

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ГОРЯ СКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

До захисту допущено
Завідувач кафедри

_____ Дмитро ЛАНДЕ
(підпис)

« _____ » _____ 2024 р.

Дипломна робота
на здобуття ступеня бакалавра за
освітньо-професійною програмою «Системи,
технології та математичні методи кібербезпеки»
спеціальності 125 «Кібербезпека»

на тему: Аналіз вразливостей в системі Kubernetes та впровадження ефективних засобів захисту

Виконав (-ла): здобувач вищої освіти **IV** курсу, групи ФБ-05

(шифр групи)

Тимченко Юрій Павлович
(прізвище, ім'я, по батькові) (підпис)

Керівник ст. викладач кафедри ІБ к.ф.-м.н Рибак О. В.
(посада, науковий ступінь, вчене звання, прізвище, ім'я, по батькові) (підпис)

Рецензент доцент кафедри ММЗІ, к. ф.-м. н. Хмельницький М. О.
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище, ім'я, по батькові) (підпис)

Засвідчую, що у цій дипломній роботі немає
запозичень з праць інших авторів без відповідних
посилань.

Здобувач вищої освіти _____

(підпис)

Київ – 2024 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 125 «Кібербезпека»

Освітньо-професійна програма «Системи, технології та математичні методи кібербезпеки»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Дмитро ЛАНДЕ
(підпис)

«__» _____ 2024 р.

ЗАВДАННЯ на дипломну роботу здобувачу
вищої освіти

Тимченко Юрій Павлович

(прізвище, ім'я, по батькові)

1. Тема роботи: Аналіз вразливостей в системі Kubernetes та впровадження ефективних засобів захисту,

керівник роботи Рибак Олександр Владиславович, старший викладач кафедри інформаційної безпеки,

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від 2024 р.

2. Термін подання здобувачем вищої освіти роботи 12 червня 2024 р.

3. Вихідні дані до роботи: Згідно з завданням дипломної роботи

4. Зміст роботи: Вступ; Що таке система керування контейнерами. Архітектура та компоненти Kubernetes; Аналіз вразливостей в системі Kubernetes; Вдосконалення безпеки Kubernetes; Висновки;

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо):

Презентація до захисту дипломної роботи.

6. Дата видачі завдання: 28.01.2024

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів дипломної роботи	Примітка
1	Формування теми дипломної роботи	09.12.2023	Виконано
2	Узгодження теми з дипломним керівником.	15.01.2023 - 19.01.2023	Виконано
3	Отримання завдання	28.01.2024 - 28.01.2024	Виконано
4	Опрацювання вибраної літератури	01.02.2024 - 07.04.2024	Виконано
5	Написання оглядових розділів дипломної роботи	08.04.2024 - 14.04.2024	Виконано
6	Проходження переддипломної практики	15.04.2024 - 19.05.2024	Виконано
7	Дослідження методів вдосконалення безпеки систем	20.05.2024 - 31.05.2024	Виконано
8	Оформлення дипломної роботи	01.03.2024 – 16.04.2024	Виконано

Здобувач вищої освіти

(підпис)

Юрій Тимченко

(Власне ім'я, ПРІЗВИЩЕ)

Керівник роботи

(підпис)

Олександр Рибак

(Власне ім'я, ПРІЗВИЩ)

РЕФЕРАТ

Обсяг дипломної 57 сторінок, 32 ілюстрації, 20 джерел літератури.

Об'єкт дослідження: система керування контейнерами Kubernetes.

Предмет дослідження: вразливості в системах керування контейнерами Kubernetes.

Мета дослідження: Покращення захисту системи керування контейнерами за допомогою впровадження додаткових рекомендацій

Практичне значення: Результати дослідження можуть бути використані DevOps інженерами під час налаштування екосистеми безперервної доставки та автоматичного розгортання. Це сприятиме стабільності та надійності роботи програмних систем, що є критично важливим для сучасного бізнесу.

Наукова новизна отриманих результатів полягає в класифікації та визначенні причин виникнення вразливостей під час роботи з системами керування контейнерами, та формулюванні методів вдосконалення безпеки Kubernetes систем.

Ключові слова: Kubernetes, pod, CNI, вразливості, контейнер

ABSTRACT

The work consists of 3 sections, 32 illustrations, 20 sources of literature, the volume of work is 57 pages

Object of Research: Kubernetes container orchestration system.

Subject of Research: Vulnerabilities in Kubernetes container orchestration systems.

Objective of Research: Improving the security of the container management system by implementing additional recommendations.

Practical significance: The results of the research can be used by DevOps engineers when setting up a continuous delivery ecosystem and automatic deployment. This will enhance the stability and reliability of software systems, which is critically important for modern businesses.

The scientific novelty of the obtained results lies in the classification and identification of the causes of vulnerabilities when working with container management systems, and in formulating methods to improve the security of Kubernetes systems.

Keywords: Kubernetes, pod, CNI, vulnerabilities, containers.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ	7
ВСТУП.....	8
1 ЩО ТАКЕ СИСТЕМА КЕРУВАННЯ КОНТЕЙНЕРАМИ. АРХІТЕКТУРА ТА КОМПОНЕНТИ KUBERNETES	9
1.1 Передумови створення Kubernetes.....	9
1.2 Види контейнерної технології.....	15
1.3 Переваги Kubernetes.....	16
1.4 Концепція Kubernetes.....	17
1.5 Концепція Service.....	20
1.6 Контролери	21
Висновки до розділу 1	23
2 АНАЛІЗ ВРАЗЛИВОСТЕЙ В СИСТЕМІ KUBERNETES	24
2.1 Найпопулярніші вразливості.....	24
2.2 Перелік методів захисту.....	26
Висновки до розділу 2	33
3 ВДОСКОНАЛЕННЯ БЕЗПЕКИ KUBERNETES	34
3.1 Впровадження плагіну Cilium.....	34
3.2 Впровадження Open Policy Agent.....	47
Висновки до розділу 3	54
ВИСНОВКИ	55
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ.....	56

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

EBPF – extended Berkley Packet Filter.

DDoS – Distributed Denial of Service.

API – Application programming interface

OCI – Open Container Initiative

AWS – Amazon Web Services

CTF – Capture The Flag

CNI – Container Network Interface

OPA – Open Policy Agent.

CRD – Custom Resource Definition.

ВСТУП

Сучасний інформаційний ландшафт приводить до постійної еволюції технологій, що ставить перед сучасними організаціями безпекові виклики невиданої складності. Зокрема, зростання популярності мікросервісної архітектури та контейнеризації призвело до широкого використання систем керування контейнерами, серед яких однією з найбільш популярних є Kubernetes. Kubernetes, розроблений компанією Google, став де-факто стандартом для оркестрації контейнерів у хмарних середовищах та віртуалізованих інфраструктурах. Його потужність та гнучкість в комбінації з відкритим джерелом робить його вельми привабливим для розробників та адміністраторів.

Однак, зростання популярності Kubernetes також приносить з собою нові виклики у сфері кібербезпеки. З великими можливостями приходить і значне збільшення потенційних загроз безпеці, включаючи атаки, витік інформації, та вразливості, які можуть призвести до серйозних наслідків для бізнесу.

Метою дипломної роботи є оцінювання вразливостей, які виникають через недбале налаштування систем Kubernetes, та виділення рекомендації, що дозволять їх уникнути.

Об'єктом дослідження є система керування контейнерами Kubernetes.

Предметом дослідження є вразливості в системах керування контейнерами Kubernetes.

Наукова новизна отриманих результатів полягає в класифікації та визначенні причин виникнення вразливостей під час роботи з системами керування контейнерами, та формулюванні методів вдосконалення безпеки Kubernetes систем.

Практичне значення: Результати дослідження можуть бути використані DevOps інженерами під час налаштування екосистеми Kubernetes. Це сприятиме стабільності та надійності роботи програмних систем, що є критично важливим для сучасного бізнесу.

1 ЩО ТАКЕ СИСТЕМА КЕРУВАННЯ КОНТЕЙНЕРАМИ. АРХІТЕКТУРА ТА КОМПОНЕНТИ KUBERNETES

1.1 Передумови створення Kubernetes

1.1.1 Ера розгортання на фізичні сервера

На початку розвитку інформаційних технологій організації зазвичай запускали свої застосунки на фізичних серверах. Це був простий і зрозумілий спосіб, але він мав свої обмеження. Оскільки ресурси на сервері були обмеженими, можливі були ситуації, коли один застосунок забирав більше ресурсів, ніж інші, що призводило до проблем з продуктивністю та доступністю[1].

Нижче наведено переваги розгортання на фізичні сервера:

- Ваша ІТ-команда має повний доступ до ваших відведених серверних ресурсів цілодобово, 7 днів на тиждень, 365 днів на рік, що може бути вигідним для операцій з високим попитом або для критичних бізнес-операцій.
- Сервер може бути розташований на місці або у зручно розташованому колокаційному центрі, залежно від ваших потреб.
- Ваша ІТ-команда може повністю налаштовувати і конфігурувати сервер за вашими специфікаціями та бізнес-вимогами.

Недоліки фізичних серверів:

- Відносно висока вартість придбання, у разі відмови обладнання, треба замінити сервер.
- Ваша ІТ-команда буде відповідальною за налаштування, конфігурацію та обслуговування.

- Вам може знадобитися найняти постачальника для допомоги у разі складнощів з налаштуванням або ремонтом.
- Вимагає фізичного простору або місця в колокаційному центрі.

1.1.2 Ера віртуалізації

В епоху віртуалізованого розгортання відкриваються нові горизонти для оптимізації ресурсів та забезпечення ефективності в управлінні інфраструктурою. Цей період в історії ІТ-індустрії характеризується широким застосуванням віртуалізації - технології, яка дозволяє запускати кілька віртуальних машин на одному фізичному сервері. Віртуалізація робить можливим ізольоване середовище для кожного застосунку шляхом використання віртуальних машин. Кожна віртуальна машина має свою власну операційну систему та ресурси, і вони можуть працювати паралельно на одному фізичному сервері без взаємодії або впливу одна на одну[1].

Нижче наведено переваги віртуалізації:

- Ефективне використання ресурсів

Перед віртуалізацією ІТ-персонал виділяв відведену фізичну ЦП на кожний сервер застосунку, купуючи та налаштовуючи окремий сервер для кожного застосунку. Цей підхід, який віддавав перевагу одному застосунку та одній операційній системі на кожному комп'ютері, був обраний заради надійності. В кожному випадку кожний фізичний сервер залишався маловикористаним. У порівнянні з цим, віртуалізація серверів дозволяє вам запускати кілька застосунків - кожен на власній віртуальній машині з власною операційною системою - на одному фізичному комп'ютері (зазвичай на сервері x86) без жертвування надійності. Це дозволяє максимально використовувати обчислювальні можливості фізичного обладнання.

- Спрощене управління

Заміна фізичних комп'ютерів віртуалізованими машинами, що визначені програмним забезпеченням, полегшує використання та управління політиками, написаними у програмному забезпеченні. Це дозволяє створювати автоматизовані робочі процеси управління ІТ-послугами. Наприклад, автоматизовані інструменти розгортання та налаштування дозволяють адміністраторам визначати колекції віртуальних машин та застосунків як послуги, у програмних шаблонах. Це означає, що вони можуть повторно та послідовно встановлювати ці послуги без важкого, часомісткого та помилкового ручного налаштування. Адміністратори можуть використовувати політики безпеки віртуалізації для обов'язкового визначення певних конфігурацій безпеки на основі ролі віртуальної машини. Політики навіть можуть підвищувати ефективність використання ресурсів, видаляючи не використовувані віртуальні машини для збереження місця та обчислювальної потужності.

- Мінімальний час простою

Падіння операційної системи та застосунків можуть призвести до простою та порушити продуктивність користувачів. Адміністратори можуть запускати кілька резервних віртуальних машин одночасно та автоматично перемикаються між ними, коли виникають проблеми. Запуск кількох резервних фізичних серверів є більш дорогим.

- Швидше надання

Придбання, встановлення та налаштування обладнання для кожного застосунку займає багато часу. Якщо обладнання вже встановлене, надання віртуальних машин для всіх ваших застосунків відбувається значно швидше. Ви

навіть можете автоматизувати це за допомогою програмного забезпечення управління та включити його в існуючі робочі процеси.

