляхом використання ESB, який виступає в ролі медіатора, досягається слабка зв'язність між модулями, оскільки сервісу А не потрібно знати нічого про те, як зробити запит до сервісу Б. Кожен сервіс оголошує чіткі контракти повідомлень, які надалі використовуються викликаючими сервісами й передаються через ESB, така

комунікація називається асинхронною. Як правило, використовується Request-Response підхід, але є й інші варіації SOA, наприклад з використанням Event-Driven підходу. Однак, SOA має свої недоліки, а саме: не досконалі можливості для масштабування через використання таких спільних ресурсів як БД; зростання складності системи через співзалежність сервісів та їх ресурсів; єдина точка відмови - ESB.

В результаті експериментів над SOA, наступним, більш вдосконаленим етапом у розвитку архітектури розподілених систем стали мікросервіс (рис. 1.4) [5]. Вони допомогли усунути багато недоліків, притаманних SOA. Мікросервісний підхід складається з дуже маленьких і повністю незалежних модулів, кожен з яких виконує одну конкретну задачу, визначену обмеженим контекстом конкретного сервісу, до якого можуть відноситися платежі, користувачі або продукти. Свій обмежений контекст, а звідси слабка зв’язність, вимагає від сервісів мати свою БД. Таким чином, робиться вибір в сторону дуплікації даних, замість використання спільних ресурсів.

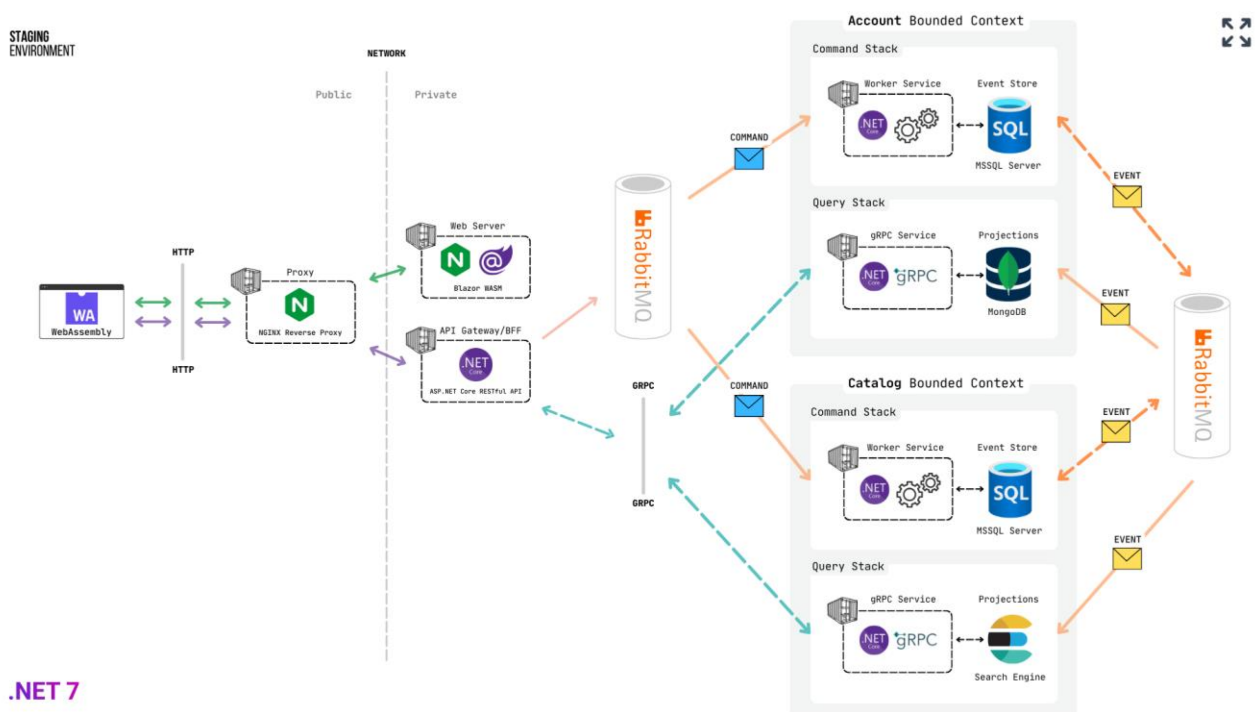

Рисунок 1.4 - Приклад мікросервісної архітектури

Комунікація між мікросервісами може бути як синхронною для деяких операцій, так і асинхронною. Кожен мікросервіс має API, яке зазвичай

реалізоване на протоколах HTTP або gRPC. Аналогічно SOA, в мікросервісах використовується брокер повідомлень (аналог EBS) для асинхронного надсилання команд або для публікації подій. Основними перевагами мікросервісів є:

- Прості сервіси та автономність команд розробників. Команди відповідають за визначений набір сервісів, які надають API для інших команд. Таким чином, команди можуть незалежно використовувати будь-які технології, змінювати деталі бізнес-логіки, впроваджувати різні сховища даних.
- Швидке розгортання. Оскільки компоненти мікросервісної архітектури, відносно інших підходів, це маленькі виконувані пакети, то розгортання виконується набагато швидше.
- Масштабування. Горизонтальне та вертикальне масштабування реалізовується набагато простіше, через можливість індивідуального підходу як до самих сервісів, так і до сховищ даних які вони використовують.

Такі якості мікросервісів роблять їх ідеальним вибором для використання в розподілених системах, в тому числі з використанням хмарних рішень.

З усім тим, при всіх перевегах мікросервісів, є і недоліки:

- Складність управління залежностями та версіонування. Коли кількість сервісів збільшується, то стає дедалі складніше відстежувати залежності та встигати оновлювати залежні сервіси при оновленні використовуваного API.
- Комунікація та ізоляція призводять до дублювання даних, функціоналу та великого мережевого трафіку.
- Вартість розгортання та підтримки більша, через потребу в більшій кількості ресурсів і використання складніших технологічних рішень.
- Розподіл системи на множину сервісів створює більше вікно можливостей для появи вразливостей та точок входу потенційних зловмисників.

Цікавим випадком став перехід від мікросервісів до моноліта в компанії Amazon. Компоненти додатка для моніторингу відео та аудіо перенесли в один

процес, що спростило та оптимізувало передачу даних та оркестрацію, які були найдорожчими операціями. Таким чином компанія знизила витрати на додаток на 90% [5].

1.2 Контейнеризація та віртуальні машини

Контейнеризація - це форма віртуалізації, коли програми запускаються в ізольованих користувацьких просторах, які називаються контейнерами, використовуючи при цьому одну спільну операційну систему (ОС) [6].

Однією з переваг контейнеризації є те, що контейнер - це, по суті, повністю упаковане і портативне обчислювальне середовище. Все, що потрібно для запуску програми - двійкові файли, бібліотеки, залежності та конфігураційні файли (у вигляді файлів маніфесту розкладання) - інкапсульовано та ізольовано в контейнері. Таким чином, контейнеризований додаток може бути протестований та розгорнутий як єдине ціле на хостовій ОС.

Сам контейнер абстрагований від хостової ОС і має лише обмежений доступ до базових ресурсів - подібно до легкої віртуальної машини (ВМ). В результаті, контейнерний додаток можна запускати на різних типах інфраструктури - на «голому металі», у віртуальних машинах і в хмарі - без необхідності рефакторингу для кожного середовища. Перевагою контейнеризації також є масштабованість, саме швидкість розгортання, дозволяюча створювати нові контейнери для короткострокових завдань. З погляду додатків, створення екземпляра образу (створення контейнера) схоже на створення екземпляра процесу, наприклад, сервісу або вебпрограми. Однак для надійності, коли запускається кілька екземплярів одного образу, зазвичай потрібно, щоб кожен контейнер (екземпляр образу) працював на різних хост-серверах або віртуальних машинах у різних доменах збоїв.

Основна мета образу полягає в тому, що він робить оточення (залежності) однаковим для різних розгортань. Це означає, що налаштований сервіс на одній машині, буде гарантовано запущений зі збереженням середовища на іншій. Контейнери ізолюють додатки один від одного у спільній ОС. Контейнеризовані програми запускаються на контейнер-хості, який в свою чергу працює під

управлінням ОС (Linux або Windows). Тому контейнери займають значно менше місця, ніж образи ВМ (рис.1.5) [7] [8].

Що стосується віртуальних машин, то на хост-сервері є три базові рівні (знизу вгору): інфраструктура, хост-ОС та гіпервізор, а зверху кожна віртуальна машина має власну ОС та всі необхідні бібліотеки. На противагу, для контейнерів, хост-сервер має лише інфраструктуру та ОС, а на вершині - контейнерний движок, який забезпечує ізоляцію контейнерів, але надає їм доступ до базових сервісів ОС.

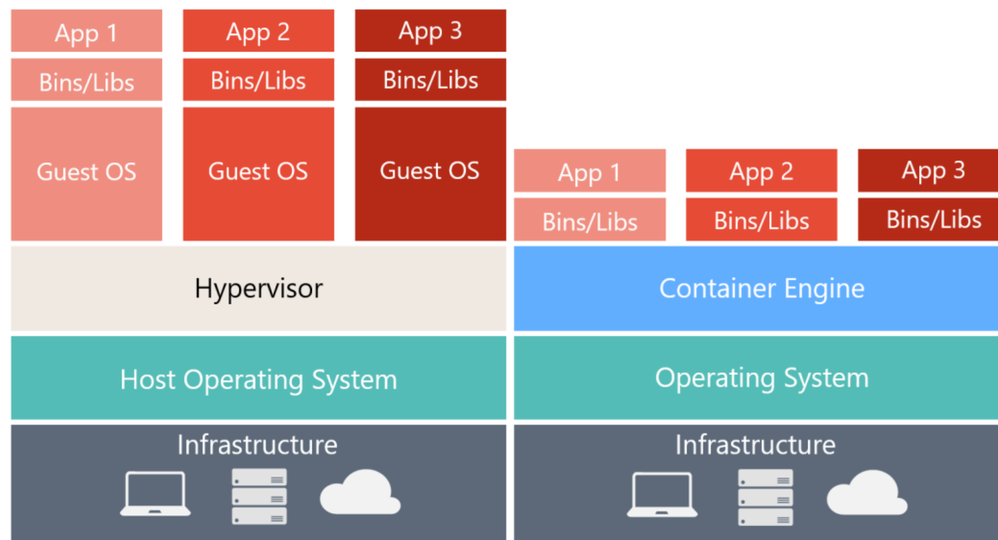

Рисунок 1.5 - Порівняння складових компонентів віртуальних машин та технологій контейнеризації

У зв'язку з тим, що контейнерний движок працює поверх ОС, існує 2 види контейнерів: для Linux та Windows. Як побічний ефект роботи контейнерів на одному ядрі, такий підхід забезпечує меншу ізоляцію, ніж у випадку з віртуальними машинами.

Хостові ОС також поділяються на 2 основі групи: універсальні ОС, такі як Linux Mint, Ubuntu та Windows Server, які можуть використовуватися для запуску багатьох типів програм, і до них може бути додано функціональність, специфічну для контейнерів; контейнерні ОС, такі як CoreOs Container Linux, Project Atomic та Google Container-Optimized OS, які являють собою мінімалістичні ОС, явно призначені для запуску лише контейнерів [8]. Зазвичай вони не постачаються з менеджерами пакетів, мають лише підмножину основних

інструментів адміністрування й активно знеохочують спроби запуску програм поза контейнерами. Часто ОС, орієнтовані на контейнери, використовують файлову систему, призначену лише для читання, щоб зменшити ймовірність того, що зловмисник зможе зберегти дані в ній, а також використовують спрощений процес оновлення, оскільки не беруть до уваги сумісність перед встановлених додатків через їх відсутність.

Оскільки контейнери вимагають набагато менше ресурсів (наприклад, їм не потрібна повна ОС), їх легко розгортати й вони швидко запускаються. Це дозволяє мати вищу щільність з погляду ресурсів системи, тобто запускати більше сервісів на одній апаратній одиниці, тим самим знижуючи витрати.

