МІНІСТЕРСТВО ОСВІТИ І НАУКИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ЕЛЕКТРОНІКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КІБЕРБЕЗПЕКИ

«До захисту допущено»
Завідувач кафедри
________ Володимир ЛЮБЧАК
(підпис) (Ім'я та ПРІЗВИЩЕ)
___ ___________ 20__р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавр

зі спеціальності 125 Кібербезпека, освітньо-професійної програми Кібербезпека
на тему: Розробка моделі архітектури захищеного вебдодатку на основі технології Kubernetes.

Здобувача групи КБ-01 Борща Дмитра Олександровича.

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

___________ Дмитро БОРЩ
(підпис)

Керівник ст. викладач, к. п. н., доц. Тетяна ЛАВРИК _____________
(підпис)

Суми — 2024

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра кібербезпеки

«Затверджую»
Завідувач кафедри

___________ Володимир ЛЮБЧАК
(підпис)

«___»_____________ 20___ р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавр

зі спеціальності 125 – Кібербезпека, освітньо-професійної програми «Кібербезпека»
здобувача групи КБ-01 Борща Дмитра Олександровича

1. Тема роботи: «Розробка моделі архітектури захищеного вебдодатку на основі технології Kubernetes».
затверджено наказом по СумДУ №0212-VI від 04.03.2024 р. зі змінами згідно Наказу №0566-VI від 21.05.2024 р.
2. Термін подання студентом роботи: «____» ________________ 20 ___ р.
3. Вихідні дані до роботи: 1) маршрутизатор з підтримкою VLAN; 2) гіпервізор для розгортання системи; 3) наукова та технічна література; 4) нормативні документи та міжнародні стандарти.
4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити): 1) Загальна характеристика архітектур вебдодатків, їх класифікація; 2) Архітектура кластеру Kubernetes; 3) Архітектура контейнеризованого додатку; 4) Аналіз вразливостей в кластерах Kubernetes; 5) Кращі практики забезпечення захисту в кластерах Kubernetes; 6) Опис моделі архітектури захищеного вебдодатку; 7) Опис програмної реалізації; 8) Тестування програмного рішення.
5. Перелік графічного матеріалу (із зазначенням плакатів, презентацій тощо) Презентація
6. Консультанти до проекту (роботи), із зазначенням розділів, що їх стосуються

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «____» ________________ 20 ___ р.

Завдання прийняв до виконання ________________ (підпис)

Керівник ________________ (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1	Визначення мети роботи, об’єкту і предмету дослідження		
2	Збір, систематизація й узагальнення матеріалу для використання у кваліфікаційній роботі		
3	Робота над першим та другим розділом		
4	Робота над третім розділом		
5	Оформлення текстової і графічної частини		

Здобувач вищої освіти ________________ Керівник ________________
(підпис) (підпис)

АНОТАЦІЯ

Записка: 54 стор., 25 рис., 4 додатки, 20 джерел.

Об'єкт дослідження: модель архітектури захищеного вебдодатку.

Мета роботи: розробити та протестувати модель архітектури захищеного вебдодатку на основі технології Kubernetes.

Метод дослідження: метод аналітичного огляду, метод порівняння, метод моделювання, метод експерименту.

Результати роботи: проаналізовано найпоширеніші архітектури вебдодатків, визначено переваги та недоліки кожної з них; проаналізовано шляхи забезпечення високої доступності вебдодатків; проаналізовано архітектуру, безпеку та вразливості системи Kubernetes, проаналізовано кращі практики забезпечення безпеки в кластерах Kubernetes; спроєктовано модель архітектури захищеного вебдодатку на основі технології Kuberentes; здійснено програмну реалізацію спроєктованої моделі та проведено тестування безпеки.

Ключові слова: вебдодаток, архітектура вебдодатку, модель захищеного вебдодатку, мікросервісна архітектура, тестування безпеки, Kubernetes, контейнеризація додатків.

ЗМІСТ

ВСТУП..6
1 АРХІТЕКТУРИ ВЕБДОДАТКІВ...8
1.1. Загальна характеристика архітектур вебдодатків, їх класифікація............8
1.2. Архітектура кластеру Kubernetes...11
1.3. Архітектура контейнеризованого додатку...14
2 ХАРАКТЕРИСТИКА БЕЗПЕКИ КЛАСТЕРІВ KUBERNETES........................16
2.1. Аналіз вразливостей в кластерах Kubernetes...16
2.2. Кращі практики забезпечення захисту в кластерах Kubernetes................18
3 ПРОГРАМНЕ РІШЕННЯ ТА ТЕСТУВАННЯ...25
3.1. Опис моделі архітектури захищеного вебдодатку.....................................25
3.2. Опис програмної реалізації...29
3.3. Тестування програмного рішення...33
ВИСНОВКИ..43
СПИСОК ЛІТЕРАТУРИ...44
ДОДАТКИ..47
Додаток А. Дерево атак на кластер Kubernetes...47
Додаток Б. Конфігурація Ignition для хостів...48
Додаток В. Конфігурація ініціалізації кластеру..50
Додаток Г. Маніфест для розгортання простору імен “app”............................51

ВСТУП

У наш час важко собі уявити життя без доступу до мережі Інтернет. Сотні та тисячі різноманітних сервісів та вебдодатків, якими користуються мільярди людей щодня. Ці сервіси агрегують величезними обсягами даних, чимось покращують наше життя, деякі з них є розважальні, інші є критично важливими для функціонування державних органів. Однак для всіх них важливим є дотримання трьох критеріїв інформаційної безпеки: конфіденційність, цілісність та доступність.

Зі збільшенням обсягів трафіку в мережі Інтернет виникає необхідність у проєктуванні більш складних та потужних сервісів, які здатні обслуговувати всіх користувачів. Існує безліч різних архітектур вебдодатків, кожна з яких має свої переваги та недоліки. Деякі з найпоширеніших це: монолітна та мікросервісна архітектури.

Для розподілення навантажень та забезпечення відмовостійкості найкраще підходить мікросервісна архітектура. Ця архітектура розбиває програму на невеликі, незалежні служби, кожна з яких відповідає за певну функцію. Це робить програму легко масштабованою та гнучкою, але також може ускладнити розробку та обслуговування. Тому вона не використовується широко в невеликих проєктах, натомість набула великої популярності у компаній, які розробляють глобальні вживані у всьому світі сервіси.

Зі зростанням складності архітектури з’являються нові площини атак. Тому, для специфічної архітектури необхідно використовувати специфічні заходи забезпечення інформаційної безпеки.

Метою даної роботи є розроблення моделі архітектури захищеного вебдодатку на основі технології Kubernetes, її програмна реалізація та тестування.

Для досягнення цієї мети поставлено такі завдання:

1) провести аналіз архітектури кластеру Kubernetes;

2) визначити потенційні загрози та вразливості безпеки в кластерах Kubernetes;
3) дослідити механізми забезпечення захисту в кластерах Kubernetes;
4) здійснити проєктування моделі архітектури захищеного вебдодатку та її програмну реалізацію;
5) протестувати розроблену модель і сформулювати висновки.

1 АРХІТЕКТУРИ ВЕБДОДАТКІВ

1.1. Загальна характеристика архітектур вебдодатків, їх класифікація

Існує дві основні архітектури побудови вебдодатку: монолітна та мікросервісна [1]. Монолітна архітектура передбачає що весь додаток буде вкладено в один умовний сервіс, який буде розгорнутий в певному середовищі. В той час як мікросервісна архітектура передбачає розбиття додатку на логічні частини, що можуть функціонувати незалежно та взаємодіяти між собою. Обидва підходи можуть забезпечити масштабування та доступність. Однак, перш ніж говорити про архітектуру побудови додатку, потрібно визначитися з загальною моделлю, що буде ключовим при виборі архітектури. Загалом моделі можна поділити на три види: один вебсервер і одна база даних, кілька вебсерверів і одна база даних, кілька вебсерверів і кілька баз даних.

Єдиний сервер і одна база даних доволі застаріла модель(як і монолітна архітектура, що має бути використана в цьому випадку), оскільки наявні ресурси не мають надлишковості та резерву. Збій одного компонента унеможливлює роботу всього додатку. Це непоганий вибір для випадків коли ресурси обмежені та доступність не є критичним фактором. Однак, це було б не вдалим рішенням для наприклад вебсайту відомої компанії з достатніми ресурсами.

В моделі з кількома вебсерверами існує менший ризик втрати даних, оскільки резервний сервер завжди доступний у разі збою першого. Однак не у випадках коли йдеться про мікросервіси, в такому випадку потрібно забезпечити копії кожного з них. Незважаючи на це, ймовірність збою додатку все ще може існувати через те, що база даних все ще немає резерву.

Для того, щоб забезпечити максимальну доступність додатку, необхідно мати резерв усіх сервісів та ресурсів, у тому числі бази даних. Розподілення сервісів залежних від стану(stateful; зокрема баз даних, бо нам потрібно, щоб їх дані надійно зберігались і не втрачались у разі збою) завжди складна задача, але

необхідна в разі великого масштабування додатку, наприклад у випадках, коли один сервер з базою даних вже не може фізично обробляти обсяги запитів до БД. Для цього можуть використовуватися різні способи. Наприклад створення кілька standby екземплярів MySQL, які б обробляли запити читання даних(подібно до серверів кешування в CDN), натомість будь які модифікації все-одно потрібно буде проводити через master екземпляр, який буде їх доставляти до standby по мірі можливості. На жаль така схема майже не масштабується і кількість запитів до master буде обмежена можливостями хосту на якому він знаходиться. Так, компанія Oracle пропонує технологію Oracle RAC(Real Application Clusters). Oracle RAC дозволяє декільком комп’ютерам одночасно запускати програмне забезпечення СУБД Oracle, таким чином забезпечуючи кластеризацію, але при цьому для клієнтів це буде виглядати як доступ до єдиної бази даних. Хоча ця технологія й потребує високого рівня підготовки інженерів для розгортання та підтримки.

Повертаючись до моделей з кількома вебсерверами, це теж може бути реалізовано кількома способами:

- використання кількох незалежних серверів та розподілення навантажень за допомогою DNS сервісу у випадку монолітної архітектури;
- використання HA clusters (кластери високої доступності) [2]. Такий підхід використовує групи комп’ютерів або логічних серверів, що можуть замінити один одного в разі необхідності, при цьому не втрачаючи стану(state). При цьому незалежно від типу додатку, монолітного чи мікросервісного. Приклад такої архітектури зображений на рис. 1.1. Однак масштабування такого кластеру доволі трудомісткий процес;
- використання технологій контейнеризації та інструментів оркестрації контейнерів(зокрема Docker та Kubernetes). В основі цього підходу лежить створення універсального середовища, що об’єднує кілька хостів в один кластер, а далі, використовуючи набір примітивів, організування керування сервісами незалежно від кількості їх контейнерів чи

середовища виконання або технології контейнеризації. Кластер Kubernetes дуже легко горизонтально масштабується, що спрощує переконфігурацію додатків в рази.

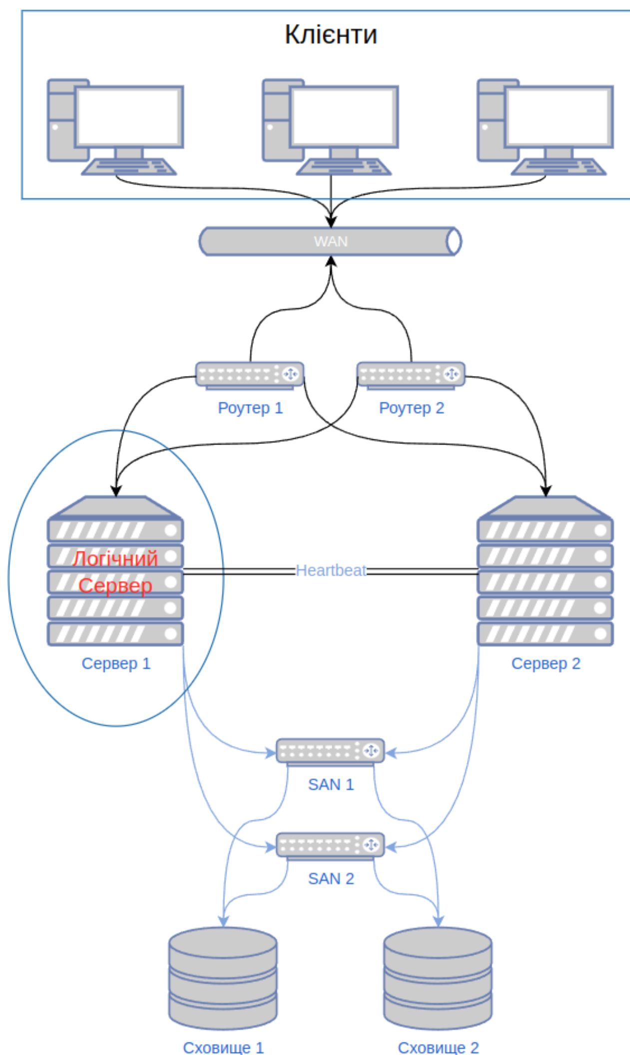

Рисунок 1.1 – Архітектура HA кластеру.