Незважаючи на корисність віртуалізації, вона має свої недоліки:

- Зниження продуктивності

Справді, віртуалізація дозволяє оптимально використовувати всі ресурси. Однак це також виклик, коли вам іноді потрібен додатковий толчок, але його немає. Ресурси віртуалізації є спільними. Ті самі ресурси, які може використовувати один користувач, тепер діляться між трьома чи чотирма користувачами. Загальні доступні ресурси можуть не розподілятися рівномірно або можуть розподілятися у певному відношенні в залежності від виконуваних завдань. Зі збільшенням складності завдань збільшується і потреба в продуктивності системи. Це призводить до значного збільшення часу, необхідного для завершення завдання.

- Ризик втрати даних

Працюючи з віртуальними екземплярами на спільних ресурсах, наші дані розміщуються на ресурсі третьої сторони, що ставить наші дані у вразливе становище. Будь-який хакер може атакувати наші дані або намагатися отримати несанкціонований доступ. Без захисного рішення наші дані перебувають у загрозливій ситуації.

1.1.3 Ера розгортання контейнерів

Контейнери схожі на віртуальні машини (ВМ), але вони працюють на спрощеній системі ізоляції та використовують загальну операційну систему для всіх програм. Це робить контейнери більш легкими, порівняно з ВМ. Кожен контейнер

має власну файлову систему, процесор, пам'ять та інші ресурси, подібно до віртуальних машин. Так як контейнери не залежать від певної інфраструктури, їх легко переносити між різними хмарними провайдерами чи операційними системами[1].

Контейнеризація є важливою технологією в сучасній розробці програмного забезпечення, що дозволяє ефективніше створювати, розгортати та підтримувати додатки. Основні переваги контейнеризації включають:

- 1. Портативність** - контейнеризація дозволяє розробникам розгортати програми в різних середовищах без необхідності переписувати код. Програму можна створити один раз і запускати на різних операційних системах, таких як Linux і Windows. Це також спрощує оновлення застарілого коду до сучасних версій.
- 2. Можливість масштабування** - Контейнери є легкими програмними компонентами, які працюють ефективно. Вони запускаються швидше, ніж віртуальні машини, оскільки не потребують завантаження операційної системи. Це дозволяє легко додавати кілька контейнерів для різних програм на одному комп'ютері, використовуючи ресурси однієї загальної операційної системи без взаємного впливу контейнерів.
- 3. Відмовостійкість** - Контейнери сприяють створенню стійких до відмов програм. Використовуючи кілька контейнерів для запуску мікросервісів у хмарі, розробники забезпечують ізолюваність кожного сервісу. Це означає, що несправний контейнер не впливає на інші, підвищуючи стійкість та доступність програми.
- 4. Гнучкість** - Контейнерні програми працюють в ізолюваних обчислювальних середовищах, що дозволяє розробникам усувати неполадки та змінювати код без втручання в операційну систему або інші сервіси. Це прискорює цикли випуску програмного забезпечення та полегшує роботу над оновленнями.

Ці переваги роблять контейнеризацію потужним інструментом для сучасних команд розробників, забезпечуючи ефективність, надійність і гнучкість у розробці та розгортанні програмного забезпечення[2].

Контейнеризація передбачає створення самодостатніх пакетів програмного забезпечення, які працюють стабільно, незалежно від комп'ютерів, на яких вони працюють. Розробники програмного забезпечення створюють та розгортають образи контейнерів, тобто файли, що містять необхідну інформацію для запуску контейнерної програми. Розробники використовують інструменти контейнеризації для створення образів контейнерів на основі специфікації образів Open Container Initiative (OCI). OCI - це проект Linux Foundation зі створення відкритих стандартів віртуалізації на рівні операційної системи, особливо контейнерів Linux[4].

На рисунку 1.1 наведено з чого складається контейнеризація.

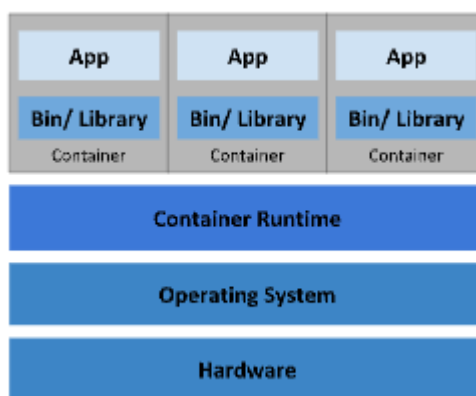


Рисунок 1.1 – Архітектура контейнеризації

1. Hardware - це апаратний рівень моделі контейнерів. Це стосується фізичного комп'ютера або сервера без встановленої операційної системи, на якому виконується контейнерна програма.
2. Операційна система - другий рівень архітектури контейнеризації. Linux – найпопулярніша операційна система для контейнеризації з локальними комп'ютерами.
3. CRI - це програмне забезпечення, яке створює контейнери на основі образів контейнерів. Він виступає посередником між контейнерами і операційною системою, надаючи ресурси, необхідні додатку, і керуючи ними. Наприклад, контейнерні двигуни можуть керувати кількома контейнерами в одній

операційній системі, зберігаючи їхню незалежність від базової інфраструктури та один від одного[3].

4. Додаток та залежності - найвищий рівень архітектури контейнеризації – це код програми та інші файли, які потрібно запустити, такі як залежності бібліотек та пов'язані з ними файли конфігурації.

1.2 Види контейнерної технології

Понад 67,619 компаній використовують інструменти контейнеризації. Docker займає 82.69% ринку (55,913 клієнтів), Google Kubernetes Engine - 11.98% ринку (8,098 клієнтів), CoreOS - 1.36% ринку (920 клієнтів)[6].

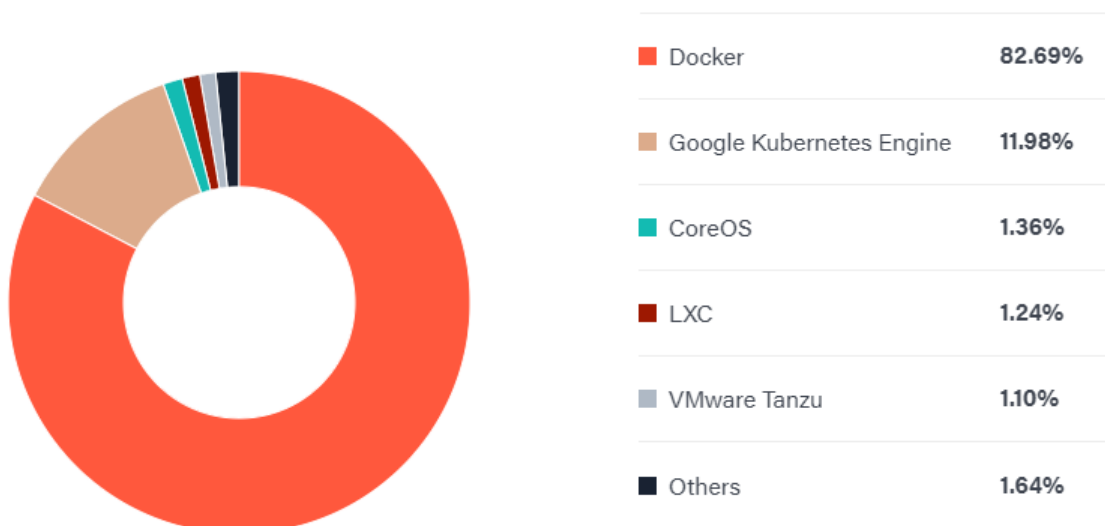


Рисунок 1.2 – Найпопулярніші контейнерні технології

1. Docker використовує технології віртуалізації та контейнеризації. Завдяки цим технологіям Docker виступає як ізольований пісочниця для створення легких контейнерів, що спрощує розробку та розгортання застосунків. Docker –це проєкт з відкритим вихідним кодом, який доступний безкоштовно, хоча існують і платні версії.
2. Google Kubernetes Engine – це платформа з відкритим вихідним кодом від Google для управління контейнерами та супутніми службами. Її головні

переваги: кросплатформність, розширюваність, можливість використання декларативної конфігурації та автоматизації[4].

3. LXC – одна з найдавніших систем для роботи з контейнерами, проте і досі залишається дуже поширеною. Платформа доступна лише для Linux та вже вбудована в Ubuntu. Програма — безплатна. Проте, щоб створювати контейнери, вам знадобиться ядро Linux 3.8 або новіше. LXC складніший у використанні, ніж Docker, що пояснює чому другий є беззаперечним лідером[5].

1.3 Переваги Kubernetes

Kubernetes – це система оркестрації для автоматизованого розгортання, управління, масштабування контейнеризованими застосунками. Давайте розглянемо ключові завдання Kubernetes.

1. Доступність: Використовуючи Kubernetes, ви забезпечуєте високу доступність своїх додатків завдяки автоматизованим процесам, таким як перезапуск несправних контейнерів та перепланування на працюючі вузли. Це гарантує, що ваш додаток не буде простоювати під час розгортання нової версії через відкати та оновлення.
2. Масштабованість: Kubernetes дозволяє легко масштабувати ваш додаток шляхом додавання або видалення модулів відповідно до потреб. Ви можете використовувати автоматичне масштабування, яке реагує на такі характеристики, як завантаження процесора.
3. Портативність: З Kubernetes ви маєте доступ до портативної платформи для управління та розгортання додатків, що працює в різних середовищах, включаючи хмарові обчислення, центри обробки даних та гібридні установки.

Портативність також дозволяє легко переміщувати додатки між різними платформами.

4. Ефективне використання ресурсів: Kubernetes забезпечує оптимізацію ресурсів, пакуючи декілька подів у кожному вузлі. Ви можете встановлювати обмеження ресурсів та запити для контролю за використанням пам'яті та процесорних ресурсів кожним подом.

1.4 Концепція Kubernetes

В цьому підрозділі розглянемо з яких компонентів повинен складатися Kubernetes, представлена архітектура на рис 2.1. Основна архітектура Kubernetes складається з двох частин: master nodes або ще їх називають control nodes та worker nodes. Нода може бути, як фізичною, так і віртуальною. Кожна Kubernetes-нода запускає свої Kubernetes-компоненти, дивлячись яка вона master чи worker.

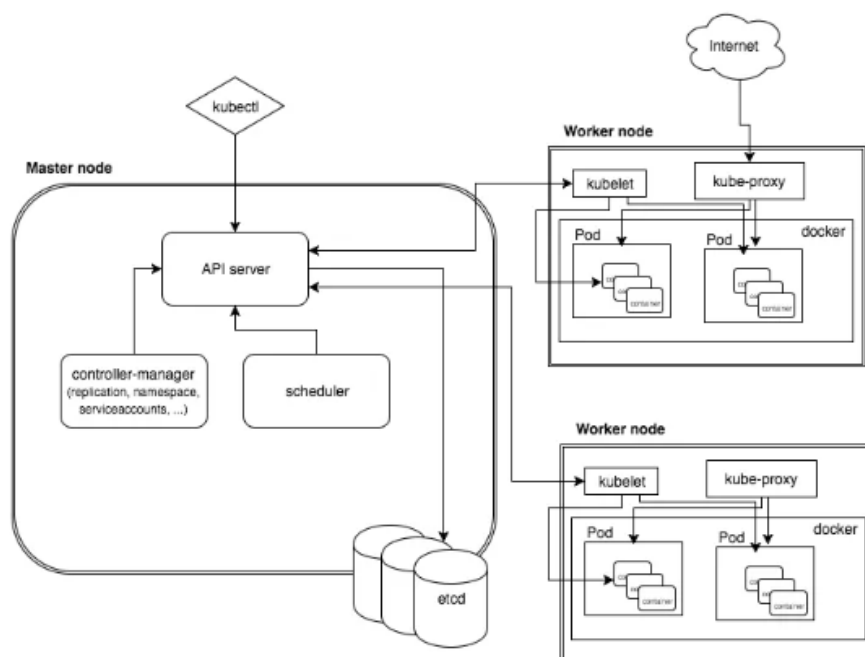


Рисунок 1.3 – Діаграма архітектури Kubernetes, що показує кластер з мастер нодою і двома робочими вузлами

Master-нода є відповідальною за те, щоб управляти кластером. Master-нода Kubernetes-кластеру містить такі компоненти:

- **API server** – це центральний компонент кластера Kubernetes, який отримує REST-запити на модифікацію (щодо подів, сервісів, наборів реплікації/контролерів та інших), виступаючи фронтом для кластера. Крім того, це єдиний компонент, який спілкується з кластером etcd, переконуючись, що дані зберігаються у etcd та узгоджуються з деталями сервісу розгорнутих подів.
- **Scheduler** – це компонент, який допомагає розподіляти поди (згруповані контейнери, всередині яких працюють процеси нашого додатка) на різні вузли на основі використання ресурсів. Він аналізує вимоги до операційної роботи сервісу та розміщує його на найбільш підходящому вузлі. Наприклад, якщо додаток потребує 1 ГБ пам'яті та 2 ядра ЦП, то поди для цього додатка будуть заплановані на вузлі з щонайменше такими ресурсами. Планувальник запускається кожного разу, коли є потреба в розподілі подів. Він повинен знати загальні доступні ресурси, а також ресурси, що виділені наявним завданням на кожному вузлі. Він працює у поєднанні з іншими компонентами керуючого плану Kubernetes, щоб забезпечити ефективне розміщення та розподіл ресурсів у кластері. Такий підхід дозволяє оптимізувати використання ресурсів і підвищувати продуктивність роботи кластера.
- **Controller Manager** – контролер спостерігає за об'єктами, які він керує в кластері. Він спостерігає за ними за їхнім бажаним станом та поточним станом через сервер API. Якщо поточний та бажаний стан об'єктів, якими керує контролер, не збігаються, контролер вживає корекційних заходів, щоб довести стан об'єктів до бажаного.
- **Etcd** – це розподілене сховище даних типу "ключ-значення", яке використовується як основне сховище для зберігання інформації про стан

кластера Kubernetes. Він входить до складу основних компонентів Kubernetes і грає ключову роль у створенні функціонального та стійкого до відмов кластера. Etcd забезпечує можливість моніторингу змін стану кластера за допомогою функції "watch" та переконфігурування Kubernetes у разі виникнення змін. Це дозволяє підтримувати узгодженість між фактичним та бажаним станом кластера та реагувати на будь-які зміни в реальному часі[1].

Далі детальніше про вже робочі вузли (worker nodes), такі компоненти як:

- CRI (Container Runtime Interface) – середовище виконання контейнерів, які, власне, відповідають за запуск контейнерів, зазвичай це docker runtime.
- Pod – є найменшою базовою одиницею, яку можна розгорнути в кластері Kubernetes, і сама платформа Kubernetes побудована на основі концепції пода. Кожен под містить один або кілька контейнерів.
- Kubelet – це програма, яка запускається демоном systemd на кожному вузлі. Вона виконує роль планувальника для модулів пода та агента на локальному вузлі. Він регулярно отримує нові або змінені специфікації подів (переважно через kube-apiserver) і переконуючись, що поди та їхні контейнери працюють в потрібному стані.
- Kube-proxy[7] – це служба-проксі, яка працює на кожному вузлі в ролі забезпечення мережевого зв'язку в кластері Kubernetes. Основна мета kube-proxy полягає в забезпеченні комунікації між різними частинами кластера, що дозволяє додаткам та службам працювати безперебійно.

1.5 Концепція Service

Kubernetes - це потужна платформа для оркестрації контейнеризованих додатків у розподілених середовищах. Однією з ключових функцій Kubernetes є керування мережевими сервісами, яке забезпечує комунікацію між різними компонентами додатків.

Сервіс (Service) у Kubernetes – це абстрактний об'єкт, який визначає логічний набір подів та політику доступу до них. Сервіси створюють зв'язок між подами, які від них залежать.

Хоча кожен под має унікальну IP-адресу, ці адреси не є доступними за межами кластера без використання сервісу. Сервіси дозволяють додаткам отримувати трафік. Сервіси можуть мати різний рівень доступності, залежно від типу сервісу, вказаного в yaml-файлі.

Існує чотири типи сервісів Kubernetes – ClusterIP, NodePort, LoadBalancer та ExternalName.

- ClusterIP – це тип сервісу за замовчуванням і найбільш поширений. Це внутрішній балансувальник, що розподіляє навантаження між модулями Pod у кластері Kubernetes. Ви не можете надсилати запити до сервісу ззовні кластера.
- Сервіс NodePort є розширенням сервісу ClusterIP. Автоматично створюється ClusterIP сервіс, до якого маршрутизується сервіс NodePort. NodePort відкриває сервіс на тому самому порту кожного обраного вузла в кластері за допомогою NAT. Робить сервіс доступним поза кластером через <NodeIP>:<NodePort>. Порт вузла повинен бути в діапазоні від 30000 до 32767.
- Сервіс LoadBalancer є розширенням сервісу NodePort. Цей сервіс інтегрує NodePort з балансувальниками навантаження, що базуються на хмарних ресурсах. Він дозволяє зробити сервіс доступним ззовні за допомогою балансувальника навантаження хмарного провайдера. Кожен хмарний

провайдер (AWS, Azure, GCP тощо) має свою власну реалізацію нативного балансувальника навантаження.

- ExternalName – надає доступ до сервісу завдяки тому, що використовується externalName[8].

1.6 Контролери

Контролери в Kubernetes відіграють важливу роль у забезпеченні стабільності та надійності кластера. Вони відповідають за автоматичне керування різними ресурсами та підтримку їх бажаного стану. Ось кілька ключових контролерів, які забезпечують цю функціональність[9]:

- ReplicaSet: Цей контролер забезпечує гарантовану кількість екземплярів певних подів у кластері. Якщо який-небудь з цих подів відмовляє, ReplicaSet автоматично створює нові, щоб замінити їх. (рис 1.3)

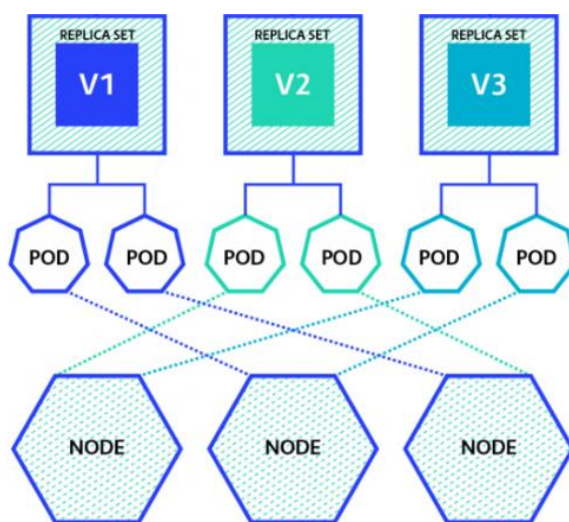


Рисунок 1.4 – Replica Set

- Deployment – відповідає за управління репліками подів. Проще кажучи, це абстракція над ReplicaSet, яка дозволяє вам керувати оновленнями додатків.

Вона гарантує можливість плавного оновлення версії додатка, а також плавного відкочування до попередніх версій у разі проблем[1]. (рис 1.4)

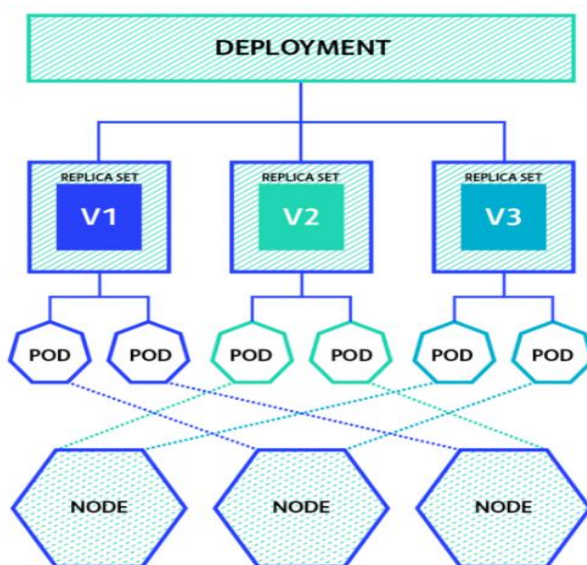


Рисунок 1.5 – Deployment

- StatefulSet – це контролер в Kubernetes, який дозволяє управляти додатками з унікальними ідентифікаторами та стійкими мережевими іменами. Він гарантує, що кожен екземпляр додатку має стабільний ідентифікатор і може зберігати свій стан навіть після перезапуску або масштабування. Наприклад, він корисний для баз даних
- DaemonSet – гарантує, що на кожному вузлі кластера працює один і той самий екземпляр певного пода. Це дозволяє розгортати агенти або служби, які повинні працювати на кожному вузлі кластера.
- Job – це механізм у Kubernetes, який забезпечує виконання конкретного завдання (пода) і успішно завершує його. Job – запускає под як пакетний процес. Коли завдання успішно виконано, Job вважається завершеним. У разі, якщо завдання не виповнилося, Kubernetes може автоматично перезапустити його, щоб забезпечити успішне виконання. CronJob - це планувальник, який дозволяє запускати завдання (поди) періодично, схоже до планувальника завдань в операційних системах.

Висновки до розділу 1

У першому розділі роботи було детально розглянуто поняття системи керування контейнерами та проаналізовано ключові аспекти архітектури і компонентів Kubernetes. Система керування контейнерами є важливим інструментом для сучасної розробки та розгортання програмного забезпечення, забезпечуючи ефективну обробку, масштабованість та ізоляцію додатків. Контейнери, які представляють собою легковагові та автономні пакети програмного забезпечення, спрощують процеси розробки, тестування та розгортання. Вони забезпечують консистентне середовище для запуску додатків, незалежно від того, де ці додатки розгортаються. Архітектура Kubernetes складається з основних компонентів, таких як API-сервер, etcd, контролер-менеджер, планувальник, та вузли (nodes), що включають kubelet та kube-proxy. Кожен з цих компонентів виконує свою унікальну роль у забезпеченні функціонування кластера

2 АНАЛІЗ ВРАЗЛИВОСТЕЙ В СИСТЕМІ KUBERNETES

2.1 Найпопулярніші вразливості

Система Kubernetes є потужним інструментом для управління контейнерами, але, як і будь-яка складна система, вона має свої вразливості. У цьому розділі ми розглянемо основні з них, зосереджуючи увагу на вразливостях, визначених у OWASP Top 10 для Kubernetes, щоб забезпечити комплексний аналіз можливих загроз. У таблиці 2.1 нижче наведено ці вразливості[10].

Таблиця 2.1 – Головні вразливості

Вразливості	Опис
Небезпечні конфігурації робочого навантаження	Неправильні конфігурації призводять до вразливих робочих навантажень.
Надто дозволені конфігурації RBAC	Контроль доступу на основі ролей Kubernetes (RBAC) забезпечують зручний спосіб контролювати доступ до критичних системних ресурсів Kubernetes. Однак, якщо скомпрометовано, надмірно дозволені RBAC, такі як роль адміністратора кластера може призвести до атак на самі ресурси.
Відсутність централізованого забезпечення виконання політики	Неправильне налаштування безпеки через відсутність централізованого автоматизованого керування політикою.

