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

«На правах рукопису» «До захисту допущено»

УДК 004.056 В.о. завідувача кафедри

__________ Дмитро ЛАНДЕ

“___”_____________2021 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою «Системи, технології та математичні методи кібербезпеки»

зі спеціальності 125 «Кібербезпека»

на тему: Методи та засоби аудиту безпеки системи автоматичної оркестрації контейнерів Kubernetes

Виконав здобувач ступеня магістра _2 курсу, групи ___ФБ-01мп_
(шифр групи)

Коваленко Ангеліна Миколаївна___________________________ _________
(прізвище, ім’я, по батькові) (підпис)

Науковий керівник к.т.н., доц. Коломицев М.В. ____________ _________
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент ____ _________
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Здобувач ступеня магістра _____________
(підпис)

Київ – 2021 року

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

Рівень вищої освіти – другий (магістерський)
Спеціальність – 125 «Кібербезпека»
Освітньо-професійна програма «Системи, технології та математичні методи кібербезпеки»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
__________ Дмитро ЛАНДЕ
(підпис)
«___»_____________2021 р.

ЗАВДАННЯ
на магістерську дисертацію здобувачу ступеня магістра

Коваленко Ангеліна Миколаївна
(прізвище, ім'я, по батькові)

Тема дисертації **Методи та засоби аудиту безпеки системи автоматичної оркестрації контейнерів Kubernetes**
науковий керівник дисертації к.т.н. Коломицев М.В. ______________ ,
____ ______ ,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «_____» ________ 2021 р. №

2. Термін подання здобувачем дисертації 06.12.2021 р.

3. Об'єкт дослідження Прогалини безпеки контейнерів Kubernetes_______

4. Вихідні дані 1. Попередні дослідження

2. Сформований перелік рекомендацій

5. Перелік завдань, які потрібно розробити:

- Спроектувати дерева рішень на аналізі векторів атак; Створити моделі загроз та провести первинний аналіз, щодо можливих векторів атак і їх мінімізації
- Спроектувати роботу застосунку пасивного аудиту безпеки Kubenretes на основі обраної методології, з графічним інтерфейсом попередньо, обравши

технології та середовище виконання для розробленого програ
забезпечення аудиту Kubernetes

6. Орієнтовний перелік ілюстративного матеріалу

Презентація – «Методи та засоби аудиту безпеки системи автоматичної оркестрації контейнерів Kubernetes»

7. Орієнтовний перелік публікацій

Публікація на тему : Вплив автоматичних засобів аудиту безпеки Kubernetes на якість перевірки застосунку експертом

8. Дата видачі завдання _10.04.2021_____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Отримання завдання	10.04.21-12.04.21	Виконано
2	Збір інформації	13.04.21-21.05.21	Виконано
3	Дослідження предметної області	22.06.21-01.07.21	Виконано
4	Розробка плану роботи	02.07.21-10.07.21	Виконано
5	Дослідження Kubernetes	10.07.21-11.08.21	Виконано
6	Створення моделей загроз та їх аналіз	12.08.21-19.08.21	Виконано
7	Аналіз методів аудиту безпеки Kubernetes	20.08.21-02.09.21	Виконано
8	Аналіз існуючих засобів аудиту	03.09.21-08.09.21	Виконано
9	Розробка прототипів застосунку	09.09.21-15.10.21	Виконано
10	Збір та аналіз отриманої інформації	15.10.21-26.10.21	Виконано
11	Оформлення дипломної роботи	26.10.21-28.11.21	Виконано
12	Попередній захист дипломної роботи	29.11.2021	Виконано

Здобувач ступеня магістра ______________ Ангеліна КОВАЛЕНКО
(підпис) (власне ім’я, ПРІЗВИЩЕ)

Науковий керівник ______________ Михайло КОЛОМИЦЕВ
(підпис) (власне ім’я, ПРІЗВИЩЕ)

АНОТАЦІЯ

Робота обсягом 133 сторінки включає 18 ілюстрацій, 28 таблиць, 29 джерел літератури та 1 додаток.

Об'єктом дослідження у даній магістерській дисертації є програлини безпеки в системі окрестрації контейнерів Kubernetes.

Предметом дослідження є інструмент виявлення уразливостей в Kubernetes.

Методи дослідження – поєднання та аналіз існуючих технологій та розробка нових для виявлення уразливостей в системі оркестрації.

Метою дослідження є аналіз ризиків пов'язаних з використанням Kubernetes та розробка варіантів їх усунення чи зменшення в вигляді методики і її реалізації в застосунку .

Результати роботи можуть використовуватися для сканування безпекових політик та конфігурацій Kubernetes

Ключові слова: кібербезпека, Docker, загрози, Kubernetes, контейнери, сканування аудиту.

ABSTRACT

The 133-page work includes 18 illustrations, 28 tables, 29 references and 1 appendix.

The object of research in this master's dissertation are security gaps in the Kubernetes container identification system.

The subject of the study is a vulnerability detection tool in Kubernetes.

Research methods - combination and analysis of existing technologies and development of new ones to identify vulnerabilities in the orchestration system.

The aim of the study is to analyze the risks associated with the use of Kubernetes and to develop options for their elimination or reduction in the form of a methodology and its implementation in the application.

The results can be used to scan Kubernetes configurations.

Keywords: Cybersecurity, Docker, Threats, Kubernetes, Containers, Audit Scan.

ЗМІСТ

Перелік умовних позначень, скорочень і термінів 8
Вступ 9
1 Аналіз предметної області 11
1.1 Монолітна архітектура 11
1.2 Мікросервісна архітектура 15
1.3 Контейнери 22
1.4 Docker: визначення, опис архітетури 25
1.5 Необхідність використання оркестрації, передумови впровадження Kubernetes 29
1.6 Kubernetes: опис, загальна архітектура, актуальність 33
Висновки до розділу 1 50
2 Безпека Kubernetes та Docker 52
2.1 Безпека Docker контейнерів 52
2.2 Безпека Kubernetes 68
2.3 Порівняння підходів забезпечення безпеки Kubernetes 77
Висновки до розділу 2 80
3 Аналіз існуючих рішень та постановка завдання 82
3.1 Сканер kube-hunter 82
3.2 Сканер Chechov 85
3.3 Недоліки існуючих рішень 86
3.4 Аналіз гілок досліджуваного застосунку 87
3.5 Постановка задачі 89
Висновки до розділу 3 90

4 Опис роботи розробленого рішення для систем оркестрації Kubernetes 92

4.1 Огляд технологій розробки системи 92

4.2 Опис функціональності застосунку 103

4.3 Результати роботи застосунку 107

Висновки до розділу 4 110

5 Маркетинговий аналіз стартап-проекту 111

5.1 Опис ідеї проекту 112

5.2 Технологічний аудит ідеї проекту 114

5.3 Аналіз ринкових можливостей запуску стартап-проекту 115

5.4 Розроблення ринкової стратегії проекту 122

5.5 Розроблення маркетингової програми стартап-проекту 124

Висновки до розділу 5 127

Висновки 128

Перелік джерел посилань 130

Додаток А Дерево рішень «Здобуття конфіденційних даних» 133

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

БД – база даних;

HTTP – протокол передачі гіпертексту;

ОС – операційна система;

API – application programming interface;

CPU – central processing unit;

GPU – graphics proccesing unit.

ВСТУП

Проект Kubernetes останнім часом набув популярності як найбажаніше середовище для розгортання серверного програмного забезпечення. Це зростання відбулося у зв’язку з переходом до мікросервісної архітектури, в якій складні додатки розділяються на окремі компоненти, що обмінюються інформацією через сервіс-сервіс API. Платформа Kubernetes принципово відрізняється від усіх до теперішніх обчислювальних інфраструктур. Вона використовує власну політику розміщення даних для розподілу компонентів застосунку, при цьому забезпечується динамічна зміна розкладу або масштабування. Крім того, можна в будь-який момент видалити або додати нові компоненти програми. Kubernetes потрібен якщо:

1. Потрібно забезпечити високу доступність системи.
2. Система динамічно розвивається, і потрібно оперативно доставляти зміни для продуктової середовища.
3. Колектив розробників функціонує як єдина команда від коду до продуктової середовища.
4. Створюється динамічна система, що розвивається і яка буде експлуатуватися роками.

Однак, використання Kubernetes не знімає з розробників і тестувальників завдання пошуку вразливостей в застосунках. Це завдання може вирішуватися різними способами, як традиційними так і такими, що використовують можливості середовища контейнеризації.

Актуальність дослідження: Контейнеризація застосунків - один з головних трендів сучасних IT-розробок. Однак, у контейнерів є один істотний недолік для масового споживача - складна настройка масштабування.

Рішенням стали автоматичні системи управління контейнеризацією, найбільш популярною з яких є Kubernetes. Це програмне забезпечення з відкритим вихідним

кодом від компанії Google завоювало визнання завдяки поєднанню гнучкості, безпеки і потужності. Забезпечення безпеки Kubermetes є наразі найбільш критичним напрямком дослідження у сфері кібербезпеки, адже не існує певного єдиного інструменту, що б допоміг проводити аудит безпеки в повній мірі, без ймовірності пошкодженння застосунку для тестування та покривав більшість загроз.

Об'єктом дослідження у даній магістерській дисертації є програлини безпеки в системі окрестрації контейнерів Kubernetes.

Предметом дослідження є інструмент виявлення уразливостей в Kubernetes.

Методи дослідження – поєднання та аналіз існуючих технологій та розробка нових для виявлення уразливостей в системі оркестрації.

Метою дослідження є аналіз ризиків пов'язаних з використанням Kubernetes та розробка варіантів їх усунення чи зменшення в вигляді методики і її реалізації в застосунку .

Практичне значення роботи полягає в тому що зроблені наопрацювання можна використовувати в роботі компаній для швидкої перевірки безпекових налаштувань та політик безпеки.

Наукова новизна дослідження: розробка програмного забепечення, який при малому втручанні користувача сканує Kubernetes на наявність прогалин безпеки, на основі аналізу моделей загроз, та на виході видає зручний збірний звіт.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

У даному розділі буде розглянуто детальніше про досліджувані технолгії, основні відомості про будову та архітектуру Kubernetes, Docker, а також проаналізована актуальність їх використання та які є переваги та недоліки даних інструментів.

1.1 Монолітна архітектура

Монолітна архітектура є класичним підхід у реалізації застосунків будь-якої спрямованості та навантаження. Вона являє собою один великий сервіс (моноліт), який містить та обробляє в собі всю бізнес-логіку програми. Такою архітектурою користується переважно ті, хто не має на меті зробити великий і масштабний застосунок, або ті ж, котрі готові змиритися і прийняти певні недоліки даного підходу. Спрощена схема роботи такого сервісу представлена на рис. 1.1.

На прикладі шляху запиту користувача розглянемо особливості роботи монолітної архітектури. Наприклад, існує запит від користувача з описом товару від сервісу, який потратить в подальшому на спеціальний сервер, що використовується для віддачі статичного контенту користувачеві, якщо користувач запросив саме його, або відбудеться перенаправлення запиту на моноліт, якщо запит потребує додаткових даних чи інформацію.

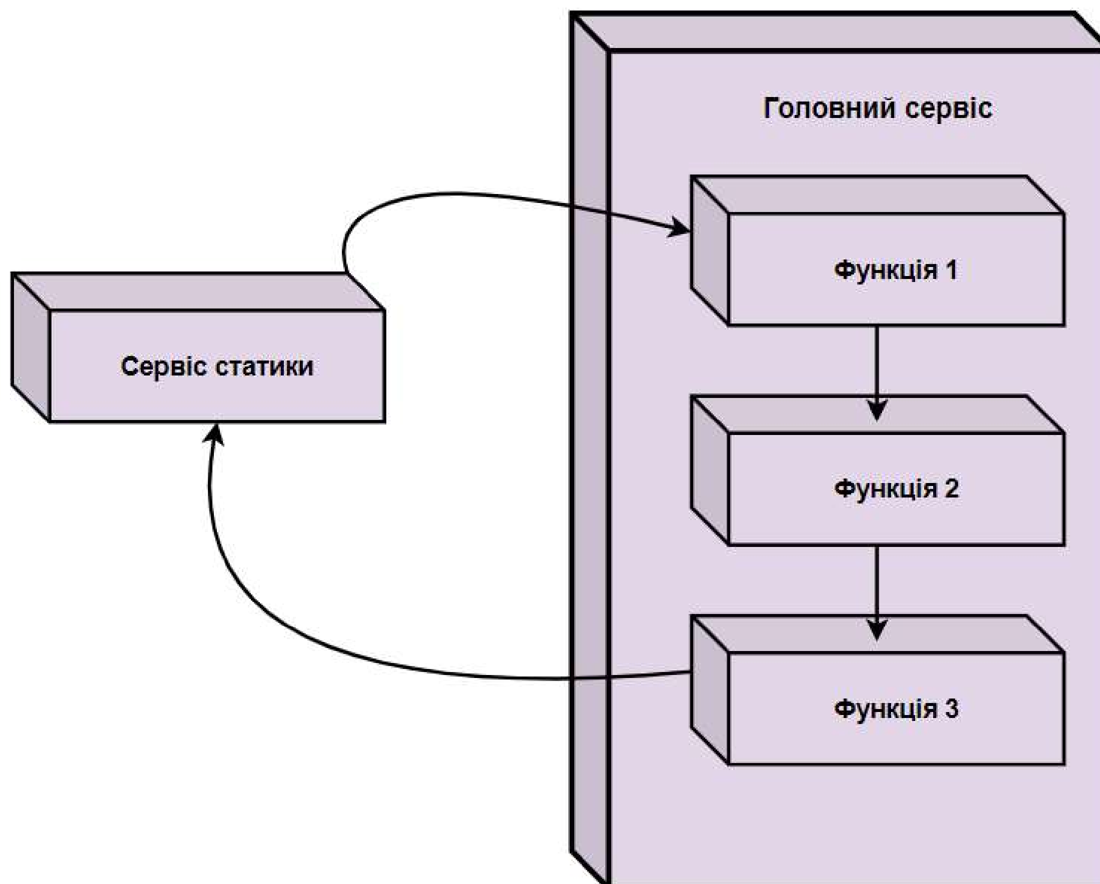


Рисунок 1.1 – Схема монолітної архітектури застосунку

Цей сервіс покликаний розвантажити роботу основного моноліту від певних категорій запитів. Після цього запит потрапляє на моноліт безпосередньо, де спочатку розбивається по певним параметрам, для того щоб визначити які саме функції необхідно буде в подальшому викликати і яку саме інформацію від них одержати. Далі відбувається вже виклик певної потрібної функції, кожна з яких має свою зону відповідальності. Функціями можуть бути наприклад запити до бази даних для отримання інформації з приводу певного об'єкту, фільтрація відомого набору даних, збирання та злиття даних від декількох інших функцій, тощо. Після отримання відповіді від усіх функцій формується відповідь, яка відправляється назад користувачеві.