Також важливо розуміти, що часто ці рішення комбінуються. Так наприклад використання ресурсів залежних від стану (stateful) в Kubernetes не завжди є зручним рішенням, тому часто БД для додатків розгортають поза кластером Kubernetes.

1.2. Архітектура кластеру Kubernetes

Kubernetes – це відкрита система управління контейнерами, розроблена компанією Google [3]. Вона надає рішення для автоматизації розгортання, масштабування та управління додатками, які упаковуються в контейнери. Kubernetes забезпечує оркестрацію контейнерів, що дозволяє ефективно керувати і розташовувати їх у кластерах серверів. За допомогою Kubernetes розробники та оперативні команди можуть спростити процес розгортання та масштабування додатків, а також забезпечити високу доступність та стійкість до відмов. Він надає безліч функцій, включаючи автоматичне масштабування, розподіл трафіку, самолікування та управління конфігурацією.

Кластер Kubernetes складається з набору робочих машин, які називаються нодами(робочими вузлами), які запускають контейнеризовані додатки. Ноди є основною одиницею кластеру. Кожен кластер має принаймні одну ноду.

Ноди несуть поди, які є компонентами робочого навантаження програми і фактично є абстракцією над одним або групою контейнерів. Наступною важливою одиницею є control plane, він керує нодами та подами в кластері. У виробничих середовищах control plane зазвичай працює на кількох фізичних машинах, а кластер зазвичай має кілька нод, забезпечуючи відмовостійкість і високу доступність.

Компоненти Kubernetes кластеру можна розділити на ті, що утворюють control plane, та ті, що утворюють ноди. Загальна схема зображена на рис. 1.2.

Компоненти control plane приймають глобальні рішення щодо кластера, наприклад планування(scheduling, процес створення будь-яких подів), а також моніторять події кластера та реагують на них, наприклад планування екземпляру поду, коли не виконується вимога мінімальної кількості його реплік. Нижче описано кожен з компонентів кластеру більш детально [3].

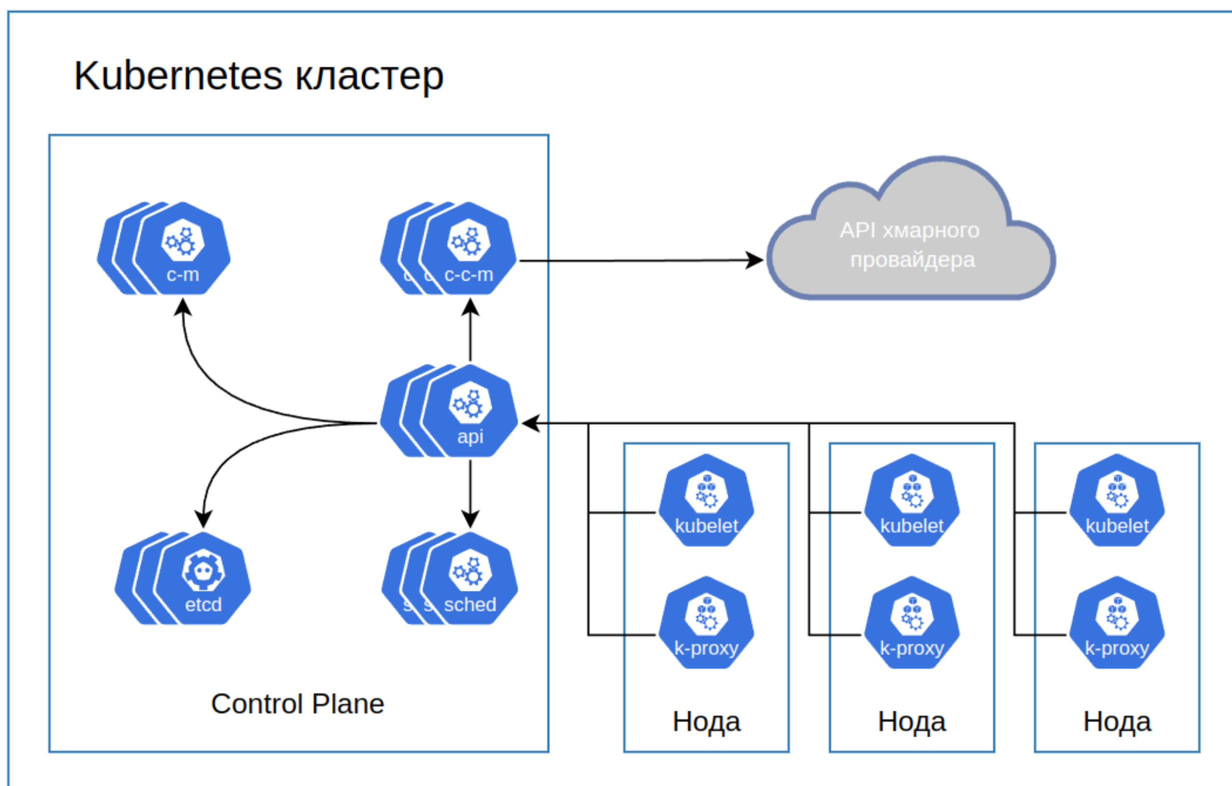

Рисунок 1.2 – Компоненти Kubernetes кластеру.

kube-apiserver

Сервер API імплементує доступ до API Kubernetes та є фронт-ендом для control plane Kubernetes. Основною реалізацією сервера Kubernetes API є kube-apiserver. kube-apiserver розроблено для горизонтального масштабування, тобто він може бути масштабований шляхом розгортання більшої кількості екземплярів.

etcd

ETCD – це консистентне та високодоступне сховище типу ключ-значення. Etcd використовується як резервне сховище Kubernetes для всіх даних кластера.

kube-scheduler

Компонент control plane, який стежить за новоствореними подами без призначеної ноди та обирає ноду для їх запуску. На прийняття рішень щодо планування можуть впливати індивідуальні та колективні вимоги до ресурсів, обмеження апаратного/програмного забезпечення/політики.

kube-controller-manager

Компонент control plane, який запускає процеси контролерів. Логічно кожен контролер є окремим процесом, але для зменшення складності всі вони скомпільовані в один двійковий файл і виконуються в одному процесі. Деякі типи цих контролерів:

- Node controller: відповідає за моніторинг вузлів та реагування на вихід їх з ладу.
- Job controller: стежить за об'єктами Job, які представляють одноразові завдання, а потім поди для виконання цих завдань.
- EndpointSlice controller: заповнює об'єкти EndpointSlice (для забезпечення зв'язку між сервісами та подами).
- ServiceAccount controller: створює облікові записи ServiceAccount за замовчуванням для нових просторів імен.

Компоненти нод працюють на кожній ноді, підтримуючи запущені поди та забезпечуючи середовище виконання Kubernetes.

kubelet

Компонент ноди, який гарантує, що всі поди мають запущені контейнери. Kubelet використовує набір PodSpec, які надаються через різні механізми, і гарантує, що контейнери, описані в цих PodSpec, працюють і справні. Kubelet не керує контейнерами, створеними не Kubernetes.

kube-proxy

Kube-proxy – це мережевий проксі, який працює на кожній ноді кластера, реалізуючи частину концепції сервісу Kubernetes. Kube-proxy підтримує мережеві правила на вузлах. Ці мережеві правила реалізують мережевий зв’язок із вашими подами під час мережевих сеансів у вашому кластері чи поза ним. kube-proxy використовує рівень фільтрації пакетів операційної системи(iptables наприклад), якщо він є та доступний. В іншому випадку kube-proxy пересилає трафік самостійно.

Середовище виконання контейнерів

Головний елемент будь-якої ноди це середовище виконання. Основна ідея створення подів як абстракції над контейнерами полягає в тому, щоб не бути залежним він container runtime та мати змогу відтворити деплойменти в будь-якому підтримуваному середовищі. Kubernetes підтримує середовища виконання контейнерів, такі як containerd, CRI-O та будь-яку іншу реалізацію Kubernetes CRI (Container Runtime Interface).

1.3. Архітектура контейнеризованого додатку

Мікросервісна архітектура це підхід до розробки програмного забезпечення, в якому додаток розбивається на невеликі самостійні сервіси, які працюють разом із власними набором функцій та мають власні інтерфейси (API). У мікросервісній архітектурі кожен мікросервіс може бути розроблений, розгорнутий і масштабований незалежно. Кожен сервіс може мати свою власну базу даних, конфігурацію та інші ресурси. Комунікація між мікросервісами зазвичай відбувається за допомогою мережевих протоколів, таких як HTTP або Message Queue.

Говорячи детальніше про масштабованість, то кожен мікросервіс може бути масштабований окремо від інших шляхом змінення кількості його реплік в кластері, що дозволяє ефективно реагувати на збільшення обсягу роботи або

навантаження на окремі компоненти системи. А це в свою чергу робить використання наявних ресурсів ефективнішим в рази. Також мікросервісна архітектура значно спрощує процес розробки, команди розробників можуть працювати над різними сервісами незалежно одна від одної, що полегшує внесення змін. Кожен сервіс може використовувати необхідну технологію та мову програмування для своєї конкретної задачі. Натомість вся їхня взаємодія буде реалізована за допомогою API.

Крім вищезазначеного, мікросервісна архітектура забезпечує високу доступність та стійкість. Навіть якщо один сервіс зазнає збою, або навіть випадково отримає оновлення з критичною помилкою, інші сервіси можуть продовжувати працювати, зберігаючи загальну функціональність системи. Помилки в одному сервісі не поширюються на інші, якщо тільки вони не залежні від дефектного.

Також одним з плюсів мікросервісного додатку є полегшена інтеграція зі сторонніми продуктами. Так наприклад до існуючого API інтерфейсу можна легко додати вебдодаток, який може бути розроблений іншою командою, але використовувати той самий набір мікросервісів. Приклад такої реалізації зображений на рис. 1.3.

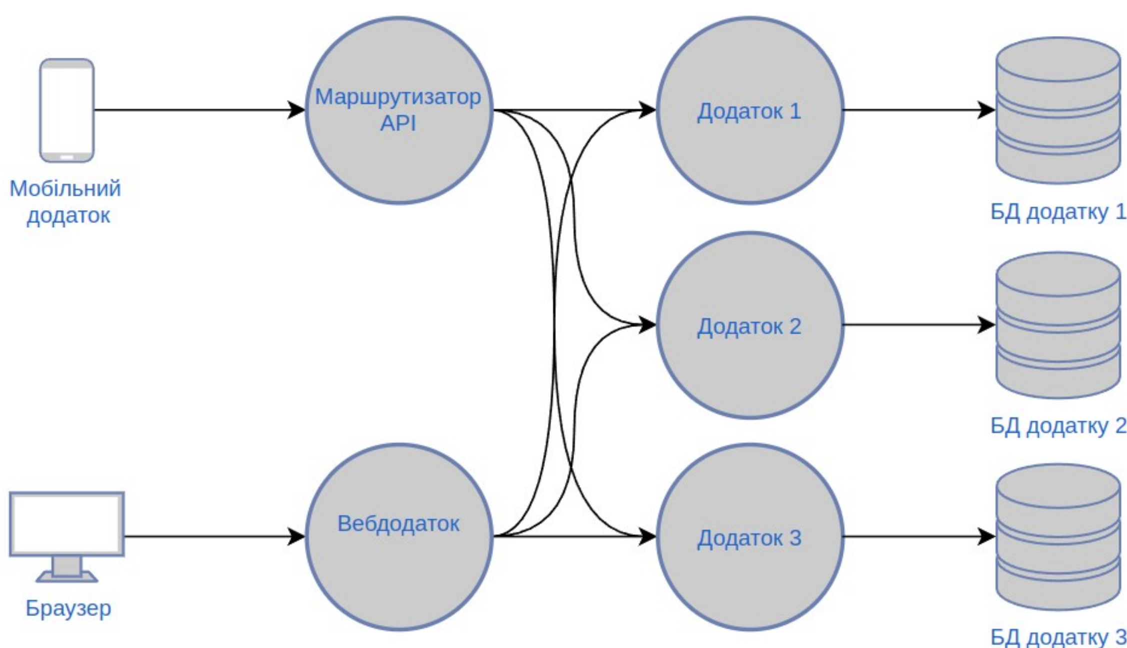

Рисунок 1.3 — Приклад архітектури мікросервісного додатку.

2 ХАРАКТЕРИСТИКА БЕЗПЕКИ КЛАСТЕРІВ KUBERNETES

2.1. Аналіз вразливостей в кластерах Kubernetes.