Кінець таблиці 2.1

Помилки в управлінні секретами	Секрети дають цінну інформацію та доступ до різних захищених ресурсів, але в разі порушення може дозволити зловмисник для доступу до цих ресурсів. Kubernetes надає механізми для зберігання та захисту цих секретів, і їх слід використовувати регулярно перевірено на належну конфігурацію.
Неадекватний аудит та моніторинг	Виявлення атак і криміналістика складні без реєстрації подій, орієнтованих на безпеку.
Зламани механізми автентифікації	Несанкціонований доступ до системних ресурсів може призвести до переміщення, пошкодження або крадіжки даних.
Відсутні засоби керування сегментацією мережі	Проблема мережі Kubernetes полягає в тому, що мережевої політики недостатньо складні. Мережеве сканування, тунелювання, командні з'єднання неможливо зупинити.
Неправильно налаштовані компоненти кластера	Неправильна конфігурація компонентів системи, таких як сервер API, kubelet тощо, створює ризики.
Застарілі та вразливі компоненти Kubernetes	Критичні CVE в контейнерах Kubernetes або інших систем (nginx, Istio) призводять до експлойту.
Вразливості ланцюга постачання	Через широке використання пакетів сторонніх розробників багато образів контейнерів за своєю суттю є небезпечними для перетягування в надійне середовище та запуску

2.2 Перелік методів захисту

1. Збереження безпечної конфігурації у розподіленому середовищі Kubernetes може бути складною задачею. Хоча багато параметрів безпеки часто встановлюються в securityContext маніфеста, є ряд інших конфігураційних помилок, які можна виявити в інших місцях. Для запобігання таких помилок їх необхідно виявляти як на етапі виконання, так і на етапі розробки коду. Ми можемо накладати такі вимоги на додатки[10]:

- Запускати додатки з правами непривілейованого користувача.
- Запускати додатки в непривілейованому режимі.
- Встановлювати параметр AllowPrivilegeEscalation: False для заборони дочірнім процесам отримувати більше привілеїв, ніж у їхніх батьківських процесах.
- Встановлювати LimitRange для обмеження виділення ресурсів для кожного відповідного об'єкта в просторі імен (namespace).

```
apiVersion: v1
kind: Pod
metadata:
  name: root-user
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 10001
  containers:
    name: ubuntu-nonroot
    image: ubuntu
```

Рисунок 2.1 – Приклад використання безпечної конфігурації

2. Щоб зменшити ризик зловживання налаштуваннями RBAC з боку атакувальників, важливо постійно аналізувати налаштування і забезпечувати

завжди виконання принципу найменшого привілею. Ось деякі рекомендації[10]:

- Зменшуйте прямий доступ до кластера для кінцевих користувачів, якщо це можливо.
- Не використовуйте токени облікових записів служб поза межами кластера.
- Уникайте автоматичного монтування токенів стандартної службової облікової записи.
- Проводьте аудит RBAC, включеного в встановлені сторонні компоненти.
- Розгортайте централізовані політики для виявлення і блокування небезпечних дозволів RBAC.
- Використовуйте RoleBindings для обмеження обсягу дозволів на конкретні простори імен порівняно з політиками RBAC на рівні всього кластера.
- Дотримуйтесь рекомендацій щодо RBAC у офіційній документації Kubernetes.

3. Централізоване забезпечення виконання політик є важливим для забезпечення безпеки кластерів Kubernetes. Без нього немає способу гарантувати, що всі ресурси в кластері дотримуються одних і тих же політик. Це може призвести до вразливостей безпеки, таких як несанкціонований доступ, перевикористання ресурсів та неправильна конфігурація. Наприклад, якщо адміністратор встановлює політику обмеження кількості подів, які можуть працювати на вузлі, а інший адміністратор розгортає більше подів, ніж це дозволяє ліміт, це може призвести до вичерпання ресурсів і збоїв у роботі служб. Централізоване забезпечення виконання політик гарантує, що такі ситуації не виникають, забезпечуючи однорідний набір політик для всього кластера. Найкращі практики для централізованого забезпечення виконання політик у Kubernetes такі:

- **Визначення політик:** Перший крок у централізованому забезпеченні виконання політик - це визначення політик для мережевої безпеки, контролю доступу, квот ресурсів та інших аспектів. Політики повинні базуватись на найкращих практиках галузі і враховувати конкретні потреби організації.
- **Автоматизація виконання політик:** Автоматизація є ключовою для забезпечення однорідного виконання політик по всьому кластеру. Це можна зробити за допомогою інструментів, таких як оператори Kubernetes або робочі процеси GitOps, які автоматично застосовують політики відповідно до їх визначення.

4. Щоб впровадити найкращі практики управління секретами Kubernetes існують такі практики:

- **Використовуйте шифрування:** шифруйте секрети, що зберігаються у зовнішній системі керування секретами.
- **Використовуйте механізми шифрування для захисту секретів у стані спокою та під час передачі.**
- **Зовнішнє управління секретами - Зберігання секретних ключів поза кластером** Kubernetes за допомогою менеджера секретів пропонує безліч переваг: посилений захист за рахунок зменшення вразливості та обмеження доступу до секретів лише під час виконання програми, легша ротація секретів, оскільки вони знаходяться поза кластером, що виключає прості, а також додаткова абстракція за рахунок централізованого, безпечного сховища для зберігання секретів та доступу до них.
- **Регулярно робіть ротацію секретів:** Вимога відповідності вказує на необхідність встановлення політики та процесу регулярної ротації секретів, що зберігаються в зовнішній системі управління секретами. Автоматизуйте процес ротації, де це можливо, щоб забезпечити своєчасне оновлення та мінімізувати ручні помилки.

- **Контроль та аудит:** Впровадьте механізми реєстрації та моніторингу для відстеження доступу до секретів і виявлення будь-якої підозрілої активності. Регулярно переглядайте журнали та встановіть механізми оповіщення для швидкого реагування на інциденти безпеки. Хорошим способом досягнення цих цілей є ефективне управління ідентифікацією, яке гарантує, що тільки авторизовані користувачі мають доступ до секретів.
- **Автоматизуйте розповсюдження секретів:** Використовуйте власні механізми Kubernetes, такі як змінні середовища або змонтовані файли, для автоматизації розподілу секретів між відповідними компонентами. Автоматизація покращує узгодженість, зменшує кількість людських помилок та спрощує керування секретами[20].

5. Ведення журналів і моніторинг є важливими компонентами безпеки Kubernetes, забезпечуючи видимість стану, продуктивності і безпеки кластера. Належне ведення журналів дозволяє збирати та аналізувати журнали з різних компонентів Kubernetes.

- **Увімкніть детальне ведення журналів:** Переконайтеся, що ведення журналу увімкнено для всіх відповідних компонентів Kubernetes, таких як модулі, контейнери, вузли та компоненти площини керування. Налаштуйте ведення журналів для збору детальних журналів, зокрема системних журналів, журналів програм і журналів, пов'язаних із безпекою.
- **Налаштуйте централізований збір логів:** Реалізуйте централізований механізм збору та агрегації логів за допомогою таких інструментів, як Elasticsearch, Fluentd і Kibana. Налаштуйте збір та агрегацію логів для безпечного зберігання логів та їх ефективного аналізу для виявлення подій безпеки.

- Проведення аудиту: Впровадьте механізми аудиту для відстеження та моніторингу змін у конфігурації та налаштуваннях кластера.
Використовуйте вбудовані функції аудиту Kubernetes або сторонні інструменти, такі як Falco, щоб переконатися, що всі зміни перевіряються та перевіряються на відповідність вимогам безпеки.
- Протестуйте реагування на інциденти: Розробіть план реагування на інциденти, який включає заздалегідь визначені ролі, обов'язки та процедури для обробки інцидентів безпеки. Регулярно проводьте навчання та симуляції, щоб перевірити план реагування на інциденти та переконатися, що IT-команда готова ефективно реагувати на порушення безпеки.

6. Уникайте використання сертифікатів для автентифікації кінцевого користувача. Сертифікати зручні для автентифікації в Kubernetes API, але їх слід використовувати з особливою обережністю. API не має можливості відкликати сертифікати, що призводить до необхідності перешифрування кластера у випадку компрометації або витоку приватних ключів. Сертифікати також складніше налаштовувати, підписувати та розповсюджувати. Сертифікат можна використовувати як механізм автентифікації "розбитого скла", але не для первинної автентифікації. Ніколи не використовуйте власну автентифікацію. Так само, як і в криптографії, ви не повинні створювати щось нове, коли в цьому немає необхідності. Використовуйте те, що підтримується і широко застосовується. Використовуйте багатофакторну автентифікацію (MFA), коли це можливо. Незалежно від обраного механізму автентифікації, змушуйте людей надавати другий метод автентифікації, зазвичай це частина OpenID Connect (OIDC). Не використовуйте токени службових облікових записів за межами кластера. Для використання всередині кластера токени Kubernetes Service Account отримуються безпосередньо за допомогою API TokenRequest і монтуються в Pods з використанням передбачуваного обсягу. Для використання поза кластером, ці токени повинні бути надані вручну через

Kubernetes Secret і не мають терміну дії. Використання токенів Service Account з тривалим терміном дії за межами кластера наражає кластер на значні ризики. Якщо потрібен підхід на основі токенів, короткострокові токени можна створити за допомогою API TokenRequest або за допомогою команди *kubectl create token* з прапором *--duration*.