Головна перевага такої системи – вся бізнес-логіка застосунку знаходиться в одному місці. При необхідності змінити якийсь параметр, можна досить легко (при

хорошій організації структури проекту) знайти залежність усіх параметрів між собою. До мінусів такого підходу можна віднести одну точку входу програми, яка є єдиною точкою і вразі збою відбудеться відмова всієї системи. Також у разі великої кількості коду стає складно внести до системи певні зміни. Однак, при використанні такого підходу на великих високонавантажених проектах виникає низка проблем. Наприклад, кілька сотень користувачів застосунку одночасно надішлють запит до подібного сервера. У моноліту існує одна точка входу, а отже, всі ці запити будуть повинні пройти через неї. Це збільшує час очікування від програми у декілька разів. Якщо потрібно зробити кілька запитів до бази даних і навіть за умови, що сервер працює у багатопотоковому режимі, а база даних підтримує велику кількість з'єднань, то з часом виникне ситуація, коли функції доведеться чекати у черзі задля виконання потрібного запиту до БД. Функція не може продовжити свою роботу через те, що у моноліті функції виконуються послідовно і складно розпаралелити обробку одного запиту, а отже необхідний запит буде простоювати в черзі, очікуючи на можливість відновити свою роботу. А отже час, що буде затрачений на очікування, буде зростати, а тому rps буде маленьким. Дану ситуацію дещо можна покращити, розпаралеливши виконання функцій за допомогою закупівлі дороговартісного додаткового обладнання. Наприклад, вирішити дану проблему можна частково паралельно поточному серверу запустити ще один такий самий, для якого необхідно буде налаштувати ідентичну інфраструктуру та явно вказати серверу статики про наявність можливості посилати запити до іншого сервера. Після проведених маніпуляцій rps значення дещо покращиться, однак цей спосіб є фінансово не вдалий. Збільшення потрібних метрик може бути не таким суттєвим як хотілось би. Крім того, у певний момент роботи може виникнути ситуація, коли впровадження нового фунціоналу стане дороговартісним завданням. Так як все перебуває в одному місці, згідно даної архітектури, то буває дещо складно знайти потрібну частину програми, яку потрібно виправити.

Розглянемо інший важливий критерій розробки продукту – можливість забезпечити якість продукту, що розробляється. Тестування є невід'ємною частиною

комплексу заходів для перевірки та забезпечення якості продукту. Наразі у сучасних веб-застосунках використовується кілька різних напрямів тестування, таких як : юніт-тестування, авто-тестування, інтеграційне тестування, тощо. Надалі будемо розглядати докладніше дані види тестування з точки зору розробки монолітного застосунку.

Юніт-тестування – вид тестування, що перевіряє функціонал окремо взятої функції без прив’язки до оточення. Підхід даного виду тестування полягає в виклику функції з усіма можливими комбінаціями вхідних змінних та наступним порівняннями отриманих результатів з очікуваними. Незалежно від архітектурного підходу функціоналу, що розробляється, необхідно писати тестувальниками юніт-тести. Але в випадку застосунку з монолітною архітектурою виникають проблеми з підтриманням та написанням даних тестів. На початкових етапах розробки дане тестування не викликає жодних труднощів, проте зі збільшенням кількості коду, починаються проблеми з підтримкою. Через велику кодову базу бувають ситуації, коли складно знайти, розібратися та поправити цей же ж юніт-тест, або ж написати новий. Тому надалі виникає ситуація, коли є велика кількість частин коду, які частково покриті тестами, що не можуть бути актуалізовані під новий функціонал, а на іншу частину функціоналу взагалі їх немає.

Авто-тестування – вид тестування, за якого в автоматичному режимі перевіряється будь-який функціонал або сценарій роботи програми. Відрізняється від юніт-тестування тим, що перевіряє не окремий невеликий функціонал, а цілу частину роботи бізнес-логіки програми. Даний вид тестування ж важливою частиною забезпечення якості продукту, оскільки дозволяє не витрачати час на ручну перевірку функціоналу, а виконати її в автоматичному режимі. Проте все ж стикається з такими проблемами як і юніт-тестування – складність розробки при наявності великої кодової бази. Через різноманітність бізнес-логіки в моноліті застосунку, написати та підтримувати автотести для всього функціоналу є проблематичним, адже зміна будь-якої частини бізнес-логіки вимагатиме виправлень у всіх тестах, в яких використовується дана частина логіки.

Інтеграційне тестування – вид тестування, за якого перевіряється взаємодія сервісу під час роботи з іншими сервісами, та може бути частиною авто-тестування. Прикладом даного виду тестування може слугувати перевірка можливості отримання будь-яких даних із без даних. У монолітній архітектурі все зосереджено в одному сервісі, тому інтеграційне тестування може звестися до перевірки поведінки під час роботи з серверами баз даних.

Окрім усіх вищезгаданих видів тестування розглянемо ще один вид – навантажувальне. Метою даного виду тестування є визначення максимально можливого навантаження на систему, а також перевірка поведінки при його поступовому збільшенні. Необхідність навантажувального тесту полягає в тому, що при нездатності системи витримати велику кількість запитів, запити від користувачів мають в кращому випадку потрапляти в чергу і виконуватися в міру обробки інших запитів черги. Це спричинить лише збільшення часу обробки запиту. У гіршому випадку система взагалі перестане працювати при досягненні певного навантаження. У переважній більшості бази даних є тим місцем, швидкість роботи якого залежить від навантаження. Тому першочергово саме на них необхідно звертати увагу при запуску тестів на навантаження. У результаті можна помітити, що проблема при тестуванні архітектури монолітного типу зводиться до великої кількості елементів, що необхідно перевірити. Через певний проміжок часу стає досить складно підтримувати необхідну якість програми.

1.2 Мікросервісна архітектура

Мікросервіс – незалежний, автономний ресурс, що спроектований як окремо виконуваний файл або ж процес, і взаємодіє з іншими мікросервісами через стандартні, але легковагі міжпроцесорні зв’язки, такі як протокол передачі гіпертексту (HTTP), веб-служби RESTful, черги повідомлень та інші. Унікальністю мікросервісів є те, що кожен з них розробляється, тестується, розгортується і маштабується незалежно від інших мікросервісів.

Ідея використання мікросервісів заснована на сучасних, кращих принципах розробки програмного забезпечення, в тому числі таких як – слабкий взаємозв'язок, високе масштабування і орієнтованість на служби. Наразі це є трендом у сфері розробки і розробляється велика кількість застосунків (близько 82%) саме за принципами даної архітектури.

Кожен мікросервіс виконує рівно одну функцію, яка поводиться однаково для всіх споживачів. Візьмемо, наприклад, службу управління замовленнями, яка тільки обробляє замовлення і більше нічого (навіть повідомлень не надсилає). Але вона може викликати інший мікросервіс, що відповідає за відправлення повідомлень про обробку. Такий поділ функцій забезпечує достатню гнучкість, тому що кожен мікросервіс можна розвивати, підтримувати, масштабувати, розширювати та замінювати незалежно від інших мікросервісів.

Додаток на основі мікросервісів – це просто група з кількох незалежних та автономних мікросервісів, кожен з яких реалізує чітко визначену функцію та для забезпечення спільної функціональність програми взаємодіє з іншими мікросервісами через чітко визначені протоколи. Цю парадигму можна описати як архітектуру, в якій кожен мікросервіс виконується в окремому процесі.

На відміну від монолітної архітектури, мікросервісна архітектура є більш сучасним підходом до організації великого веб-програми. Спрощена схема програми, що працює за принципами мікросервісної архітектури, представлена на рис. 1.2

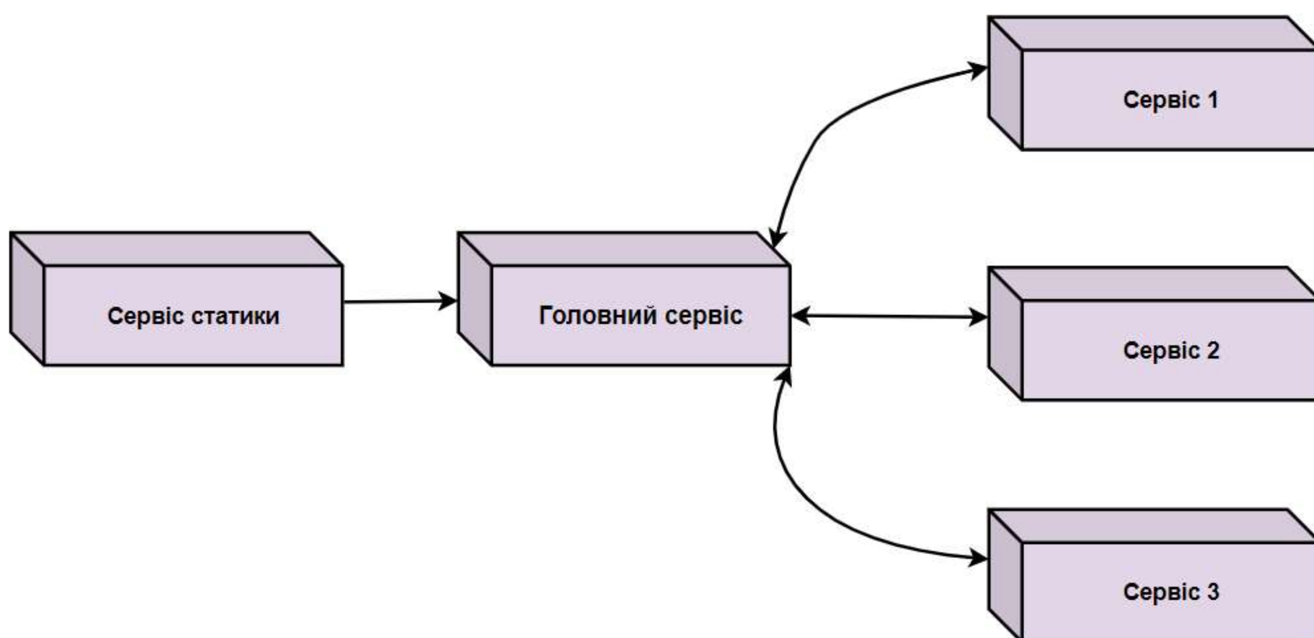


Рисунок 1.2 – Схема роботи застосунку з мікросервісною архітектурою

Таким же чином як і у монолітної архітектури, надісланий запит клієнта спочатку потрапляє на сервер статики, а потім перенаправляється на основний сервіс. У ньому також визначаються параметри та дані, що необхідно одержати для успішного виконання запиту. Але подальший принцип роботи значно відрізняється від моноліту: замість того, щоб обробляти всі запити в одному місці, основний сервіс розсилає підзапити до мікросервісів, що повертають назад певні дані. Мікросервіси дещо схожі на функції моноліту: кожен такий сервіс має свою зону відповідальності і найчастіше не має прямого доступу до інших мікросервісів. Після обробки кожним мікросервісом підзапиту та надсилання відповіді основному сервісу, дані збираються та прямують назад до клієнта. Головною перевагою даного підходу є розбиття функціоналу програми на маленькі незалежні модулі, які можуть працювати в паралельному режимі. За такого підходу до архітектури програми сервер може надіслати запит в мережу для одержання запиту від мікросервісу. Мережевий запит в деяких ситуаціях може виконуватись довше, аніж виклик звичайнісінької функції, оскільки потрібен ще додатковий час на встановлення з’єднання, обмін даними між сторонами-учасниками та обробку безпосередньо самого запиту мікросервісом.

Проте більшість сучасних фреймворків та бібліотек, що використовуються при сервісній розробці, здатні виконувати асинхронні запити або утримувати відкриті з'єднання для того, щоб не витрачати час на його встановлення. Подібні запити не блокують виконання основного коду в очікування відповіді сервісу. Таким чином, можна підвищити кількість оброблюваних запитів за рахунок виконання великої кількості асинхронних процесів.

1.2.1 Переваги мікросервісів у порівнянні з монолітом

Підсумовуючи, крім перерахованих вище переваг, мікросервіси можуть запропонувати такі вигоди:

Простота. Кожен мікросервіс виконує лише одну чітко визначену функцію, тому потрібно менше коду, а отже і існує менше залежностей від іншого коду та зменшується ймовірність помилок;

Масштабованість. Для масштабування монолітної програми необхідно розгорнути застосунок на декількох серверах і налаштувати балансувальник навантаження, бо неможливо масштабувати лише частину програми. Тут діє принцип «усе або нічого». З використанням мікросервісної архітектури можна масштабувати тільки компоненти, що піддаються високому навантаженню. Можливість диференційованої масштабованості є дуже простою і важливою особливістю мікросервісів;

Безперервне розгортання. Завдяки меншій кількості взаємозалежностей у коді та більш швидкому циклу розробки парадигма мікро сервісів підтримує культуру безперервного розгортання та інтеграції розробки та експлуатації (DevOps) і фактично підштовхує до її використання;

Більше свободи та менше залежностей. Мікросервіси за визначенням уже є автономні та незалежні. Команда розробників може зосередитися на своєму мікросервісі і вільно розширювати його можливості, не побоюючись зруйнувати

роботу іншого мікросервісу, поки вони гарантують незмінність інтерфейсу або реалізують новий інтерфейс, він є зворотно сумісний з колишнім;

Ізоляція відмов. Ізоляція відмов - це явище, коли відмова в одній

частини системи не призводить до відмови всієї системи, тобто відмови виявляються ізольовані від системи. У монолітному додатку збій у будь-якій його частини призведе до збою всієї програми, тому що вона являє собою єдиний процес. У випадку з мікросервісами ситуація виявляється дещо інакшою: збій в одному мікросервісі може призвести до відмови цього мікросервісу, але не обов'язково призведе до відмови всієї програми, тому що мікросервіс, що відмовив, виконується в окремому процесі. Наприклад, у системі електронної комерції, заснованої на архітектурі мікросервісів, у разі збою мікросервісу «відгуки про продукт», користувачі, як і раніше зможуть переглядати список товарів, що є в наявності, вибирати товари для покупки, переглядати вміст купівельного кошика та розміщувати замовлення. Єдине, чого вони не зможуть, – це побачити відгуки, доки не буде виправлено мікросервіс. Якби додаток був монолітним, помилка в компоненті, що відповідає за перегляд відгуків, могла б перервати роботу всього застосунку;

Поділ та децентралізація даних. На відміну від монолітних додатків, де всі дані зазвичай зберігаються разом у центральній базі даних, мікросервіси дають можливість розділити дані. Кожен мікросервіс може володіти лише своїми даними та не ділитися ними з іншими мікросервісами;

Широта вибору. На відміну від монолітної програми, де всі компоненти використовують єдину базу даних, платформу і повинні бути написані однією мовою програмування, мікросервіси дають можливість використовувати інструменти, які краще підходять для кожного конкретного випадку та вирішення нових задач. Один мікросервіс може використовувати Oracle та ОС Linux, а інший – NoSQL та Microsoft Windows. Більше немає необхідності пов'язувати себе з певними стеками технологій.