Існує три поширені джерела компрометації безпеки Kubernetes: ризики ланцюга поставок, дії зловмисників і загрози зсередини. Ризики ланцюга поставок часто важко пом'якшити, вони можуть виникнути під час створення контейнерів або придбання інфраструктури. Зловмисники можуть використовувати вразливості та неправильні конфігурації компонентів архітектури Kubernetes. Загрозами зсередини можуть бути адміністратори, користувачі або постачальники хмарних послуг. Особи з особливим доступом до інфраструктури Kubernetes організації можуть зловживати цими правами.

Важливо визначити модель загроз. Потенційними цілями для зловмисників у кластерах Kubernetes можуть бути як власне інформація, яка циркулює в середині, так і обчислювальні ресурси. Зазвичай злочини пов’язані з компрометацією інформації вважаються первинною мотивацією атакуючого, але також кібер злочинці можуть бути зацікавлені в експлуатації обчислювальних можливостей жертви. В еру криптовалют більш масовими стають випадки втручання в комп’ютерні системи різного рівня для використання їх з метою видобутку криптовалюти. У випадку з Kubernetes кластерами, мова йдеться про потужності спроможні витримувати навантаження, що спрямовуються на вебресурси, які обслуговує кластер.

Для того, щоб наочно продемонструвати джерела небезпек, було вирішено використати методику дерев атак [4]. Узагальнене дерево атаки на кластер Kubernetes приведене в додатку А. Схема демонструє можливі шляхи досягнення мети компрометації інформації кібер злочинцем. Діаграма є загальною та багато деталей в ній опущено з метою спрощення подання. Але загальна ідея цілком збережена. Варто зупинитися на кожному джерелі загроз окремо.

Останніми роками проєкти з відкритим вихідним кодом набирають все більшої популярності, оскільки працюють на засадах прозорості та в

абсолютній більшості мають ліцензії, що дозволяють повторне використання коду. Тож запорукою створення якісного та ефективного проєкту у наш час є використання уже готових напрацювань з інших проєктів, які це дозволяють та заохочують. Але в таких випадках не варто сліпо покладатися на якісь цього коду та його надійність, безпеку. Яскравим прикладом створення загрози шляхом втручання в ланцюги поставок є вразливість CVE-2024-3094 [5] про яку було повідомлено в березні 2024 року. GitHub акаунт з іменем JiaT75 більше двох років займався контрибуціями коду в проєкт XZ [6]. Шляхом довгих та складних маніпуляцій над сирцями, зловмиснику вдалося додати шкідливий код, що створював загрозу отримання несанкціонованого доступу за допомогою протоколу SSH. Фактично під загрозою могли опинилися майже всі комп’ютери, що використовують операційні системи на основі GNU/Linux. Оскільки вразливість виявили вже після її впровадження, деякі системи вже отримали на той момент вразливу версію XZ. Даний інцидент є гарним прикладом втручання в ланцюги поставки й створення критичного рівня загроз на великих кількостях систем.

Kubernetes є комплексною системою з великою кількістю компонентів. Здебільшого хоствими операційними системами для кластерів Kubernetes виступають дистрибутиви на основі Linux завдяки реалізації ізольованих груп процесів на рівні ядра. Звідси будь-які вразливості, до яких схильні хостові дистрибутиви, становлять загрозу для кластеру Kubernetes та даних, які в ньому обробляються. Окрім того, самі компоненти кластеру можуть мати вразливості. Основними типами атак на систем контейнеризації та оркестрації є втеча з контейнеру(варіація атаки з підвищенням привілеїв), відмова в обслуговуванні (DoS), несанкціонований доступ до адміністративних компонентів. Усім ним може бути схильний кластер Kubernetes.

І останнім джерелом загроз є внутрішні загрози. Вони можуть походити від авторизованих працівників організації, яка володіє кластером Kubernetes. Наприклад, адміністратори Kubernetes мають повний контроль над роботою

контейнерів, включаючи виконання довільних команд всередині середовищ контейнерів. Також внутрішня загроза може надходити з боку провайдерів інфраструктури чи хмарних послуг. У випадку співпраці з третіми сторонами з метою оренди обчислювальних потужностей, організація, що володіє кластером, довіряє їм інформацію, що циркулює в системі.

Отже, маючи уявлення про можливі джерела, характери загроз та будову кластерів Kubernetes ми можемо сформувати уявлення про формування та впровадження технічних та організаційних заходів для покращення стану безпеки інформації в системі.

2.2. Кращі практики забезпечення захисту в кластерах Kubernetes

Першим етапом до забезпечення захисту кластеру Kubernetes буде впровадження базових адміністративних заходів з правильного налаштування кластеру, обрання надійних секретів (сертифікатів, паролів), налаштування мережі та керування апаратними ресурсами відповідно до документації Kubernetes [7].

Варто розуміти, що система Kubernetes за своєю філософією слідує стандартам хмарної архітектури [8]. У питаннях безпеки він спирається на методи та поради Cloud Native Computing Foundation (CNCF) [9]. За даним документом, автори розділяють процес розробки хмарних робочих навантажень на чотири етапи: розробка, доставка, розгортання, виконання. Але використання периметричного підходу над загальним процесом розробки робить його занадто комплексним та ресурсоємним. На противагу класичним практикам пропонується вживляти елементи забезпечення захисту в кожен з етапів розробки. Такий підхід надає більш повний контроль та розділяє комплексний захист на такі ж етапи, які і процес розробки власне об’єктів захисту. Окрім того, розділення дозволяє в значній мірі автоматизувати кожен з кроків.

Розробка

На етапі розробки варто забезпечувати цілісні середовища розробки, організовувати огляд коду з турботою про аспекти безпеки, для ідентифікації ризиків побудувати модель загроз для конкретної цільової архітектури, в архітектуру закладати автоматизацію тестування безпеки, впроваджувати надійні архітектури безпеки, наприклад Zero Trust [10], це мінімізує площини атак навіть зі сторони внутрішніх загроз.

Доставка

Важливим кроком є мінімізація утворення вразливостей за ланцюгом поставок. Основною атомарною одиницею постачання коду, якою оперують системи контейнеризації, є образ контейнеру. Такий образ містить у собі базову систему, всі необхідні залежності додатку та власне додаток (його вихідний код, або бінарні файли). Образи контейнерів зазвичай зберігаються в спеціальних реєстрах, наприклад Docker Hub або GitHub Container Registry. Важливо не використовувати під час розгортання образи від невідомих постачальників та з невідомих реєстрів. Вони можуть містити в собі будь-які модифікації, які створять загрози безпеці. Так, наприклад, Docker Hub має спеціальні відмітки для образів, що опубліковані авторизованими постачальниками (рис. 2.1).

Рисунок 2.1 — Образ контейнеру, що містить СУБД MySQL від Google та має відповідну відмітку.

Окрім того, окремі образи мають спеціальну відмітку “Docker Official Image”(рис. 2.2), що є спеціальною програмою організації Docker надання

образів з використанням найкращих практик їх створення. Тобто такий образ є “взірцевим” та одним з найбезпечніших для використання.

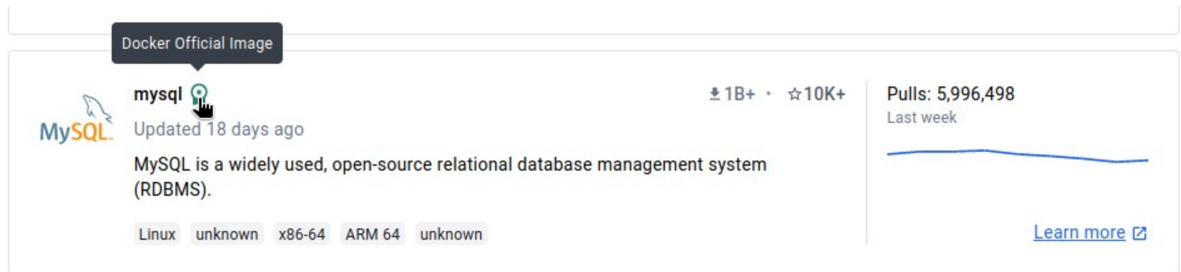

Рисунок 2.2 — Образ контейнеру, що містить СУБД MySQL та має відмітку “Docker Official Image”.

Для того, щоб переконатися в цілісності компонентів варто використовувати механізми валідації, наприклад цифрові сертифікати.

Перш ніж доставляти на цільову систему, вихідні контейнери варто сканувати на наявність відомих вразливостей.

Також доброю практикою вважається використання приватних реєстрів образів, що гарантує наявність вмісту образу, який туди власне вкладає розробник робочого навантаження. Реєстри контейнерів мають реалізацію процесу авторизації і шифрування процесу транспортування образів на цільову систему. Таким чином можна убезпечити себе від атак типу “man-in-the-middle” та забезпечити цілісність доставки компонентів.

Розгортання

На етапі розгортання варто вжити адміністративних заходів та чітко визначити, що можна розгортати, де це можна робити, хто на це уповноважений. Також на етапі розгортання можна впроваджувати деякі практики етапу доставки, такі як системи перевірки цілісності.

Виконання

Етап виконання включає три критичні області: обчислення, доступ і зберігання.

Всі компоненти кластеру взаємодіють між собою за допомогою інтерфейсу Kubernetes API. Захист цього інтерфейсу один із найголовніших заходів захисту кластеру на етапі виконання робочих навантажень. Важливо забезпечити взаємодію з API з використанням протоколу TLS, також налаштувати автентифікацію та авторизацію для доступу до API.

Для захисту обчислювальних потужностей необхідно застосовувати стандарти безпеки для подів, щоб гарантувати, що вони працюють лише з необхідними привілеями. На нодах варто використовувати спеціалізовану операційну систему, розроблену спеціально для виконання контейнеризованих робочих навантажень. Зазвичай вони базуються на операційній системі лише для читання (read-only OS), яка надає лише служби, необхідні для роботи контейнерів і не може змінювати свою конфігурацію під час виконання робочих навантажень. Контейнерно-орієнтовані операційні системи допомагають ізолювати системні компоненти та зменшувати поверхню атаки у разі атаки типу "втечі з контейнера". Для робочих навантажень необхідно визначати квоти ресурсів, щоб раціонально розподілити спільні обчислювальні ресурси.

Щоб забезпечити захист даних у постійному сховищі найкращим рішенням буде інтегрувати свій кластер з плагіном зовнішнього сховища , який забезпечує шифрування розділів у стані спокою та має реалізацію автентифікації підключень. Не варто забувати про резервне копіювання даних. Створення резервних копій найкраще автоматизувати й регулярно перевіряти надійність їх створення.

Мережа та безпека

Деякі мережеві плагіни для Kubernetes забезпечують шифрування для мережі кластера за допомогою таких технологій, як накладення віртуальної приватної мережі (VPN). За своєю будовою Kubernetes дозволяє використовувати зовнішній мережевий плагін для кластера. Прикладами таких

плагінів є Calico, Flannel, Cilium, Multus. Усі вони надають певний додатковий функціонал до мереж кластеру Kubernetes, та вибір одного чи декількох з них визначається лише потреба конкретного додатку. Обраний мережевий плагін і спосіб його інтеграції можуть сильно вплинути на безпеку інформації, що обробляється в кластері.

Спостережність та захист у реальному часі

Kubernetes дозволяє розширити можливості спостереження та забезпечення безпеки під час виконання робочих навантажень за допомогою додаткових інструментів. Деякі основні функції спостереження вбудовані в сам Kubernetes. Навантаження, що працюють в контейнерах, можуть генерувати журнали, публікувати показники або надавати інші дані для спостереження; під час розгортання потрібно переконатися, що кластер забезпечує належний рівень захисту.

Для кращого виявлення загроз та захисту в реальному часі під час виконання можна використовувати комплексні інструменти. Їх призначення можна поділити на два види: захист робочих навантажень(додатків в контейнерах) та захист інфраструктури.

Прикладом системи захисту робочих навантажень є KubeArmor (рис. 2.3) [11]. Вона є системою посилення безпеки у реальному часі, яка реалізує практично повний контроль над поведінкою компонентів у кластері Kubernetes. KubeArmor використовує компоненти фреймворку Linux Security Modules, такі як AppArmor та SELinux, для реалізації політик безпеки заданих користувачем. Також KubeArmor генерує детальну телеметрію щодо подій в контейнерах, подах та просторах імен.