7. Щоб запобігти проблемам сегментації мережі та забезпечити безпечне, добре організоване середовище Kubernetes, дотримуйтесь цих вказівок:
 - Визначення мережевих політик. Використовуйте Kubernetes NetworkPolicy для визначення способу взаємодії подів між собою та іншими кінцевими точками мережі.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-backend-egress
  namespace: default
spec:
  podSelector:
    matchLabels:
      tier: backend
  policyTypes:
    - Egress
  egress:
    - to:
        - podSelector:
            matchLabels:
              tier: backend
```

Рисунок 2.2 – Приклад використання Native Controls (NetworkPolicies)

8. Неправильні конфігурації основних компонентів Kubernetes можуть призвести до повного компрометації кластера або навіть до гірших наслідків. Тому проведення регулярних сканувань та аудитів за стандартами CIS Benchmark, зосереджених на неправильних конфігураціях компонентів буде хорошим початком. Використання керованих Kubernetes, таких як EKS або GKE, також може допомогти обмежити деякі варіанти конфігурації компонентів та спрямувати операторів до більш безпечних налаштувань за замовчуванням.

9. Через велику кількість стороннього програмного забезпечення, що працює всередині кластера Kubernetes, потрібен багатогранний підхід для усунення вразливих компонентів:

- Відстеження баз даних CVE: Перш за все, Kubernetes та пов'язані компоненти не можна залишати поза процесом сканування вразливостей CVE.
- Мінімізація залежностей від стороннього програмного забезпечення: Усе стороннє програмне забезпечення слід незалежно перевіряти перед розгортанням на наявність надмірно дозволеного доступу RBAC, низькорівневого доступу до ядра та історичних записів про вразливості.

10. У процесі розробки програмного забезпечення контейнерні зображення є набором програмних артефактів та метаданих, які передаються від виробника до споживача через складний CI/CD процес або прямо в Kubernetes з інтегрованого середовища розробника. Забезпечення цілісності цих зображень критично важливе на кожному етапі: від використання атестацій за стандартом in-toto для перевірки цілісності до генерації списків програмних матеріалів (SBOM) за стандартами CycloneDX чи SPDX для документування включених пакетів програмного забезпечення та ліцензій. Кожен крок у постачальному ланцюзі вимагає підписування та перевірки зображень за допомогою криптографічних ключів, щоб виявити потенційні втручання або несанкціоновані зміни. Рекомендується використовувати мінімальні пакети ОС та обирати базові зображення, такі як Distrolless або Scratch, для зменшення атакованої поверхні та покращення безпеки. Важливо також систематично сканувати зображення на відомі вразливості за допомогою інструментів, таких як Clair або Trivy, та оперативно виправляти знайдені проблеми. Інтеграція цих практик допомагає забезпечити безпеку та надійність додатків в середовищі Kubernetes.

Висновки до розділу 2

У другому розділі дипломної роботи були проаналізовані потенційні вразливості в системах Kubernetes і розглянуті відповідні методи захисту. Кожна із виявлених уразливостей має відповідний метод захисту, який нативно впроваджений в систему Kubernetes для безпечного використання. Загальні рекомендації включають:

1. Мережеві політики (Network Policies): Використання для обмеження трафіку між компонентами кластера і захисту від атак з боку мережі.
2. RBAC (Role-Based Access Control): Встановлення точних прав доступу для керування ресурсами і обмеження несанкціонованого доступу.
3. Моніторинг і аудит безпеки: Систематичний аналіз і реагування на аномальні події для своєчасного виявлення потенційних загроз.
4. Використання безпечних базових образів (Base Images): Використання мінімальних образів з налаштуваннями безпеки (наприклад, Distrosless або Scratch) для зменшення атакованої поверхні.
5. Автоматизовані сканування вразливостей: Використання інструментів для сканування зображень контейнерів на наявність відомих вразливостей і оновлення з виправленням вразливих компонентів.

Ці заходи спрямовані на ефективний захист систем Kubernetes від потенційних загроз і забезпечення безпеки під час їх експлуатації у великих інформаційних системах.

3 ВДОСКОНАЛЕННЯ БЕЗПЕКИ KUBERNETES

3.1 Впровадження плагіну Cilium

3.1.1 Container Networking Interface

CNI – це стандартний інтерфейс для підключення мережі до контейнерів. CNI був створений для того, щоб забезпечити стандартизований спосіб налаштування мережі для контейнерів, що дозволяє різним мережевим рішенням легко інтегруватися з платформами для оркестрації контейнерів, такими як Kubernetes[11].

Cilium є одним із популярних плагінів CNI, що забезпечує мережеву взаємодію, безпеку та доступність для хмарних середовищ, таких як Kubernetes та інші платформи для оркестрації контейнерів. В основі Cilium лежить технологія ядра Linux під назвою eBPF, яка дозволяє динамічно впроваджувати потужну логіку безпеки, видимості та мережевого управління в ядро цієї ОС[12].

Cilium активно використовується для забезпечення високої продуктивності та гнучкості мережесхем. Його інтеграція з Kubernetes дозволяє спростувати управління мережевими політиками та покращувати безпеку контейнерних середовищ. Крім того, eBPF надає можливість знижувати затримки та підвищувати ефективність роботи мережесхем додатків.

Cilium багатифункціональний, розрахований на сучасні запити розробників, насичений великою кількістю функцій. eBPF — головна особливість Cilium. eBPF — це революційна технологія, що зародилася в ядрі Linux, яка дозволяє запускати ізольовані програми в привілейованому контексті. Вона використовується для безпечного та ефективного розширення можливостей ядра без необхідності змінювати вихідний код ядра або завантажувати модулі ядра.

Програми BPF, міняючи додаткові рівні абстракції, працюють безпосередньо в ядрі Linux, що робить їх дуже швидкими. Це відбувається через відсутність копіювання пакетів з привілейованого режиму в режим юзера, а також обхід стандартних ланцюгів iptables.

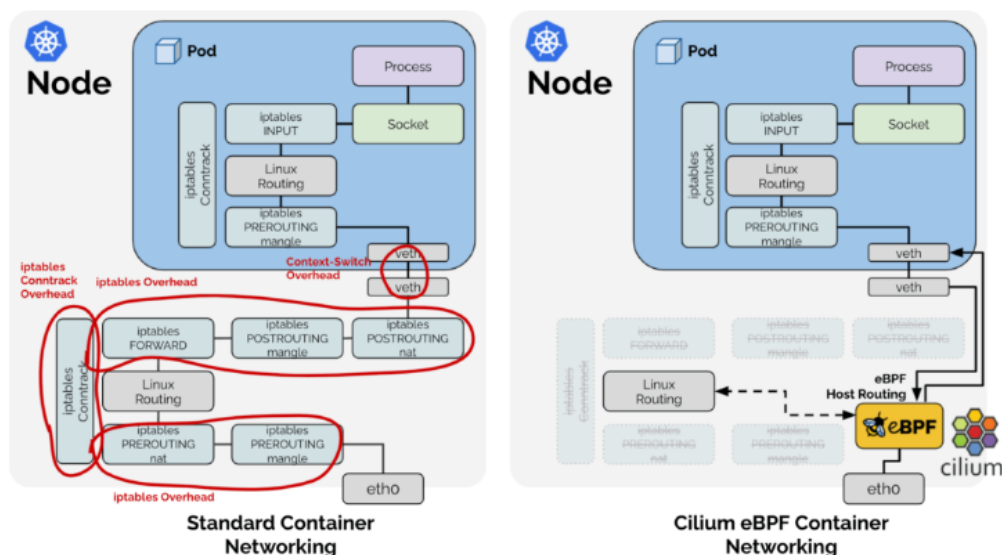


Рисунок 3.1 – Порівняння стандартного мережевого підключення контейнерів та eBPF

Ось деякі найкращі практики та корисні поради для безпечного використання Cilium:

- Використовуйте розширені можливості мережевої політики Cilium для визначення та застосування детальних заходів безпеки. Використовуйте CiliumNetworkPolicies для створення політик, які розуміють протоколи та концепції на рівні застосувань. Регулярно переглядайте та оновлюйте мережеві політики, щоб адаптуватися до змін у вимогах безпеки.
- Ретельно плануйте та налаштовуйте функції eBPF. Набір функцій Cilium, що працює на основі eBPF, забезпечує розширені можливості мереж та безпеки. Плануйте та налаштовуйте ці функції з урахуванням ваших конкретних вимог та впливу на продуктивність і безпеку. Увімкніть і вимкніть функції за потребою та ретельно тестуйте конфігурацію перед розгортанням у продуктивне середовище.
- Скористайтеся можливостями спостереження Cilium, щоб отримати видимість мережевого трафіку. Використовуйте інтеграцію Cilium з платформою спостереження Hubble для моніторингу та аналізу мережевих потоків, метрик

на рівні застосувань і подій безпеки. Регулярно переглядайте та аналізуйте зібрані дані для виявлення будь-яких аномалій або потенційних загроз безпеці.

- Оскільки Cilium значною мірою залежить від eBPF і взаємодіє з ядром Linux, забезпечте належні заходи безпеки для цих компонентів. Оновлюйте ядро Linux з останніми патчами та оновленнями безпеки. Впроваджуйте відповідні заходи контролю доступу, механізми ізоляції та конфігурації безпеки для захисту програм eBPF і базового ядра[13].

3.1.2 Налаштування мережевих політик

Мережеві політики в Kubernetes дозволяють контролювати трафік між подами на різних рівнях мережевої моделі OSI. Політики рівня L3 (мережевий рівень) та L4 (транспортний рівень) використовують IP-адреси для визначення джерела та призначення трафіку. L3 політики контролюють трафік на основі IP-адрес і субмереж, тоді як L4 політики враховують порти та протоколи (TCP, UDP). Cilium пропонує підтримку політик рівня L7, що не є нативною функцією Kubernetes. Політики рівня L7 (прикладний рівень) працюють на рівні додатків і можуть аналізувати вміст трафіку, такий як HTTP-запити, для прийняття рішень про дозволений чи заборонений трафік. Вони можуть аналізувати HTTP -запити, включаючи методи (GET, POST). Основні характеристики L7 політик включають детальний контроль над трафіком, забезпечення безпеки на рівні додатків та можливість дозволяти лише певні типи запитів до API. Для налаштування мережевих політик, спочатку створимо кластер[12].

```

kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  extraPortMappings:
  - containerPort: 32042
    hostPort: 32042
  - containerPort: 31234
    hostPort: 31234
  - containerPort: 31235
    hostPort: 31235
  extraMounts:
  - hostPath: /opt/images
    containerPath: /opt/images
- role: worker
  extraMounts:
  - hostPath: /opt/images
    containerPath: /opt/images
- role: worker
  extraMounts:
  - hostPath: /opt/images
    containerPath: /opt/images
networking:
  disableDefaultCNI: true

```

Рисунок 3.2 – Конфігураційний файл кластера

Кластер складається з трьох вузлів: одного вузла control-plane, який виконує роль керуючої площини Kubernetes та працює з etcd, і двох робочих вузлів, призначених для розгортання додатків. У розділі мережових налаштувань конфігураційного файлу за замовчуванням вимкнено CNI, тому кластер не матиме мережі Pod під час запуску. Замість цього для надання цієї функціональності в кластер встановимо Cilium.

```

(yura@kali)-[~]
$ cilium install
🌀 Auto-detected Kubernetes kind: kind
🌀 Running "kind" validation checks
✓ Detected kind version "0.23.0"
i Using Cilium version 1.15.5
🌀 Auto-detected cluster name: kind-kind
🌀 Auto-detected kube-proxy has been installed

```

Рисунок 3.3 – Результат команди cilium install

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: apiconfig
  labels:
    app: api
    type: production
data:
  .yaml: |+
    simlet: secret

    request:
    - method: GET
    - uriPathPattern: /secret/{secret}

    Secret:
    is: parameter
    from: uriPathPattern
    pattern: /secret/{secret}

    Value:
    is: parameter
    from: list
    list: ['root']
    pick: 1

    response:
    from: template
    status: 200
    headers:
    - "Content-Type: application/json"
    body: `
    {
      "name": "${ Secret }",
      "secret": "${ Value }"
    }
    `

apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: api
    type: production
  name: api
spec:
  replicas: 1
  selector:
    matchLabels:
      app: api
      type: production
  template:
    metadata:
      labels:
        app: api
        type: production

```

(a)

```

spec:
  containers:
    - command:
      - apisimulator
      - start
      - /config
      image: apimastery/apisimulator:latest
      name: apisimulator
      lifecycle:
        preStop:
          exec:
            command:
            - apisimulator
            - stop
            - /config
      ports:
        - name: http
          containerPort: 6090
      volumeMounts:
        - mountPath: /config
          name: api-config
      volumes:
        - name: api-config
          configMap:
            name: apiconfig

apiVersion: v1
kind: Service
metadata:
  labels:
    app: api
    type: production
  name: api
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: http
  selector:
    app: api
    type: production

apiVersion: v1
kind: Pod
metadata:
  labels:
    app: client
    type: production
  name: api-client
spec:
  containers:
    - command:
      - sleep
      - infinity
      image: nginx:alpine
      imagePullPolicy: IfNotPresent
      name: nginx

```

(б)

Рисунок 3.4 (а,б) – Створення структури кластера

Цей YAML-файл описує наступні ресурси для Kubernetes: ConfigMap з ім'ям `apiconfig`, який містить файл `5.yaml` з правилами обробки HTTP-запитів та формування відповідей для симулятора API. Deployment з ім'ям `api`, який створює один реплікаційний под з контейнером `apisimulator`, що використовує `image apisimulator`, монтує конфігурацію з ConfigMap `apiconfig` в `/config`, і слухає на порту 6090. Service з ім'ям `api`, який відкриває порт 80 та направляє трафік до контейнера `apisimulator` на порт 6090 та Pod з ім'ям `api-client`, який створює контейнер `nginx` для тестування доступу до API. Протестуємо HTTP-запити.

```
(yura@kali)-[~]
$ kubectl exec -it api-client -- curl -H 'X-Security-Header: mysecret' api/secret/mysecret
{
  "name": "mysecret",
  "secret": "root"
}

(yura@kali)-[~]
$ kubectl exec -it api-client -- curl api/secret/mysecret
{
  "name": "mysecret",
  "secret": "root"
}
```

Рисунок 3.5 – HTTP-запити різними способами без мережевої політики
Оскільки ми не визначили L7 NetworkPolicies, API доступний без обмежень.

```
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: allow-api-ingress
spec:
  endpointSelector:
    matchLabels:
      app: api
      type: production
  egress:
  - {}
  ingress:
  - fromEndpoints:
    - matchLabels:
        app: client
        type: production
    toPorts:
    - port: "6090"
      protocol: TCP
    rules:
      http:
      - method: "GET"
        path: "/secret/*"
        headers:
        - 'X-Security-Header: mysecret'
```

Рисунок 3.6 – Мережева політика

Цей YAML-файл визначає політику мережі Cilium для кластера Kubernetes. Вона дозволяє вхідний трафік на порт 6090 з методом GET та шляхом `/secret/.*` тільки від кінцевих точок з мітками `app: client` та `type: production`, за умови наявності заголовка `X-Security-Header` зі значенням `"mysecret"`.

Відкриємо Hubble та подивимось зміни. Hubble - інструмент моніторингу мережі для Kubernetes, який дозволяє вам візуалізувати та аналізувати трафік вашого кластера Kubernetes в реальному часі.