1.3 Оркестратори

З початком використанням контейнерів, розробники та системні адміністратори зустріли ряд викликів при управлінні та масштабуванні контейнеризованими додатками. У відповідь на ринку з’явилися інструменти, як Kubernetes, Docker Swarm, і Mesos, розроблені на потребу більш ефективного керування численними контейнерами, які можуть швидко збільшуватися в рамках великих систем.

Одним з основних завдань оркестрації контейнерів є спрощення процесу розгортання, масштабування та управління життєвим циклом додатків. За допомогою автоматизації, оркестратори здатні забезпечувати швидке відновлення послуг після збоїв, ефективно розподіляти ресурси між контейнерами, та забезпечувати безперервну доставку оновлень до додатків без зупинки послуг. Це також включає автоматизацію мережевих з'єднань між контейнерами, управління балансуванням навантаження, та забезпечення безпеки в межах цих комунікацій (рис. 1.6) [9].

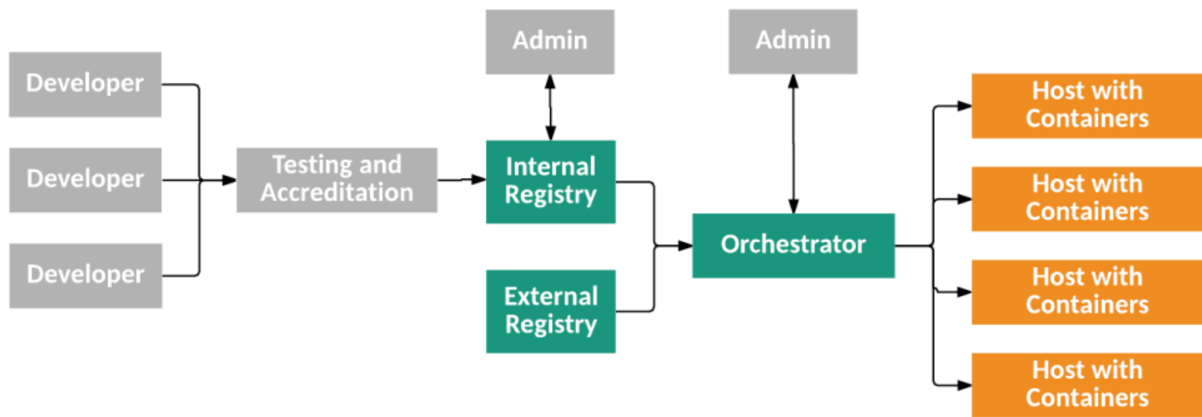

Рисунок 1.6 - Рівні та компоненти архітектури контейнерної технології

Оркестратори дозволяють мати централізований контроль над складними системами з великою кількістю контейнерів. Контролюються всі аспекти життєвого цикла контейнеризованого додатку. Такий підхід значно спрощує моніторинг станів системи, логування подій та дозволяє налаштовувати систематизований підхід для реакції на події та інциденти.

Хоча розподіл системи на багато сервісів вносить додаткові виклики, засоби оркестрації включають розширені можливості для захисту даних та забезпечення доступності послуг. Ці засоби сприяють високому рівню відмовостійкості та безперервності роботи, знижуючи ризики збоїв та покращуючи загальне управління безпекою.

Використання оркестрації контейнерів може значно підвищити аптайм систем залежно від конфігурації та використання відповідних практик управління та моніторингу, що свідчить про високу ефективність впровадження оркестрації в управління контейнеризованими додатками та важливість її інтеграції в загальні стратегії цифрової трансформації компаній при використанні технологій розподілених систем.

У цьому розділі було досліджено архітектурні стилі для проєктування розподілених систем. Було розглянуто три основні архітектурні стилі: моноліти, SOA та мікросервіси. Монолітна архітектура, яка є традиційною системою для проєктування, вирізняється простотою і швидкістю розробки, а також простотою розгортання; вона передбачає єдину кодову базу, яка включає всю бізнес-логіку та інфраструктурну логіку. SOA, яка стала першим великим кроком до

розподілених систем, передбачає розбиття кодової бази на менші й більш гнучкі модулі, що дозволяє долати ряд проблем, характерних для монолітних систем. Вона використовує ESB для пересилання повідомлень між сервісами. Мікросервісна архітектура, яка стала наступним етапом у розвитку розподілених систем, складається з маленьких, повністю незалежних модулів, кожен з яких виконує одну конкретну задачу.

Отже, мікросервіси забезпечують високий рівень гнучкості та дозволяють кожному модулю мати власну базу даних.

2 КІБЕРБЕЗПЕКА В РОЗПОДІЛЕНИХ СИСТЕМАХ З ВИКОРИСТАННЯМ КОНТЕЙНЕРИЗАЦІЇ

2.1 Вступ до кібербезпеки в контейнеризованих системах

Кібербезпека є основою сучасних інформаційних систем, адже зростання кількості кіберзагроз вимагає постійної уваги та вдосконалення захисних заходів [10]. Основні принципи кібербезпеки включають конфіденційність, цілісність та доступність. Конфіденційність означає, що доступ до даних мають лише авторизовані користувачі та системи, що запобігає несанкціонованому доступу (НСД) та витоку інформації. Цілісність передбачає захист даних від несанкціонованих змін або видалення, що забезпечує їхню точність та повноту. Доступність гарантує, що системи та дані будуть доступні для авторизованих користувачів у потрібний час, що важливо для безперервної роботи бізнесу.

Окрім основних принципів, важливими аспектами кібербезпеки є автентифікація, авторизація та журналювання. Автентифікація – це процес перевірки особи або системи, що здійснює запит на доступ. Вона забезпечує, що тільки легітимні користувачі можуть отримати доступ до системи. Авторизація визначає, які ресурси й дії дозволені для кожного користувача на основі їхніх прав доступу, що обмежує можливості несанкціонованих дій. Журналювання передбачає ведення записів про всі події та дії в системі, що дозволяє проводити аудит та виявляти аномалії, які можуть вказувати на спроби зловживання або атаки.

Контейнеризація стає все більш популярною завдяки своїй здатності забезпечувати ізольоване середовище для запуску додатків. Проте використання контейнерів супроводжується специфічними викликами безпеки. Одним із головних ризиків є безпека контейнерних образів. Якщо образи не проходять регулярне сканування та оновлення, вони можуть містити відомі вразливості, які зловмисники можуть використовувати для атак. Ще однією проблемою є мережева ізоляція. Контейнери можуть взаємодіяти один з одним через мережу, що створює потенційні ризики для мережевої безпеки, зокрема можливість атак

типу «людина посередині». Управління секретами, такими як паролі та ключі, також стає складнішим в контейнеризованих середовищах. Для цього необхідно використовувати безпечні методи управління секретами, щоб запобігти їх витоку або НСД.

Оркестрація контейнерів додає ще один рівень складності в управлінні безпекою [11]. Використання оркестраторів, таких як Kubernetes, вимагає забезпечення безпеки як на рівні окремих контейнерів, так і на рівні всього кластера. Це включає налаштування політик доступу, контроль над ресурсами та захист від внутрішніх загроз. Відсутність належного моніторингу та журналювання також може ускладнити виявлення та розслідування інцидентів безпеки, що робить необхідним впровадження ефективних інструментів для цих цілей.

Контейнеризація має ряд значних переваг з погляду безпеки. По-перше, вона забезпечує ізоляцію середовищ, що мінімізує ризики взаємного впливу додатків один на одного. Це дозволяє забезпечити вищу стабільність та безпеку системи в цілому. По-друге, контейнеризовані додатки є портативними, що дозволяє їх легко переносити між різними середовищами, зберігаючи при цьому їхню цілісність та конфігурацію. Це значно спрощує процеси тестування та розгортання, забезпечуючи консистентність на всіх етапах. Іншою перевагою є швидкість розгортання контейнеризованих додатків.

Завдяки контейнерам можна швидко розгортати та масштабувати додатки, що сприяє оперативному реагуванню на загрози та потреби бізнесу. Контейнери також зменшують поверхню атаки, оскільки використовують мінімалістичні образи, що містять лише необхідні компоненти, зменшуючи кількість вразливих точок.

Однак, контейнеризація має і свої недоліки. Основним ризиком є вразливості контейнерних образів. Якщо образи не проходять регулярне сканування, вони можуть містити відомі вразливості, які можуть бути використані зловмисниками. Мережева взаємодія між контейнерами також створює ризики, оскільки можливі атаки типу «людина посередині» можуть порушити цілісність та конфіденційність даних, що передаються.

Ускладнення управління безпекою є значним недоліком контейнеризації. Використання великої кількості контейнерів та оркестраторів вимагає належного управління політиками безпеки та конфігураціями. Це включає контроль доступу, налаштування мережевих політик та забезпечення відповідності стандартам безпеки. Хоча контейнери забезпечують певний рівень ізоляції, вони менш ізольовані порівняно з віртуальними машинами. Це може призвести до експлуатації вразливостей на рівні ядра операційної системи, що підвищує ризик успішних атак.

Отже, контейнеризація надає значні переваги для розвитку та масштабування додатків, але вимагає уважного підходу до управління безпекою. Необхідно впроваджувати сучасні методи захисту, щоб ефективно мінімізувати ризики та забезпечити надійну роботу контейнеризованих систем.

2.2 Порівняння безпеки традиційних монолітних та кластеризованих додатків

Монолітні додатки являють собою традиційний підхід до розробки програмного забезпечення, де всі компоненти системи інтегровані в один єдиний блок. Така архітектура має свої переваги з погляду безпеки. По-перше, монолітні додатки часто простіше для захисту, оскільки всі компоненти працюють в одному процесі та можуть бути легко контролюватися через єдину точку входу. Це спрощує налаштування і контроль доступу, а також дозволяє централізовано керувати політиками безпеки. Крім того, монолітна архітектура зазвичай має нижчу затримку комунікацій між компонентами, що зменшує ризики атак на мережевому рівні.

Відсутність мережевих комунікацій між внутрішніми компонентами знижує можливість атак типу «людина посередині» (MitM) та інших мережевих вразливостей. Однак, монолітні додатки мають і значні недоліки. Зокрема, їх велика кодова база і тісна інтеграція компонентів можуть стати серйозною проблемою у разі виявлення вразливості. Якщо зловмисник отримує доступ до однієї частини додатка, він може потенційно отримати доступ до всього додатку через тісно пов'язані компоненти. Ще одним недоліком монолітних додатків є

складність масштабування. У випадку необхідності масштабування окремих компонентів додатка, монолітна архітектура змушує масштабувати всю систему, що може бути неефективним і витратним. Це також підвищує ризики виникнення проблем з безпекою через більшу кількість точок потенційного зламу при збільшенні кількості екземплярів додатка.

Мікросервісна архітектура, на відміну від монолітної, складається з невеликих, незалежних сервісів, кожен з яких виконує окрему функцію. Цей підхід надає значні переваги з погляду безпеки. Однією з головних переваг є ізоляція сервісів, яка зменшує вплив потенційних вразливостей. Якщо один сервіс компрометовано, інші сервіси залишаються захищеними, що підвищує загальну відмовостійкість системи. Мікросервіси також дозволяють легко застосовувати різні політики безпеки до різних сервісів, що робить систему більш гнучкою й адаптивною до змінних вимог безпеки. Кожен сервіс може використовувати окремі методи автентифікації та авторизації, що підвищує рівень контролю доступу.

Проте мікросервісна архітектура має і свої ризики. Основний з них - це складність управління безпекою великої кількості незалежних сервісів. Необхідність забезпечення безпеки на рівні кожного окремого сервісу може створювати додаткові виклики для системних адміністраторів. Крім того, мікросервісна архітектура покладається на інтенсивну мережеву взаємодію між сервісами. Це збільшує ризики мережевих атак, таких як DoS або MitM. Для мінімізації цих ризиків необхідно впроваджувати надійні методи шифрування даних та безпечні протоколи зв'язку. Вразливості в оркестраторах контейнерів, таких як Kubernetes, також можуть стати точкою входу для атак, тому необхідно ретельно налаштовувати й постійно моніторити ці інструменти.