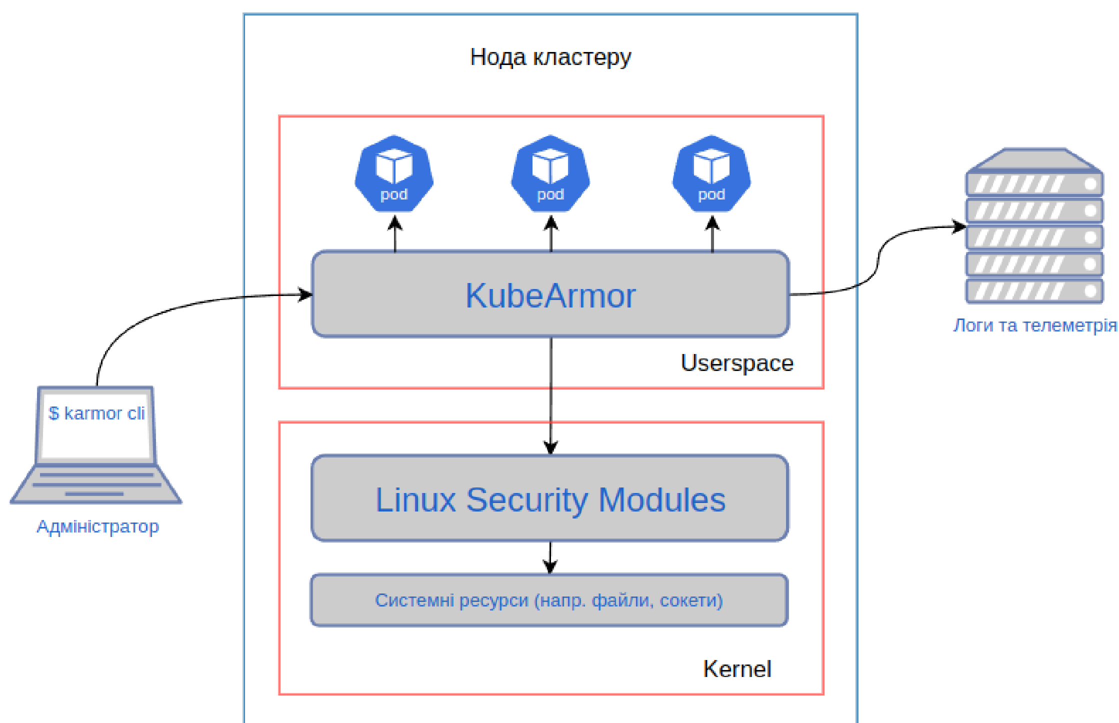

Рисунок 2.3 — Архітектура системи KubeArmor.

Більш цілісним рішенням є платформа Aqua Security [12]. Вона реалізує автоматизацію тестувань на проникнення, процесу атестування ризиків та проведення бенчмарків CIS (Центру Безпеки в Інтернеті). Окрім цього також надає контроль над робочими навантаженнями подібний до того, що реалізує KubeArmor.

Сервісні акаунти (Service Accounts)

Сервісні акаунти в Kubernetes є важливою частиною керування доступом на основі ролей (Role-Based Access Control, RBAC) [13]. На відміну від акаунтів користувачів, сервісні акаунти призначаються сутностям кластеру, наприклад подам з метою застосування до них політик безпеки. За замовчуванням, система створює сервісний аккаунт default для кожного простору імен. Цей аккаунт призначається усім новоствореним подам, для яких явно не вказане інше. Для впровадження більш строгих політик безпеки, або надання подам розширених дозволів (доступ до внутрішніх секретів кластеру, доступ до інших

просторів імен, автентифікація доступу до приватного реєстру образів за допомогою сервісного акаунту) треба створити відповідні сервісні акаунти та надати їм необхідні повноваження за допомогою механізмів авторизації (RBAC).

Захист секретів

Секретами (Secretes) в Kubernetes є сутності, що зберігають певні дані, які необхідно захистити від несанкціонованого доступу. За замовчуванням, секрети зберігаються в сховищі типу ключ-значення etcd [14]. Дане сховище підтримує шифрування даних при збереженні на диск. За замовчуванням, секрети фізично зберігаються в etcd у відкритому вигляді, тому для забезпечення їх безпеки варто налаштувати шифрування секретів[15].

Окрім шифрування, необхідно слідкувати за доступом. Використовуючи RBAC та сервісні аккаунти, кращою практикою буде надавати доступ до секретів лише необхідним компонентам та користувачам. Також гарним рішенням буде забрати права на записування секретів у сутностей, які потребують лише читання.

Для більш широкого керування та захисту секретів, Kubernetes підтримує використання зовнішніх сховищ секретів. Провайдерами сховищ секретів можуть бути [16]:

- Akeyless Provider;
- AWS Provider;
- Azure Provider;
- GCP Provider;
- Vault Provider.

Зазвичай зовнішні рішення використовуються у разі побудови архітектури додатки на основі існуючої хмари. Часто великі хмарні провайдери надають комплекс інструментів та компонентів для цього.

3 ПРОГРАМНЕ РІШЕННЯ ТА ТЕСТУВАННЯ

3.1. Опис моделі архітектури захищеного вебдодатку

Першим кроком до забезпечення контрольованості системи є забезпечення ізоляції процесів, що виконуються в ній. В ідеально захищеній системі процеси та користувачі мають доступ лише до необхідної інформації та обчислювальних ресурсів. В проєктованій моделі пропонується використовувати віртуальні машини для створення середовища виконання контейнерів та додаткової ізоляції їх від апаратного забезпечення. Діаграма шарів ізоляції зображена на рис 3.1.

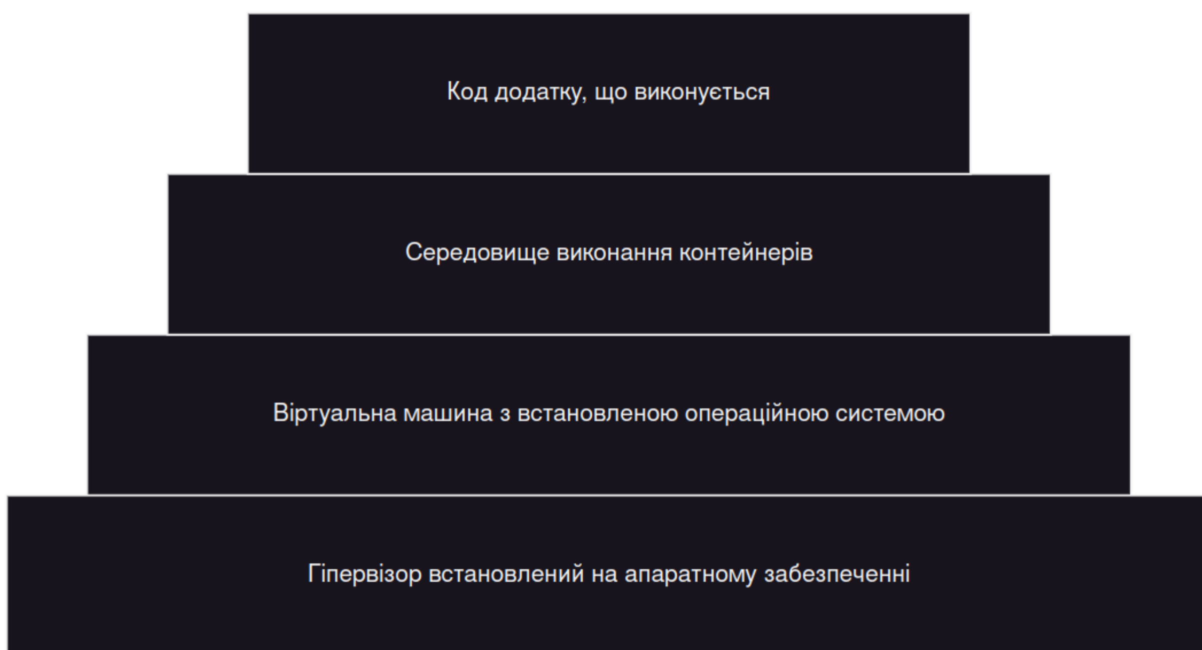

Рисунок 3.1 — Шари ізоляції в запропонованій моделі.

Почати проєктування кластеру слід почати з оцінки наявної кількості хостів. Для кластеру Kubernetes прийнято вважати мінімальною прийнятною кількістю шість хостів(три, що будуть обслуговувати control plane, та три робочі ноди). Для забезпечення відмовостійкості control plane буде використано kube-vip, спеціальний компонент який впроваджує VIP (Virtual IP) адресу в кластері Kubernetes та здатний автоматично реагувати на ситуації, коли хост з VIP стає недоступним та переносити його на інший операбельний. На рис. 3.2 зображено пропоновану логічну топологію control plane кластеру.

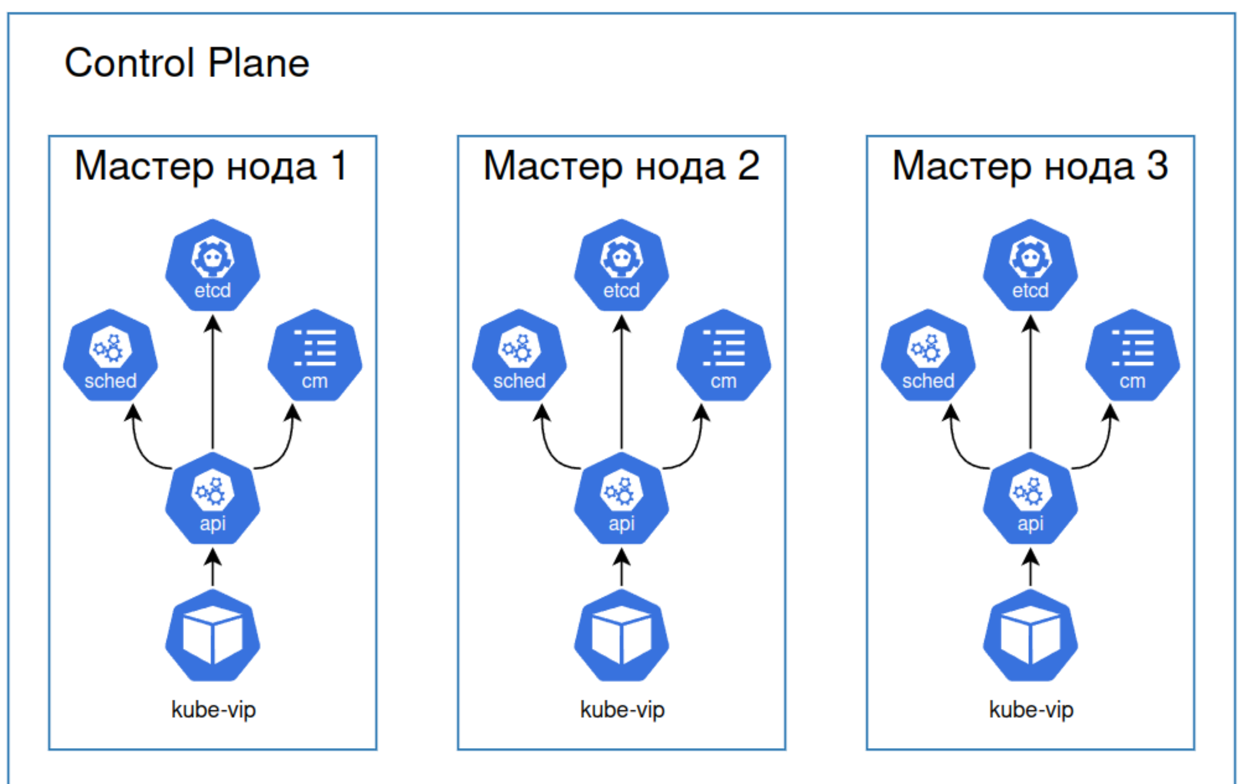

Рисунок 3.2 — Логічна топологія Control plane.

Варто зазначити, що кількість хостів, що використовується для розгортання control plane має бути обов’язково не меншою за три. Це обумовлено тим, що etcd є також кластером, який проводить вибори лідера. Під час обрання лідера etcd кластера використовується принцип більшості. Тобто у випадку якщо в кластері буде лише дві ноди і одна вийде з ладу, то голос іншої у виборі самої себе лідером не буде означати більшість. Таким чином служба etcd не зможе запуститися без лідера.

Сервіс kube-vip у свою чергу буде слідкувати за тим, щоб в мережі була активна IP адреса для доступу до API кластеру, оскільки це є вхідна точка для всієї роботи з кластером.

На рис. 3.3 зображена пропонована логічна топологія робочих нод кластеру.