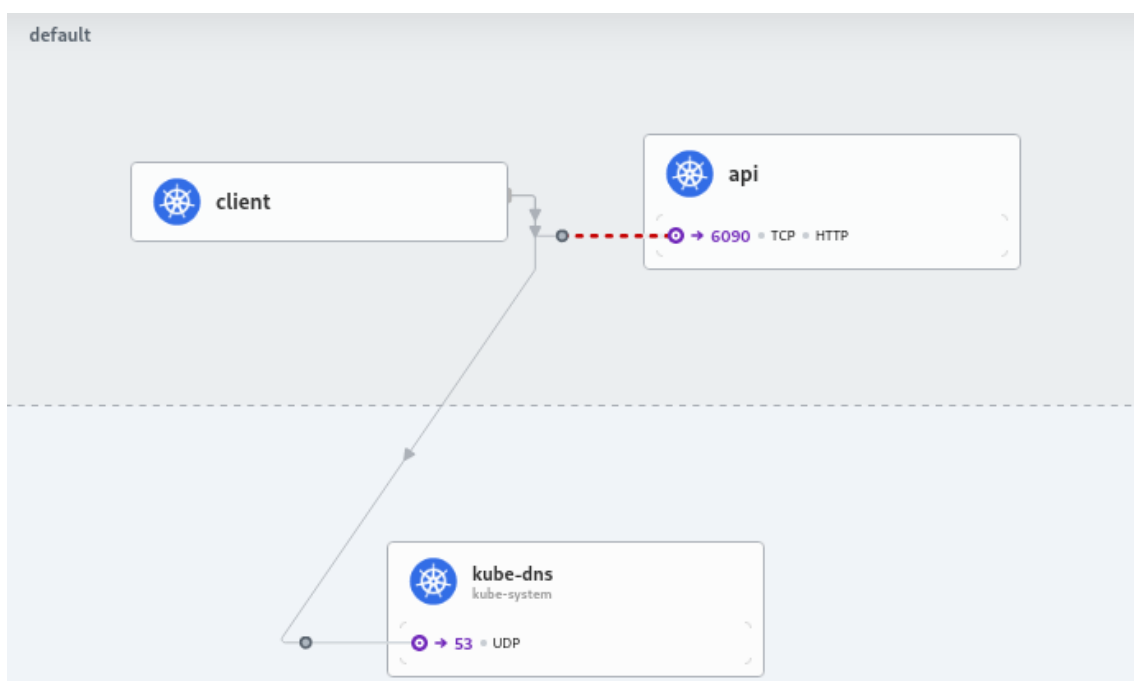


Рисунок 3.7 – Hubble UI

```
(yura@kali)-[~]
$ kubectl exec -it api-client -- curl api/secret/mysecret
Access denied

(yura@kali)-[~]
$ kubectl exec -it api-client -- curl -H 'X-Security-Header: mysecret' api/secret/mysecret
{
  "name": "mysecret",
  "secret": "root"
}
```

Рисунок 3.8 – HTTP-запити різними способами з мережевою політикою

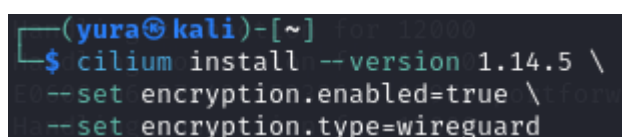
Як ми бачимо, мережева політика L7 рівня успішно спрацювала.

3.1.3 Шифрування трафіку за допомогою WireGuard

Kubernetes не пропонує шифрування даних під час передачі. Щоб забезпечити можливість шифрування, часто потрібно впроваджувати його безпосередньо в додатки або розгортати Service Mesh. Обидва варіанти додають складності та оперативних труднощів.

Cilium пропонує інтеграцію з IPsec та WireGuard для забезпечення шифрування даних під час передачі між вузлами та подами кластера. Коли WireGuard увімкнено в Cilium, агент, запущений на кожному вузлі кластера, створить захищений тунель WireGuard між ним та всіма іншими відомими вузлами в кластері. Кожен вузол автоматично створює власну пару ключів шифрування та розповсюджує свій відкритий ключ через анотацію в об'єкті спеціального ресурсу `network.cilium.io/wg-pub-key` Kubernetes.

Пакети не шифруються, якщо вони призначені на той самий вузол, з якого вони були надіслані. Така поведінка призначена. У цьому випадку шифрування не принесе жодних переваг, враховуючи, що необроблений трафік все одно можна спостерігати на вузлі. Кінцева точка тунелю WireGuard доступна через порт UDP 51871 на кожному вузлі[14].



```
(yura@kali)-[~]
$ cilium install --version 1.14.5 \
  --set encryption.enabled=true \
  --set encryption.type=wireguard
```

Рисунок 3.9 – Встановлення Cilium з включеним параметром шифрування

Однією з переваг WireGuard перед IPsec є те, що кожен вузол автоматично створює власну пару ключів шифрування і поширює свій публічний ключ через анотацію `network.cilium.io/wg-pub-key` в об'єкті користувацького ресурсу `CiliumNode` у Kubernetes. Ви можете перевірити це, переглянувши анотацію, наприклад, на вузлі `Cilium kind-worker2`, який містить публічний ключ.

```
(yura@kali)-[~]
$ kubectl get ciliumnode kind-worker2 \
-o jsonpath='{.metadata.annotations.network\.cilium\.io/wg-pub-key}'
JxCs0IE2FAJq9uADpLwSPhu0CZrWCFC/CnVjZT9hJwM=
```

Рисунок 3.10 – Публічний ключ на ноді kind-worker2

Дізнаємося назву Cilium агента на ноді kind-worker2.

```
(yura@kali)-[~]
$ CILIUM_POD=$(kubectl -n kube-system get po -l k8s-app=cilium --field-selector spec.nodeName=kind-worker2 -o name)
echo $CILIUM_POD
pod/cilium-7pzzq
```

Рис 3.11 – назва Cilium агента на ноді kind-worker2

Зайдемо в термінал агента та перевіримо чи встановили ми Wireguard.

```
(yura@kali)-[~]
$ kubectl -n kube-system exec -ti $CILIUM_POD -- bash
Defaulted container "cilium-agent" out of: cilium-agent, config (init), mount-cgroup (init), apply-sysctl-overwrites (init), mount-bpf-fs (init), clean-cilium
root@kind-worker2:/home/cilium# cilium status | grep Encryption
Encryption: Wireguard [NodeEncryption: Disabled, cilium_wg0 (Pubkey: JxCs0IE2FAJq9uADpLwSPhu0CZrWCFC/CnVjZT9hJwM=, Port: 51871, Peers: 2)]
```

Рисунок 3.12 – Результат команди `kubectl -n kube-system -ti $CILIUM_POD -- bash`

Давайте коротко пояснимо цей запис: У нас є 2 однорангові вузли: агент, який працює на кожному вузлі кластера, встановив захищений тунель WireGuard між собою та всіма іншими відомими вузлами в кластері. Інтерфейс тунелю WireGuard називається `cilium_wg0`. Кінцеві точки тунелю WireGuard доступні через UDP-порт 51871. Зверніть увагу, що значення відкритого ключа таке ж, як на рис 3.10. NodeEncryption (можливість шифрувати трафік між вузлами Kubernetes) вимкнено.

```
root@kind-worker2:/home/cilium# tcpdump -n -i cilium_wg0
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on cilium_wg0, link-type RAW (Raw IP), snapshot length 262144 bytes
16:52:59.197036 IP 10.244.2.166.56616 > 10.244.1.97.6090: Flags [S], seq 1386819889, win 32430, options [mss 1410,sackOK,TS val 1789683094 ecr 0,nop,wscale 7],
16:52:59.197190 IP 10.244.1.97.6090 > 10.244.2.166.56616: Flags [S.], seq 4185923359, ack 1386819890, win 32154, options [mss 1410,sackOK,TS val 218052738 ecr 1
16:52:59.197966 IP 10.244.2.166.56616 > 10.244.1.97.6090: Flags [.], ack 1, win 254, options [nop,nop,TS val 1789683095 ecr 218052738], length 0
16:52:59.198303 IP 10.244.2.166.56616 > 10.244.1.97.6090: Flags [P.], seq 1:111, ack 1, win 254, options [nop,nop,TS val 1789683096 ecr 218052738], length 110
16:52:59.198539 IP 10.244.1.97.6090 > 10.244.2.166.56616: Flags [.], ack 111, win 251, options [nop,nop,TS val 218052739 ecr 1789683096], length 0
16:52:59.302602 IP 10.244.1.97.6090 > 10.244.2.166.56616: Flags [P.], seq 1:184, ack 111, win 251, options [nop,nop,TS val 218052843 ecr 1789683096], length 183
16:52:59.305697 IP 10.244.2.166.56616 > 10.244.1.97.6090: Flags [.], ack 184, win 253, options [nop,nop,TS val 1789683203 ecr 218052843], length 0
16:52:59.305925 IP 10.244.2.166.56616 > 10.244.1.97.6090: Flags [F.], seq 111, ack 184, win 253, options [nop,nop,TS val 1789683203 ecr 218052843], length 0
16:52:59.313665 IP 10.244.1.97.6090 > 10.244.2.166.56616: Flags [F.], seq 184, ack 112, win 251, options [nop,nop,TS val 218052855 ecr 1789683203], length 0
16:52:59.349641 IP 10.244.2.166.56616 > 10.244.1.97.6090: Flags [.], ack 185, win 253, options [nop,nop,TS val 1789683240 ecr 218052855], length 0
```

Рисунок 3.13 – Результат команди `tcpdump -n -i cilium_wg0`

Запустивши утиліту `tcpdump`, було проведено захоплення трафіку на інтерфейсі тунелю WireGuard, `cilium_wg0`, паралельно відправивши запит в іншому терміналі `kubectl exec -it api-client -- curl -H 'X-Security-Header: mysecret' api/secret/mysecret`. Трафік між подами на різних вузлах надсилається через тунелі WireGuard і тому зашифрований. Ось наскільки просте прозоре шифрування з використанням WireGuard із Cilium.

3.1.4 Використання mTLS

Взаємна автентифікація — це процес, у якому дві сторони, відправник і одержувач, ідентифікують особу одна одною, щоб переконатися, що обидві спілкуються зі стороною, з якою вони мають намір спілкуватися. Це не слід плутати з цілісністю та конфіденційністю. Цілісність гарантує, що повідомлення, якими обмінюється, не було підроблено. Конфіденційність гарантує, що повідомлення залишатимуться конфіденційними. Часто вважають, що «шифрування» гарантує всі три, але вони можуть надходити і надходять окремо. Насправді ми всі щодня використовуємо TLS для забезпечення конфіденційності, цілісності та автентифікації сервера, але зазвичай не покладаємося на взаємну автентифікацію, тобто сеанс TLS гарантує, що ми спілкуємося з потрібним сервером, але сервер не обов'язково має підтвердження, що спілкується з автентичним клієнтом. Взаємна автентифікація додає ще один рівень безпеки, забезпечуючи, що і сервер, і клієнт можуть впевнено підтвердити особу один одного. Відображаючи мережеву комунікацію концепції деталізації, ми можемо розглянути два типові режими виконання взаємної автентифікації: сеансова та мережева автентифікація[15].

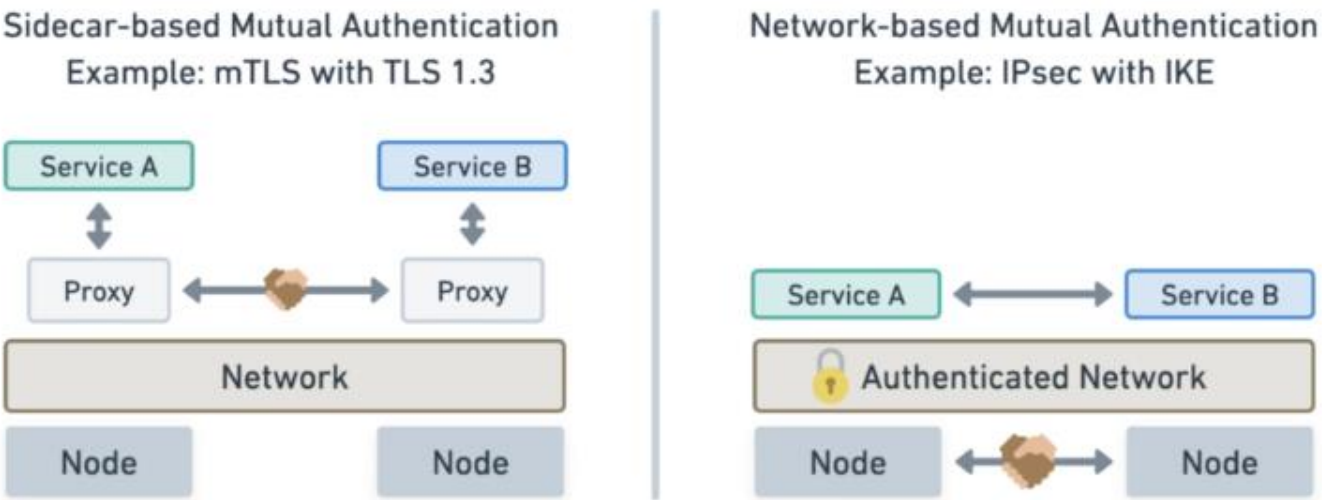


Рис 3.14 – Архітектура сеансової та мережевої автентифікації

Таблиця 3.1 – Порівняння сеансової та мережевої автентифікації

mTLS is Sidecar	Аутентифікація вузла за допомогою IPsec/WireGuard
Впроваджений проксі-сервер виконує взаємне рукостискання автентифікації за допомогою TLS 1.3 для кожного з’єднання між службами.	Вузли автентифікують один одного та інкапсують увесь мережевий трафік, щоб створити автентифіковану та зашифровану віртуальну мережу.
Плюси. Ідентифікація/сертифікат на рівні обслуговування забезпечує кращий рівень безпеки. TLS має сучасне рукостискання з низькою затримкою, яке спеціально розроблено для перетину складних мереж, як і для Інтернету.	Плюси. Повністю прозорий для додатків, без модифікації модулів. Здатність підтримувати будь-який мережевий протокол. Значно ефективніша (в 3 рази краща затримка, у 2 рази краща кількість запитів/с) і масштабована, оскільки не потребує додаткових проксі-серверів.