Розподілені системи та контейнеризація значно впливають на загальний рівень безпеки додатків. Розподіленість дозволяє підвищити відмовостійкість та масштабованість системи, але водночас ускладнює управління безпекою. Контейнеризація забезпечує ізоляцію додатків, що зменшує ризики взаємного впливу між ними, але потребує використання передових методів захисту для збереження безпеки на всіх рівнях. Контейнери дозволяють швидко створювати

та розгортати нові екземпляри додатків, що спрощує процес оновлення та виправлення вразливостей. Проте, якщо контейнерні образи не проходять регулярного сканування, вони можуть містити вразливості, які будуть репліковані на всі екземпляри. Це вимагає впровадження автоматизованих процесів сканування і перевірки контейнерних образів на наявність вразливостей перед їх використанням у продакшені. Розподілені системи також покладаються на мережеву комунікацію, що створює додаткові ризики. Використання надійних протоколів шифрування і безпечних методів аутентифікації та авторизації є критично важливим для забезпечення безпеки. Оркестратори контейнерів, такі як Kubernetes, надають потужні інструменти для управління та моніторингу контейнеризованих додатків, але потребують належного налаштування і постійного моніторингу для виявлення та усунення потенційних загроз.

Загалом, контейнеризація та оркестрація можуть значно підвищити безпеку додатків за умови належного управління ризиками та впровадження передових методів захисту. Це вимагає інтегрованого підходу до безпеки, що включає регулярне сканування, моніторинг, контроль доступу та використання надійних методів шифрування даних.

2.3 Основні ризики контейнеризації

Контейнеризація стала ключовою технологією в сучасній розробці програмного забезпечення (ПЗ), але її впровадження супроводжується низкою специфічних ризиків. OWASP визначив десять основних ризиків [12], пов'язаних із використанням Docker, які можуть суттєво вплинути на безпеку контейнерних середовищ.

Першим ризиком є неналежне управління образами контейнерів. Це включає використання небезпечних або неоновлених образів, які можуть містити відомі вразливості. Використання ненадійних джерел для завантаження образів підвищує ризик впровадження шкідливого коду в систему. Регулярне сканування образів за допомогою інструментів, таких як Clair або Trivy, є критично важливим для забезпечення безпеки.

Другим важливим ризиком є відсутність обмежень ресурсів для контейнерів. Якщо не встановлювати обмеження на використання ресурсів (CPU, пам'ять), один контейнер може споживати всі доступні ресурси, що призведе до відмови в обслуговуванні (DoS) інших контейнерів. Це підкреслює необхідність налаштування обмежень ресурсів за допомогою відповідних параметрів у конфігураціях Docker.

Третій ризик стосується неналежного управління секретами. Зберігання конфіденційних даних, таких як паролі або ключі, без належного захисту в контейнерах може призвести до їх витоку. Використання інструментів для управління секретами, таких як Docker Secrets або HashiCorp Vault, є необхідним для захисту цієї інформації.

Четвертий ризик пов'язаний із надмірними правами контейнерів. Запуск контейнерів з привілеями (привілейовані контейнери) підвищує ризик експлуатації вразливостей на рівні хостової ОС. Необхідно уникати використання привілейованих контейнерів і налаштовувати мінімально необхідні права для виконання завдань.

Окрім цих основних ризиків, існують і інші, такі як неналежне налаштування мережевих політик, використання застарілих версій Docker та недоліки в системах моніторингу і журналювання, які також можуть суттєво впливати на безпеку контейнерних середовищ.

Контейнерні образи є основою для створення контейнерів, і їх безпека має вирішальне значення. Вразливості в контейнерних образах можуть мати серйозний вплив на безпеку всього додатку. Наприклад, якщо базовий образ містить вразливість, усі контейнери, створені на його основі, будуть також вразливими. Це підвищує ризик експлуатації вразливостей і компрометації системи.

Однією з основних причин вразливостей контейнерних образів є використання застарілих або ненадійних компонентів. Це може включати використання старих версій ОС, бібліотек або інших залежностей, які мають відомі вразливості. Регулярне оновлення образів і використання перевірених джерел для завантаження базових образів є важливими заходами для

забезпечення безпеки. Вразливості в контейнерних образах також можуть виникати через неправильно налаштовані конфігурації. Наприклад, неправильно налаштовані дозволи файлів або використання небезпечних налаштувань за замовчуванням можуть створити можливості для атак. Регулярні перевірки та сканування конфігурацій образів за допомогою інструментів, таких як Docker Bench for Security, можуть допомогти виявити й усунути ці проблеми.

Використання контейнерів передбачає спільне використання ресурсів хостової ОС, що створює додаткові ризики. Одним з основних ризиків є використання загальних ядерних просторів. Контейнери, що працюють на одному ядрі, можуть потенційно впливати один на одного через спільне використання системних ресурсів. Це може призвести до витоку інформації або ескалації привілеїв.

Використання механізмів, таких як Control Groups та Namespaces, допомагає ізолювати контейнери та зменшити ці ризики. Іншим важливим аспектом є безпека файлових систем. Контейнери часто мають доступ до файлової системи хостової ОС або спільних томів, що створює ризики для цілісності та конфіденційності даних. Використання тільки необхідних прав доступу до файлових систем, а також шифрування даних є критичними для забезпечення безпеки. Інструменти для управління доступом, такі як AppArmor або SELinux, можуть додатково захистити файлові системи від НСД.

2.4 Кібербезпека в оркестрації контейнерів

Оркестрація контейнерів є ключовим аспектом сучасних розподілених систем, забезпечуючи автоматизацію розгортання, масштабування та управління контейнеризованими додатками. Однак, оркестратори контейнерів, такі як Kubernetes, мають свої специфічні проблеми безпеки, які можуть вплинути на загальну безпеку системи. OWASP Kubernetes Top 10 [13] визначає основні ризики, пов'язані з безпекою Kubernetes.

Однією з основних проблем є неналежне управління автентифікацією та авторизацією. Якщо автентифікація та авторизація налаштовані неправильно, зловмисники можуть отримати доступ до кластера і виконувати несанкціоновані

дії. Важливо використовувати надійні механізми автентифікації, такі як TLS сертифікати, і налаштовувати ролі та права доступу відповідно до принципу найменших привілеїв. Іншою важливою проблемою є неналежне управління конфігураціями. Це може включати використання небезпечних налаштувань за замовчуванням або відсутність політик безпеки. Використання інструментів для автоматизованої перевірки конфігурацій, таких як kube-bench, допомагає виявляти й виправляти небезпечні налаштування. Також значною проблемою є відсутність моніторингу та журналювання. Без належного моніторингу і журналювання важко виявити та розслідувати інциденти безпеки. Впровадження систем для збору і аналізу журналів, таких як Elasticsearch, Fluentd і Kibana, є необхідним для забезпечення належного рівня безпеки.

Kubernetes, як оркестратор контейнерів, є ціллю для різних типів атак, і розуміння цих атак допомагає вжити ефективних заходів для їхнього запобігання. Один з поширених типів атак – це атаки на API сервер. Зловмисники можуть намагатися отримати доступ до API сервера Kubernetes, щоб виконувати несанкціоновані дії. Для запобігання таким атакам необхідно використовувати аутентифікацію на основі сертифікатів, налаштовувати обмеження на IP-адреси, з яких можна звертатися до API, та регулярно оновлювати ПЗ.

Ще однією поширеною атакою є атака через привілейовані контейнери. Запуск привілейованих контейнерів може дати зловмисникам можливість отримати доступ до ресурсів хостової ОС. Щоб уникнути цього, слід обмежити використання привілейованих контейнерів і застосовувати політики безпеки для контейнерів, такі як Pod Security Policies. Також часто зустрічаються атаки на мережеві взаємодії між подами, зловмисники можуть спробувати перехопити або змінити трафік між ними. Використання мережевих політик для контролю трафіку між подами й забезпечення шифрування мережевого трафіку допомагає захистити систему від таких атак.

Для забезпечення безпеки в Kubernetes важливо налаштовувати та дотримуватися політик безпеки. Однією з основних політик є Role-Based Access Control (RBAC). RBAC дозволяє керувати доступом до ресурсів на основі ролей, що привласнюються користувачам та сервісам. Це дозволяє обмежити права

доступу і забезпечити, щоб кожен користувач та сервіс мали лише необхідні для їхніх завдань права. Network Policies дозволяють контролювати трафік між подами й службами. Ці політики визначають, які з'єднання дозволені або заборонені, що допомагає захистити систему від НСД та атак типу MitM. Налаштування мережевих політик включає визначення дозволених вхідних та вихідних з'єднань для кожного пода, що значно підвищує рівень безпеки мережі. Pod Security Policies дозволяють визначати політики безпеки для подів. Ці політики контролюють такі аспекти, як використання привілейованих контейнерів, доступ до хостової файлової системи, можливість ескалації привілеїв та інші налаштування безпеки. Використання PSP допомагає забезпечити, що поди відповідають вимогам безпеки й знижує ризик експлуатації вразливостей.

2.5 Передові методи захисту контейнеризованих додатків

Одним з найважливіших аспектів забезпечення безпеки контейнеризованих додатків є використання безпечних контейнерних образів. Контейнерні образи повинні бути регулярно скановані на наявність вразливостей і підписані, щоб гарантувати їхню цілісність та походження. Сканування образів дозволяє виявляти відомі вразливості у бібліотеках та залежностях, що використовуються в образі. Це можна здійснювати за допомогою таких інструментів, як Clair, Trivy та Anchore.

Підписання образів допомагає забезпечити, що образи не були змінені після їх створення. Підписані образи дозволяють перевіряти їхню цілісність перед розгортанням, що забезпечує додатковий рівень захисту від впровадження шкідливого коду. Для цього можна використовувати інструменти, такі як Docker Content Trust або Notary. Застосування політики щодо використання лише підписаних і сканованих образів є критичним для підтримки високого рівня безпеки в контейнеризованих середовищах. Цей підхід гарантує, що всі контейнери, які розгортаються в системі, пройшли належний аудит і відповідають стандартам безпеки.

Для ефективного забезпечення безпеки контейнеризованих додатків існує ряд спеціалізованих інструментів, які допомагають виявляти та усувати вразливості, контролювати доступ та моніторити активність контейнерів [14]. Одним з таких інструментів є Aqua Security, який надає комплексне рішення для захисту контейнеризованих середовищ. Aqua дозволяє сканувати образи, контролювати поведінку контейнерів у реальному часі, а також впроваджувати політики безпеки для контейнерів і оркестраторів. Іншим важливим інструментом є Sysdig, який забезпечує моніторинг і безпеку контейнерів та кластерів Kubernetes. Sysdig пропонує розширені можливості для аналізу активності контейнерів, виявлення аномалій та реагування на інциденти. Цей інструмент також підтримує інтеграцію з іншими системами безпеки та дозволяє налаштовувати політики відповідно до потреб організації. Kube-bench є ще одним важливим інструментом, який допомагає перевіряти конфігурації Kubernetes на відповідність найкращим практикам безпеки. Kube-bench виконує автоматизований аудит конфігурацій кластера і надає рекомендації щодо виправлення виявлених проблем. Це дозволяє забезпечити належний рівень безпеки на рівні оркестратора контейнерів.

Принципи Zero Trust (нульової довіри) стають все більш популярними в сучасних підходах до кібербезпеки. Основна ідея Zero Trust полягає в тому, що нікому і нічому не можна довіряти за замовчуванням, навіть якщо вони знаходяться всередині корпоративної мережі. Це особливо актуально для контейнеризованих середовищ, де велика кількість компонентів взаємодіють один з одним. Застосування принципів Zero Trust у контейнеризованих середовищах включає кілька ключових аспектів. Перш за все, необхідно забезпечити надійну автентифікацію та авторизацію для всіх компонентів системи. Це може включати використання багатофакторної автентифікації та політик доступу на основі ролей.