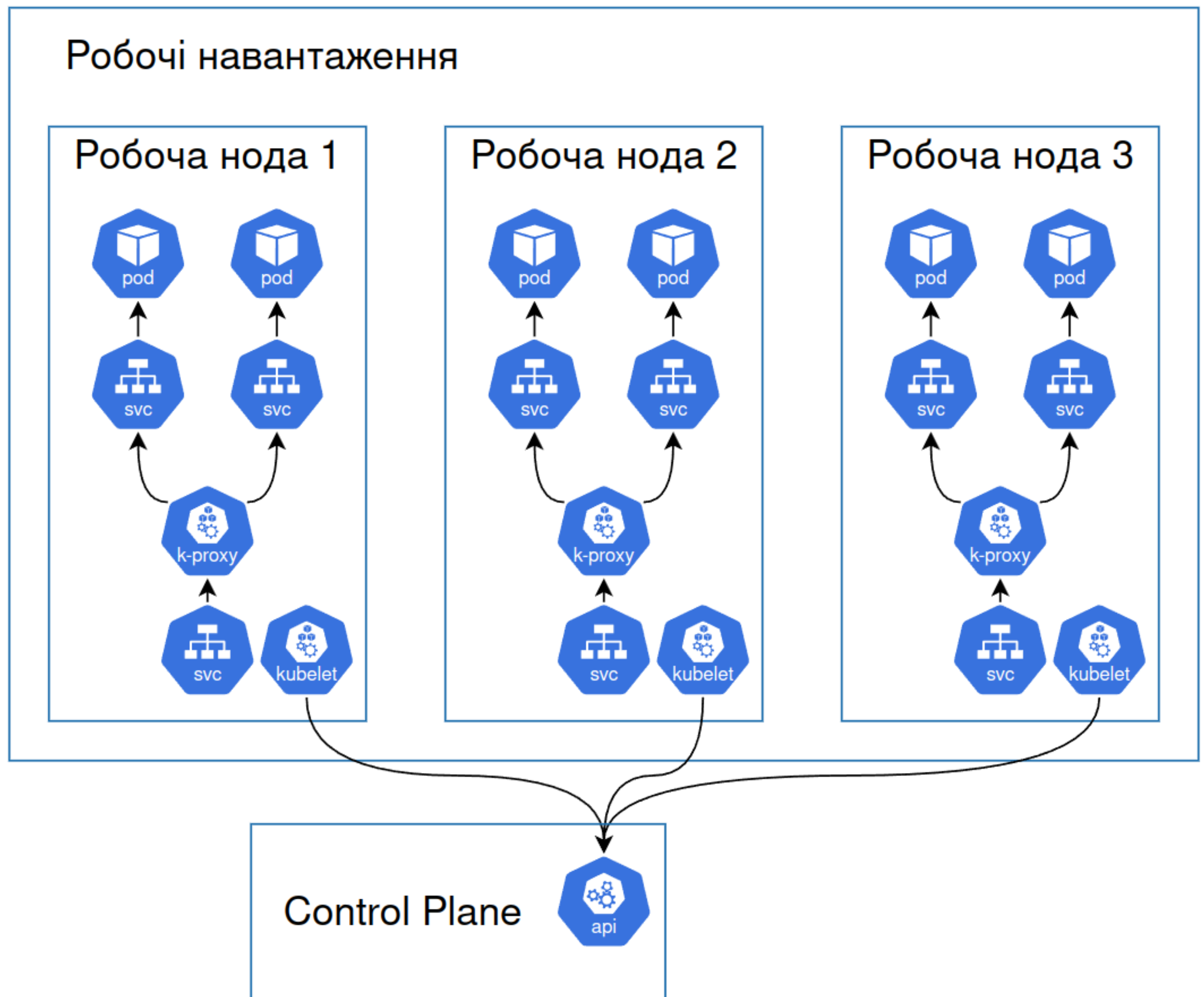

Рисунок 3.3 — Логічна топологія робочих нод кластеру.

Для функціонування кластеру пропонується створити дві мережі. Одна внутрішня, для комунікації всередині кластеру, і друга зовнішня, для роботи додатку з мережею Інтернет. Логічна топологія запропонованої моделі мережі зображена на рисунку 3.4.

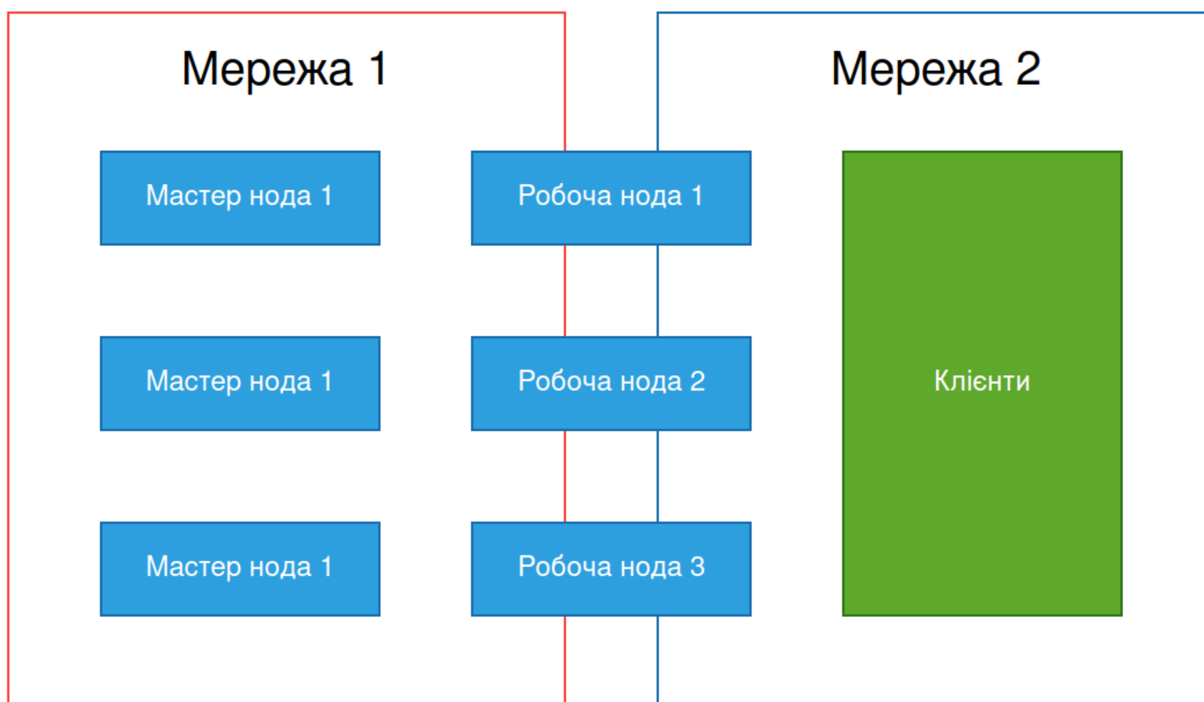

Рисунок 3.4 — Логічна топологія мережі.

Робочим нодам пропонується надати по два мережеві інтерфейси для комунікації з двома мережами. Вся внутрішня комунікація додатку (передача даних між сервісами, зв’язок з базою даних) проходить через внутрішню накладну мережу подів кластеру.

Для керування розгортанням робочих навантажень пропонується використовувати інструмент Portainer, а для захисту кластеру у реальному часі пропонується використати систему KubeArmor.

3.2. Опис програмної реалізації

Першим кроком є обрання операційної системи для нод кластеру. В моделі пропонується використовувати Fedora CoreOS. Дана операційна система розробляється компанією RedHat та створена спеціально для створення середовищ виконання контейнерів. Крім того, CoreOS використовує систему rpm-ostree [17] для керування версіями операційної системи. Цей інструмент додає атомарності в оновлення системи імплементуючи систему транзакцій на основі образів. Таким чином система може бути відкочена до попереднього стану майже після будь-яких змін в її ключові компоненти. В якості середовища виконання контейнерів пропонується використати CRI-O [18], створене спеціально для використання разом з Kubernetes. Також буде використано мережевий плагін Flannel [19].

Fedora CoreOS не має засобів графічного встановлення, натомість використовує спеціальний конфігураційний файл для системи що називається Ignition(запал). Ignition виконується лише один раз під час першого запуску системи, тому його конфігурацію необхідно підготувати завчасно.

Файл конфігурації Ignition наведено в додатку Б. Основними його пунктами є:

- створення користувача “op” та надання йому групи “sudo”;
- додавання файлів закритого та відкритого ключа для авторизації за допомогою протоколу SSH. Варто зазначити, що у користувача не буде паролю, тобто доступ буде надаватися виключно за допомогою ключа;
- додавання файлу конфігурації репозиторіїв Kubernetes для подальшого встановлення компонентів кластеру;
- додавання файлу конфігурації сервісу Zincati (вбудованої служби автоматичного оновлення).

В якості стратегії автоматичних оновлень було обрано використання вихідних днів й початок о 3:00 за системним часом та не довше 60 хвилин.

Наступним кроком буде планування мереж. Для зручності ідентифікації хостів пропонується використовувати зручні псевдоніми. В конкретному прикладі використано назви хімічних елементів (металів для Control Plane та неметалів для робочих нод). Таким чином ноди Control Plane будуть Cuprum, Ferrum та Titanium, а робочі ноди будуть Carbon, Nitrogen та Oxygen.

Загалом в даному прикладі кластер буде побудовано в мережі 192.168.3.0/24. Для зручності, робочим нодам буде виділено адреси з діапазону 192.168.3.1*, а робочим хостам з діапазону 192.168.3.2*. Для VIP адреси API буде взято 192.168.3.20. Всього в мережі буде шість хостів (рис. 3.5).

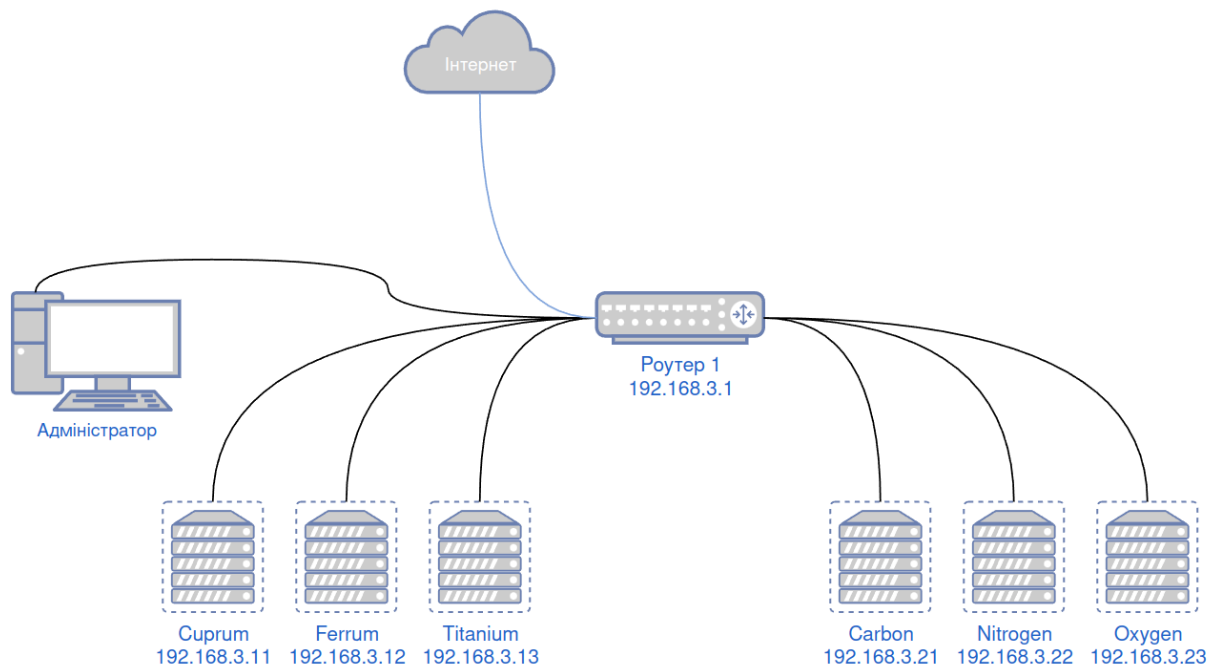

Рисунок 3.5 — Топологія службової мережі.

Друга мережа слугуватиме лише для доступу користувачів з мережі інтернет до корисних навантажень кластеру та буде мати адресу 192.168.4.0/24. IP адреси для робочих хостів було обрано такі ж як і в службовій мережі, але з іншою маскою підмережі для зручності. Топологія другої мережі зображена на рис. 3.6.