1.2.2 Недоліки мікросервісів

У даній архітектурі є не тільки ряд переваг, а ще й деякі недоліки. Деякі проблеми, характерні для мікросервісів:

Складність пошуку та усунення несправностей. Мікросервіси пропонують свої можливості за допомогою механізму взаємодій між мікросервісами, що збільшує кількість потенційних точок відмови. Це робить відповіді на такі питання складнішими: «Наскільки нормально працює моя система зараз? Якщо кінцевий користувач повідомляє про таку проблему, як низька продуктивність або тривалі періоди очікування, з чого почати усунення несправностей?». Відстежити шлях обробки запиту у монолітному застосунку набагато простіше. Але у застосунку, що складається з мікросервісів, кожен запит може бути розбитий на кілька запитів, що обробляються різними мікросервісами. Пошук та усунення несправностей у цьому випадку можуть стати трохи складніше;

Збільшені затримки. Внутрішньо процесні взаємодії (як і в монолітних застосунках) виконуються набагато швидше між процесних (як у випадку з мікросервісами);

Складність супроводу. Коли програма складається з сотень або навіть тисяч мікросервісів, групам оперативного супроводу доводиться долати складності, пов'язані з налаштуванням інфраструктури, розгортанням, моніторингом, резервним копіюванням та управлінням. Можна навіть сказати, що складності монолітної архітектури переносяться на системний бік мікросервісів. Але даний недолік нівелюється збільшенням штату працівників та покращення ефективності їх роботи та за допомогою високого рівня автоматизації;.

Керування версіями. Через те, що програма може складатися з тисяч мікросервісів, управління версіями стає трохи складніше. Для його здійснення потрібно використовувати більш досконалі системи управління версіями.

Як і у випадку монолітної архітектури, виникає необхідність перевірки якості роботи окремих мікросервісів. Для цього потрібно розглянути різні варіанти тестування мікросервісів. На відміну від моноліту, вони мають меншу кодову базу, а це означає, що покрити мікросервіс юніт-тестами та підтримувати їх значно легше. Подібною ситуація є і з автотестами. Набагато простіше буде покрити тестами мікросервіс, оскільки він містить в собі лише частину бізнес-логіки програми. Але у разі покриття самого сервісу тестами, необхідно переконатися, що сам застосунок працює коректно разом з іншими мікросервісами. Для цього необхідно написати додаткові автотести, що будуть повторювати всі сценарії користувача. Що стосуться інтеграційного тестування, то його важливість для мікросервісу, на відміну від монолітної архітектури, стає більш вагомим і необхідним. Так як мережна взаємодія – єдиний вид взаємодії між мікросервісами, то даному виду тестування потрібно приділити більше уваги, аніж для монолітних функцій. У план для тесту для інтеграційного тестування входить перевірка можливості взаємодії з іншими мікросервісами, а також результат даної взаємодії. За результатом тестування можна буде будувати висновки про доступність сервісу для інших. Разом зі збільшенням важливості інтеграційних тестів, зростає необхідність у правильно складених тестах навантаження. Адже може виникнути ситуація, коли мікросервіси не зможуть між собою обмінюватись інформацією\запитами, бо велике навантаження між ними не дозволяє швидко обробити запити, що дещо ускладнює задачу у складенні подібних тестів. У ж моноліті потрібно було покривати подібними тестами лише спілкування його з базою даних.

Аналізуючи вищеописані можливості проведення тестування, можна помітити, що там, де виникають труднощі у тестуванні застосунку монолітної архітектури, у тестуванні мікросервісу їх немає, і навпаки. У зв’язку з тим не можна однозначно стверджувати на основі даної інформації, що мікросервіс або моноліт краще, так як у кожного підходу існують свої складнощі під час підготовки та розробки тестів. Проте переваги мікросервісної архітектури є значно більшими, аніж складнощі, що виникаю у роботі з нею.

1.3 Контейнери

Якщо застосунок складається з невеликої кількості великих компонент, то цілком можливо надати кожному з них виділену віртуальну машину та ізолювати середовища, надавши кожному їх власний екземпляр операційної системи.

Проте коли компоненти є не досить великими, але їхня кількість зростає, то стає не можливим надати кожному свою віртуальну машину. Це може потягти за собою більші витрати апаратних ресурсів, але справа не тільки у їх розтраті. Так як кожна віртуальна машина зазвичай налаштовується та керується індивідуально, то збільшення їхньої кількості також призводить до витрачання ще й людських ресурсів. Тому замість використання віртуальних машин для ізоляції середовищ кожного з мікросервісів, розробники звертаються до контейнерних технологій. Дані технології дозволяють запускати кілька сервісів на одній хост-машині, не тільки забезпечуючи доступ до різних середовищ, але і ізолюючи їх один від одного, подібно до тих самих же віртуальних машин, проте з набагато меншими витратами.

Отже, контейнер — це ізольований, невибагливий до ресурсів приймач команд, призначений для запуску програми операційної системи сервера. Контейнери реалізуються поверх ядра операційної системи вузла (яке можна вважати своєрідним фундаментом операційної системи) і містять лише додатки та деякі API-інтерфейси та служби операційної системи, що працюють у режимі користувача.

Процес, що є запущений у контейнері, виконується всередині операційної системи хоста, як і інші процеси, але він є ізольованим. Для самого процесу це виглядає, ніби він є єдиним працюючим на машині і в її операційній системі.

Контейнери є набагато злегшені, у порівнянні з віртуальними машинами, що дає змогу запускати більшу кількість програмних компонентів на тому самому обладнанні. Це відбувається тому, що кожна машина повинна запускати свій власний набір системних процесів, який вимагає ще більше обчислювальних ресурсів на

додаток до тих, які споживаються власним процесом компонента. У цілому, контейнер – ніщо інше, як окремий ізольований процес, що виконується в центральній ОС, що споживає тільки ті ресурси, які додаток вимагає, без накладних витрат у вигляді додаткових процесів. Варто зазначити, що створення віртуальної машини є доцільним, коли ви хочете самостійно керувати налаштуваннями ядра, у той час коли контенери ділять одне ядро, що може стати проблемою в безпеці.

Наприклад, це може знадобитися:

- якщо встановлено специфічне обладнання або виникає конфлікт апаратного забезпечення стандартним ядром;

- щоб задіяти властивості, яких немає в зборках ядра, що поставляються (наприклад, підтримка high memory);

- для оптимізації ядра, видаляючи непотрібні драйвери зменшення часу завантаження;

- Створення монолітного ядра, без модулів;

- для встановлення оновленого або ядра, що розробляється.

Детальніше відмінності між контейнерами та віртуальними машинами можна переглянути в табл. 1.1 [1].

Таблиця 1.1 – Функції віртуальної машини у порівнянні з контейнерами

Функція	Віртуальна машина	Контейнер
Ізоляція	Забезпечує повну ізоляцію від операційної системи вузла та інших віртуальних машин. Це корисно, коли важлива строга межа безпеки, наприклад, для поділу додатків від компаній, що конкурують, на одному сервері або в кластері.	Зазвичай надає спрощену ізоляцію від вузла та інших контейнерів, але не надає такої надійної межі безпеки, як у випадку віртуальних машин.
Операційна система	Містить повноцінну операційну систему, включаючи ядро, тому	Запускає частину операційної системи в режимі користувача і

	потребує більше системних ресурсів (ЦП, пам'яті та сховища).	її можна адаптувати, щоб вона містила лише необхідні служби для програми, що дозволить використовувати менше системних ресурсів.
Сумісність з гостьовою системою	Працює практично з будь-якою операційною системою у віртуальній машині	Працює на тій самій версії операційної системи, що і вузол
Постійне сховище	Використовує віртуальний жорсткий диск (VHD) для локального сховища для однієї віртуальної машини або спільний файловий ресурс SMB для спільного використання сховища	Використовує диски Azure для локального сховища для одного вузла або служби файлів Azure (спільні ресурси SMB) для спільного використання кількома вузлами або серверами сховища.
Балансування навантаження	Балансування навантаження віртуальної машини переміщає віртуальні машини, що виконуються, на інші сервери у відмовностійкому кластері.	Самі контейнери не переміщуються. Натомість Orchestrator може автоматично запускати або припиняти роботу контейнерів на вузлах кластера для керування змінами навантаження та доступності.
Відмовостійкість	Віртуальні машини можуть виконувати відпрацювання відмови на інший сервер у кластері з перезапуском операційної системи віртуальної машини на новому сервері.	У разі збою вузла кластера всі контейнери, що працюють на ньому, швидко перетворюються Orchestrator на іншому вузлі кластера.

Мережа	Використовує віртуальні мережеві адаптери.	Використовує ізольоване уявлення віртуального мережного адаптера, надаючи меншу віртуалізацію: брандмауер вузла використовується контейнерами спільно - використовується менше ресурсів.

1.4 Docker: визначення, опис архітетури

Хоч і контейнерні технології існують вже досить довго, проте найбільшої популярності вони набули з появою контейнерної платформи Docker. Docker була першою контейнерною системою, що зробила контейнери, які легко переносяться на різні машини. Це спростило процес упаковки не тільки безпосереднь програми, але і всіх його бібліотек та навіть файлової системи операційної системи, в простий , транспортабельний пакет, який може використовуватись для підготовки застосунку до роботи на будь-якій іншій машині, де встановлений Docker.

Виконувана програма, запакована за допомогою Docker, може бачити точний вміст файлової системи, що постачається разом з нею. Незалежно від того чи працює вона на основній машині, призначеній для розробки, або ж на машині з робочого оточення, програма бачить ті ж самі файли, навіть якщо на робочому сервері запущена зовсім інша ОС. Не має значення, має чи ні сервер зовсім інший набір встановлених бібліотек порівняно з машиною для розробки, бо застосунок не буде бачити нічого з сервера, на якому він виконується.

Замість використання великих монолітних образів віртуальних машин у Docker застосовуються образи контейнерів, які зазвичай менші. Ключова різниця між образами контейнерів на основі Docker та образами віртуальних машин полягає в

тому, що образи контейнерів складаються з шарів, які можна спільно повторно використовувати в декількох різних образах. Це означає, що повинні завантажуватись лише певні шари образу, якщо інші шари вже були завантажені раніше під час запуску іншого образу контейнера, який містить ті ж самі шари.

Отже, Docker - це платформа для пакування, виконання та розповсюдження застосунків; дозволяє упаковувати програму разом з усім його середовищем. Це може бути або бібліотеки, що необхідні застосунку, або навіть всі файли, що зазвичай доступні в файловій системі встановленої ОС. Docker дозволяє переносити цей пакет в центральний репозиторій, з якого він потім може бути перенесеним на будь-який комп'ютер, на якому працює Docker, та бути виконаним там.

Архітектура Docker

Docker використовує клієнт-серверну архітектуру і складається з клієнта – утиліти docker, яка звертається до сервера за допомогою REST API, і демона в операційній системі GNU/Linux (Dockerd) [2]. Хоча Docker працює і в інших операційних системах, проте надалі буде матися на увазі, при детальному огляді, саме ця ОС.

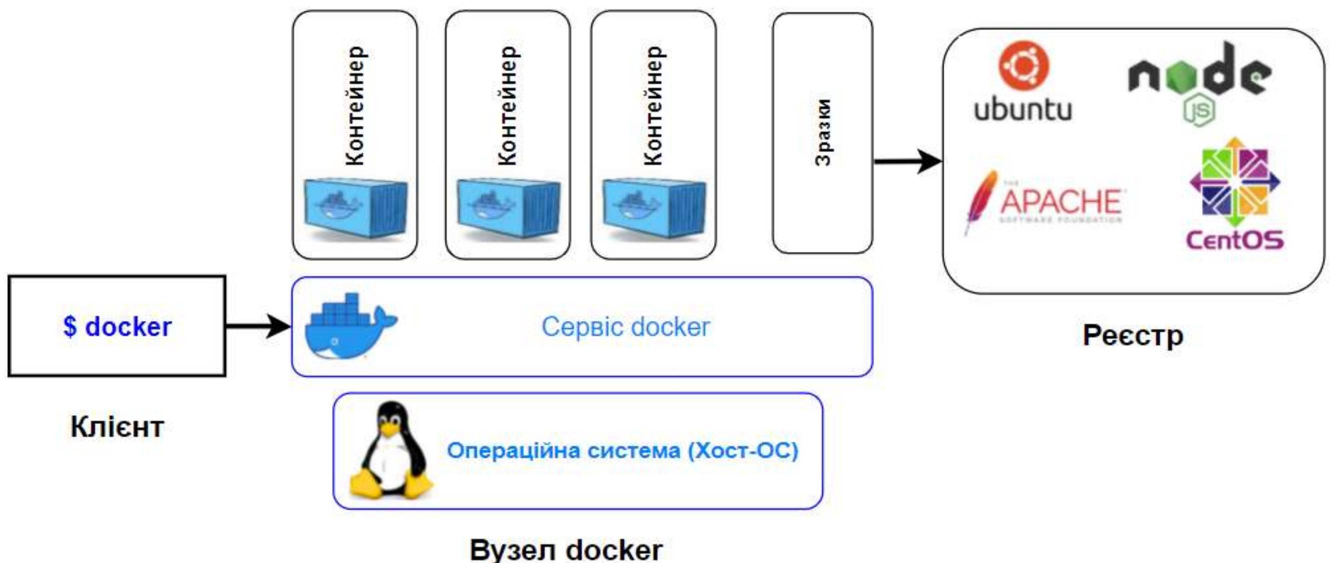


Рисунок 1.3 – Архітектура контейнерів Docker у GNU/Linux

Для того щоб перейти до огляду безпеки даної платформи та контейнерів уцілому потрібно спочатку детальніше ознайомитися з архітектурою. Основними компонентами Docker є:

- **контейнери** – ізольовані за допомогою технологій операційної системи оточення користувачів,в яких виконуються компоненти застосунку. Розробники проекту Docker сповідують принцип: один контейнер – це один компонент застосунку. Контейнер на основі Docker – це звичайний контейнер Linux створений з образу контейнера на основі Docker. Виконуваний контейнер Docker – це процес, запущений на хості, на якому працює Docker, але повністю ізольований як від хоста, так і від усіх інших процесів, запущених у ньому. Процес є також обмежений ресурсами, тобто він може отримувати доступ і використовувати лише той обсяг ресурсів (ЦП, ОЗУ тощо), який йому виділено.
- **образи** - це те, у що ви пакуєте свою програму та її середовище. Він містить файлову систему, яка буде доступна застосунку, та інші метадані, такі як шлях до виконуваного файлу, який має бути виконаний під час запуску образу. Поверх існуючих образів можуть додаватися нові рівні образів, які спільно репрезентують файлову систему, змінюючи або доповнюючи попередній рівень. Зазвичай новий образ створюється або за допомогою зберігання вже запущеного контейнера в новий образ поверх існуючого, або за допомогою спеціальних інструкцій для утиліти Dockerfile. Для поділу різних рівнів контейнера на рівні файлової системи можуть використовуватися UnionFS, aufs, btrfs, vfs, OverlayFS і Device Mapper, тощо;
- **сховища** (registry), що містять репозиторії (repository), у яких зберігаються образи Docker та який спрощує обмін цими образами між різними комп'ютерами. Створюючи образ, можна або запустити його на комп'ютері, на якому він був створений, або відправити образ у сховище, а потім за необхідності витягти (завантажити) його на іншому комп'ютері та запустити його там. Сховища можуть бути як приватними, так і загальнодоступними. Найвідомішим сховищем є Docker Hub [3].