Друге, необхідно сегментувати мережу, щоб обмежити доступ між різними компонентами системи. Використання мережевих політик в Kubernetes дозволяє контролювати трафік між подами й службами, що зменшує ризик поширення атак всередині кластеру. Третє, важливо впроваджувати постійний моніторинг і

аудит активності в системі. Інструменти для моніторингу, такі як Prometheus та Grafana, допомагають відстежувати стан контейнерів і виявляти аномалії в їхній поведінці. Це дозволяє оперативно реагувати на потенційні загрози та знижувати ризики.

Загалом, застосування передових методів захисту контейнеризованих додатків, використання спеціалізованих інструментів для забезпечення безпеки та впровадження принципів Zero Trust дозволяє значно підвищити рівень безпеки в контейнеризованих середовищах. Це забезпечує надійний захист від сучасних кіберзагроз і гарантує безперебійну роботу додатків

2.6 Контроль доступу та автентифікація в контейнеризованих додатках

У контейнеризованих додатках контроль доступу та автентифікація є критичними аспектами забезпечення безпеки. Одним із найпоширеніших методів автентифікації та авторизації є використання JSON Web Tokens (JWT) та OAuth2.

JWT – це відкритий стандарт для створення токенів доступу, які дозволяють безпечно передавати інформацію між сторонами у вигляді JSON об'єктів. JWT складається з трьох частин: заголовка, тіла та підпису. Заголовок містить інформацію про тип токена й алгоритм підпису, тіло – корисне навантаження з даними про користувача, а підпис гарантує цілісність токена. Використання JWT дозволяє легко перевіряти автентифікацію користувачів без необхідності зберігати сесійні дані на сервері, що особливо зручно для розподілених систем [15].

OAuth2 – це протокол авторизації, який дозволяє додаткам отримувати обмежений доступ до ресурсів користувачів без необхідності передачі паролів. OAuth2 використовує токени доступу, які надаються після успішної автентифікації користувача. Ці токени можна використовувати для доступу до ресурсів, визначених власником ресурсів. OAuth2 є гнучким і дозволяє налаштовувати різні потоки авторизації залежно від потреб додатків [16].

Впровадження JWT та OAuth2 в контейнеризованих додатках забезпечує надійний контроль доступу та автентифікацію користувачів. Це дозволяє зменшити ризики НСД та підвищити загальний рівень безпеки системи [17].

Управління секретами, такими як паролі, ключі шифрування та інші конфіденційні дані, є ще одним важливим аспектом забезпечення безпеки контейнеризованих додатків. Неправильне зберігання секретів може призвести до витоку інформації та компрометації системи. Існують кілька найкращих практик для безпечного управління секретами, які можна використовувати в контейнеризованих середовищах. Kubernetes Secrets – це вбудований механізм Kubernetes для зберігання конфіденційних даних. Kubernetes Secrets дозволяють зберігати та управляти секретами у вигляді об'єктів, які можна передавати контейнерам через середовищні змінні або файли.

Використання Kubernetes Secrets забезпечує ізоляцію конфіденційних даних від контейнерних образів, що знижує ризик їхнього витоку. Однак, необхідно забезпечити належний контроль доступу до секретів, щоб лише авторизовані поди могли їх використовувати.

HashiCorp Vault – це потужний інструмент для управління секретами та динамічної видачі облікових даних. Vault дозволяє зберігати та контролювати доступ до секретів за допомогою політик доступу. Він також підтримує автоматичне оновлення та ротацію секретів, що зменшує ризик компрометації через використання застарілих або скомпрометованих облікових даних. Vault інтегрується з Kubernetes, що дозволяє використовувати його можливості для управління секретами в контейнеризованих середовищах.

Практики безпечного управління секретами включають використання шифрування для зберігання та передачі конфіденційних даних, обмеження доступу до секретів лише авторизованим користувачам та сервісам, а також регулярне оновлення та ротацію секретів. Це допомагає забезпечити належний рівень безпеки й зменшити ризик НСД до конфіденційної інформації.

2.7 Моніторинг та аудит безпеки в контейнеризованих системах

Використання інструментів для моніторингу дозволяє відстежувати стан системи в режимі реального часу, виявляти аномалії та оперативно реагувати на потенційні загрози. Одним з найпопулярніших інструментів для моніторингу є Prometheus. Prometheus збирає метрики з додатків та інфраструктури, зберігає їх у базі даних і дозволяє виконувати запити для аналізу зібраних даних. Prometheus легко інтегрується з Kubernetes і забезпечує високу продуктивність та масштабованість [18].

Grafana використовується для візуалізації даних, зібраних Prometheus, та інших джерел. Використовуючи Grafana, можна створювати інтерактивні дашборди, які надають візуалізацію метрик у реальному часі. Це дозволяє оперативно виявляти проблеми та аналізувати їхні причини. Grafana підтримує налаштування сповіщень, що дозволяє отримувати повідомлення про критичні події через різні канали зв'язку, такі як електронна пошта або месенджери.

Kube-prometheus-stack є комплексним рішенням для моніторингу Kubernetes кластерів, яке об'єднує Prometheus, Grafana, Alertmanager та інші інструменти [19]. Kube-prometheus-stack надає готові до використання конфігурації для збору метрик, візуалізації даних та налаштування сповіщень. Це спрощує процес впровадження моніторингу та забезпечує комплексний підхід до управління безпекою контейнеризованих додатків.

Логи дозволяють відстежувати активність у системі, виявляти аномальні дії та аналізувати події для розслідування інцидентів безпеки. Kubernetes надає потужні можливості для логування та аудиту, які можна використовувати для забезпечення прозорості та контролю над системою. Kubernetes Audit Logging дозволяє записувати всі запити до API сервера, що забезпечує повний журнал подій в кластері. Це включає інформацію про те, хто і коли виконував запити, які ресурси були доступні та які дії були виконані.

Налаштування Kubernetes Audit Logging дозволяє визначити політики логування, які контролюють, які події повинні бути зафіксовані та з яким рівнем деталізації. Для налаштування Kubernetes Audit Logging необхідно створити файл конфігурації аудиту, який визначає правила для логування подій. Цей файл

конфігурації вказується при запуску API сервера. Зібрані логи можуть бути збережені у файлах або передані в системи обробки логів, такі як Elasticsearch, Fluentd та Kibana (EFK), для подальшого аналізу та зберігання.

Інтеграція безпекових інструментів у CI/CD конвеєри дозволяє виявляти вразливості на ранніх етапах розробки та знижує ризик впровадження небезпечного коду в продакшн середовище.

KubeScape є інструментом для сканування безпеки Kubernetes кластерів, який автоматично перевіряє конфігурації кластера на відповідність найкращим практикам безпеки. KubeScape аналізує налаштування кластера, подів, мережевих політик та інших компонентів, надаючи звіти з рекомендаціями щодо усунення виявлених проблем. Інтеграція KubeScape у CI/CD конвеєри дозволяє автоматично перевіряти безпеку конфігурацій перед їх розгортанням.

Trivy є інструментом для сканування контейнерних образів на наявність вразливостей. Trivy перевіряє образи на відомі вразливості в ОС, бібліотеках та залежностях. Інтеграція Trivy у CI/CD конвеєри забезпечує автоматичне сканування образів на кожному етапі розробки, що дозволяє виявляти та виправляти вразливості до розгортання.

3 РЕАЛІЗАЦІЯ ПІДВИЩЕННЯ РІВНЯ КІБЕРБЕЗПЕКИ В РОЗПОДІЛЕНОМУ ЗАСТОСУНКУ

У рамках дипломної роботи було створено розподілений контейнеризований додаток, який складається з двох сервісів, черги повідомлень та бази даних. Він демонструє, як контейнеризація може ізолювати процеси та ресурси між контейнерами, що підвищує загальну безпеку системи.

Застосунок є прикладом інструменту для управління проектами у великих компаніях. Команда розробників може використовувати цей інструмент для відстеження проектів, а адміністратори (менеджери або архітектори) можуть додавати важливі нотатки. Ці нотатки можуть містити коментарі або завдання, які потрібно виконати, і будуть доступні лише адміністраторам. Такий підхід допомагає координувати роботу команди та підвищує ефективність управління проектами.

3.1 Створення розподіленого додатка

Додаток дозволяє зберігати всю інформацію про проект в одному місці й забезпечує доступ до неї всім членам команди.

Застосунок складається з декількох компонентів:

- Сервіс управління проектами (Write Service)
- Сервіс читання проектів (Read Service)
- Черга повідомлень (RabbitMQ)
- База даних (Microsoft SQL Server)

Write Service відповідальний за управління проектами: створення, оновлення та видалення. Сервіс реалізує операції CUD надаючи API для використання клієнтами. Програмний код створений на мові програмування C# з використанням платформи .NET.

Read Service надає можливість користувачам отримувати інформацію про проекти, аналогічним чином надаючи API для операцій читання. Цей сервіс також написаний на мові програмування C# з використанням платформи .NET.

RabbitMQ використовується для асинхронного обміну повідомленнями між сервісами. Черга повідомлень дозволяє сервісу управління проектами відправляти події про створення та оновлення проектів до сервісу читання. Забезпечує надійну доставку повідомлень між сервісами, знижуючи навантаження на кожен окремий сервіс і підвищуючи стійкість системи до збоїв [20].

База даних використовується для зберігання інформації про проекти. Для простоти реалізації, БД взаємодіє з обома сервісами. Зазвичай при такій схемі побудови архітектури використовуються окремі та різні БД для сервісів запису і читання.

3.2 Комунікація між компонентами застосунку

Комунікація між компонентами здійснюється як синхронно, так і асинхронно, що забезпечує гнучкість і надійність роботи системи.

Асинхронна комунікація у даному застосунку реалізована за допомогою черги повідомлень RabbitMQ та моделі Publisher-Subscriber. Сервіс управління проектами публікує події (наприклад, створення або оновлення проекту) в RabbitMQ. Кожен раз, коли створюється новий проект або оновлюються дані існуючого проекту, відповідна подія відправляється до черги повідомлень. Це означає, що основний сервіс не чекає на завершення обробки даних іншими компонентами й може продовжувати обробку інших запитів, що значно підвищує продуктивність. Сервіс читання проектів, своєю чергою, споживає ці події з черги та відповідно оновлює свою базу даних. Цей сервіс працює незалежно від сервісу управління проектами, витягаючи події з черги у своєму власному ритмі й оновлюючи дані в базі даних.

Такий підхід надає декілька важливих переваг з погляду кібербезпеки, надаючи відмовостійкість, масштабованість та ізоляцію компонентів.

Черга повідомлень забезпечує збереження подій, навіть якщо один з сервісів тимчасово не доступний. Наприклад, якщо сервіс читання проектів виходить з ладу, події залишаються у черзі та будуть оброблені, коли сервіс знову стане доступним. Це гарантує, що жодна подія не буде втрачена.

Якщо навантаження на сервіс читання проектів збільшується, можна додати більше екземплярів цього сервісу, які опрацьоватимуть події з черги паралельно, що підвищує загальну продуктивність системи.

Використання черги повідомлень ізолює сервіси один від одного, Якщо один з сервісів буде скомпрометований, він не зможе напряму вплинути на інші сервіси, оскільки взаємодія відбувається через RabbitMQ.

Синхронна комунікація здійснюється через HTTP API. Клієнти взаємодіють із сервісами за допомогою HTTP запитів для виконання операцій створення, оновлення або отримання проектів. Наприклад, при запиті на створення нового проекту клієнт надсилає HTTP запит до сервісу управління проектами, який обробляє запит і повертає відповідь клієнту.

Синхронна комунікація являє собою традиційну, просту і зрозумілу модель взаємодії в клієнт-серверній архітектурі, дозволяючи клієнтам безпосередньо взаємодіяти з сервісами, отримуючи необхідну інформацію або виконуючи операції в реальному часі, що дозволяє їм миттєво дізнатися про успіх або помилку операції. На рис. 3.1 зображена схема взаємодії всіх компонентів додатку, де суцільними лініями позначена синхронна комунікація, а пунктирними - асинхронна.