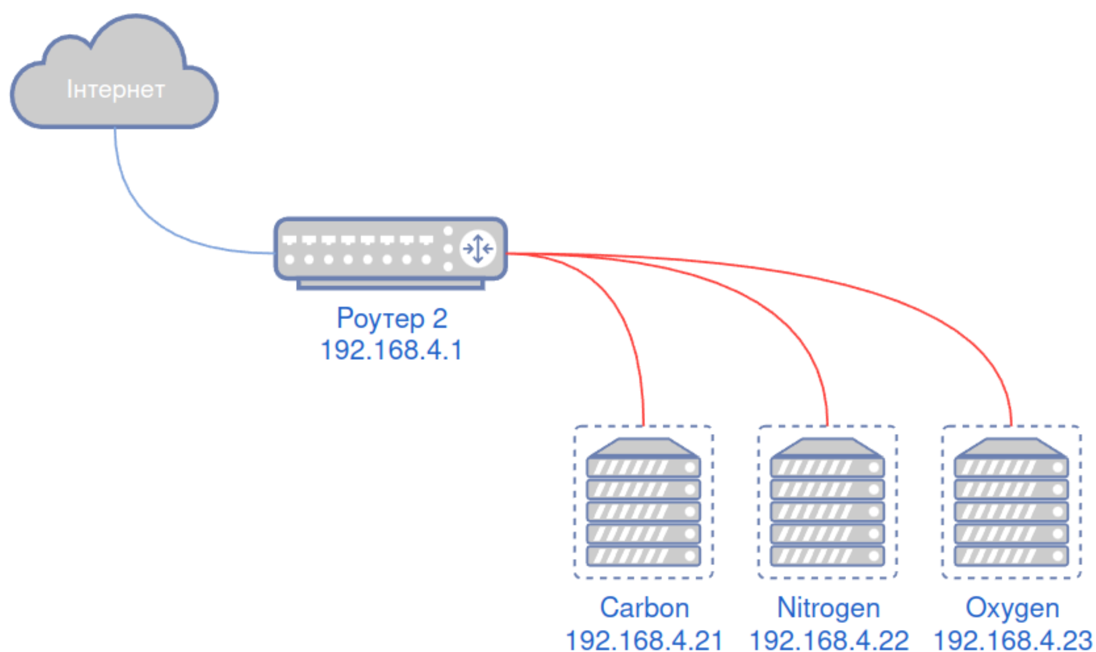

Рисунок 3.6 — Топологія другої мережі.

Повні налаштування мережі для хостів наведено в таблиці 3.1.

Таблиця 3.1 — Налаштування мережі для хостів

Хост	**Інтерфейс**	**IP адерса**	**Шлюз за замовчуванням**
Cuprum	ens18	192.168.3.11	192.168.3.1
Ferrum	ens18	192.168.3.12	192.168.3.1
Titanium	ens18	192.168.3.13	192.168.3.1
Carbon	ens18	192.168.3.21	192.168.3.1
	ens19	192.168.4.21	192.168.4.1
Nitrogen	ens18	192.168.3.22	192.168.3.1
	ens19	192.168.4.22	192.168.4.1
Oxygen	ens18	192.168.3.23	192.168.3.1
	ens19	192.168.4.23	192.168.4.1

В кластері буде мінімум три середовища імен: для розгортання самого додатку, розгортання системи KubeArmor та для інструменту Portainer. На етапі

проєктування архітектури, кількість мікро сервісів не має великого значення, нам важливо

Отже перший простір імен буде називатися “app” та в ньому будуть деплойменти додатку, у кожного з них буде свій набір реплік та сервіс (Service). Варто зазначити, що зазвичай деплоймент не створюється без сервісу. Крім деплойментів буде наявний Ingress контролер (буде використано Nginx Ingress), що буде маршрутизувати запити на мікросервіси. Набір компонентів першого простору імен зображено на рис. 3.7.

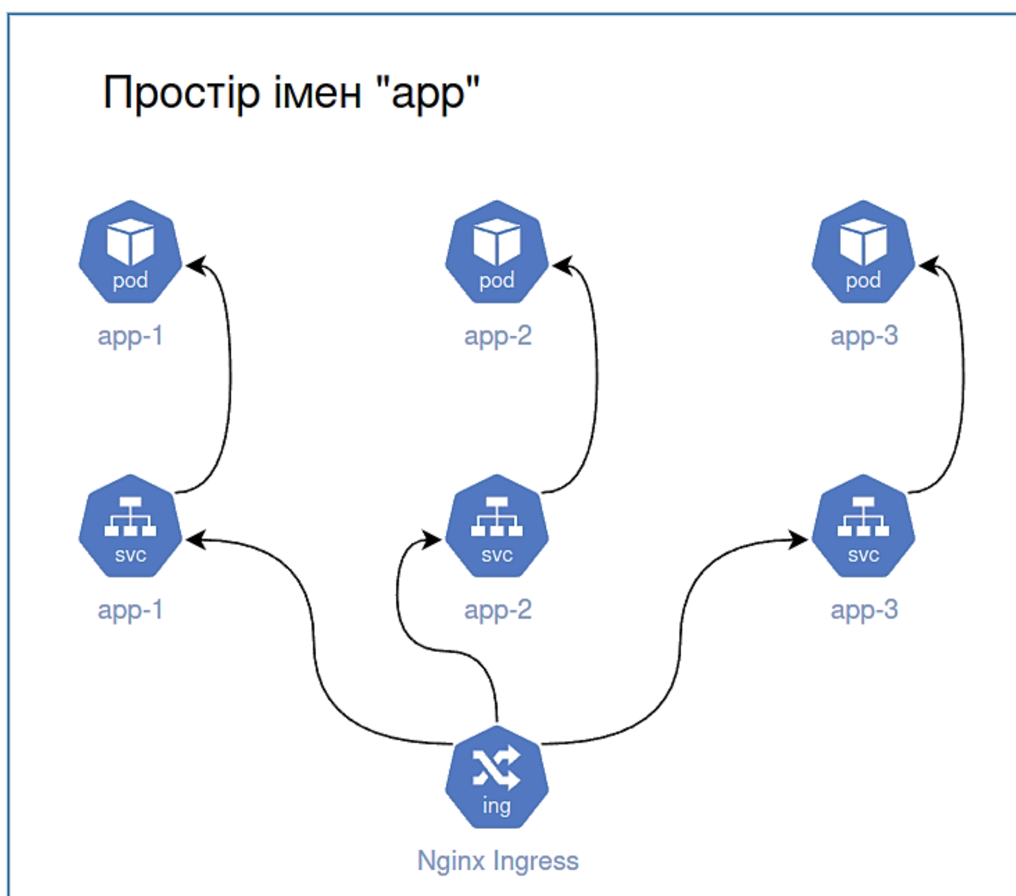

Рисунок 3.7 — Компоненти першого простору імен.

Інші простори будуть містити суто компоненти KubeArmor, Portainer та Nginx Ingress.

Таким чином, здійснено підготовку плану програмної реалізації моделі системи. Крім цього, спроєктовано мережу, підготовлено конфігураційні файли та спроєктовано простори імен. Здійснивши цей етап і маючи ці дані, можна переходити до тестування програмного рішення.

3.3. Тестування програмного рішення

Перед початком встановлення операційних систем на віртуальні машини необхідно зібрати конфігурацію Ignition в спеціальний JSON-файл. Зробити це можна за допомогою CLI інструмента butane наступним чином:

butane --pretty --strict host.bu > diploma.ign

Після цього Ignition файл необхідно помістити на вебсервер, доступний для віртуальної машини, щоб вона його отримала під час встановлення.

Коли файл Ignition підготовлено, залишається лише запустити LiveCD образ CoreOS та запустити програму встановлення з налаштованим конфігураційним файлом (рис. 3.8).

```
core@localhost:~$ sudo coreos-installer install /dev/sda --ignition-url http://138.2.156.19/diploma.ign --insecure-ignition
Installing Fedora CoreOS 40.20240416.3.1 x86_64 (512-byte sectors)
> Read disk 2.3 GiB/2.3 GiB (100%)
Writing Ignition config
Install complete.
core@localhost:~$
```

Рисунок 3.8 — Встановлення CoreOS за допомогою Ignition файлу з мережі.

З метою тестування в рамках роботи мережі емульовано за допомогою VLAN (рис. 3.9) в операційній системі OpenWRT, що встановлена на використаному маршрутизаторі.

У тестовій системі розподілення IP адреси налаштовано за допомогою статичних оренд на DHCP сервері маршрутизатору. Налаштування статичних оренд зображені на рис. 3.10.

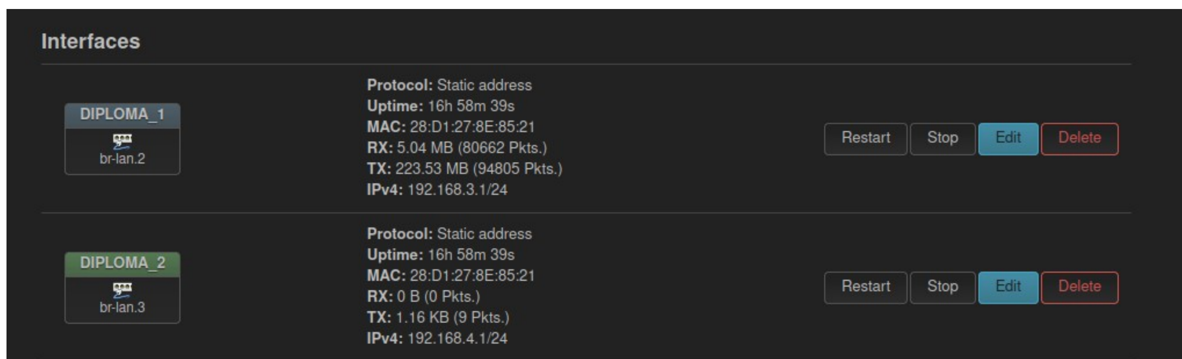

Рисунок 3.9 — Віртуальні інтерфейси на маршрутизаторі, що створюють дві нові мережі.

Cuprum	2E:82:8E:25:26:48	192.168.3.11
Ferrum	9E:22:0B:D2:F9:0C	192.168.3.12
Titanium	06:20:9C:39:DA:7C	192.168.3.13
Carbon	96:98:B3:EF:49:43	192.168.3.21
Carbon-ext	36:56:CC:03:86:86	192.168.4.21
Nitrogen	2A:88:A7:1F:37:83	192.168.3.22
Nitrogen-ext	5E:27:46:A5:A8:72	192.168.4.22
Oxygen	0A:A5:40:6E:AC:7C	192.168.3.23
Oxygen-ext	96:B8:20:97:C4:2E	192.168.4.23

Рисунок 3.10 — Налаштування статичних оренд DHCP серверу.

Коли мережа налаштована, тимчасово необхідно призначити VIP адресу вручну на початкову ноду Control Plane, щоб ініціалізувати кластер. Зробити це можна наступним чином:

sudo ip a add 192.168.3.20 dev ens18

Коли VIP адреса готова, можна ініціалізувати кластер. Для цього пропонується початковий конфігураційний файл (додаток В). У ньому зазначено VIP адресу та внутрішні підмережі для подів та сервісів.

Після ініціалізації всіх нод кластеру варто перевірити їх готовність за допомогою утиліти kubectl (рис. 3.11).

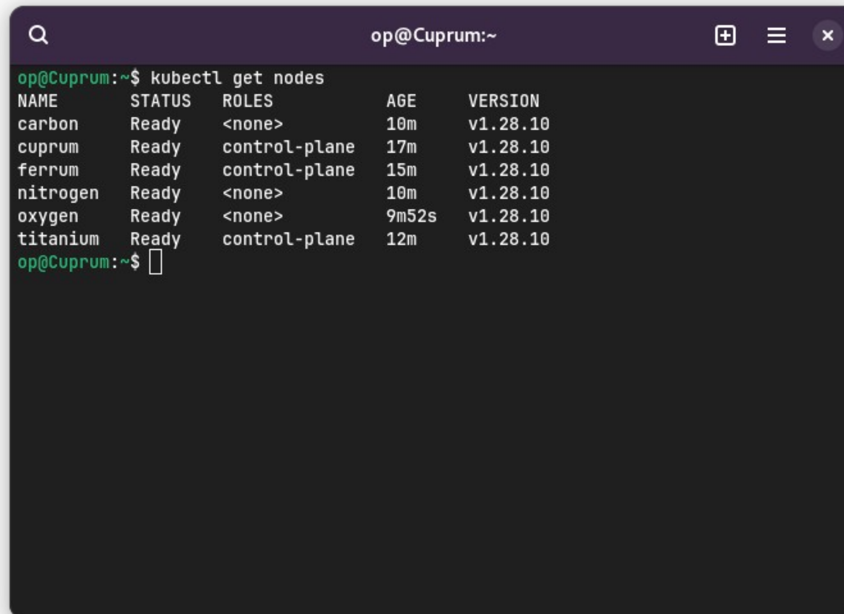

Рисунок 3.11 — Статус нод кластеру після ініціалізації.

Kube-vip розгортається шляхом генерації статичного маніфесту та копіювання його на ноди Control Plane.

Протестувати kube-vip можна шляхом вимкнення ноди, на якій в поточний момент часу знаходиться VIP адреса. Наприклад, під час тестування VIP знаходиться на ноді Cuprum (рис. 3.12).