- **сервер або демон Docker.** Виконується в хост-системі та керує всіма запущеними контейнерами;
- **клієнт Docker.** Інтерфейс користувача, або командний інтерфейс рядка для взаємодій з демоном Docker;
- **файл Dockerfile.** Це простий текстовий файл, що містить команди, за допомогою яких виконується складання образів Docker. За допомогою цих команд можна також встановлювати додаткові програмні компоненти, налаштовувати змінні оточення, робочі каталоги та точку входу ENTRYPOINT, а також додавати нові компоненти коду;
- **Docker Machine.** Docker Machine дозволяє розгортати вузли Docker на локальній машині або всередині загальнодоступної або приватної хмари, включаючи хмарні системи таких постачальників, як Amazon і Microsoft Azure. Також він забезпечує керування вузлами за допомогою команд start, stop, inspect та ін;
- **Docker Swarm -** це готовий до використання механізм кластеризації, що дозволяє об'єднати кілька вузлів Docker в один великий хост Docker. Це окремий інструмент, який можна встановити за допомогою Docker Machine або вручну, отримавши образ Swarm.Найбільшою перевагою– це те, що існує можливість просто наказати Swarm запустити наші контейнери, а той сам вирішить, на яких вузлах їх запустити, позбавивши адміністраторів застосунку від складності вибору;
- **компонувальник Docker Compose**. Програми часто складаються з безлічі компонентів і відповідно вони будуть виконуватися в кількох контейнерах. До складу Docker входить інструмент Compose, за допомогою якого можна легко запустити програму в декількох контейнерах. Можна визначити оточення для програми в загальному файлі Dockerfile і перелік служб у файлі docker-compose.yml, після чого Docker автоматично буде створювати і запускати необхідні контейнери, як визначено в цих файлах. Інструмент Compose так само, як Docker Machine, має свій набір команд для керування службами програми.

Для ізоляції контейнерів та забезпечення безпеки в операційних системах GNU/Linux використовуються стандартні технології ядра Linux [4], такі як:

- простір імен (Linux Namespaces);
- контрольні групи (cgroups);
- засоби управління привілеями (Linux Capabilities);
- додаткові, мандатні системи забезпечення безпеки, такі як AppArmor або SELinux.

Приклади просторів імен, які використовуються Docker:

- PID, Process ID – ізоляція ієрархії процесів;
- User – ізоляція ідентифікаторів користувачів (UID) та груп (GID);
- NET, Networking – ізоляція мережевих інтерфейсів;
- IPC, InterProcess Communication – управління взаємодією між процесами;
- MNT, Mount – керування точками монтування;
- UTS, Unix Timesharing System – ізоляція ядра, ідентифікаторів версії, імені хоста та доменного імені NIS.

Існуючий механізм під назвою Capabilities дозволяє розбити привілеї користувача root на невеликі групи та призначати їх окремо. Цей функціонал у GNU/Linux з'явився з версії ядра 2.2. Спочатку контейнери запускаються вже з обмеженим набором привілеїв. За допомогою різних команди docker існує можливість дозволяти та забороняти такі дії, як: операції монтування; доступ до сокетів; виконання частини операцій з файловою системою, наприклад зміна атрибутів файлів чи власника.

1.5 Необхідність використання оркестрації, передумови впровадження Kubernetes

Контейнери є повноправними застосунками; кожен із них містить необхідний код програми, бібліотеки, залежності та системні інструменти для роботи на різноманітних платформах та інфраструктурі. Контейнери в тій чи іншій формі

існують з кінця 1970-х років, але інструменти, які використовуються для їх створення, керування та захисту, різко змінилися.

Оскільки контейнери за своєю природою легкі та ефемерні, запуск їх у виробництво може швидко перетворитися на велику кількість завдань. Зокрема, у поєднанні з мікросервісами — кожен із яких зазвичай працює у своїх власних контейнерах — контейнерна програма може перевести в експлуатацію сотні чи тисячі контейнерів, особливо при створенні й експлуатації будь-якої великомасштабної системи. Масштабування контейнерів на підприємстві може бути дуже складним без автоматизованих методів балансування навантаження, розподілу ресурсів та забезпечення безпеки. Також велика кількість контейнерів обумовлює складнощі у їхньому керуванні, налаштуванні та зберіганні вцілому. Це може спричинити значну складність, якщо керувати вручну. Тому виникло таке поняття як оркестація.

Оркестрація контейнерів — це те, що робить цю операційну складність керованою для розробки та операцій — або для DevOps команди — оскільки вона забезпечує декларативний спосіб автоматизації більшої частини роботи. Це робить його ідеальним для команд DevOps і культури компаній, які зазвичай прагнуть працювати з набагато більшою швидкістю та спритністю, ніж традиційні команди програмного забезпечення. Автоматизація, єдине повторюване завдання, яке можна виконувати регулярно без участі людини, може зробити багато з цих завдань DevOps спеціалістів більш ефективними.

Оркестрація контейнерів стала гарячою темою за останні кілька років, і багато підприємств публічно оголосили про свій перехід на хмарні технології та мікросервіси. Google, Facebook, Netflix, Capital One та IBM – це лише кілька прикладів компаній, які отримують вигоду від використання платформи оркестрування контейнерів. Згідно з дослідженням Forrester Consulting 2020 Container Adoption & Usage in the Enterprise [5] , 65% технологічних лідерів вже в найближчий рік звернуться до сторонніх платформ для управління контейнерами, а не покладатимуться на внутрішній досвід попередніх років.

Оркестрація контейнерів — це автоматизація більшої частини операційних зусиль, необхідних для виконання контейнерних робочих навантажень і служб. Це включає широкий спектр речей, необхідних командам програмного забезпечення для керування життєвим циклом контейнера, включаючи надання, розгортання, масштабування (збільшення та зменшення), мережу, балансування навантаження тощо.

Нижче будуть наведені переваги оркестрації контейнерів:

- Спрощені операції: це найважливіша перевага оркестрації контейнерів і основна причина її прийняття. Контейнери представляють велику складність, яка може швидко вийти з-під контролю належного керування ними.
- Стійкість: інструменти оркестрування контейнерів можуть автоматично перезапускати або масштабувати контейнер або кластер, підвищуючи їх стійкість.
- Додаткова безпека: автоматизований підхід до оркестровки контейнерів допомагає захистити контейнерні програми, дещо зменшуючи або усуваючи ймовірність людської помилки.

Як працює оркестрація контейнерів?

Інструменти оркестрування контейнерів за своєю природою є декларативними. Просто потрібно вказати бажаний результат, і платформа забезпечить виконання цього стану. Чому це важливо для контейнерів? Для початку порівняємо декларативний та імперативний підхід в програмуванні

Імперативне програмування, у його найпростішій формі, можна описати як «як» має змінитися стан об’єкта та точний порядок, у якому ці зміни мають виконуватися. Декларативне програмування розроблено трохи інакше. У цьому випадку нас хвилює лише результат або те, «що» ми хочемо досягти. Іншими словами, який бажаний стан потрібно досягти? За допомогою декларативного програмування ви визначаєте вихід програми, не турбуючись про кроки, необхідні для цього. Простіше кажучи, існує повна абстракція від деталей того, «як» щось зробити, що надзвичайно важливо при великій кількості контейнерів.

1.6.1 Архітектура платформ оркестрування

Кожна платформа оркестрування контейнерів реалізована зовсім по різному. Проте нижче буде наведені загальні їхні спільні поняття та частини архітектури [6].

1. Job - програма, що складається з взаємозалежних і різнорідних завдань, визначених користувачем.
2. Менеджер кластера - ядро платформи оркестрації і відповідає за моніторинг ресурсів, облік, планування завдань, контроль адміністрування та рішення щодо переміщення завдань.
3. Compute Cluster – усі завдання плануються на наборі робочих вузлів, на кожному з яких працює агент робочого вузла, який передає інформацію про контейнер назад менеджеру кластерів.
4. Інфраструктура - мережа та ресурси, на яких розгорнута платформа оркестрування. Оскільки контейнери настільки гнучкі та портативні, вони можуть бути локальними, приватними або публічними хмарами.

Інструменти оркестрування покладаються на легкодоступні формати, такі як YAML та JSON, для декларативних визначень, за допомогою яких можна легко виконувати управління змінами. Ці файли конфігурації описують, де знайти образ контейнера, які апаратні ресурси слід зарезервувати та як встановити мережу.

Розгортаючи новий контейнер за допомогою інструмента оркестрування контейнерів, платформа керуватиме плануванням контейнерів на основі найкращого доступного хоста, який відповідає будь-яким попередньо визначеним обмеженням. Якщо ресурси на одному хості будуть обмежені, контейнери будуть автоматично перепланowані на новий хост.

Підтримка оркестрації контейнерів можлива через різноманітні платформи, такі як Kubernetes (часто оформлені як K8s або k8s), Docker Swarm, Apache Mesos та Amazon Elastic Container Service (ECS). Оркестрація контейнерів за допомогою Kubernetes є однією з найпопулярніших. За останні п’ять років на платформі оркестрування контейнерів Kubernetes було створено кілька інших повністю

керованих інструментів. Деякі з цих інструментів оркестрування Kubernetes включають Azure Kubernetes Service (AKS), Amazon Elastic Container Service for Kubernetes (EKS) і Google Kubernetes Engine (GKE).

1.6 Kubernetes: опис, загальна архітектура, актуальність

Контейнеризація застосунків - один з головних трендів сучасних IT-розробок. Однак, у контейнерів є один істотний недолік для масового споживача - складна настройка масштабування.

Рішенням стали автоматичні системи управління контейнеризацією, найбільш популярною з яких є Kubernetes. Це програмне забезпечення з відкритим вихідним кодом завоювало визнання завдяки поєднанню гнучкості, безпеки і потужності.

Kubernetes — це компактна портативна платформа з відкритим кодом для керування контейнерними робочими навантаженнями, службами та додатками. Розроблений компанією Google Kubernetes, він був зосереджений на автоматизації розгортання, масштабування та інших форм керування контейнерними додатками. За лише декілька років Kubernetes виріс і зарекомендував себе як певний «золотий стандарт» для оркестрації контейнерних застосунків. Це один із флагманських проектів Cloud Native Computing Foundation (CNCF), яким пожертвував Google.

Замість того, щоб турбуватися про те, як сервери будуть масштабуватися, щоб прийняти значний трафік і дані, команди розробки зосереджуються лише на створенні програми - движок Kubernetes зробить усе інше за них. Це навіть надає кращий контроль над тим, як відбувається використання апаратних ресурсів.

Завдяки Kubernetes відстежувати й підтримувати контейнеризовану програму легко - все, що потрібно зробити, це визначити ємність обладнання та вимоги до окремих контейнерів, і Kubernetes буде розумно використовувати ресурси, щоб відповідати умовам.

Основні завдання Kubernetes:

- Розгортання контейнерів і всі операції для запуску необхідної конфігурації. В їх число входять перезапуск зупинених контейнерів, їх переміщення для виділення ресурсів на нові контейнери та інші операції. Kubernetes спрощує розгортання та керування кількома контейнерами для програми. Це робиться шляхом створення рівня абстракції на доступних кластерах апаратних вузлів, що дозволяє командам розробників легко розгортати свої програми.
- Масштабування і запуск декількох контейнерів одночасно на великій кількості хостів.

- Балансування безлічі контейнерів в процесі запуску. Для цього Kubernetes використовує API, завдання якого полягає в логічному групуванні контейнерів. Це дає можливість визначити їх пули, задати їм розміщення, а також рівномірно розподілити навантаження.

Хоча у автоматизації таких завдань, як розгортання, масштабування і управління додатками в контейнерах, є безліч переваг, необхідно пам'ятати про злами в безпеці, які виникнуть, якщо не вжити певних заходів. У Kubernetes немає обширного вбудованого захисту.

1.6.1 Переваги використання Kubernetes

Kubernetes, як засіб оркестрації контейнерів, є однією з найкращих речей, що траплялися з DevOps на сьогоднішній день. Впровадження Kubernetes дозволило командам розробників програмного забезпечення перенести свою увагу з рутинного обслуговування та оновлення на розробку додатків [7]. Надалі будуть розглянуть деякі основні переваги Kubernetes, які були виокремлені за час його аналізу.

Підвищення продуктивності завдяки величезній екосистемі

Однією з найбільших переваг, яку пропонує Kubernetes, є можливість бути більш продуктивними під час створення додатків. Kubernetes допомагає швидко

створювати додатки на платформі, які включають рівень апаратної абстракції. Цей рівень допомагає командам розробників швидше впроваджувати зміни та керувати всім набором обладнання (вузлами) як єдиним об'єктом, керованим за допомогою механізму Kubernetes.

Спрощений DevOps

В Kubernetes існує певна концепція GitOps, в якій репозиторій git є основним джерелом «правди» для розгортання програми. Це означає, що якщо фактичне розгортання та історія git розходяться, Kubernetes автоматично оновлює розгортання відповідно до статусу git. Тому навантаження на розробників дедалі зменшується, які вручну зазвичай узгоджували розгортання з вихідним кодом. Тепер процес полягає в тому, щоб лише оновити історію git із потрібними змінами, і Kubernetes подбає про відповідне оновлення вашої програми.

Розподіл і звільнення ресурсів є все більше простим - не потрібно налаштовувати іншу машину вручну. Все, що потрібно зробити, це підготувати інший вузол за допомогою інтерфейсу Kubernetes. Історично масштабування програми було одним із найскладніших завдань, тому команди DevOps працювали щодня витрачаючи чимало годин, щоб успішно узгодити потужність системи з трафіком користувачів і вимогами. З Kubernetes цей процес на даний час повністю автоматизований.