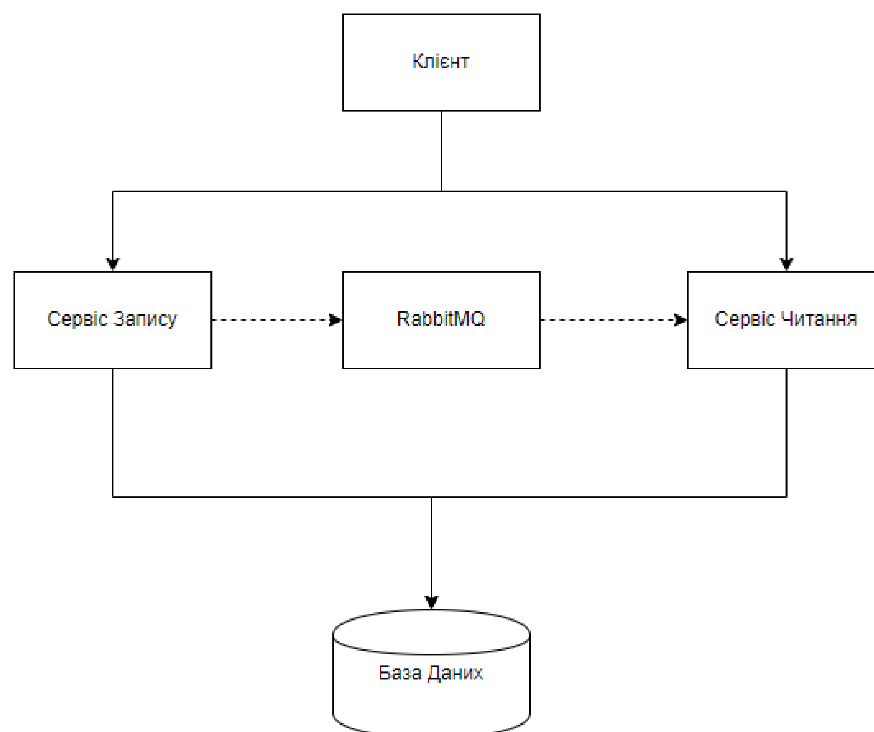

Рисунок 3.1 – Схема взаємодії компонентів додатку

3.3 Автентифікація та авторизація

Для забезпечення безпеки доступу до ресурсів застосунку використовуються JWT. Цей підхід дозволяє ефективно керувати доступом до різних частин системи, забезпечуючи при цьому високий рівень захисту. Приклад тіла JWT токена у відкритому вигляді, який використовується у застосунку, представлений на лістингу 3.1.

Лістинг 3.1 – JWT токен

```
{
    "sub": "1234567890",
    "name": "John Doe",
    "role": "Admin",
    "iat": 1516239022,
    "exp": 1516242622
}
```

У системі передбачені дві основні ролі користувачів: звичайні користувачі та адміністратори. Звичайні користувачі мають доступ до ендпоїнтів, які дозволяють отримувати інформацію про проекти. Адміністратори мають розширені права доступу, включаючи можливість створювати та оновлювати проекти, а також додавати нотатки до проектів. Налаштування політик авторизації у застосунку представлено на лістингу 3.2.

Лістинг 3.2 – Налаштування політик авторизаці

```
services.AddAuthorization(options =>
{
    options.AddPolicy("Default", policy => policy.RequireAssertion(_ =>
true));

    // Роль адміністратора вимагає ствердження ролі "Admin"
    options.AddPolicy("Admin", policy =>
    {
        policy.RequireClaim(ClaimTypes.Role, "Admin");
    });

    // Роль користувача вимагає ствердження ролі
    // "Admin" або  "User"
    options.AddPolicy("User", policy =>
    {
        policy.RequireClaim(ClaimTypes.Role, "Admin", "User");
    });
});
```

Для спрощення реалізації, ключ підпису для JWT зберігається в секреті, який безпечно передається в контейнер. Зазвичай рекомендується використання

віддалених провайдерів, але використання секретів Kubernetes або Docker в зашифрованому вигляді також є цілком прийнятним, оскільки дозволяє зберігати конфіденційну інформацію окремо від коду застосунку і передавати її контейнерам під час їх створення.

Для забезпечення безпеки та цілісності комунікації між клієнтом і сервером, JWT токен передається у заголовку кожного HTTP запиту. Це дозволяє серверу автентифікувати користувача та перевірити його права доступу без необхідності зберігати сесійні дані на сервері. Прикладом такої передачі може бути використання заголовка Authorization, де токен включається у форматі “Bearer <токен>”. Це забезпечує зручний і безпечний спосіб передачі автентифікаційної інформації з клієнта на сервер при кожному запиті. GET-запит (Ліст. 3.3) до сервісу читання та відповідь, отриманні за допомогою утиліти Postman (рис. 3.2).

Лістинг 3.3 – Налаштування авторизації для кінцевої точки

```
// GET-метод та відносний шлях кінцевої точки
app.MapGet("/api/projects/{id}/admin",
async (...) =>
{
 …
}).RequireAuthorization("Admin");
// Вимога авторизації за політикою "Admin"
```

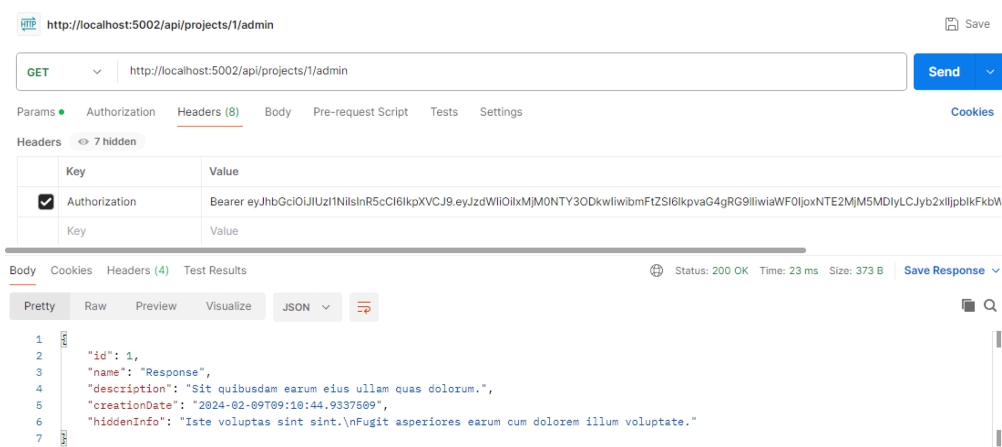

Рисунок 3.2 – виконання запиту до Read Service

3.4 Контейнеризація

Для створення та розгортання розподіленої системи з використанням Docker, де кожен сервіс ізольований у своєму контейнері, спочатку потрібно

створити образ кожного сервісу. Для цього використовується Dockerfile, в якому декларацію етапи та операції на кожному з них. В цьому випадку створюється багаторівневий образ, для більшої гнучкості, швидкості зборки образу та для того, щоб у фінальному образі знаходилися тільки необхідні файли (ліст. 3.4). Образ складається з 4-х рівнів:

1) Базовий.
2) Збірка проекту та його залежностей.
3) Компіляція проекту та його залежностей.
4) Фінальний.

Базовий шар, за допомогою інструкції FROM, створюється на основі офіційного образу Microsoft .NET ASP.NET Core, який забезпечує необхідне середовище виконання для запуску ASP.NET Core додатків. Далі створюється окремий користувач інструкцією USER, директорія в якій буде зберігатися фінальні файли за допомогою WORKDIR, та декларуються порти командою EXPOSE, які будуть відкриті у контейнера, дозволяючи додатку приймати HTTP запити.

Другим етапом відбувається копіювання вихідних файлів, збірка проекту, його залежностей. Цей рівень створюється аналогічним базовому, на основі образу Microsoft. Командами COPY копіюються вихідні файли для компіляції їх всередині образу, а потім вже скомпільовані копіюються в окрему директорію. Інструкціями RUN виконуються дві команди: restore та build, які потрібні для відновлення залежностей та збірки проекту.

На етапі публікації компілюються фінальні файли з раніше зібраного образу. Додаток збирається та оптимізується для розгортання, зберігаючи кінцеві файли в окремій директорії.

Фінальний рівень створюється на основі базового, де вже був створений користувач та задекларовані порти. На цьому етапі кінцеві файли копіюються у робочу директорію, таким чином залишаючи фінальний рівень тільки з потрібними артефактами. Останньою інструкцією ENTRYPOINT об’являється точка входу для контейнера, яка запускає додаток.

Лістинг 3.4 – Створення багаторівневого образу

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
USER app
WORKDIR /app
EXPOSE 8080
EXPOSE 8081

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
ARG BUILD_CONFIGURATION=Release
WORKDIR /src
COPY ["ProjectReadService/ProjectReadService.csproj", "ProjectReadService/"]
RUN dotnet restore "./ProjectReadService/ProjectReadService.csproj"
COPY . .
WORKDIR "/src/ProjectReadService"
RUN dotnet build "./ProjectReadService.csproj" -c $BUILD_CONFIGURATION -o
/app/build

FROM build AS publish
ARG BUILD_CONFIGURATION=Release
RUN dotnet publish "./ProjectReadService.csproj" -c $BUILD_CONFIGURATION -o
/app/publish /p:UseAppHost=false

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "ProjectReadService.dll"]
```

Для зборки образу використовується команда docker build. Ця команда збирає Docker образ використовуючи файл образу та контекст виконання. В результаті зібраний образ поміщається в регістр, в розглянутому випадку цей регістр є локальним до машини, де збирається цей образ. На рис 3.3 представлений консольний вивід при виконанні консольної команди зборку контейнера.

```
>> docker build . -f .\ProjectReadService\Dockerfile
[+] Building 0.7s (18/18) FINISHED
 => [internal] load build definition from Dockerfile                                                      0.0s
 => => transferring dockerfile: 32B                                                                       0.0s
 => [internal] load .dockerignore                                                                         0.0s
 => => transferring context: 35B                                                                          0.0s
 => [internal] load metadata for mcr.microsoft.com/dotnet/sdk:8.0                                         0.1s
 => [internal] load metadata for mcr.microsoft.com/dotnet/aspnet:8.0                                      0.0s
 => [build 1/7] FROM mcr.microsoft.com/dotnet/sdk:8.0@sha256:935902ef9eee58a9226b906e3d6ff1b2abaca240c9d5b4ac8dca9943b2  0.0s
 => [internal] load build context                                                                         0.2s
 => => transferring context: 9.68kB                                                                       0.1s
 => [base 1/2] FROM mcr.microsoft.com/dotnet/aspnet:8.0                                                   0.0s
 => CACHED [base 2/2] WORKDIR /app                                                                        0.0s
 => CACHED [final 1/2] WORKDIR /app                                                                       0.0s
 => CACHED [build 2/7] WORKDIR /src                                                                       0.0s
 => CACHED [build 3/7] COPY [ProjectReadService/ProjectReadService.csproj, ProjectReadService/]           0.0s
 => CACHED [build 4/7] RUN dotnet restore "./ProjectReadService/ProjectReadService.csproj"                0.0s
 => CACHED [build 5/7] COPY . .                                                                           0.0s
 => CACHED [build 6/7] WORKDIR /src/ProjectReadService                                                    0.0s
 => CACHED [build 7/7] RUN dotnet build "./ProjectReadService.csproj" -c Release -o /app/build            0.0s
 => CACHED [publish 1/1] RUN dotnet publish "./ProjectReadService.csproj" -c Release -o /app/publish /p:UseAppHost=fals  0.0s
 => CACHED [final 2/2] COPY --from=publish /app/publish .                                                 0.0s
 => exporting to image                                                                                    0.0s
 => => exporting layers                                                                                   0.0s
 => => writing image sha256:2d3a7221b0bc9529c4b208e9a78129800b6628be748d1456d46d17fcc405d1fb               0.0s
```

Рисунок 3.3 – Консольний вивід зборки контейнера

Для одночасного розгортання декількох сервісів, часто залежних, використовується утиліта docker compose та відповідний файл конфігурації. Файл docker-compose.yml використовується для визначення налаштувань та правил для сервісів. В цьому випадку він визначає сервіси для управління проектами, читання проектів, базу даних та RabbitMQ для асинхронної комунікації. Для сервісів читання та запису використовуються створені Dockerfile, а для БД та черги повідомлень - їх офіційні образи. У файлі визначені шляхи до Dockerfile, контекст для збірки, змінні середовища, назви контейнерів, залежності між сервісами, а також налаштовується перенаправлення портів між хостом та контейнером.

Такий підхід спрощує процес побудови та конфігурації розподіленого контейнеризованого застосунку, що надає можливість легко розгортати масштабувати та керувати кількома контейнерами, забезпечуючи ізоляцію та безпеку кожного сервісу.

Лістинг 3.5 – Представлення спрощеного змісту файла docker-compose.yml