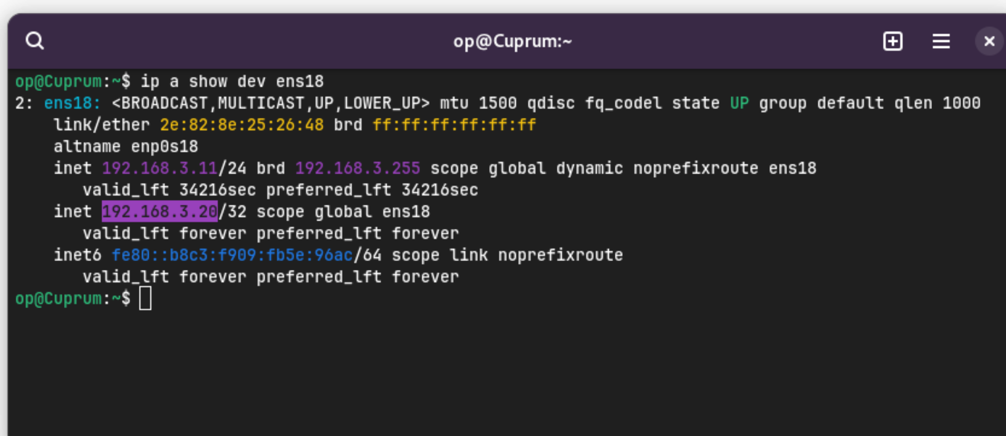

Рисунок 3.12 — VIP адреса на ноді Cuprum.

Після вимкнення віртуальної машини ноди Cuprum, в логах сервісів kube-vip з’являється наступний запис:

time="2024-05-22T14:52:01Z" level=info msg="Node [ferrum] is assuming leadership of the cluster"

Даний запис означає, що тепер лідером кластеру є нода Ferrum і відповідно VIP адреса з’являється на ній (рис. 3.13).

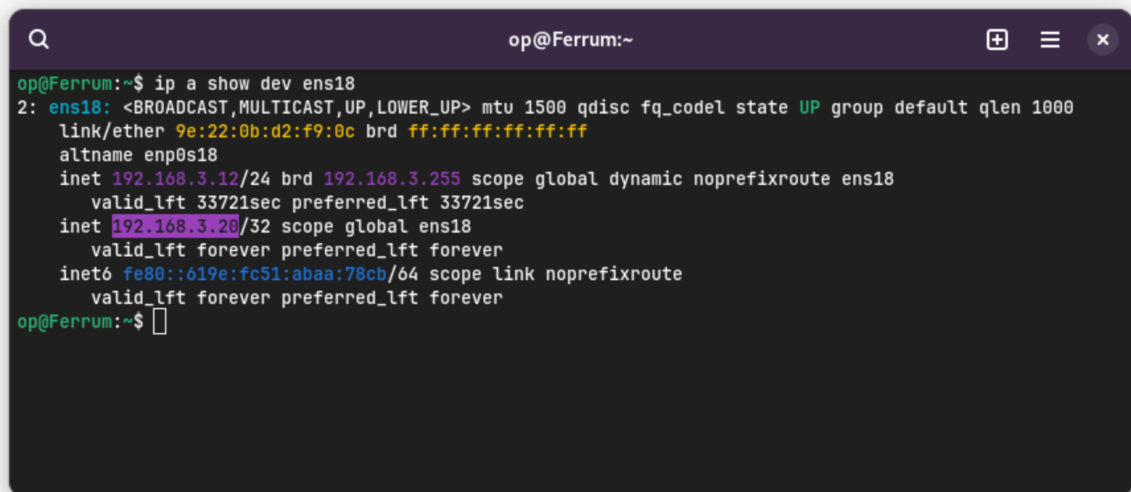

Рисунок 3.13 — VIP адреса на ноді Ferrum.

Отже, kube-vip працює належним чином. Після розгортання основних компонентів кластеру необхідно розгорнути мережевий плагін (було обрано Flannel). Зробити це можна за допомогою маніфесту, що надається розробниками наступною командою:

kubectl apply -f https://github.com/flannel-io/flannel/releases/latest/download/kube-flannel.yml

Коли кластер повністю підготовлено, можна розгортати додаткові компоненти. Portainer та Nginx Ingress подібно до Flannel також можна розгорнути за допомогою маніфесту, який надається розробниками. Для цього необхідно лише застосувати його в новоствореному кластері (рис.3.14).

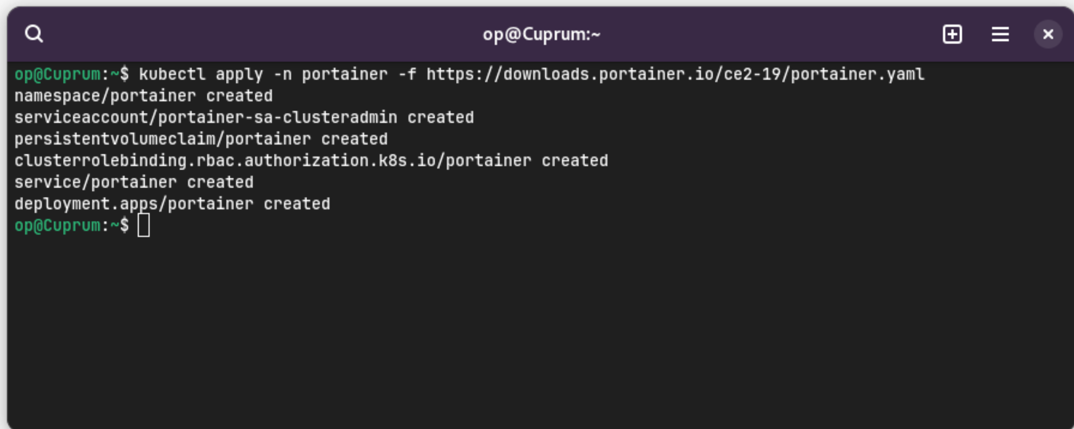

Рисунок 3.14 — Розгортання інструменту Portainer.

Після розгортання для сервісу Portainer будуть відкриті три порти на робочих нодах:

- 30777 для HTTP доступу до API серверу;
- 30778 для доступу до агенту Portainer;
- 30779 для HTTPS доступу до API серверу.

За замовчуванням дані порти відкриті на всіх інтерфейсах. Тому необхідно налаштувати мережеві екрани на робочих нодах таким чином, щоб порти Portainer були недоступні через публічну мережу.

Найпростішим способом це можна зробити з використанням вбудованого інструментарію iptables. За замовчуванням Kubernetes використовує ланцюги iptables для правильної маршрутизації трафіку в кластері, тому правила необхідно додати в ланцюги raw PREROUTING, які обробляють пакети у першу чергу до того, як вони потрапляють у ланцюги створені мережею Kubernetes. Отже, саме правило буде мати наступний вигляд:

iptables -t raw -A PREROUTING -p tcp -i ens19 --match multiport --dports 30777:30779 -j DROP

Воно включає в себе вказівку на таблицю raw в ланцюгу PREROUTING, пакети з протоколом TCP, які надійшли на інтерфейс “ens19” та мають порти призначення в діапазоні 30777-30779, та дії DROP.

Також варто закрити доступ до порту 22/TCP, на якому працює сервер sshd, щоб унеможливити віддалений доступ до консолі операційної системи. Правило для блокування порту 22 буде подібним до портів Portainer:

iptables -t raw -A PREROUTING -p tcp -i ens19 --dport 22 -j DROP

Отже, після застосування даних правил до робочих нод, Portainer та SSH буде доступний лише з внутрішньої мережі. Перевірити це можна за допомогою утиліти Nmap (рис. 3.15).

KubeArmor необхідно встановити за допомогою пакетного менеджеру helm [20]. Даний інструмент дозволяє швидко встановлювати компоненти до Kubernetes кластеру подібно до використання маніфестів, одначе додає певної цілісності операціям. Таким чином за допомогою helm можна не тільки швидко встановлювати, а ще й видаляти компоненти.

```
dmytro@baphomet: ~ | xonsh
dmytro@baphomet ~ @ nmap -p 22,30777,30778,30779 192.168.3.21
Starting Nmap 7.95 ( https://nmap.org ) at 2024-05-22 17:15 EEST
Nmap scan report for Carbon.lan (192.168.3.21)
Host is up (0.0015s latency).

PORT      STATE SERVICE
22/tcp    open  ssh
30777/tcp open  unknown
30778/tcp open  unknown
30779/tcp open  unknown

Nmap done: 1 IP address (1 host up) scanned in 0.02 seconds
dmytro@baphomet ~ @ nmap -p 22,30777,30778,30779 192.168.4.21
Starting Nmap 7.95 ( https://nmap.org ) at 2024-05-22 17:15 EEST
Nmap scan report for Carbon-ext.lan (192.168.4.21)
Host is up (0.0010s latency).

PORT      STATE    SERVICE
22/tcp    filtered ssh
30777/tcp filtered unknown
30778/tcp filtered unknown
30779/tcp filtered unknown

Nmap done: 1 IP address (1 host up) scanned in 1.22 seconds
dmytro@baphomet ~ @
```

Рисунок 3.15 — Перевірка доступності сервісів з різних мереж.

В якості корисного навантаження (app-1, app-2, app-3) в рамках тестування використано вебсервери Nginx, розгорнуті як окремі сервіси. Повний маніфест, що був використаний для розгортання простору імен “app” наведено в додатку Г. У кожен з контейнерів Nginx було примонтовано простий файл з вмістом, який допоможе ідентифікувати додаток. Перевірити правильність маршрутизації Ingress можна за допомогою просто скрипта та утиліти curl (рис. 3.16).

```
dmytro@baphomet ~ @ for i in ["app-1", "app-2", "app-3"]:
..................     print(f"Response from {i}: ", $(curl @(f"http://app.local:30384/{i}/") 2>/dev/null))
..................
Response from app-1:  Hi, I'm app-1

Response from app-2:  Hi, I'm app-2

Response from app-3:  Hi, I'm app-3

dmytro@baphomet ~ @
```

Рисунок 3.16 — Перевірка налаштувань Ingress.

Політики KubeArmor будуть відрізнятися залежно від використання корисних навантажень. В конкретному прикладі використані контейнери з вебсервером Nginx, що містять мінімальний необхідний набір виконуваних файлів та бібліотек. Утім, щоб унеможливити встановлення пакунків та отримання даних з мережі зсередини контейнерів, можна створити політику KubeArmor яка забороняє виконання пакетних менеджерів та інструментів curl і wget. Така політика буде мати наступний вигляд:

```
apiVersion: security.kubearmor.com/v1
kind: KubeArmorPolicy
metadata:
  name: block-pkg-tools-nginx
  namespace: app
spec:
  selector:
    matchLabels:
      group: app
  process:
    matchPaths:
      - path: /usr/bin/apt
      - path: /usr/bin/apt-get
      - path: /usr/bin/dpkg
```

```
        - path: /usr/bin/wget
        - path: /usr/bin/curl
    action:
      Block
```

Після її застосування можна перевірити статус системи KubeArmor за допомогою інструментів командного рядка karmor (рис. 3.17).

```
Armored Up pods :
+---------------+---------------------------------+-----------+---------------------------------------------+---------------------+
|  NAMESPACE    |        DEFAULT POSTURE          | VISIBILITY|                   NAME                      |       POLICY        |
+---------------+---------------------------------+-----------+---------------------------------------------+---------------------+
| app           | File(audit),                    | none      | app-1-deployment-c8fc8789b-5qf2s            | block-pkg-tools-nginx |
|               | Capabilities(audit), Network    |           |                                             |                     |
|               | (audit)                         |           |                                             |                     |
+               +                                 +           +---------------------------------------------+---------------------+
|               |                                 |           | app-2-deployment-6cfdb54654-thc5t           | block-pkg-tools-nginx |
+               +                                 +           +---------------------------------------------+---------------------+
|               |                                 |           | app-3-deployment-dc97cbfdd-chrlj            | block-pkg-tools-nginx |
+---------------+                                 +           +---------------------------------------------+---------------------+
| ingress-nginx |                                 |           | ingress-nginx-admission-patch-x8l4f         |                     |
+               +                                 +           +---------------------------------------------+---------------------+
|               |                                 |           | ingress-nginx-controller-6dc9c5fb7c-k4bhp   |                     |
+---------------+---------------------------------+-----------+---------------------------------------------+---------------------+
op@Cuprum:~$
```

Рисунок 3.17 — Стан захисту подів кластеру системою KubeArmor.

Після налаштування політики виконання вказаних виконуваних файлів стає забороненим навіть для користувача root (рис. 3.18).

```
op@Cuprum:~
op@Cuprum:~$ kubectl exec -n app -it app-1-deployment-c8fc8789b-5qf2s -- bash
root@app-1-deployment-c8fc8789b-5qf2s:/# apt
bash: /usr/bin/apt: Permission denied
root@app-1-deployment-c8fc8789b-5qf2s:/# curl
bash: /usr/bin/curl: Permission denied
root@app-1-deployment-c8fc8789b-5qf2s:/#
```

Рисунок 3.18 — Робота політики безпеки.