Широка спільнота та екосистема

Навколо та для Kubernetes розрослася велика екосистема інструментів. Деякі з найпопулярніших включають Helm, який керує діаграмами Kubernetes (попередньо налаштованими ресурсами), і Kubectl, інструмент командного рядка, який керує менеджером кластерів Kubernetes. Природа технології Kubernetes з відкритим кодом дозволяє розробникам створювати спеціальні інструменти, що додатково допомагає автоматизувати монотонні завдання та спрощує процес DevOps. На даний час спільнота Kubernetes є сильнішою, ніж будь-коли, бо все більше ентузіастів вивчають та застосовують його у своїй роботі. Дана спільнота моментами може допомагати на початковому шляху інженера у вирішенні деяких проблем, що пов'язані з Kubernetes.

Багато постачальників хмарних послуг об’єдналися з платформами ed-tech, щоб надати структуровані практичні курси з хмарних технологій, подібних до Kubernetes. Qwiklabs, AWS Training тощо — це лише кілька імен у океані ініціатив, спрямованих на надання освітніх послуг інженерам щодо цієї технології.

Наявність великої кількості функцій, що підвищують продуктивність, допомагає підприємствам швидше створювати та масштабувати програми. Це призводить до кращої якості для користувачів, оскільки команди програмного забезпечення тепер можуть краще зосередитися фактично на створенні програмного забезпечення, що у подальшому перетворюється на більший дохід для бізнесу.

Майбутнє

Наразі Kubernetes є дуже обговорюваною технологією в світі IT. Технології будуть корисними лише в тому випадку, якщо через 10 або 20 років вони все ще матимуть порівняно велику частку використання. З Kubernetes це твердження схоже на дійсність. Переваги, які пропонує Kubernetes перед традиційним процесом DevOps, значно полегшують працю команд програмного забезпечення та підприємств по всьому світу. Ось деякі з причин, чому успіх Kubernetes матиме довгострокову перспективу:

1. Швидке зростання

Однією з головних вимог до того, щоб технологія мала й надалі успіх в майбутньому, є швидке зростання відповідно до мінливих вимог програмних продуктів. Екосистема Kubernetes постійно зростає, і корпорації та спільноти постійно випускають нові інструменти та API. Наявність спільнот з відкритим кодом, які працюють з даною технологією, полегшує розуміння та покращує досвід бізнесу з Kubernetes.

2. Адаптивна природа

Основна задача Kubernetes полягає в можливості адаптуватися до вимог використання та обслуговування розподіленої програми. Це означає, що якщо база користувачів збільшиться до більшої кількості в майбутньому, то старі програми, розміщені за допомогою Kubernetes, зможуть швидко наздогнати нові. Контейнерна

структура програми також сприятиме легкому переходу міжількома постачальниками хмар, наприклад, з Amazon Web Services (AWS) до Microsoft Azure, Google Cloud Platform (GCP) до Amazon Web Services (AWS).

3. Універсальна підтримка

Все більше і більше застосунків переносить свою діяльність із центрів обробки даних на власних серверах до основних постачальників хмарних платформ, таких як AWS, GCP, Azure тощо. У цю еру переходу до хмар нові технології, що у зв'язку з цим зароджуються, повинні співпрацювати з провідними хмарними платформами, щоб полегшити працю розробника та роботу бізнесу сучасних організацій. Тому не дивно, що Kubernetes повністю сумісний з усіма основними хмарними платформами і навіть має готові до використання стартери на багатьох з них для легкого налаштування.

Різноманітність робочих навантажень і варіантів розгортання

Існує велике різноманіття розгортання з Kubernetes. Можна запускати практично будь-які програми на движку Kubernetes, такі як:

1. Повторювані програми

Запуск кількох екземплярів програми в Kubernetes не потребує додаткових особливих зусиль. Основною причиною його задуму було достатньо гарне масштабування додатку таким чином, щоб охопити різний трафік. Кількість запущених екземплярів узгоджується з вхідним трафіком програми, щоб максимізувати економічну ефективність шляхом запуску кількох екземплярів програми.

2. Додатки, що зберігають стан

Деякі застосунки потребують стабільного та постійного зберігання для нормального функціонування, тому Kubernetes пропонує чудову підтримку для них. Технологія підтримує різні типи сховища, включаючи локальне сховище, хмарне сховище, мережеве сховище, програмно-визначене сховище тощо. За допомогою доступних плагінів існує можливість підключити будь-яке популярне рішення для зберігання даних до контейнерних програм і надійно їх зберегти.

3. Спільні програми

Найчастіше програми створюються і запускаються паралельно один одному, а також можуть допомагати один одному і інколи потребують можливості внутрішнього спілкування. Kubernetes пропонує краще рішення, ніж варіант спілкування через зовнішню мережу. Спільні програми розміщені за допомогою Pod Kubernetes, що дозволяє вузлам розміщувати всередині них кілька типів контейнерів, дозволяючи тим самим контейнерам безпечно спілкуватися та обмінюватися ресурсами.

Робота

Такий функціонал, як елемент фонової обробки потрібен для безперебійної роботи багатьох програм, який дозволяє проводити такі операції як: очищення баз даних, надсилання періодичних електронних листів, створення резервних образів, тощо. За допомогою примітиву CronJobs API можна легко розміщувати служби завдань у кластері Kubernetes і керувати ними.

Наявність ряду ресурсів і примітивів API робить Kubernetes дуже динамічною платформою, що допомагає розмістити будь-яке навантаження на єдиному порталі.

Посилена безпека

Безпека є вирішальним фактором при виборі будь-якої технології для застосунків, адже перш за все потрібно закрити репутаційні ризики. Kubernetes пропонує власні функції безпеки для захисту від широкого спектру загроз і вразливостей. Деякі з провідних базових функцій безпеки, які пропонує Kubernetes, містять в собі:

Шифрування мережі

Kubernetes використовує TLS для шифрування мережевого трафіку всередині та за межами хост-мережі, що забезпечує захист від хакерів, які намагаються перехопити дані на шляху між серверами та користувачами, тобто захищає від атаки «людина посередині».

Контроль доступу на основі ролей

Kubernetes дозволяє адміністраторам контролювати рівень доступу, наданий кожному користувачеві в команді розробників, подібно до популярних стандартів IAM (управління ідентифікацією та доступом), які надаються в провідних розподілених обчислювальних платформах, таких як хмара. Ролі та ClusterRoles можна визначити, для того щоб чітко розподілити, які ресурси доступні для яких користувачів у просторі імен. Це є чудовим способом регулювання доступу до ресурсів і забезпечення безпечної експлуатації.

Політика безпеки Pod і мережева політика

Адміністратори можуть контролювати рівень доступу, наданий окремим контейнерам і модулям, щоб запобігти атакам root-доступу. Також даний вид захисту дає можливість запобігти одержанню контейнерів непотрібних прав на всю систему, що перешкоджає зловмисникам, зламавши лише один контейнер, отримати контроль над усією системою. Адміністратори можуть контролювати, як модулі спілкуються один з одним, завдяки модерованим мережевим політикам, що може запобігти несанкціонованому зв'язку між контейнерами, щоб ще більше відключити порушників у системі.

Сумісність з основними постачальниками хмар

Можливість працювати з будь-яким основним постачальником хмари є надзвичайно важлива. Такі лідери, як Amazon Web Services (AWS), Google Cloud Platform (GCP) та інші, пропонують швидку та просту інтеграцію для додатків на основі Kubernetes. Тому компанії можуть розмістити екземпляри цих служб у локальному центрі обробки даних і все одно використовувати їх безперебійні робочі процеси для Kubernetes.

«Без блокування» постачальників

Полегшує перехід між хмарними постачальниками можливість інтеграції з кількома з них, при цьому практично не потрібно перепроектувати певну задану програму. Оскільки зменшується залежність від одного постачальника і відкривається широкий спектр можливостей для переміщення програми, якщо це необхідно, то це дуже вигідно для бізнесу у цілому. Якщо певний постачальник

хмарних послуг не відповідає вимогам команїї, то вона зможе спробувати нових учасників та інноваторів, таких як Cloud Foundry [8] , Kublr [9] тощо. Тепер це можливо завдяки однаковій природі розгортання Kubernetes для всіх постачальників.

Допоміжне управління

Більшість хмарних постачальників пропонують готові робочі процеси для налаштування модулів Kubernetes і керування ними. Деякі з них йдуть ще далі й пропонують навіть більш індивідуальні пропозиції, як Kubernetes-as-a-Service. Amazon EKS , Azure Kubernetes Service (AKS) , Red Hat OpenShift і Google Cloud Kubernetes Engine є одними з провідних постачальників хмарних послуг, які пропонують дану послугу. Усе це надає повноцінну платформу керування Kubernetes і є головним трендом в подальшому способі використання даної технології, тож можна перенести увагу компаній з керування додатками на певний користувацький досвід [10].

1.6.2 Недоліки застосування Kubernetes

Хоча Kubernetes пропонує широкий набір функцій, які в кінцевому підсумку перетворюються на переваги використання технології, він також має деякі недоліки, про які варто зазначити. Складність і додаткове планування, пов'язане з кожним движком Kubernetes, часом може бути складним. Деякими з основних недоліків підходу Kubernetes є:

Крута крива навчання

Однією з найбільш очевидних проблем із запровадженням Kubernetes є те, що навчання використанню та впровадженню є назвичайно складне. Навіть для найдосвідченіших розробників та інженерів DevOps вибір шлюху Kubernetes може стати фатальним, якщо його не вивчати з належною дорожньою картою. Вивчення Kubernetes – це довга подорож, яка починається від розуміння основних концепцій оркестрації контейнерів і контейнер-технологій уцілому до отримання міцного

досвіду в передових концепціях розробки та операцій. Цей процес може бути трудомістким і виснажливим.

Потрібні різноманітні знання

Інженеру DevOps також важливо добре знати інші хмарні технології окрім Kubernetes. Розподілені програми, розподілене ведення журналів і хмарні обчислення — це лише невеликий приклад пов’язаних тем, які необхідно знати будь-якому інженеру DevOps, щоб ефективно працювати з Kubernetes. Особливу увагу також слід приділяти таким завданням, як забезпечення безпеки, підтримка доступності, забезпечення зберігання та моніторинг усіх видів, що може бути справжнім викликом для спеціалістів, які тільки почали робити перші кроки у світі оркестровки контейнерів. Тому зазвичай вони припускаються великих помилок, що веде до прогалин у безпеці застосунку, якими можуть скористатися у подальшому зловмисники

Складна початкова конфігурація

Варто згадати про складну початкову установку та конфігурацію, обговорюючи стрімку криву навчання, яку несе з собою даний вид тенологій. Kubernetes складається з кількох «рухомих» частин, які потрібно налаштовувати та встановлювати окремо для ініціалізації системи. У той час як провідні постачальники хмарних послуг пропонують стартові (подібні до пропозицій Kubernetes-as-a-Service) готові налаштування, процес може бути складним, якщо потрібно налаштувати самостійно кластер Kubernetes, або внести певні корективи до нього.

Дорогий талант

Через стрімку криву навчання технології, досвід та таланти в цій області можуть бути дуже дорогими, навіть обходитися більше ніж спеціалісти, що розробляють, або ж керують розробкою самого застосунку. Так як організації не можуть готувати власних експертів Kubernetes у тому масштабі, який необхідний, їм доводиться часто наймати «готових талантів» з даної галузі. Кількість запитів на вакансії досвідченого інженера Kubernetes значно більша, ніж для інженерів без досвіду, що лише мають

бажання та певні початкові навички для навчанню цьому. Тому таких спеціалістів не так багато, яким все ж таки вдалось якось отримати досвід роботи у даній сфері.

Індустрія визнає зусилля, спрямовані на створення досвідченого інженера Kubernetes, оскільки середня зарплата PayScale [11] становить 115 тисяч доларів за навички лише працювати з дааним стеком технологій. Бюджети багатьох малих і середніх організацій можуть бути недостатньо великими, щоб покрити витрати такого інженера.

Некомфортне переміщення існуючих програм на Kubernetes

Будь-який перехід у розробці програмного забезпечення є дуже складним, незалежно від того, чи змінюються функції чи змінюється стек технологій, зміни завжди виявляються трудомістким і чутливим процесом. Початкова ідея Kubernetes спрямована на спрощення цього процесу переходу за допомогою контейнерних застосунків. Але перехід неконтейнерного додатка до контейнерного – зовсім непросте завдання.

Швидкість і можливість міграції повністю залежать від того, як було написано програмне забезпечення. Якщо міграція здається складною, можливо, навіть доведеться розглянути можливість переробки всієї програми, з урахуванням розгортання Kubernetes.

Потрібне бачення з самого початку

Виходячи з попереднього пункту недоліків, потрібно завчасно планувати до початку розробки архітектуру програми на основі Kubernetes. Контейнерні програми мають кардинально відмінну структуру у порівнянні з неконтейнернерними додатками, і якщо ця відмінність не врахована при розробці, то можливо буде необхідно перебудувати всю програму з нуля.

Для керування додатками на базі Kubernetes потрібне також гарне фінансове планування, оскільки для отримання прибутку потрібна кваліфікована робоча сила та достатньо трафіку. Загалом, фаза планування програми вимагає детального та продуманого підходу, щоб забезпечити максимальну віддачу від Kubernetes для дизайну застосунку.

Дорожче, ніж його аналоги

З додатковими труднощами в навчанні та налаштуванні системи Kubernetes, бюджет програм на основі Kubernetes іноді може вийти за межі дозволеного. У більшості випадків, де технологія використовується помірно та налаштована з нуля, Kubernetes виявляється дешевше, ніж його альтернативи. Однак, якщо переноситься стара програма на Kubernetes і в компанії немає внутрішніх експертів з даного стеку технологій Kubernetes, то це несе за собою набагато більші додаткові витрати.

Складність перетворюється на вартість

Як згадувалося раніше, Kubernetes важко освоїти, так як там багато тонкощів, що потрібно і важливо досліджувати. Це означає, що компанії або потрібно інвестувати більше часу в навчання своїх інженерів, або більше грошей на вже готових підготовлених інженерів. У будь-якому випадку, в кінцевому підсумку витрати будуть більші, ніж у налаштуванні без використання Kubernetes.

Крім того, у компанії витратиться багато часу, якщо команда зіткнеться з будь якими труднощами під час впровадження конфігурації. А в бізнесі очевидно, що втрачений час безпосередньо стосується втрачених грошей.