Кінець таблиці 3.1

Мінуси. Потрібне ін'єкції побічної коляски, що спричиняє погіршення продуктивності та додаткову складність. Складно або неможливо підтримувати інші протоколи, окрім TCP і QUIC. Витік топології програми та невідтримуваних протоколів у базову мережу.	Мінуси. Порушена ідентифікація/сертифікат вузла призводить до зламаних робочих навантажень на вузлі. Обмежена сумісність із системами керування ідентифікацією на основі сертифікатів TLS, такими як SPIFFE.
--	--

Концепція ідентифікації, вбудована в Cilium, дозволяє ідентифікувати служби та впроваджувати мережеві політики, що робить її ідеальною основою для інтеграції з розширеними системами керування ідентифікацією та сертифікатами, такими як SPIFFE, Vault, SMI, cert-manager або Istio. Це дозволяє використовувати існуючі рішення для керування ідентифікацією та сертифікатами для створення та керування ідентифікацією служб і їх сертифікатами. Потім ці сертифікати можуть бути використані для взаємної аутентифікації між ідентифікаторами Cilium, що представляють модулі.

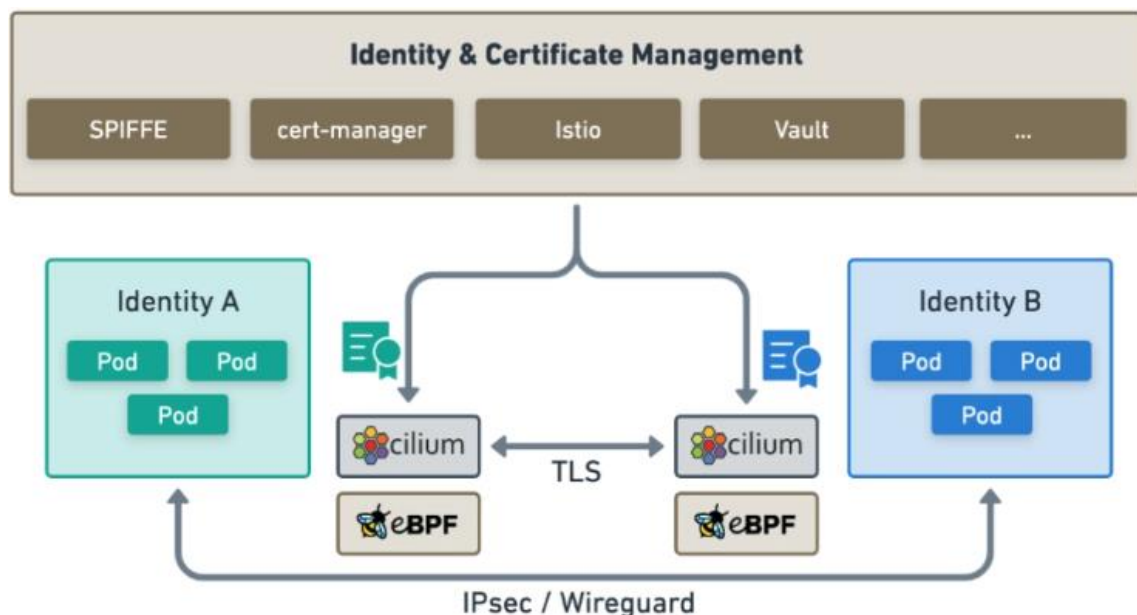


Рисунок 3.15 – Взаємна автентифікація за допомогою Cilium

При встановленні Cilium з включеним параметром *spire*, автоматично розгортається SPIRE server, детально подивимось *cilium config*.

```
(yura@kali)-[~]
$ cilium config view | grep mesh-auth
mesh-auth-enabled                true
mesh-auth-gc-interval            5m0s
mesh-auth-mutual-enabled         true
mesh-auth-mutual-listener-port   4250
mesh-auth-queue-size             1024
mesh-auth-rotated-identities-queue-size 1024
mesh-auth-spiffe-trust-domain    spiffe.cilium
mesh-auth-spire-admin-socket     /run/spire/sockets/admin.sock
mesh-auth-spire-agent-socket     /run/spire/sockets/agent/agent.sock
mesh-auth-spire-server-address   spire-server.cilium-spire.svc:8081
mesh-auth-spire-server-connection-timeout 30s
```

Рисунок 3.16 – Результат команди *cilium config view | grep mesh-auth*

Вище зазначено, що:

- Взаємна автентифікація увімкнена.
- Порт, на якому за замовчуванням будуть виконуватися взаємні рукопотискання між агентами, встановлено на 4250.
- Домен довіри SPIFFE за замовчуванням встановлений на *spiffe.cilium*.
- Час очікування підключення до SPIRE за замовчуванням становить 30 секунд

Ми додамо *authentication.mode: required* до наших правил вхідного трафіку. Це забезпечить, окрім існуючих вимог політики, вхідний доступ лише для робочих навантажень із взаємною автентифікацією.

Давайте тепер подивимось на трафік від *api-client* до самого *api*. Мережна політика повинна була скинути перший потік від *api-client* *r* до служби *api* через */request-landing*. Чому? Оскільки перший пакет, який відповідає мережевій політиці на основі взаємної автентифікації, розпочне взаємне рукопотискання автентифікації.

```
(yura@kali)-[~]
$ hubble observe --type drop --from-pod default/api-client
Jun 9 07:55:44.213: default/api-client:51910 (ID:13528) <> default/api-6b7f4b9689-mrdlr:6090 (ID:12388) Authentication required DROPPED (TCP Flags: SYN)
```

Рисунок 3.17 – Результат команди *hubble observe --type drop default/api-client*

```
(yura@kali)~$ hubble observe --type policy-verdict --from-pod default/api-client
Jun  9 07:55:44.206: default/api-client:51910 (ID:13528) <-> default/api-6b7f4b9689-mrdlr:6090 (ID:12388) policy-verdict:L3-L4 INGRESS DENIED (TCP Flags: SYN; Auth: SPIRE)
Jun  9 07:55:45.221: default/api-client:51910 (ID:13528) -> default/api-6b7f4b9689-mrdlr:6090 (ID:12388) policy-verdict:L3-L4 INGRESS ALLOWED (TCP Flags: SYN; Auth: SPIRE)
Jun  9 07:58:40.902: default/api-client:54562 (ID:13528) -> default/api-6b7f4b9689-mrdlr:6090 (ID:12388) policy-verdict:L3-L4 INGRESS ALLOWED (TCP Flags: SYN; Auth: SPIRE)
```

Рисунок 3.18 – Результат команди *hubble observe --type policy-verdict --from-pod default/api-client*

Логи виглядають ось так. Перша спроба під'єднатися була відхилена, тому що взаємна аутентифікація ще не була налаштована. Друга спроба та подальші спроби були успішними, оскільки аутентифікація пройшла успішно.

Варто зазначити, що взаємна автентифікація на рівні обслуговування не просто замінює мережеву політику на мережевому рівні. Це додає додатковий рівень безпеки. Cilium все ще ідентифікує окремі кінцеві точки, як це робить сьогодні, і сегментація мережі все ще застосовується до цих окремих кінцевих точок. Якщо в мережевій політиці визначено ідентифікатор SPIFFE і селектори кінцевої точки, для зловмисного робочого навантаження недостатньо імітувати цю службу за допомогою скомпрометованого сертифіката рівня обслуговування

3.2 Впровадження Open Policy Agent

3.2.1 Контролер доступу в Kubernetes

Контролер доступу — це фрагмент коду, який перехоплює запити до сервера Kubernetes API до збереження об'єкта, але після автентифікації та авторизації запиту. Простіше кажучи, контролери доступу дозволяють установлювати політики безпеки для ваших кластерів Kubernetes. З першого погляду контролери доступу забезпечують контроль доступу для користувачів і програм у кластері. Вони дозволяють адміністраторам встановлювати різні політики, зокрема, які користувачі можуть додавати або видаляти об'єкти в кластері, а також які програми можна розгортати в кластерах. Контролери допуску зазвичай використовуються, щоб

гарантувати, що лише безпечний код і конфігурації допускаються до виробничого середовища. Перехоплюючи та перевіряючи запити до Kubernetes API, перш ніж їм буде дозволено зберігатися під час виконання в кластері, контролери доступу зменшують ризики, пов'язані з неавторизованими діями, шкідливими об'єктами або помилковими конфігураціями. Це превентивне застосування політики гарантує, що розгортаються лише безпечні та сумісні робочі навантаження. На практиці контролери допуску володіють великою універсальністю, полегшуючи низку спеціальних політик, адаптованих до конкретних потреб організації. Їх можна використовувати для перевірки запитів ресурсів, оцінки міток і анотацій, оцінки квот ресурсів і навіть запуску веб-перехоплень для розширених сценаріїв перевірки.

Пропонуючи таку гнучкість, контролери доступу значно підвищують стійкість платформи та загальну безпеку. Kubernetes навіть надає Kubernetes Pod Security Admission із трьома рівнями обмежувальних політик[16][17].

3.1.2 Типи контролерів допуску

Контролери допуску поділяються на два основні типи: мутаційні та перевіряючі. Кожен із них відіграє важливу роль у захисті вашого середовища Kubernetes.

- Контролери, що змінюються, першими взаємодіють із запитами. Вони мають повноваження змінювати або видозмінювати запити до того, як вони матеріалізуються в кластері. Їх основна роль полягає у внесенні змін до вхідних запитів, гарантуючи, що вони відповідають стандартам і стандартам. Це може передбачати зміну вмісту запиту, як-от додавання міток, анотацій або навіть зміну специфікацій контейнерів таким чином, щоб вони відповідали певним стандартам або параметрам за замовчуванням до перевірки запиту[1].
- Контролери перевірки – це другі, хто взаємодіє із запитами. Їхня роль — це сфера перевірки, де кожен запит порівнюється з набором встановлених політик і стандартів. На відміну від своїх мінливих аналогів, перевіряючі контролери не

змінюють запити; вони просто забезпечують відповідність правилам кластера[1].

3.1.3 Різниця між веб-хуками та контролерами доступу

Веб-хуки служать конекторами, які пов'язують зовнішні служби з процесом контролю доступу Kubernetes. По суті, це зворотні виклики HTTP, ініційовані певними подіями на сервері Kubernetes API, що дозволяє взаємодіяти в режимі реального часу із зовнішніми системами для модифікації або перевірки запиту.

Вебхуки відрізняються від контролерів доступу тим, що вони запускаються лише тоді, коли кластер отримує запит. Вебхук дозволяє розробникам адаптувати конкретні інструкції під час створення нових ресурсів у кластері Kubernetes. Він діє як інтерфейс між вашою програмою та сервером Kubernetes API, дозволяючи налаштовувати правила перевірки, такі як квоти та обмеження ресурсів, вимоги до автентифікації користувачів і клієнтів або політики контролю доступу.

З іншого боку, контролери доступу запускаються щоразу, коли хтось намагається створити або змінити ресурс у кластері. І щоб зробити це ще більш заплутаним, існують веб-хуки Kubernetes, які працюють з контролерами доступу; а саме мутація та перевірка типів. Поведінка цих веб-хуків визначається файлами конфігурації контролера доступу. Типи веб-хуків доступу:

- **MutatingAdmissionWebhook** перехоплює та обробляє вхідні запити до сервера Kubernetes API, перш ніж вони будуть збережені в кластері. Основною функцією типового прикладу мутаційного контролера доступу в Kubernetes є модифікація (зміна) вмісту створюваного або оновлюваного ресурсу. Це, як правило, нав'язливий варіант, який не дозволяє ретельно керувати тим, що розгортається у виробництві, оскільки мутація відбувається автоматично, а застосування політики в першу чергу може призвести до зниження виробництва[18].
- **ValidatingAdmissionWebhook** зосереджується на перевірці та потенційному відхиленні вхідних запитів до того, як вони будуть збережені в кластері. Його

основна роль полягає в оцінці відповідності запиту визначеним політикам і стандартам кластера. Це набагато менш нав'язливо, ніж змінний контролер допуску, оскільки його результати, як правило, є двійковими; розгорнути робоче навантаження чи ні[18].