Окрім того, попередження про спробу порушення політики безпеки з‘являється в логах системи KubeArmor.

```
== Alert / 2024-05-23 15:59:31.726963 ==
ClusterName: default
HostName: Carbon
NamespaceName: app
PodName: app-1-deployment-c8fc8789b-5qf2s
Labels: app=app-1,group=app
ContainerName: app-1
ContainerID: 5046188be736c47d6a0f2d230b6ab52e58840ffc53e796d9856d9a3703ba7609
ContainerImage: docker.io/library/nginx:lateste784f4560448b14a66f55c26e1b4dad2c2877cc73d001b7cd0b18e24a700a070
Type: MatchedPolicy
PolicyName: block-pkg-tools-nginx
Severity: 1
Source: /usr/bin/bash
Resource: /usr/bin/apt
Operation: Process
Action: Block
Data: lsm=SECURITY_BPRM_CHECK
Enforcer: BPFLSM
Result: Permission denied
Cwd: /
HostPID: 57682
HostPPID: 57085
Owner: map[Name:app-1-deployment Namespace:app Ref:Deployment]
PID: 56
PPID: 48
ParentProcessName: /usr/bin/bash
ProcessName: /usr/bin/apt
UID: 0
```

Рисунок 3.19 — Сповіщення в логу KubeArmor.

Отже під час тестування програмного рішення було реалізовано модель системи в тестовому середовищі, налаштовано мережеві екрани, додаткове програмне забезпечення (KubeArmor, NginxIngress, Potrainer), розроблено тестову політику безпеки та протестовано коректність налаштувань.

За підсумками проведеної роботи, можна констатувати, що налаштування мережевого екрану та політик безпеки KubeArmor працюють належним чином, забезпечуючи очікуваний рівень захисту.

ВИСНОВКИ

У процесі виконання даної роботи було проведено інформаційний огляд архітектур вебдодатків та їх класифікацію, зокрема, монолітну та мікросервісну архітектури. Розглянуто архітектуру кластеру Kubernetes, яку використовують для побудови мікросервісних додатків, джерела загроз та площини можливих атак на нього. Окрім того, основний акцент у роботі поставлено на безпеку в кластерах Kubernetes. У зв’язку з цим проаналізовано кращі практики та методи захисту на різних етапах життєвого циклу вебдодатків від розробки до виконання в кластерах Kubernetes.

У рамках практичної частини спроєктовано та протестовано модель архітектури для захищеного вебдодатку з високою доступністю з використанням вбудованих інструментів операційної системи Fedora CoreOS та системи Kubernetes. Під час тестування програмного рішення реалізовано модель в тестовому середовищі, налаштовано мережеві екрани, додаткове програмне забезпечення (KubeArmor, NginxIngress, Potrainer), розроблено тестову політику безпеки та протестовано коректність налаштувань. Результати тестування показали, що запропоновані налаштування мережевого екрану та політик безпеки KubeArmor працюють належним чином, забезпечуючи очікуваний рівень захисту. В цілому модель є загальною та може модифікуватися в конкретних випадках застосування залежно від потреб конкретного додатку.

Система KubeArmor має доволі розвинений інтерфейс для інтеграції, тому в подальшій перспективі можна імплементувати також її інтеграцію з SIEM системами. Окрім того, в наявній моделі є також перспектива для проєктування статичного сховища для подій кластеру з ухилом на практики забезпечення кібербезпеки.

СПИСОК ЛІТЕРАТУРИ

1. Common web application architectures - .NET. *Microsoft Learn: Build skills that open doors in your career*. URL: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures (date of access: 25.05.2024).
2. High Availability Architecture and Best Practices | FileCloud. *FileCloud blog*. URL: https://www.filecloud.com/blog/an-introduction-to-high-availability-architecture/ (date of access: 25.05.2024).
3. Kubernetes Documentation. *Kubernetes*. URL: https://kubernetes.io/docs/home/ (date of access: 25.05.2024).
4. Academic: Attack Trees - Schneier on Security. *Schneier on Security*. URL: https://www.schneier.com/academic/archives/1999/12/attack_trees.html (date of access: 04.05.2024).
5. NVD - CVE-2024-3094. *NVD - Home*. URL: https://nvd.nist.gov/vuln/detail/CVE-2024-3094 (date of access: 04.05.2024).
6. xz-utils backdoor situation (CVE-2024-3094). *Gist*. URL: https://gist.github.com/thesamesam/223949d5a074ebc3dce9ee78baad9e27 (date of access: 04.05.2024).
7. Administer a Cluster. *Kubernetes*. URL: https://kubernetes.io/docs/tasks/administer-cluster/ (date of access: 04.05.2024).
8. Cloud Native Security and Kubernetes. *Kubernetes*. URL: https://kubernetes.io/docs/concepts/security/cloud-native-security/ (date of access: 04.05.2024).

9. Cloud Native Security Whitepaper. *Cloud Native Computing Foundation*. URL: https://www.cncf.io/wp-content/uploads/2022/06/CNCF_cloud-native-security-whitepaper-May2022-v2.pdf (date of access: 04.05.2024).
10. Zero Trust Architecture. *Cloud Native Glossary*. URL: https://glossary.cncf.io/zero-trust-architecture/ (date of access: 04.05.2024).
11. GitHub - kubearmor/KubeArmor: Runtime Security Enforcement System. Workload hardening/sandboxing and implementing least-permissive policies made easy leveraging LSMs (BPF-LSM, AppArmor). *GitHub*. URL: https://github.com/kubearmor/KubeArmor (date of access: 04.05.2024).
12. Holistic Kubernetes Security for the Enterprise - Aqua. *Aqua*. URL: https://www.aquasec.com/products/kubernetes-security/ (date of access: 04.05.2024).
13. Service Accounts. *Kubernetes*. URL: https://kubernetes.io/docs/concepts/security/service-accounts/ (date of access: 04.05.2024).
14. Good practices for Kubernetes Secrets. *Kubernetes*. URL: https://kubernetes.io/docs/concepts/security/secrets-good-practices/ (date of access: 04.05.2024).
15. Encrypting Confidential Data at Rest. *Kubernetes*. URL: https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/ (date of access: 04.05.2024).
16. Concepts - Secrets Store CSI Driver. *Introduction - Secrets Store CSI Driver*. URL: https://secrets-store-csi-driver.sigs.k8s.io/concepts.html#provider-for-the-secrets-store-csi-driver (date of access: 04.05.2024).
17. A true hybrid image/package system. *rpm-ostree*. URL: https://coreos.github.io/rpm-ostree/ (date of access: 25.05.2024).
18. cri-o. *cri-o*. URL: https://cri-o.io/ (date of access: 25.05.2024).

19. GitHub - flannel-io/flannel: flannel is a network fabric for containers, designed for Kubernetes. *GitHub*. URL: https://github.com/flannel-io/flannel?tab=readme-ov-file#deploying-flannel-manually (date of access: 25.05.2024).
20. KubeArmor/getting-started/deployment_guide.md at main · kubearmor/KubeArmor. *GitHub*. URL: https://github.com/kubearmor/KubeArmor/blob/main/getting-started/deployment_guide.md (date of access: 25.05.2024).

ДОДАТКИ

Додаток А. Дерево атак на кластер Kubernetes.

Додаток Б. Конфігурація Ignition для хостів

```
variant: fcos
version: 1.4.0
passwd:
  users:
        - name: op
          groups:
            - sudo
          ssh_authorized_keys:
            - ssh-ed25519
AAAAC3NzaC1lZDI1NTE5AAAAII9aUzGPRW2+zjMVOlK6ZX+mdJZnFY7HUuSqP6LQJjNf
storage:
  files:
        - path: /home/op/.ssh/internal
          overwrite: true
          contents:
            inline: |
               -----BEGIN OPENSSH PRIVATE KEY-----
               b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAAAMwAAAAtzc2gtZW
               QyNTUxOQAAACCPWlMxj0Vtvs4zFTpSumV/pnSWZxWOx1Lkqj+i0CYzXwAAAJj9WW+F/Vlv
               hQAAAAtzc2gtZWQyNTUxOQAAACCPWlMxj0Vtvs4zFTpSumV/pnSWZxWOx1Lkqj+i0CYzXw
               AAAEAYRhRGuGV0EniJhxpko2WPs0sDKsz1gXkVRdTdiU617I9aUzGPRW2+zjMVOlK6ZX+m
               dJZnFY7HUuSqP6LQJjNfAAAAEGRteXRyb0BtYW50aWNvcmUBAgMEBQ==
               -----END OPENSSH PRIVATE KEY-----
          mode: 0600
          user:
            name: op
          group:
            name: op
        - path: /home/op/.ssh/config
          overwrite: true
          contents:
            inline: |
               Host *
               User op
               IdentityFile /home/op/.ssh/internal
          mode: 0600
          user:
            name: op
          group:
            name: op
        - path: /etc/yum.repos.d/kubernetes.repo
          contents:
            inline: |
               [kubernetes]
               name=Kubernetes
               baseurl=https://pkgs.k8s.io/core:/stable:/v1.28/rpm/
               enabled=1
               gpgcheck=1
               gpgkey=https://pkgs.k8s.io/core:/stable:/v1.28/rpm/repodata/repomd.xml.key
```

```
- path: /etc/zincati/config.d/55-updates-strategy.toml
  contents:
   inline: |
       [updates]
       strategy = "periodic"
       [[updates.periodic.window]]
       days = [ "Sat", "Sun" ]
       start_time = "03:00"
       length_minutes = 60
```

Додаток В. Конфігурація ініціалізації кластеру

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: InitConfiguration
nodeRegistration:
  kubeletExtraArgs:
       volume-plugin-dir: "/opt/libexec/kubernetes/kubelet-plugins/volume/exec/"
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
controllerManager:
  extraArgs:
       flex-volume-plugin-dir: "/opt/libexec/kubernetes/kubelet-plugins/volume/exec/"
networking:
  serviceSubnet: 10.10.0.0/16
  podSubnet: 10.20.0.0/16
controlPlaneEndpoint: 192.168.3.20
---
kind: KubeletConfiguration
apiVersion: kubelet.config.k8s.io/v1beta1
cgroupDriver: systemd
```

Додаток Г. Маніфест для розгортання простору імен “app”.

```
apiVersion: v1
kind: Namespace
metadata:
  name: app
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-1-config
  namespace: app
data:
  index.html: |
       Hi, I'm app-1
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-2-config
  namespace: app
data:
  index.html: |
       Hi, I'm app-2
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-3-config
  namespace: app
data:
  index.html: |
       Hi, I'm app-3
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-1-deployment
  namespace: app
  labels:
       app: app-1
spec:
  selector:
       matchLabels:
       app: app-1
  replicas: 1
  strategy:
       type: Recreate
  template:
       metadata:
       labels:
       app: app-1
       group: app
       spec:
       containers:
       - name: app-1
       image: nginx
       ports:
       - containerPort: 80
```

```
        volumeMounts:
        - name: app-1-index
              mountPath: /usr/share/nginx/html/app-1
        volumes:
        - name: app-1-index
        configMap:
              name: app-1-config
---
apiVersion: v1
kind: Service
metadata:
  name: app-1
  namespace: app
spec:
  selector:
        app: app-1
  ports:
        - protocol: TCP
        port: 80
        targetPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-2-deployment
  namespace: app
  labels:
        app: app-2
spec:
  selector:
        matchLabels:
        app: app-2
  replicas: 1
  strategy:
        type: Recreate
  template:
        metadata:
        labels:
        app: app-2
        group: app
        spec:
        containers:
        - name: app-2
        image: nginx
        ports:
        - containerPort: 80
        volumeMounts:
        - name: app-2-index
              mountPath: /usr/share/nginx/html/app-2
        volumes:
        - name: app-2-index
        configMap:
              name: app-2-config
---
apiVersion: v1
kind: Service
metadata:
  name: app-2
  namespace: app
```

```
spec:
  selector:
       app: app-2
  ports:
       - protocol: TCP
       port: 80
       targetPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-3-deployment
  namespace: app
  labels:
       app: app-3
spec:
  selector:
       matchLabels:
       app: app-3
  replicas: 1
  strategy:
       type: Recreate
  template:
       metadata:
       labels:
       app: app-3
       group: app
       spec:
       containers:
       - name: app-3
       image: nginx
       ports:
       - containerPort: 80
       volumeMounts:
       - name: app-3-index
              mountPath: /usr/share/nginx/html/app-3
       volumes:
       - name: app-3-index
       configMap:
              name: app-3-config
---
apiVersion: v1
kind: Service
metadata:
  name: app-3
  namespace: app
spec:
  selector:
       app: app-3
  ports:
       - protocol: TCP
       port: 80
       targetPort: 80
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-ingress
  namespace: app
```

```
  annotations:
      kubernetes.io/ingress.class: "nginx"
spec:
  rules:
  - host: "app.local"
      http:
      paths:
      - path: /app-1
      pathType: Prefix
      backend:
      service:
            name: app-1
            port:
            number: 80
      - path: /app-2
      pathType: Prefix
      backend:
      service:
            name: app-2
            port:
            number: 80
      - path: /app-3
      pathType: Prefix
      backend:
      service:
            name: app-3
            port:
            number: 80
```