```
services:
  read-service:
    container_name: read-service
    depends_on:
      - rabbit
      - mssql
    image: ${DOCKER_REGISTRY-}read-service
    build:
      context: .
      dockerfile: ProjectReadService/Dockerfile
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_HTTP_PORTS=8080
    ports:
      - "5002:8080"
      - "5003:8081"
…
  rabbit:
    container_name: rabbit
    image: rabbitmq:3-management-alpine
    ports:
     - "5672:5672"
     - "15672:15672"
  mssql:
    container_name: mssql
    image: mcr.microsoft.com/mssql/server:latest
    environment:
      ACCEPT_EULA: "Y"
      SA_PASSWORD: "StrongPassword!"
    ports:
      - 1433:1433
```

Командою docker compose up, всі налаштовані сервіси розгортаються, що можна перевірити перейшовши до застосунку Docker Desktop та побачити активні контейнери (рис. 3.4). Під час виконання команда індикує статус сервісів та час зайнятий на запуск кожного з них (лістинг 3.6), аргументом -d вимикається перенаправлення виводу з консолей контейнерів.

Name	Image	Status	Port(s)	Last started
projectdistributed	-	Running (5/5)		2 minutes ago
app 554c52faf9ba	app	Running	5000:8080 Show all ports (2)	2 minutes ago
mssql b1686cfeb880	mcr.microsoft.con	Running	1433:1433	2 minutes ago
rabbit 9fd39466ff00	rabbitmq:3-manag	Running	15672:15672 Show all ports (2)	2 minutes ago
read-service 8892572464fc	read-service	Running	5002:8080 Show all ports (2)	2 minutes ago
write-service 30b783408350	write-service	Running	5004:8080 Show all ports (2)	2 minutes ago

Рисунок 3.4 – Представлення активних контейнерів

Лістинг 3.6 - Виконання команди docker compose up

```
 docker compose up -d
[+] Running 5/5
 - Container rabbit              Started     1.0s
 - Container mssql               Started     1.2s
 - Container read-service        Started     1.6s
 - Container write-service       Started     1.9s
 - Container app                 Started     2.3s
```

Ізоляція контейнерів є однією з ключових переваг використання Docker. Вона забезпечує роботу кожного контейнера у власному оточенні, незалежно від інших контейнерів. Перевіримо це шляхом порівняння змінних середовища у контейнерах rabbit та read-service. Для цього треба під'єднатися до контейнерів по черзі командою docker exec та вивести на екран список змінних середовища (лістинг 3.7).

Лістинг 3.7 – Застосування команди docker exec

```
docker exec -it rabbit bash
9fd39466ff00:/# printenv
HOSTNAME=9fd39466ff00
LANGUAGE=C.UTF-8
PWD=/
RABBITMQ_PGP_KEY_ID=0x0A9AF2115F4687BD29803A206B73A36E6026DFCA
HOME=/var/lib/rabbitmq
LANG=C.UTF-8
ERLANG_INSTALL_PATH_PREFIX=/opt/erlang
RABBITMQ_HOME=/opt/rabbitmq
```

Продовження лістингу 3.7

```
RABBITMQ_VERSION=3.13.1
OPENSSL_INSTALL_PATH_PREFIX=/opt/openssl
TERM=xterm
RABBITMQ_DATA_DIR=/var/lib/rabbitmq
SHLVL=1
LC_ALL=C.UTF-8
PATH=/opt/rabbitmq/sbin:/opt/erlang/bin:/opt/openssl/bin:/usr/local/sbin:/usr/
local/bin:/usr/sbin:/usr/bin:/sbin:/bin
_=/bin/printenv

docker exec -it read-service bash
app@8892572464fc:/app$ printenv
HOSTNAME=8892572464fc
ASPNETCORE_ENVIRONMENT=Development
DOTNET_VERSION=8.0.4
APP_UID=1654
PWD=/app
HOME=/home/app
ASPNETCORE_HTTP_PORTS=8080
TERM=xterm
SHLVL=1 ASPNET_VERSION=8.0.4
DOTNET_RUNNING_IN_CONTAINER=true
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
_=/usr/bin/printenv
```

На лістингу видно, що, по-перше – змінні середовища відрізняються, а спільними є тільки ті, що є системними. По-друге, read-service використовує звичайного користувача app, на відміну від rabbit, який використовує root користувача, що можна помітити по характерному знаку решітки (#) у командному рядку.

Також доцільно перевірити ізольованість файлової системи, а саме те, що створений файл в одному контейнері не буде видимим в іншому. Аналогічно перевірці змінних середовища, потрібно приєднатися до контейнера rabbit, створити файл testfile.txt та перевірити його наявність у read-service (лістинг 3.8).

Лістинг 3.8 – Перевірка ізольованості файлової системи

```
docker exec -it rabbit bash
9fd39466ff00:/# mkdir -m 777 /app
9fd39466ff00:/# touch /app/testfile.txt
9fd39466ff00:/# exit

docker exec -it read-service bash
app@8892572464fc:/app$ find "testfile.txt"
find: 'testfile.txt': No such file or directory
```

3.5 Процес кластеризації з використанням Kubernetes

Kubernetes є одним з найпопулярніших інструментів для управління контейнерними додатками у кластері. Він забезпечує автоматизоване розгортання, масштабування та управління контейнерними додатками.

Утиліта kompose використовується для перетворення Docker Compose файлів у Kubernetes YAML маніфести, які визначають деплойменти, сервіси та інші ресурси Kubernetes. Після виконання команди kompose -c, були створені маніфести деплойменту та сервісу для кожного з компонентів розподіленого застосунку. Спочатку доцільно розглянути зміст на прикладі файлів mssql-deployment.yaml та mssql-service.yaml.

Маніфест деплойменту (лістинг 3.9) містить наступну інформацію:

1) apiVersion: Визначає версію API Kubernetes.
2) kind: Тип ресурсу, в цьому випадку Deployment, що відповідає за розгортання та управління контейнерами.
3) metadata: Містить метадані про деплоймент, включаючи ім'я та мітки.
4) spec: Специфікація деплойменту, включаючи кількість реплік, селектори для вибору подів та шаблон поду.
5) replicas: Вказує кількість реплік, які повинні бути запущені.
6) template: Шаблон для подів, які будуть створені цим деплойментом.
7) containers: Визначає контейнери, які будуть запущені в поді.
8) env: Середовищні змінні для контейнера.
9) image: Docker-образ, який буде використаний для створення контейнера.
10) name: Ім'я контейнера.
11) ports: Порти, які будуть відкриті в контейнері.

Лістинг 3.9 – Маніфест деплойменту

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    io.kompose.service: mssql
  name: mssql
spec:
  replicas: 1
  selector:
    matchLabels:
```

Продовження лістингу 3.9

```
io.kompose.service: mssql
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        io.kompose.service: mssql
    spec:
      containers:
      - env:
        - name: SA_PASSWORD
          value: "YourStrong!Passw0rd"
        - name: ACCEPT_EULA
          value: "Y"
        image: mcr.microsoft.com/mssql/server:2019-latest
        name: mssql
        ports:
        - containerPort: 1433
```

Файл сервісу (лістинг 3.10) містить наступні дані:

1) apiVersion: Визначає версію API Kubernetes.
2) kind: Тип ресурсу, в цьому випадку Service, що забезпечує мережевий доступ до набору подів.
3) metadata: Містить метадані про сервіс, включаючи ім'я та мітки.
4) spec: Специфікація сервісу.
    - ports: Визначає порти, які будуть відкриті для доступу до сервісу.
    - port: Порт, на якому сервіс буде доступний.
    - targetPort: Порт, на який буде перенаправлено трафік всередині поду.
    - selector: Мітки для вибору подів, до яких буде направлений трафік.

Лістинг 3.10 – Файл сервісу

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    io.kompose.service: mssql
  name: mssql
spec:
  ports:
  - name: "1433"
    port: 1433
    targetPort: 1433
  selector:
    io.kompose.service: mssql
```

Далі створюється minikube кластер командою minikube start, на якому розгортаються сервіси. Сервіси розгортаються командою kubectl apply. Наступним етапом запустимо вбудований дашборд для перевірки готовності сервісів (рис.3.5).

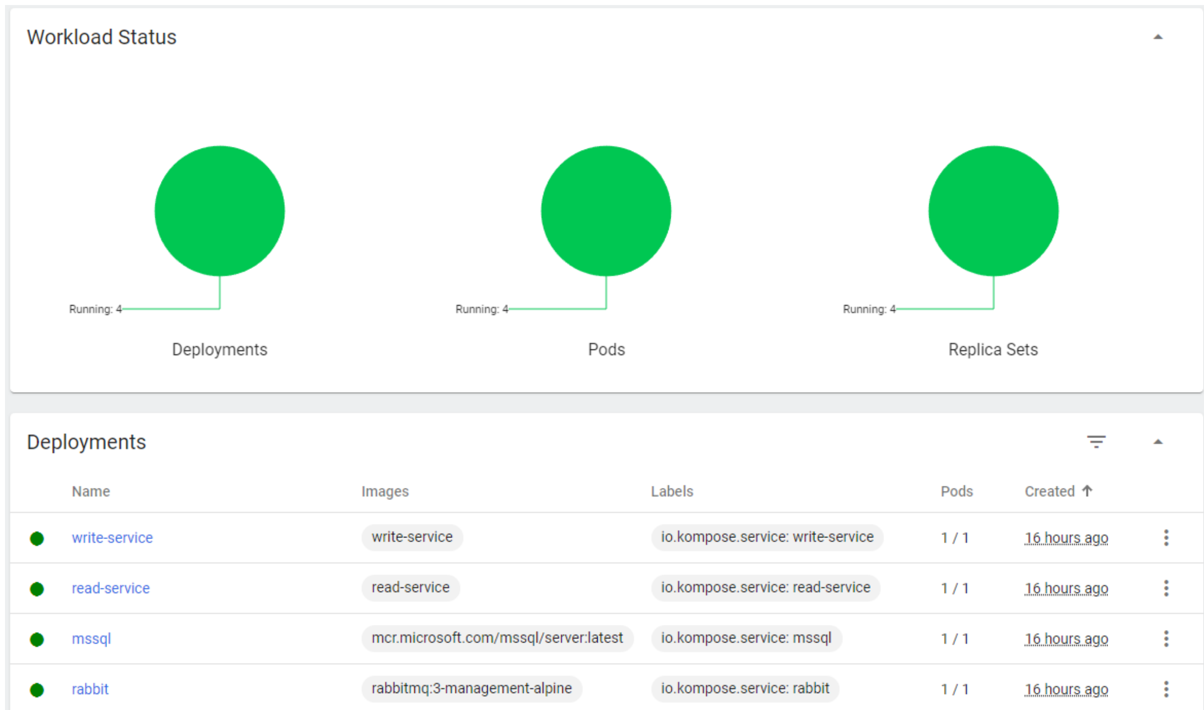

Рисунок 3.5 – Запуск вбудованого дашборду для перевірки готовності сервісів

3.6 Процес кластеризації з використанням Kubernetes

Для забезпечення повноцінного моніторингу Kubernetes кластера, включаючи збір метрик, налаштування дашбордів та оповіщення про критичні події, було встановлено чарт kube-prometheus-stack. Цей чарт, наданий спільнотою Prometheus, включає всі необхідні компоненти для створення комплексного моніторингового рішення.

Kube-prometheus-stack об'єднує в собі Kubernetes маніфести, дашборди Grafana та правила Prometheus, що дозволяють легко розгорнути комплексну систему моніторингу кластера Kubernetes. Цей чарт включає Prometheus Operator, Alertmanager, Node Exporters, kube-state-metrics, а також попередньо налаштовані дашборди Grafana та правила сповіщень Prometheus.

Для розгортання kube-prometheus-stack використовується Helm. Команди додають репозиторій Helm з чартами Prometheus, оновлюють локальний кеш і

встановлюють kube-prometheus-stack в просторі імен kubescape з необхідними параметрами (лістинг 3.11).

Лістинг 3.11 – Розгортання kube-prometheus-stack

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update
helm install observability prometheus-community/kube-prometheus-stack -n kubescape --set
prometheus.prometheusSpec.podMonitorSelectorNilUsesHelmValues=false,prometheus
.prometheusSpec.serviceMonitorSelectorNilUsesHelmValues=false
```

У сервісі Grafana доступна велика кількість попередньо створених дашбордів, які роблять спостереження за станом та подіями у кластері дуже наочними, розглянемо 2 приклади таких дашбордів.