Міграція виглядає непродуктивно

Крім витрат на людський капітал, є й інші приховані витрати, пов’язані з налаштуванням Kubernetes. Якщо відбувається перехід на Kubernetes з іншого налаштування оркестрації контейнерів, то обов’язково буде втрачатися час команди розробки на переробку існуючих програм, і вона не додаватиме жодних нових функцій до застосунку. Це означає, що час, витрачений на міграцію програми, не виявляється продуктивним для бізнесу.

Інфраструктура може коштувати більше, ніж загальний прибуток

Крім того, іноді витрати на інфраструктуру запуску Kubernetes дещо вищі, ніж для альтернативних конфігурацій, особливо що стосується невеликих застосунків. Kubernetes має власні потреби в обчисленні; тому інколи може бути дешевше налаштувати веб-сайт на простішій інфраструктурі, як-от Heroku або Firebase, ніж проходити виснажливий процес налаштування конфігурації Kubernetes.

Невиправдано складний для малих додатків

Хоча ідея створення додатків, які можна масштабувати вгору і вниз, щоб відповідати будь-яким вимогам до трафіку, звучить дуже перспективо, проте це може бути не так просто реалізувати для невеликих програм. Багато застосунків при створенні орієнтовані на невелику групу користувачів або менший бюджет. Запровадження Kubernetes для них може виявитися зайвим та непотрібним, оскільки переваги запуску програми на Kubernetes можуть не компенсувати кількість часу та ресурсів, витрачених на його налаштування.

Збільшення часу розробки

Окрім витрат на початкове налаштування, обслуговування движків Kubernetes може стати громіздким завданням, тому для більш простих застосунків важливо мати ще простіший процес розгортання. Наявність повноцінної системи Kubernetes може додати непотрібних затримок у розгортанні оновлень і змін. Знову ж таки, складність може стати проблемою, оскільки для підтримки цих систем потрібні висококваліфіковані фахівці, які можуть не цікавитися і не співпрацювати з командами простих застосунків.

Важливо розуміти, коли і коли не використовувати Kubernetes. Правильний вибір у даному випадку може позбавити команди розробки від багатьох непотрібних неприємностей, тому варто враховувати усі ризики пов’язані з впровадженням даного стеку технологій на проекті.

1.6.3 Архітектура та принцип роботи Kubernetes

Розглянемо архітектуру та компоненти Kubernetes [12], представлені на рис. 1.4. Це спрощена діаграма без урахування реалізації високої доступності компонентів. Кластер складається з як мінімум одного керуючого вузла і від одного і більше обчислювальних (або робочих) вузлів, на яких безпосередньо запускаються контейнери. Керуючий вузол може бути суміщений безпосередньо з обчислювальним, і в найпростішому випадку всі компоненти можуть працювати в

одній віртуальній машині або на одному єдиному сервері. Як приклад можна навести інструмент Minikube, який дозволяє створити подібну віртуальну машину

у VirtualBox. Можливі варіанти також з кількома керуючими вузлами для забезпечення високої доступності та з єдиною базою etcd, а також з кількома керуючими вузлами та розподіленим кластером etcd.

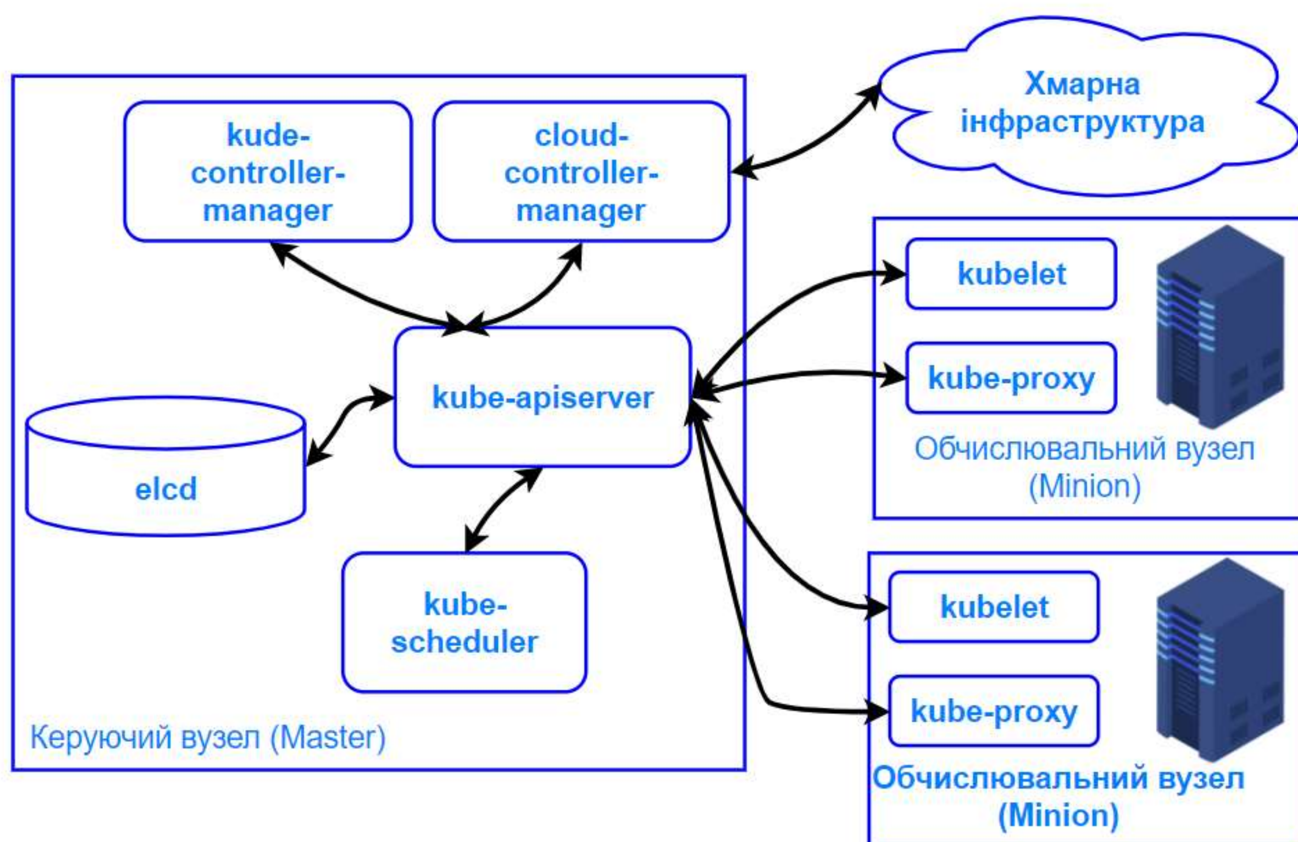


Рисунок 1.4 - Архітектура Kubernetes

Розглянемо детальніше визначення деяких функцій та важливих компонент керуючого вузла:

- kube-apiserver – це центральний компонент кластера Kubernetes. Він є зв'язуючим компонентом для всіх інших сервісів. Вся взаємодія як самих компонентів, так і зовнішні звернення до кластера проходить через kube-apiserver і валідується ним. Це єдиний компонент кластера, який спілкується з базою даних etcd, де зберігається стан кластера.

- etcd – розподілене сховище типу «включення», який не є «базою даних» у класичному розумінні СУБД. Спочатку це розвивалося як складова частина проекту CoreOS. Кластер Kubernetes зберігає в etcd всю інформацію про стан кластера, сервіси, мережі і т. д. Доступ до даних здійснюється через REST API, а при здійсненні змін в записах, замість пошуку та зміни старої копії, вона позначається просто як застаріла, при цьому нові значення дописуються в кінець. Пізніше старі значення видаляються спеціальним процесом.
- kube-scheduler – компонент, що відповідає за вибір обчислювального вузла, на якому будуть запускатися контейнери. Існують механізми, які дозволяють втрутитися в цей алгоритм і, наприклад, прив'язати контейнери до конкретного вузла. Також на запуск та розміщення контейнерів впливають існуючі квоти на ресурси.
- kube-controller-manager – компонент, що відповідає за запуск так званих контролерів, які визначають поточний стан системи, наприклад, Deployment. Фактично kube-controller-manager стежить, щоб кластер та його ресурси відповідали заданим станам.
- cloud-controller-manager – починаючи з версії Kubernetes 1.6 цей компонент відповідає за взаємодію з хмарними провайдерами і абстрагує специфічний для конкретного хмари код. Раніше за цей функціонал відповідав kube-controller-manager.

Далі детальніше про вже сервіси обчислювальних (керованих) вузлів, такі як:

- kubelet – даний сервіс управляє pod модулями, ґрунтуючись на їхній специфікації. Сервіс взаємодіє з kube-apiserver та є основним для обчислювальних вузлів.
- kubeproxy – відповідає за мережеву взаємодію pod модулів, керуючи правилами iptables (таблиця nat).

Загальними та основними термінами, що стосуються архітектури Kubernetes – кластер, вузол та под.

Кластер — це набір комп'ютерів, сховищ даних та мережевих ресурсів, за допомогою яких Kubernetes виконує різні завдання у системі. Варто відзначити, що система може складатися з кількох кластерів.

Вузол – це окремий комп'ютер (фізичний чи віртуальний). Його завдання полягає в запуску подів, про які буде інформація наведена нижче. Кожен вузол Kubernetes містить кілька компонентів, таких як kubelet і проксі kube. Усі вони перебувають під керуванням провідного вузла. Вузли - це щось на зразок робочих бджіл, які виконують всі основні завдання. Раніше вони називалися міньйонами (minions) у старій документації, де міньйони - це ті ж вузли.

Провідний вузол – це панель управління Kubernetes. Він складається з кількох компонентів, таких як API-сервер, планувальник і диспетчер контролерів, і відповідає за глобальне (рівня кластера) планування роботи подів та обробку подій. Зазвичай, всі провідні компоненти розміщуються на єдиному вузлі. Але якщо сценарій передбачає високу доступність або кластери є дуже великі, потрібно подумати про те, щоб зробити провідний вузол надлишковим.

Под (pod) – це одиниця роботи в Kubernetes. Кожен под містить один або кілька контейнерів. Поди завжди працюють спільно, тобто на одному комп'ютері. Всі контейнери всередині пода мають ті самі IP-адреси і простір портів, і вони можуть спілкуватися між собою через локальний сервер або через певні міжпроцесорні взаємодії. Крім того, всі контейнери мають доступ до загального локального сховища даних вузла, на якому знаходиться под і таке зберігання може бути підключене до кожного контейнера. В одному і тому ж Docker-контейнері можна запускати відразу кілька додатків, використовуючи сервіс supervisord для управління різними процесами, однак такий підхід зазвичай програє методу із застосуванням подів. Причинами цього може слугувати:

1. Прозорість. Коли контейнери всередині пода видно ззовні, це дозволяє надавати їм різні інфраструктурні послуги, такі як управління процесами та моніторинг ресурсів. Таким чином користувачі отримують безліч зручних можливостей.

2. Поділ програмних залежностей. Можливо керувати версіями окремих контейнерів, наново їх перебудовувати та розгортати, але в Kubernetes поки нема підтримки гарячих оновлень окремих контейнерів.
3. Простота використання. Користувачам не потрібно запускати власні диспетчери процесів, турбуватися про передачу сигналів та кодів завершення тощо.
4. Ефективність. Оскільки інфраструктура бере на себе більшу відповідальність, контейнери можуть бути дещо спрощеними.

Поди забезпечують відмінне рішення для управління групами тісно пов'язаних між собою контейнерів, які для виконання свого завдання повинні взаємодіяти на тому самому вузлі. Важливо підкреслити, що поди вважаються фіктивними сутностями, які за бажання можна видалити чи замінити, але разом з подом знищується будь-яке сховище даних, яке у ньому знаходилося. Кожен екземпляр пода отримує унікальний ідентифікатор (unique ID, або UID), щоб при необхідності їх можна було розрізнити.

ReplicaSet (Набор реплік) - об'єкт, відповідальний за опис і контроль за декылькома примірниками (репліками) подів, створених в кластері. Створення більше однієї репліки дозволяє підвищити стійкість від відмов і масштабування додатків. На практиці ReplicaSet створюється з використанням Deployment.

ReplicaSet є більш сучасною версією попереднього способу створення реплік (реплікацій) у K8s – контролер реплікації.

Розгортання - об'єкт, в якому зберігається опис подів, кількість реплік і алгоритм їх заміни в разі зміни параметрів. Контроллер розгортання дозволяє виконувати декларативні оновлення (за допомогою опису потрібного стану) на таких об’єктах, як вузли та набори реплік.

StatefulSet. Поди з'являються і зникають, і якщо важливі дані, що містяться в них, то можна задіяти постійне сховище. Це непоганий варіант, але іноді виникає необхідність автоматичного розподіленого сховища даних, наприклад MySQL Galera. Подібні кластеризовані системи розподіляють дані з різних вузлів з унікальними ідентифікаторами. Досягти цього можна за допомогою звичайних подів та сервісів

або якщо скористатися StatefulSet. StatefulSet може допомогти з виявленням вузлів, а також їх додаванням та видаленням.

DaemonSet (Набор даемона) - об’єкт, який відповідає за те, щоб на кожному окремому вузлі (або ряді вибраних) запустився один примірник вибраного пода.

Job/CronJob (Завдання по розкладу) - об'єкти для регулювання одного або запуску вибраних подів і контролю завершення їх роботи регулярним чином. Контролер Job відповідає за однократний запуск, в той час як CronJob — за запуск кількох завдань за розкладом.

Мітка - це пара "ключ - значення", за допомогою якої групуються набори об'єктів (часто подів). Цей механізм відіграє важливу роль для кількох інших концепцій, таких як контролери реплікації, набори реплік та сервіси, які працюють уже з динамічними групами об'єктів і повинні якось ідентифікувати членів цих груп. Між об'єктами та мітками існує зв'язок виду N × N. Кожен об'єкт може мати кілька міток, а кожна мітка застосовна до різних об'єктів. Мітки пов'язані з певними обмеженнями, і це зроблено навмисно. Наприклад, кожна мітка об'єкта повинна мати унікальний ключ, який відповідає суворому синтаксису. Він має складатися з двох частин – префікса та імені. Префікс необов'язковий і відокремлюється від імені косою лінією (/). Він повинен являти собою коректний піддомен DNS і не може містити понад 253 символи. Ім'я має обов’язково існувати, а його довжина обмежена 63 символами. Імена повинні починатися і закінчуватися алфавітно-цифровими символами (a - z, A - Z, 0-9) і містити тільки літери, цифри, точки, тире та підкреслення. Значення підпорядковуються тим самим правилам, як і імена. Слід зазначити, що мітки призначені лише для ідентифікації об'єктів, а для впровадження довільних метаданих використовуються інструкції.