3.1.4 Огляд OPA Gatekeeper

OPA Gatekeeper — це перевіряючий контролер доступу Kubernetes, який застосовує політику до об'єктів Kubernetes при їх створенні або зміні. Він оцінює запити на створення/модифікацію ресурсу Kubernetes на відповідність визначеним політикам, щоб визначити, дозволити чи заборонити створення, зміну чи видалення ресурсу Kubernetes. Оцінка політики відбувається на стороні сервера, коли запит API проходить через сервер API Kubernetes. Це гарантує наявність єдиної централізованої точки оцінки політики на кластер.

Gatekeeper використовує рамку обмежень OPA для створення власних політик на основі визначення ресурсу (CRD). Використання CRD забезпечує нативний досвід Kubernetes, дозволяє відокремити проблеми та відокремлює створення політики від реалізації політики. Автори політики можуть створювати шаблони політики, які називаються шаблонами обмежень. Шаблон обмеження моделює визначення політики. Ці шаблони обмежень можна спільно використовувати та повторно використовувати в кластерах. Обмеження реалізуються за допомогою шаблонів обмежень. Ці обмеження реалізують і забезпечують виконання політик, визначених шаблонами обмежень. Gatekeeper використовує ці обмеження, щоб увімкнути перевірку ресурсів і функції аудиту. Gatekeeper можна впровадити в будь-якому кластері Kubernetes[19].

3.1.5 Реалізація політики безпеки за допомогою Gatekeeper в Kubernetes

При розгортанні кластера Kubernetes, однією з ключових задач є забезпечення безпеки і контролю доступу до ресурсів. Однак інструментарій Kubernetes не завжди

надає всі необхідні функції для реалізації вимог безпеки, що може призводити до потреби використання додаткових інструментів. У таких випадках Gatekeeper стає корисним інструментом, який дозволяє встановлювати та реалізувати політики безпеки в Kubernetes. Gatekeeper - це розширення для Kubernetes, яке дозволяє визначати та контролювати деякі аспекти конфігурації кластера за допомогою власних політик на основі мови програмування Rego.

Однією з типових ситуацій є контроль доступу до контейнерних образів. Наприклад, бажаючи забезпечити безпеку, адміністратори кластера можуть бажати обмежити створення контейнерних образів лише з певних приватних репозиторіїв, уникнути використання публічних репозиторіїв.

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: checkpodrepo
spec:
  crd:
    spec:
      names:
        kind: checkpodrepo
      validation:
        openAPIV3Schema:
          properties:
            repo:
              type: array
              items:
                type: string
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package checkpodrepo

        violation[{"msg": msg}] {
          containers := input.review.object.spec.containers[_]
          satisfied := [good | repo = input.parameters.repo[_]; good = startswith(containers.image, repo)]

          not any(satisfied)

          msg := sprintf("Зображення цього контейнера <%v> не в приватному репозиторії <%v>", [containers.image, input.parameters.repo])
        }
```

Рисунок 3.19 – файл template.yaml

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: checkpodrepo
metadata:
  name: private-repo-pod
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
  parameters:
    repo:
      - "again4/privaterepo"
```

Рисунок 3.20 – файл constraint.yaml

Create repository

Namespace **again4** Repository Name * **privaterepo**

Short description **private repository**

A short description to identify your repository. If the repository is public, this description is used to index your content on Docker Hub and in search engines, and is visible to users in search results.

Visibility

Using 0 of 1 private repositories. [Get more](#)

☐ **Public** Appears in Docker Hub search results
 ☒ **Private** Only visible to you

Cancel Create

Рисунок 3.21 – створення приватного репозиторію

```
apiVersion: v1
kind: Pod
metadata:
  name: private-pod
spec:
  containers:
  - name: private-container
    image: again4/privaterepo
    command: ["sleep", "3600"]
```

Рисунок 3.22 – файл private.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: public-pod
spec:
  containers:
  - name: public-container
    image: nginx
    command: ["sleep", "3600"]
```

Рисунок 3.23 – файл public.yaml

template.yaml

Цей файл визначає структуру обмеження, яке буде використовуватися для перевірки правил безпеки. Створюємо шаблон checkpodrepo, який перевіряє, чи використовуються допустимі репозиторії для контейнерних образів.

constraint.yaml

Цей файл створює конкретну інстанцію обмеження з використанням шаблону checkpodrepo. У вашому випадку, можна створити два окремих обмеження: одне для публічних репозиторіїв і одне для приватних. Вони визначають правила, які будуть застосовуватися до подальшого створення об'єктів, таких як Поди в Kubernetes.

private.yaml

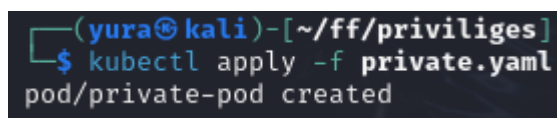
У цьому YAML-файлі описуємо под, який ми хочете створити. Він містить `image` з мого приватного репозиторію.

public.yaml

Аналогічно, цей YAML-файл також описує Под, але використовує *image* з публічного репозиторію.

Ці файли разом допомагають налаштувати політику безпеки в Kubernetes з використанням Gatekeeper, щоб контролювати, з яких репозиторіїв можуть бути використані зображення для створення Подів.

Результати



```
(yura@kali)-[~/ff/privileges]
$ kubectl apply -f private.yaml
pod/private-pod created
```

Рисунок 3.24 – Результат команди `kubectl apply -f private.yaml`

```
(yura@kali)-[~/ff/privileges]
$ kubectl apply -f public.yaml
Error from server (Forbidden): error when creating "public.yaml": admission webhook "validation.gatekeeper.sh" denied the request: [private-repo-pod] Зображення цього контейнера <nginx> не в приватному репозиторії <["again4/privaterepo"]>
```

Рисунок 3.25 – Результат команди *kubectl apply -f public.yaml*

Висновки до розділу 3

В третьому розділі були розглянуті методи вдосконалення Kubernetes системи. У рамках OPA Gatekeeper я розробив політику безпеки, яка перевіряє наявність публічних та приватних репозиторіїв. Це означає, що якщо користувач створює під з публічним репозиторієм, дія буде заборонена. OPA Gatekeeper та Pod Security Admission обидва забезпечують можливість налаштовувати політику безпеки на рівні кластера, проте OPA Gatekeeper надає більш гнучкі можливості для створення власних політик з використанням мови регулярних виразів. Впровадження Cilium дає нам змогу втілювати політики мережі Kubernetes не тільки на рівні L3/L4, а також на рівні L7 для поширених протоколів, таких як HTTP, Kafka, gRPC та інші. Крім того, Cilium підтримує шифрування трафіку за допомогою протоколів IPsec або WireGuard, що забезпечує конфіденційність даних під час передачі через мережу. Додатково, завдяки підтримці механізму взаємної аутентифікації (mTLS), Cilium забезпечує впевненість у тому, що лише довірені клієнти мають доступ до ресурсів в кластері.

ВИСНОВКИ

Кубернетес став невід'ємною складовою інфраструктури сучасних інформаційних систем, що оркеструють контейнеризовані додатки. В рамках дипломної роботи детально вивчено систему керування контейнерами Kubernetes з акцентом на аспекти безпеки та захисту. Архітектура та ключові компоненти Kubernetes не лише сприяють ефективному розгортанню програмного забезпечення, але й ставлять під загрозу безпеку системи в разі недостатньої конфігурації та захисту. Отже, впровадження ефективних заходів безпеки стає критично важливим для забезпечення стійкості і безпеки в екосистемі Kubernetes.

В другому розділі були ідентифіковані потенційні вразливості системи Kubernetes, а також розглянуті методи їх захисту. Встановлено, що використання мережових політик, RBAC, моніторингу безпеки, безпечних базових образів та автоматизованого сканування вразливостей є ключовими аспектами для забезпечення безпеки у роботі з Kubernetes.

Третій розділ розглянув можливості вдосконалення Kubernetes за допомогою інструментів OPA Gatekeeper та Cilium. OPA Gatekeeper дозволяє створювати гнучкі політики безпеки з використанням регулярних виразів, тоді як Cilium забезпечує розширені можливості мережевого контролю та шифрування трафіку для підвищення безпеки та конфіденційності даних.

Впровадження запропонованих заходів дозволить забезпечити надійність та стійкість інфраструктури контейнеризації під час її експлуатації.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Kubernetes Documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://kubernetes.io/docs/>
2. Kubernetes: Advantages and Disadvantages [Електронний ресурс] – Режим доступу до ресурсу: <https://cloudnativejourney.wordpress.com/2023/04/19/kubernetes-advantages-and-disadvantages/>
3. The differences between Docker, containerd, CRI-O and runc [Електронний ресурс] – Режим доступу до ресурсу: <https://vineetcic.medium.com/the-differences-between-docker-containerd-cri-o-and-runc-a93ae4c9fdac>
4. Open Container Initiative. [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Open_Container_Initiative
5. Навіщо нам контейнеризація: переваги для DevOps. [Електронний ресурс] – Режим доступу до ресурсу: <https://itedu.center/ua/blog/articles/navishho-nam-kontejnerizaciya-perevagi-dlya-devops/>
6. Best Containerization Software in 2024. [Електронний ресурс] – Режим доступу до ресурсу: <https://6sense.com/tech/containerization>
7. Kube-Proxy: What is it and How it Works. [Електронний ресурс] – Режим доступу до ресурсу: <https://medium.com/@amroessameldin/kube-proxy-what-is-it-and-how-it-works-6def85d9bc8f>
8. ClusterIP, NodePort, and LoadBalancer: Kubernetes Service Types. [Електронний ресурс] – Режим доступу до ресурсу: <https://www.baeldung.com/ops/kubernetes-service-types>
9. Understanding Kubernetes Controllers: A Key Concept for Managing Your Applications. [Електронний ресурс] – Режим доступу до ресурсу: <https://www.fairwinds.com/blog/kubernetes-controllers>
10. OWASP Kubernetes Top Ten. [Електронний ресурс] – Режим доступу до ресурсу: <https://owasp.org/www-project-kubernetes-top-ten/>
11. Kubernetes CNI Explained. [Електронний ресурс] – Режим доступу до ресурсу: <https://www.tigera.io/learn/guides/kubernetes-networking/kubernetes-cni/>
12. Cilium Documentation. [Електронний ресурс] – Режим доступу до ресурсу:

<https://docs.cilium.io/en/stable/overview/intro/>

13. Cilium and eBPF. [Електронний ресурс] – Режим доступу до ресурсу:

<https://www.solo.io/topics/cilium/cilium-ebpf/>

14. Transparent Encryption with IPsec and WireGuard. [Електронний ресурс] – Режим доступу до ресурсу: <https://isovalent.com/blog/post/tutorial-transparent-encryption-with-ipsec-and-wireguard/>

15. Next-Generation Mutual Authentication (mTLS) with Cilium Service Mesh [Електронний ресурс] – Режим доступу до ресурсу: <https://isovalent.com/blog/post/2022-05-03-servicemesh-security/>

16. Як захистити кластер. Kubernetes і безпека контейнерів [Електронний ресурс] – Режим доступу до ресурсу: <https://dou.ua/forums/topic/36341/>

17. Goodbye PodSecurityPolicy, Hello OPA Gatekeeper [Електронний ресурс] – Режим доступу до ресурсу: <https://rafay.co/the-kubernetes-current/goodbye-podsecuritypolicy-hello-opa-gatekeeper/>

18. Goodbye PodSecurityPolicy, Hello OPA Gatekeeper [Електронний ресурс] – Режим доступу до ресурсу: <https://medium.com/nerd-for-tech/the-difference-between-validating-and-mutating-webhooks-167757f8db5f>

19. Secure your Kubernetes environment with OPA and Gatekeeper [Електронний ресурс] – Режим доступу до ресурсу: <https://www.cncf.io/blog/2023/10/09/secure-your-kubernetes-environment-with-opa-and-gatekeeper/>

20. Best Practices of Secrets Management with Kubernetes [Електронний ресурс] – Режим доступу до ресурсу: <https://entro.security/blog/best-practices-of-secrets-management-with-kubernetes/>