“Kubernetes / API server Dashboard” призначений для моніторингу стану API сервера Kubernetes. Ключовими метриками є кількість запитів до API сервера, кількість помилок та час відповіді на ці запити. Також зображена загальна доступність сервера, доступність операцій читання та запису у процентному співвідношенні. Моніторинг API сервера є критичним для розуміння загального стану кластера, оскільки API сервер є точкою входу для всіх запитів до Kubernetes. На рис. 3.6 частково зображений загальний вигляд дашборду.

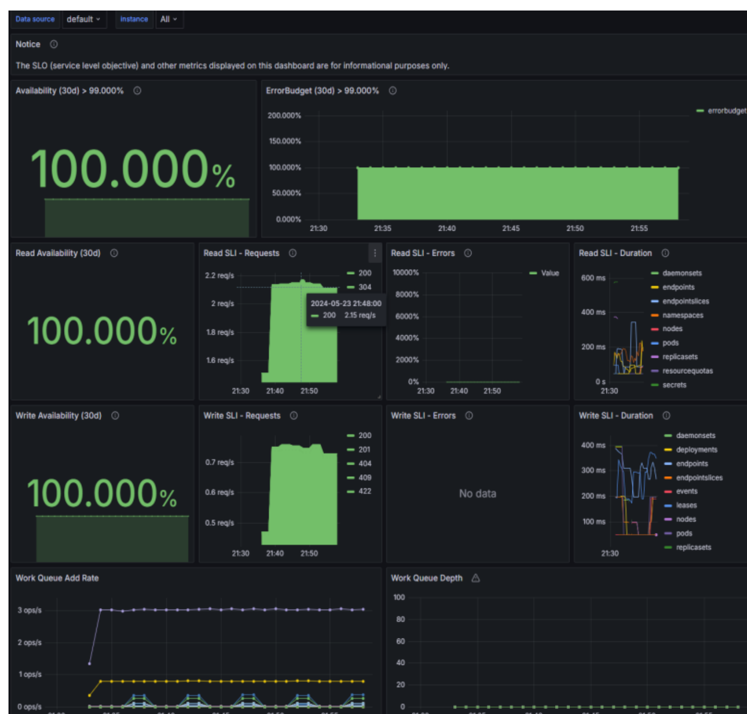

Рисунок 3.6 - Kubernetes / API server Dashboard

“Compute Resources / Cluster” призначений для моніторингу загального використання ресурсів в кластері Kubernetes (рис.3.7). Дозволяє отримати загальне уявлення про стан ресурсів кластера, що є важливим для забезпечення стабільної роботи додатків та оптимального використання доступних ресурсів. Ключові метрики використання: CPU, пам'яті, дискової пам'яті та трафіку.

Використання CPU включає загальне використання процесорного часу в кластері, що дозволяє оцінити ефективність використання процесорних ресурсів. Використання пам'яті охоплює загальне споживання оперативної пам'яті, що допомагає контролювати навантаження на пам'ять. Використання дискової пам'яті визначає обсяг використаного та доступного дискового простору. Мережевий трафік враховує обсяг вхідного та вихідного трафіку, що дозволяє відстежувати мережеву активність та виявляти можливі проблеми з пропускною здатністю.

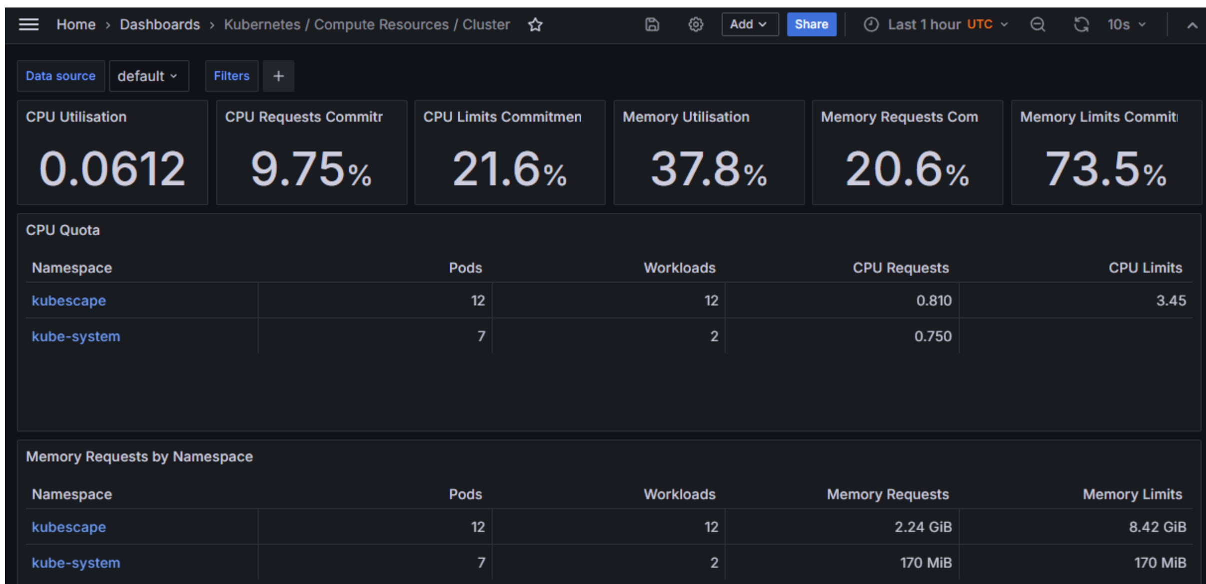

Рисунок 3.7 – Compute Resources / Cluster

Система оповіщень налаштовується через правила Prometheus, які визначають умови для відправлення сповіщень. Наприклад, можна налаштувати правило (лістинг 3.12), яке сповіщатиме про високий рівень використання CPU або пам'яті в кластері, а саме сповіщення якщо певний сервіс використовує понад 90% CPU протягом двох хвилин.

Лістинг 3.12 – Налаштування сповіщення

```
groups:
- name: example.rules
  rules:
  - alert: HighCPUUsage
    expr: sum(rate(container_cpu_usage_seconds_total[1m])) by (namespace, pod)
> 0.9
    for: 2m
    labels:
      severity: critical
    annotations:
      summary: "High CPU usage detected"
      description: "Pod {{ $labels.pod }} in namespace {{ $labels.namespace }}
is using more than 90% CPU for more than 2 minutes."
```

Сповіщення можуть бути відправлені через різні канали, такі як Email, Slack, PagerDuty (система управління кіберінцидентами) та інші. Такий підхід для моніторингу кластера та сповіщень забезпечує не тільки оперативний збір та аналіз метрик, але й підвищує загальний рівень кібербезпеки системи завдяки наступним аспектам:

- Швидке виявлення аномальних дій та показників дозволяє оперативно реагувати на потенційні загрози.
- Постійний моніторинг стану компонентів кластера забезпечує їх безперебійну роботу і запобігає можливим вразливостям.
- Налаштування оповіщень дозволяє проактивно реагувати на потенційні проблеми до їх виникнення, що знижує ризик атак та збоїв.

3.7 KubeScape та його роль в підвищенні кібербезпеки розподілених застосунків

KubeScape – це інструмент для сканування та оцінки безпеки кластерів Kubernetes. Він надає можливість автоматизованого аналізу конфігурацій і політик безпеки, допомагаючи виявляти потенційні вразливості та недоліки в налаштуваннях кластерів. KubeScape підтримує різноманітні стандарти та рекомендації, такі як NSA, MITRE ATT&CK та CIS Benchmarks. KubeScape автоматизує процес перевірки конфігурацій, зменшуючи ризик людських помилок і забезпечуючи швидкий зворотний зв'язок. Інструмент дозволяє сканувати кластери на відповідність загальновизнаним стандартам безпеки, що

підвищує загальний рівень захищеності. Також надає детальні звіти з описом знайдених вразливостей, рекомендаціями щодо їх усунення та оцінкою ризиків. Має можливість інтеграції з конвеєрами безперервної інтеграції та доставки дозволяє виявляти проблеми на ранніх стадіях розробки та розгортання.

Команда kubescape scan дозволяє користувачам здійснювати сканування кластерів Kubernetes для перевірки їхньої відповідності визначеним стандартам безпеки. Це може бути корисно як для одноразових сканувань під час налаштування середовища, так і для регулярних перевірок в процесі розробки.

Після встановлення утиліти можна виконати детальне сканування на відповідність стандартам безпеки, рекомендованим NSA. У результаті буде отримана як загальна оцінка відповідності системи (рис. 3.8), так і деталізована по кожному компоненту кластера в цілому (рис. 3.9). Покомпонентний вивід містить назву вразливості, серйозність, посилання на повну документацію вразливості та коротку підказку для її усунення.

Severity	Control name	Failed resources	All Resources	Compliance score
Critical	API server insecure port is enabled	0	1	100%
Critical	Disable anonymous access to Kubelet service	0	0	Action Required *
Critical	Enforce Kubelet client TLS authentication	0	0	Action Required *
High	Applications credentials in configuration files	2	62	97%
High	Host PID/IPC privileges	1	30	97%
High	HostNetwork access	1	30	97%
High	Insecure capabilities	0	30	100%
High	Privileged container	1	30	97%
High	CVE-2021-25742-nginx-ingress-snippet-annotation-vu...	0	0	100%
High	Ensure CPU limits are set	18	30	40%
High	Ensure memory limits are set	18	30	40%
Medium	Prevent containers from allowing command execution	1	89	99%
Medium	Non-root containers	8	30	73%
Medium	Allow privilege escalation	9	30	70%
Medium	Ingress and Egress blocked	18	30	40%
Medium	Automatic mapping of service account	18	89	80%
Medium	Administrative Roles	1	89	99%
Medium	Container hostPort	4	30	87%
Medium	Cluster internal networking	2	7	71%
Medium	Linux hardening	12	30	60%
Medium	CVE-2021-25741 - Using symlink for arbitrary host ...	0	0	100%
Medium	Secret/etcd encryption enabled	1	1	0%
Medium	Audit logs enabled	0	1	100%
Low	Immutable container filesystem	11	30	63%
Low	PSP enabled	1	1	0%
	Resource Summary	37	217	68.36%

Рисунок 3.8 – Загальна оцінка відповідності системи

```
################################################################################
ApiVersion: apps/v1
Kind: Deployment
Name: read-service
Namespace: default

Controls: 14 (Failed: 9, action required: 0)
```