Том, або ж локальне сховище в поді — тимчасовий елемент і видаляється разом з подом. Іноді цього достатньо, якщо потрібно передавати інформацію між контейнерами на одному вузлі. Але іноді дані повинні зберігатися і після знищення пода або бути доступними для кількох екземплярів. На цей випадок передбачено концепцію томів. Docker теж має підтримку томів, але досить обмежений їх

функціонал (хоча її постійно розвивають). Платформа Kubernetes використовує власні томи і підтримує додаткові типи контейнерів, такі як rkt, тому принципово не може покладатися на аналогічний механізм у Docker.

Існує безліч різновидів томів. Багато хто з них безпосередньо підтримується в Kubernetes, але сучасний підхід передбачає додавання нових типів томів через інтерфейс CSI (Container Storage Interface — інтерфейс) сховища контейнерів. Тома типу emptyDir підключаються до кожного контейнера та ґрунтуються на вмісті материнської системи і за бажання їх можна розмістити в пам'яті. Це сховище видаляється при знищенні пода. Існує багато різних видів томів для певних хмарних платформ, мережевих файлових систем та навіть репозиторіїв Git. Як цікавий приклад можна навести том persistentDiskClaim, який використовує стандартне для середовища сховище (зазвичай у хмарі), інкапсулюючи деякі деталі. Можна ще більш детальніше розглядати елементи та архітектуру Kubernetes, проте для розгляду з точки зору безпеки даної інформації буде достатньо.

Висновки до розділу 1

За останні кілька років масштаби застосування контейнерів різко зросли. Відповідні концепції контейнерів існували ще задовго до Docker. Але поява у 2013 році Docker з його зручними у використанні утилітами командного рядка дало поштовх популярності контейнерів серед спільноти розробників.

У контейнерів є безліч переваг. Як говорить рекламний слоган Docker, за допомогою даного програмного забезпечення можна «створити один раз, виконувати будь-де» завдяки об'єднанню в пакет програми та всіх її залежностей, а також ізоляції її від решти в машині, в якій вона працює.

У контейнеризованого застосунку є все необхідне, його можна легко упаковати в образ контейнера, який працюватиме однаково на будь-якій техніці, що дозволяє встановити Docker, і на сервері в центрі обробки даних (ЦОД). Наслідком ізоляції є

те, що існує можливість паралельного виконання кількох різних контейнерів, які не заважатимуть один одному, що значно економить ресурси.

Мішанина залежностей до появи контейнерів могла легко перетворитися на справжній кошмар для розробників, у якому двом застосункам були необхідні різні версії тих самих пакетів. Найпростішим рішенням було - виконувати програми на окремих машинах

Після появи даного програмного забезпечення багато розробників почали деталі частіше використовувати його, бо завдяки контейнеризації можна запускати кілька додатків на одному хості (неважливо, віртуальній машині або реальному сервері), не переймаючись залежностями.

Наступним логічним кроком у революції розробки — розподіл контейнеризованих додатків за кластером серверів. Завдяки засобам оркестрації, наприклад Kubernetes, цей процес автоматизується настільки, що більше не потрібно вручну встановлювати програми на конкретних машинах, а лишень достатньо повідомити засобу координації, які контейнери необхідно запустити, і воно саме знайде потрібну машину для кожного з них.

З точки зору безпеки контейнеризоване середовище багато в чому схоже із звичайним розгортанням. Порушники намагаються викрасти дані, або змінити поведінку системи, або, скажімо, використовувати обчислювальні ресурси інших людей для майнінгу криптовалюти. При переході на контейнеризовану архітектуру відбулися деякі зміни, які буде досліджено та показано у наступних розділах, і це призводить до інших наборів ризиків безпеки включаючи людські помилки.

2 БЕЗПЕКА KUBERNETES ТА DOCKER

У даному розділі буде досліджено безпеку Docker, Kubernetes, а також побудовано моделі загроз для основних напрямків.

2.1 Безпека Docker контейнерів

В основі управління ризиками лежить процес їх систематизації, перерахування можливих загроз, розстановки їх по пріоритету та вибору підходу до зменшення їх наслідків.

Моделювання загроз (threat modeling) — процес розпізнавання та перерахування можливих загроз системі. За рахунок планомірного аналізу її компонентів та ймовірних векторів атаки модель загроз допомагає визначити найбільш уразливі до атак місця системи.

Єдиної всеосяжної моделі загроз не існує, все залежить від ризиків конкретного середовища, організації та програм, що запускаються. Але можна перерахувати та проаналізувати деякі потенційні загрози, спільні для багатьох, якщо не всіх, контейнерних розгортань.

2.1.1 Модель загроз для контейнерів та основні вектори атак на них

Класичний підхід до того, як захистити середовище, полягає в тому, щоб розглядати його з точки зору зловмисника і перераховувати вектори для атаки. Саме про це йдеться у цьому підрозділі. Аналізуючи архітектуру Docker та керуючись відомими данними щодо знайдених вразливостей, можна виділити 7 тип загроз, які можливі для нього. Наочно можна побачити це на рис.2.1. Надалі буде проаналізовані можливі персони, що можуть бути причепні до можливих атак, а також самі вектори

атак з точки зору архітектури. Ці вектори допоможуть вам визначити, що потрібно захистити. За допомогою цього визначення можна встановити контроль безпеки, щоб забезпечити базовий захист і не тільки.

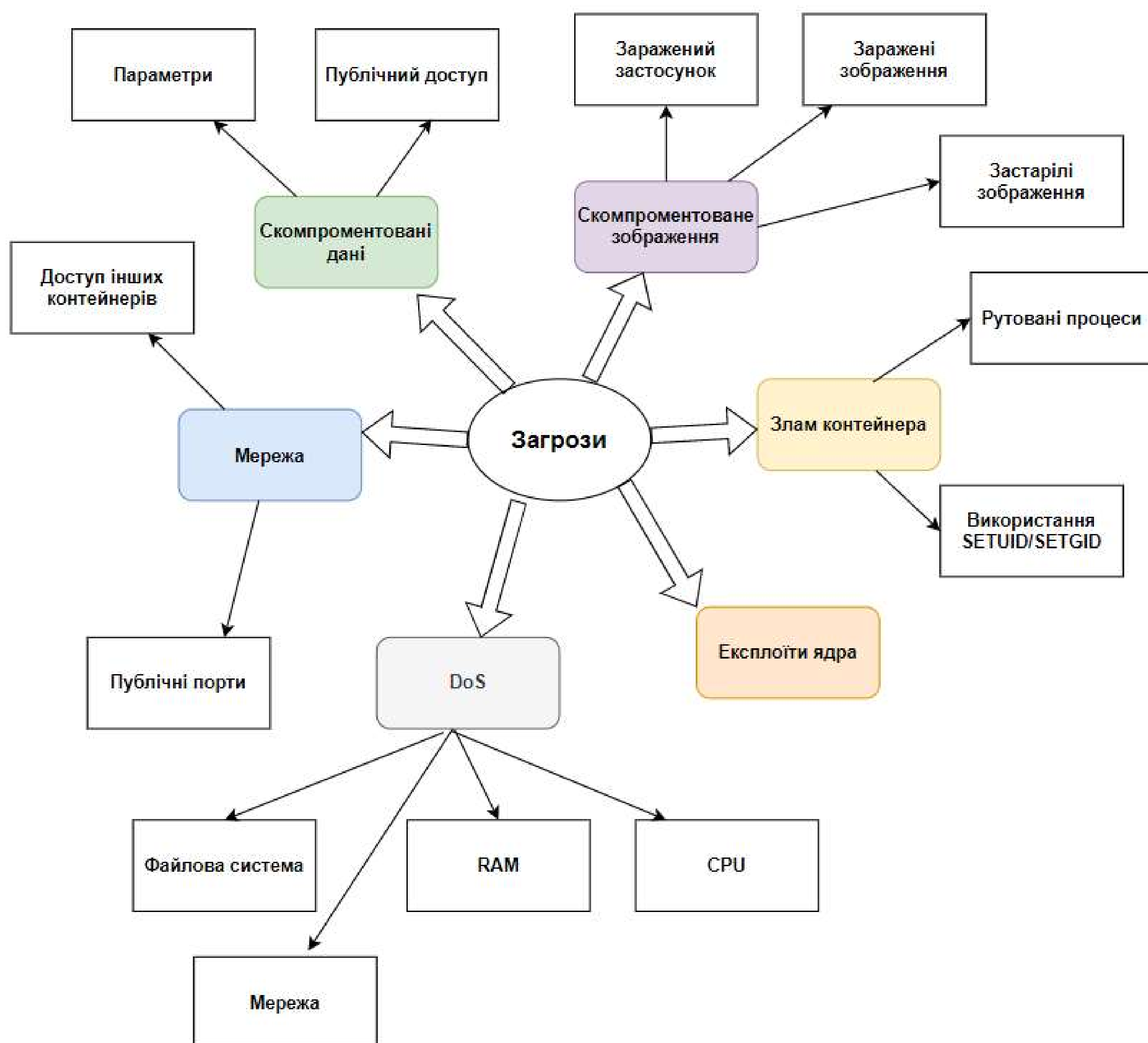


Рисунок 2.1– Основні загрози для Docker

1. Втеча з контейнеру втеча контейнера (система)

У даному випадку існує доступ до оболонки (shell), що робить вразливим застосунок. Наприклад, зловмиснику вдалося успішно провести атаку веб-застосунка, під час якого йому вдалося обійти його та опинитися безпосередньо в контейнері. На

другому етапі, він намагатиметься обійти вже контейнер, як користувач контейнера з погляду хоста, або за допомогою експлойту ядра. У першому сценарії він просто отримає привілеї користувача на хості. У другому сценарії він буде root на хост-системі, що дає йому контроль над усіма контейнерами, що працюють на цьому хості.

2. Загрози, що виходять від інших контейнерів через мережу

Даний напрям загроз має аналогічний перший етап, як і попередній напрям. Зловмисник має доступ до оболонки, проте в наступних своїх кроках він вирішує атакувати інший контейнер через мережу. Він може бути частиною того ж застосунку, або ж іншої програми того самого клієнта, або в випадку з середовищем з кількома клієнтами – від іншого клієнта.

3. Атака на елемент оркестрування через мережу.

Аналогічно цей сценарій має той самий перший вектор атаки, що й попередні два. Зловмисник має доступ до оболонки всередині контейнера, але він вирішує атакувати інтерфейси керування або інші поверхні атак інструменту оркестрування - задню панель керування. У 2018 році майже кожен інструмент мав недолік, що заключався в відкритому інтерфейсі керування за замовчуванням. «Відкритий» означає в гіршому випадку відкритий порт без аутентифікації

4. Атака на хост через мережу

Завдяки отриманому доступу до оболонки, зловмисник атакує відкритий порт хоста. Якщо ж цей порт слабко захищений або не захиений зовсім, то він отримає доступ рівня користувача, або ж root до хоста.

5. Атака на інші ресурси через мережу

По суті, це загроза, яка збирає всі мережеві загрози, що залишилися, в одне ціле. Це знову ж таки має той самий перший вектор, що й згаданий раніше. Завдяки доступу до оболонки зловмисник знаходить, наприклад, незахищену мережеву файлову систему, яка використовується спільно між контейнерами, де він може читати або навіть змінювати дані. Іншою можливістю можуть бути такі ресурси, як Active або LDAP Directory. Також ресурсом у даному випадку може бути, наприклад, Jenkins, який налаштував надто відкритий і доступний з контейнера.

Також можливо, що зловмисник встановить мережевий сніфер у контейнер, який він захопив, щоб він міг читати трафік з інших контейнерів.

6. Ресурсне голодування

Основна загроза виникає через умову безпеки іншого контейнера, що працює на тому ж хості. Створена умова може бути пов’язана з тим, що інший контейнер споживає ресурси, якими можуть бути, наприклад, цикли ЦП, оперативна пам’ять, мережа або дисковий ввод-вивод, що унеможливлює користування даних ресурсів іншими контейнерами. Можливо також, що в контейнері змонтована файлова система хоста, яку наповнив попередньо зловмисник, що спричиняє проблеми на хості, що, у свою чергу, впливає на інші контейнери.

7. Компроміс

У той час як у попередній розглянутій загрозі зловмиснику вдавалось опосередковано через хост вплинути на інші контейнери, то у даному напрямку – зловмисник скомпроментував хост через інший контейнер, або через мережу.

8. Цілісність образів

Конвеєр CD може включати кілька стрибків, де образ міні операційної системи передається від одного кроку до наступного, поки не досягне розгортання.

Кожен стрибок є потенційною поверхнею атаки для нападника. Якщо зловмиснику вдається закріпитися за один крок і немає перевірки цілісності, чи слід розгорнути те, що буде розгорнуто, існує загроза того, що від імені зловмисника розгортаються образи з його шкідливим корисним навантаженням.

Розглянувши основні вектори можливих загроз, існує необхідність висвітлити її учасників,що можуть брати участь у моделі.

Такими учасниками можуть бути :

Зовнішні зловмисники (external attackers), які намагаються ззовні отримати доступ до розгорнутої системи;

Внутрішні порушники (internal attackers), які зуміли отримати доступ до частини розгорнутої системи;

- Внутрішні учасники-зловмисники (malicious internal actors), наприклад, розробники та адміністратори з певним рівнем повноважень доступу до розгорнутої системи, що мають зловмисні наміри;
- Недбалі внутрішні дійові особи (inadvertent internal actors), які можуть ненавмисно викликати проблеми;
- Процеси додатків (application processes) - не люди-зловмисники, але вони щонайменше мають певний програмний доступом до системі.

Існує кілька можливих шляхів атаки на розгорнуту контейнеризовану систему [13], і щоб їх систематизувати, можна, наприклад, проаналізувати потенційні вектори атак на кожному з етапів життєвого циклу контейнера (рис. 2.2). Розглянемо більш детальніше дані вектори.