Severity	Control name	Docs	Assisted remediation
Medium	Allow privilege escalation	https://hub.armosec.io/docs/c-0016	spec.template.spec.containers[0].securityContext.allowPrivilegeEscalation=false
	Automatic mapping of service account	https://hub.armosec.io/docs/c-0034	spec.template.spec.automountServiceAccountToken=false
	Container hostPort	https://hub.armosec.io/docs/c-0044	spec.template.spec.containers[0].ports[0].hostPort spec.template.spec.containers[0].ports[1].hostPort
	Ingress and Egress blocked	https://hub.armosec.io/docs/c-0030	
	Linux hardening	https://hub.armosec.io/docs/c-0055	spec.template.spec.containers[0].securityContext.seccompProfile=YOUR_VALUE spec.template.spec.containers[0].securityContext.seLinuxOptions=YOUR_VALUE spec.template.spec.containers[0].securityContext.capabilities.drop[0]=YOUR_VALUE
	Non-root containers	https://hub.armosec.io/docs/c-0013	spec.template.spec.containers[0].securityContext.runAsNonRoot=true spec.template.spec.containers[0].securityContext.runAsGroup=1000
High	Ensure CPU limits are set	https://hub.armosec.io/docs/c-0270	spec.template.spec.containers[0].resources.limits.cpu=YOUR_VALUE
	Ensure memory limits are set	https://hub.armosec.io/docs/c-0271	spec.template.spec.containers[0].resources.limits.memory=YOUR_VALUE
Low	Immutable container filesystem	https://hub.armosec.io/docs/c-0017	spec.template.spec.containers[0].securityContext.readOnlyRootFilesystem=true

Рисунок 3.9 – Оцінка відповідності системи за кожним компонентом

На прикладі read-service усунемо вразливість, яка дозволяє ескалацію привілеїв у контейнері. Для цього потрібно явно заборонити таку дію в конфігурації деплойменту, виставивши прапор spec.template.spec.containers[0].securityContext.allowPrivilegeEscalation у значення false. Оновимо файл конфігурації новим значенням (лістинг 3.13) та застосуємо зміни.

Лістинг 3.13 – Оновлення файлу конфігурації

```
apiVersion: apps/v1
kind: Deployment
…
spec:
…
    spec:
      containers:
…
          image: read-service
          name: read-service
          securityContext:
            allowPrivilegeEscalation: false
…
```

Після повторного сканування можна побачити, що вразливість щодо підняття привілеїв зникла (рис. 3.10).

```
################################################################################
ApiVersion: apps/v1
Kind: Deployment
Name: read-service
Namespace: default

Controls: 14 (Failed: 8, action required: 0)
```

Severity	Control name	Docs	Assisted remediation
Medium	Automatic mapping of service account	https://hub.armosec.io/docs/c-0034	spec.template.spec.automountServiceAccountToken=false
	Container hostPort	https://hub.armosec.io/docs/c-0044	spec.template.spec.containers[0].ports[0].hostPort spec.template.spec.containers[0].ports[1].hostPort
	Ingress and Egress blocked	https://hub.armosec.io/docs/c-0030	
	Linux hardening	https://hub.armosec.io/docs/c-0055	spec.template.spec.containers[0].securityContext.seccompProfile=YOUR_VALUE spec.template.spec.containers[0].securityContext.seLinuxOptions=YOUR_VALUE spec.template.spec.containers[0].securityContext.capabilities.drop[0]=YOUR_VALUE
	Non-root containers	https://hub.armosec.io/docs/c-0013	spec.template.spec.containers[0].securityContext.runAsNonRoot=true spec.template.spec.containers[0].securityContext.runAsGroup=1000
High	Ensure CPU limits are set	https://hub.armosec.io/docs/c-0270	spec.template.spec.containers[0].resources.limits.cpu=YOUR_VALUE
	Ensure memory limits are set	https://hub.armosec.io/docs/c-0271	spec.template.spec.containers[0].resources.limits.memory=YOUR_VALUE
Low	Immutable container filesystem	https://hub.armosec.io/docs/c-0017	spec.template.spec.containers[0].securityContext.readOnlyRootFilesystem=true

Рисунок 3.10 – Результат усунення ескалації привілеїв у контейнері

Щодо сканування кластера в цілому, то KubeScape охоплює велику кількість аспектів для забезпечення кібербезпеки, а саме мережевий захист, управління обліковими записами, контроль доступу, політики безпеки контейнерів та кластерів.

Інструмент допомагає ідентифікувати відкриті порти, які можуть стати точками входу для атак, а також перевіряє налаштування політик мережевої безпеки між різними компонентами кластера, зменшуючи ризик внутрішніх атак. Один із важливих функціоналів є виявлення небезпечних налаштувань облікових записів, такі як надмірні привілеї або використання загальних облікових записів без паролів, що допомагає зменшити ризик НСД.

Для перевірки політик RBAC, KubeScape забезпечує дотримання принципу мінімально необхідних привілеїв. Він аналізує ролі та права доступу, виявляючи надмірні привілеї або небезпечні ролі, які можуть стати мішенню для атак, таким чином забезпечуючи належний рівень захисту. Таким чином, він допомагає забезпечити правильну ізоляцію контейнерів, запобігаючи можливості впливу одного контейнера на інші.

Отже, KubeScape є інструментом, за використанням якого можна створити більш захищені та надійні розподілені системи, зменшуючи ризик кіберзагроз та підвищуючи загальний рівень безпеки ІТ-інфраструктури.

ВИСНОВКИ

У першому розділі було розглянуто архітектурні стилі проєктування розподілених систем. Основна увага була приділена трьом основним стилям: моноліти, сервісно-орієнтована архітектура та мікросервіси. Було проаналізовано переваги та недоліки кожного з цих стилів.

Моноліти вирізняються простотою і швидкістю розробки, але мають обмеження у масштабованості. SOA пропонує гнучкість та повторне використання сервісів, але має складності з управлінням залежностями. Мікросервіси забезпечують високу гнучкість і можливість незалежного масштабування кожного сервісу, але потребують складного управління та безпеки мережевих взаємодій.

У другому розділі досліджено аспекти кібербезпеки в контейнеризованих системах. Було використано методи аналізу та синтезу для порівняння безпеки традиційних монолітних та кластеризованих додатків. У ході дослідження виявлено, що монолітні додатки простіше захищати завдяки єдиній точці входу, але вони мають більшу поверхню атаки через тісно пов'язані компоненти. Мікросервісна архітектура зменшує ризики завдяки ізоляції сервісів, але збільшує складність управління безпекою через інтенсивну мережеву взаємодію.

Для забезпечення належного рівня безпеки необхідно налаштувати політики безпеки, регулярно сканувати контейнерні образи, ефективно управляти секретами та використовувати передові методи моніторингу та аудиту.

Для підвищення ефективності системи рекомендується використовувати багаторівневі образи, налаштовувати обмеження ресурсів для контейнерів та впроваджувати систему моніторингу на базі Prometheus та Grafana для оперативного виявлення та реагування на загрози.

Третій розділ був присвячений реалізації підвищення рівня кібербезпеки в розподіленому застосунку. Було створено розподілений контейнеризований додаток, який демонструє, як контейнеризація може ізолювати процеси та

ресурси між контейнерами, підвищуючи загальну безпеку системи. А також впроваджено механізми автентифікації та авторизації, контроль доступу за допомогою JWT та OAuth2 і системи моніторингу та аудиту.

Виконана робота буде актуальною для розробників програмного забезпечення, інженерів DevOps, фахівців з кібербезпеки, менеджерів IT-проектів та дослідників. Розробникам вона допоможе вибрати відповідний архітектурний стиль для проектів, інженерам DevOps – налаштовувати політики безпеки та інтегрувати системи моніторингу, фахівцям з кібербезпеки – захищати розподілені додатки, а менеджерам IT-проектів – приймати обґрунтовані рішення щодо технологій. В свою чергу, дослідники зможуть використовувати результати для подальших досліджень у сфері кібербезпеки розподілених систем та контейнеризації.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Virto Commerce: Microservices vs Service-Oriented vs Monolithic. (2019, жовтень 8). Режим доступу: https://www.virtocommerce.org/t/virto-commerce-microservices-vs-service-oriented-vs-monolithic/35
2. Ford, N., Parsons, R., & Kua, P. (2017). Building evolutionary architectures: Support Constant Change. “O’Reilly Media, Inc.”
3. Figure 2: Monolithic architecture. Режим доступу: https://www.researchgate.net/figure/Monolithic-Architecture_fig2_325390165
4. Sarwar, A. (2021, грудень 13). What is Service Oriented Architecture - Adeel Sarwar - Medium. Medium. Режим доступу: https://medium.com
5. Kolny, M. (2023, березень 22). Scaling up the Prime Video audio/video monitoring service and reducing costs by 90% - Prime Video Tech. Режим доступу: https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90
6. Kane, S. P., & Matthias, K. (2018). Docker: Up & running: Shipping Reliable Containers in Production. O’Reilly Media.
7. AntonioFalcaoJr. GitHub - AntonioFalcaoJr/EventualShop: A state-of-the-art distributed system using Reactive DDD as uncertainty modeling, Event Storming as subdomain decomposition, Event Sourcing as an eventual persistence mechanism, CQRS, Async Projections, Microservices for individual deployable units, Event-driven Architecture for efficient integration, and Clean Architecture as domain-centric design. Режим доступу: https://github.com/AntonioFalcaoJr/EventualShop?tab=readme-ov-file
8. Containerized Docker Application Lifecycle with Microsoft Platform and Tools. Режим доступу: https://novacontext.com/containerized-docker-application-lifecycle-with-microsoft-platform-and-tools/index.html
9. Threat landscape. Режим доступу: https://www.enisa.europa.eu/topics/threat-risk-management/threats-and-trends

10. Souppaya, M., Morello, J., & Scarfone, K. (2017, вересень). Application container security guide. Режим доступу: https://doi.org/10.6028/nist.sp.800-190

11. Pahl, C., Brogi, A., Soldani, J., & Jamshidi, P. (2017). Security issues and challenges for container orchestration in cloud computing. IEEE Cloud Computing, 4(1), ст 22-29. Режим доступу: https://doi.org/10.1109/MCC.2017.18

12. Owasp. Docker-Security/dist/owasp-docker-security.pdf at main · OWASP/Docker-Security. Режим доступу: https://github.com/OWASP/Docker-Security/blob/main/dist/owasp-docker-security.pdf

13. OWASP Kubernetes Top Ten | OWASP Foundation. Режим доступу: https://owasp.org/www-project-kubernetes-top-ten/

14. Jorm, N., Peters, A., & Lee, M. (2019). Container security: Virtualization for DevOps. IEEE Security & Privacy, 17(2), ст 72-78. Режим доступу: https://doi.org/10.1109/MSEC.2019.2892105

15. Jones, M., Bradley, J., & Sakimura, N. (2015). JSON Web Token (JWT) (RFC 7519). Internet Engineering Task Force (IETF). Режим доступу: https://datatracker.ietf.org/doc/html/rfc7519

16. Hardt, D. (2012). The OAuth 2.0 authorization framework (RFC 6749). Internet Engineering Task Force (IETF). Режим доступу: https://datatracker.ietf.org/doc/html/rfc6749

17. Ali, M. U., Khan, S., & Ahmad, F. (2020). A comprehensive study of authentication mechanisms in containerized applications. ACM Computing Surveys, 53(4), ст 1-35. Режим доступу: https://doi.org/10.1145/3397097

18. Cinque, M., Della Corte, R., & Pecchia, A. (2022). Microservices Monitoring with Event Logs and Black Box Execution Tracing. IEEE Transactions on Services Computing, 15(1), 294–307. Режим доступу: https://doi.org/10.1109/tsc.2019.2940009

19. Prometheus-Community. Режим доступу: helm-charts/charts/kube-prometheus-stack at main · prometheus-community/helm-charts. Retrieved from https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack

20. Kedziora, D., Smith, J., & Taylor, R. (2020). An analysis of message queuing systems for microservices. Journal of Systems and Software, 160, 110512. Режим доступу: https://doi.org/10.1016/j.jss.2020.110512