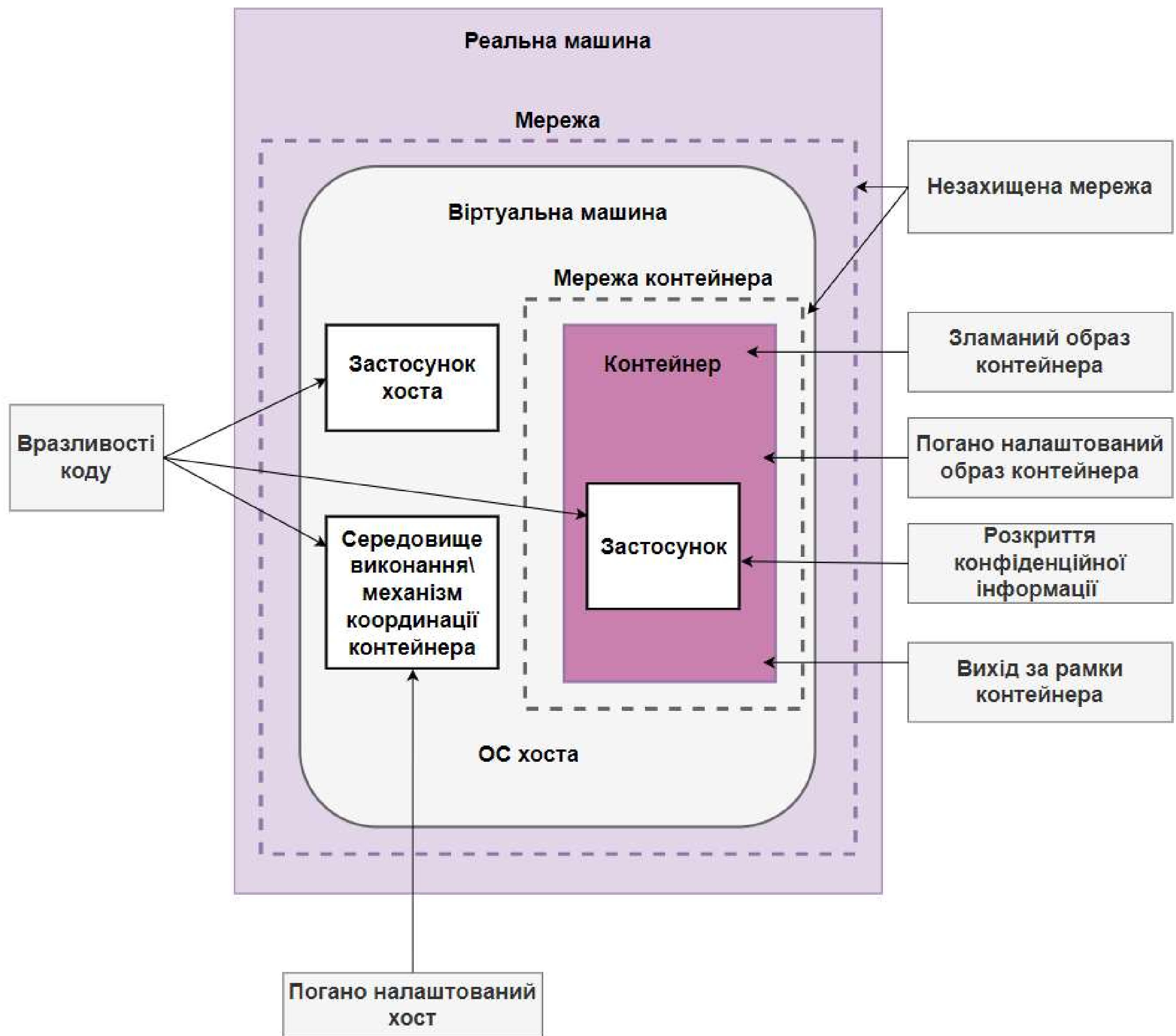


Рисунок 2.2 - Вектори атак на Docker, на кожному етапі життєвого циклу

1. Вразливий код

Життєвий цикл програми починається з написання безпосередньо коду. Він, як і його залежності, може містити певні вади (уразливості). Існують списки з тисяч відомих уразливих шляхів, якими (якщо є для цього можливості і програлини безпеки у застосунку) можуть скористатися зловмисники. Тому потрібно аналізувати кожен зібраний образ на наявність хоча б найпоширеніших вразливостей, що відомі світу. Причому робити це потрібно регулярно, оскільки вразливості виявляються у вже

існуючому коді постійно. У процесі аналізу можуть виявлятися, наприклад, контейнери із застарілим ПЗ, яке необхідно оновити, чи встановивити деякі корективи в коді з точки зору забезпечення безпеки. Крім того, існує можливість написати аналізатори, які здатні виявляти вбудоване в образи шкідливе програмне забезпечення.

2. Погане налаштування образів контейнерів.

Написаний код у процесі будує майбутній образ контейнера. У ході конфігурації складання образу контейнера виникає безліч можливостей створити вразливості, які відкривають дорогу для подальших атак на працюючий контейнер. Наприклад, виконання контейнера від імені суперкористувача, внаслідок чого у нього виявляється більше повноважень, аніж потрібно. Так як це дуже кропіткий процес, то не завжди існує можливість впоратися ідеально і забезпечити потрібний рівень безпеки лише мануальним шляхом, тому інженери намагаються автоматизувати дані процеси.

3. Атаки на систему збирання образу.

Якщо зловмисник може змінити щось в процесі збирання образу контейнера або якось вплинути на нього, то зможе здійснити імплементацію у процес шкідливий код, який потім буде запущений у середовищі промислової експлуатації. Крім того, можливість закріпитися всередині середовища збирання - плацдарм для зловмисника, що дозволяє надалі проникати у середовище промислової експлуатації.

4. Атаки на ланцюг зборки-розгортання образу

Зібраний образ контейнера зберігається у реєстрі, звідки вже витагується перед самим запуском. Тому логічно вникають питання: Як гарантувати вдповідність вмісту образу, що раніше був поміщений в реєстр? ЧИ не могли зловмисники внести якість зміни в ньому?

Будь-хто, хто може якимось чином змінити образ або модифікувати його між переходом образу в один стан з іншого, зможе виконати будь-який код у розгорнутій

системі. Тому потрібо забезпечувати механізм захисту від модифікації та доступу до образу.

5. Погане налаштування контейнерів.

Контейнери можна запусити з будь-яким налаштуванням і зачасту розробники грішать не модифікуючи їх згідно стандартів безпеки, або ж припускаються банальних людських помилок, пропустивши котресь з них. Тому іноді у контейнерів з'явля.ться непотрібні, а часом і незаплановані повноваження. Завантаживши файл конфігурації YAML з певних відкритих джерел в Інтернеті та імплементувавши його до власних налаштувань, можна також наразити на небезпеку власні контейнери.

6. Вразливі хости

Контейнери виконуються зазвичай на хост-комп'ютерах, тому необхідно перевіряти працюючий на них код на наявність певних вразливих мостів безпеки. Наприклад, відстежувати старі версії компонентів з відомими вразливостями. Існує певний сенс у тому, аби скоротити обсяг запущеного на кожному хості ПЗ до мінімуму, аби скоротити поверхню атаки. Крім того, ще й хостам потрібно задати правильну конфігурацію відповідно до екомендацій забезпечення безпеки.

7. Загальнодоступні таємні дані.

Код програми часто потребує облікових даних, токенів чи інших чутливих даних, для того щоб взаємодіяти з іншими компонентами системи. Ці таємні значення необхідно передавати до контейнеризованого коду і при розгортанні. Тому існує потреба у захисті зв'язку передачі даних, їх надійному зберіганні та шифруванні за необхідності.

8. Незахищена мережа.

Контейнерам зазвичай потрібно взаємодіяти один з одним або з навколишнім середовищем, тому потрібно налаштувати безпечні з'єднання між ними. Зачасту даний вектор атаки не розглядається з точки зору мережевої взаємодії між

контейнерами всередені в бізнесі, адже якщо зловмисник потрапляє в внутрішню мережу, то він не буде досліджувати у даному напрямку, бо має можливість скомпроментувати інші дані і підвищити привілеї в застосунку уцілому.

9. Вразливість виходу за рамки контейнера

Широко використовувані засоби виконання контейнерів, включаючи containerd і CRI-O, вже добре перевірені роками використання, проте не виключено, що в них все ж таки залишаються програмні помилки, внаслідок яких шкідливий код, що працює всередині контейнера, може просочитися межі контейнера в хост.Однією з таких проблему виявилась у 2019 році- Runcescape [14].
Збитки від виходу за рамки контейнера для низки додатків був настільки великий, що має сенс також забезпечити використання ефективніших механізмів ізоляції.

10. Неналежне зберігання коду.

Вихідний код зазвичай зберігається в репозиторіях, потенційно доступних для атак, мета яких - злом програми. Необхідно забезпечити належний контроль доступу користувачів до репозиторію.

11. Неналежний захист доступу до хостів.

Хост-комп'ютери зв'язуються між собою мережею, зазвичай підключеною до мережі Інтернету, причому з метою безпеки при цьому часто застосовується таий механізм як VPC. Як і при звичайному розгортанні, необхідно захистити комп'ютери (або віртуальні машини) від доступу зловмисників. Безпечні налаштування мережі, використання брандмауера, а також керування ідентифікацією та доступом для нативного хмарного розгортання анітрохи не менш релевантні, ніж для звичайного.

12. Небезпечне налаштування оркестраторів

Контейнери зазвичай працюють під управлінням механізму координації— у сучасних розгортаннях його роль зазвичай грає Kubernetes, хоча є й інші варіанти, наприклад Docker Swarm та Hashicorp Nomad.

Недостатня безпека налаштувань засобу координації або відсутність належного контролю за доступом з правами адміністратора відкривають зловмисникам додаткові вектори атак, про які буде більш детальніше написано в наступному розділі присвяченому Kubernetes.

2.1.2 Контрзаходи спрямовані на захист контейнерів від типів вразливостей

1. Уразливості в програмному забезпеченні під час виконання

Необхідно ретельно відстежувати під час виконання контейнера на предмет виявлення вразливостей і проблем виявлені, їх необхідно швидко усунути. Уразливе середовище виконання наражає всі підтримувані ним контейнери, а також сам хост, потенційно значному ризику. Організації повинні використовувати інструменти з пошуку вразливостей та ризиків (CVE) у розгорнутому середовищі виконання, оновлювати будь-які екземпляри, які піддаються ризику, і гарантувати, що оркестратори дозволяють розгортання лише для належним чином підтримуваного середовища виконання.

2. Необмежений доступ до мережі з контейнерів

Організації повинні контролювати вихідний мережевий трафік, що надсилається контейнерами. Принаймні, ці засоби контролю мають бути встановлені на кордонах мережі, гарантуючи, що контейнери не можуть передавати трафік через мережі різного рівня чутливості, наприклад із середовища, що містить захищені дані, в Інтернет, подібно до шаблонів, що використовуються для традиційних архітектур. Однак віртуалізована мережева модель міжконтейнерного трафіку створює додаткову проблему.

Оскільки контейнери, розгорнуті на кількох хостах, зазвичай спілкуються через віртуальну зашифровану мережу, традиційні мережеві пристрої часто не сприймають

цей трафік. Крім того, контейнерам зазвичай автоматично призначаються динамічні IP-адреси під час розгортання оркестраторами, і ці адреси постійно змінюються в міру масштабування програми та балансування навантаження.

Таким чином, в ідеалі, організації повинні використовувати комбінацію наявних пристроїв мережевого рівня та фільтрації мережі, що функціонує на основі програм. Інструменти, що відповідають за застосунок, повинні мати можливість не просто бачити міжконтейнерний трафік, а й динамічно генерувати правила, які будуть використовуватися для фільтрації цього трафіку на основі конкретних характеристик додатків, що працюють у контейнерах. Це динамічне керування правилами має вирішальне значення через масштаб і швидкість зміни контейнерних додатків, а також їх недовговічні топології мережі.

Зокрема, інструменти, що забезпечують підтримку програм, повинні забезпечувати такі можливості:

• Автоматизоване визначення належних мережевих поверхонь контейнерів, включаючи як вхідні порти, так і прив'язки портів процесів;

• Виявлення потоків трафіку як між контейнерами, так і іншими об’єктами мережі, як за трафіком «по проводу», так і за інкапсульованим трафіком;

• Виявлення мережевих аномалій, таких як несподівані потоки трафіку в мережі організації, сканування портів або вихідний доступ до потенційно небезпечних місць призначення.

3. Небезпечні конфігурації середовища виконання контейнера

Організації повинні автоматизувати дотримання стандартів конфігурації середовища виконання контейнера. Задокументовані вказівки з технічної реалізації, такі як Центр безпеки в Інтернеті Docker Benchmark [15], містять детальну інформацію про параметри та рекомендовані налаштування, але впроваджують це в дію.

Організації можуть використовувати різноманітні інструменти для «сканування» та оцінки їх відповідності в певний момент часу, але такі підходи не мають масштабу. Натомість організації повинні використовувати інструменти або процеси, які постійно оцінюють налаштування конфігурації в середовищі та активно дотримуватися їх.

Крім того, технології обов'язкового контролю доступу (MAC), такі як SELinux [16] і AppArmor [17], забезпечують покращений контроль та ізоляцію для контейнерів під керуванням ОС Linux. Наприклад, ці технології можна використовувати для забезпечення додаткової сегментації та гарантії того, що контейнери повинні мати доступ лише до певних шляхів до файлів, процесів і мережевих сокетів, що ще більше обмежує здатність навіть скомпрометованого контейнера впливати на хост чи інші контейнери.

Технології MAC забезпечують захист на рівні ОС хосту, гарантуючи, що лише певні файли, шляхи та процеси доступні для контейнерних програм. Організаціям рекомендується використовувати технології MAC, що надаються їхніми хост ОС у всіх контейнерах, що розгортаються.

Профілі безпечних обчислень (seccomp) — це ще один механізм, який можна використовувати для обмеження можливостей системного рівня, які виділяються під час виконання. Звичайні середовища виконання контейнера, як Docker містять в собі стандартні профілі seccomp, які відкидають системні виклики, що є небезпечними та зазвичай непотрібними для роботи з контейнером. Крім того, можна створювати та передавати власні профілі середовища виконання контейнера, щоб ще більше обмежити їх можливості. Принаймні, організації повинні переконатися, що контейнери запускаються з профілями за замовчуванням, наданими їх середовищем виконання, і повинні розглянути можливість використання додаткових профілів для програм із високим ризиком.

4. Уразливості програм

Існуючі процеси та інструменти виявлення вторгнень на базі хостів часто не в змозі виявити та запобігти атаки всередині контейнерів через різну технічну архітектуру та робочі методи, які обговорювалися раніше. Організації повинні впроваджувати додаткові інструменти, які призначені для роботи з контейнерами і призначені для роботи з масштабом та швидкістю змін, які зазвичай спостерігаються для контейнерів. Ці інструменти повинні мати можливість автоматично профільувати контейнерні програми за допомогою поведінкового навчання та створювати профілі безпеки для них, щоб мінімізувати взаємодію з людьми. Ці профілі повинні мати можливість запобігати й виявляти аномалії під час виконання, включаючи такі події, як:

• Неочікуване виконання процесу,

• Недійсні або неочікувані системні виклики,

• Зміни до захищених файлів конфігурації та двійкових файлів,

• Запис в неочікувані місця та використання неочікуваних типів файлів,

• Створення неочікуваних мережевих перехоплювачів (“слухачів”),

• Трафік, надісланий до\з неочікуваних мережевих місць призначення,

• Зберігання або виконання шкідливих програм.

Контейнери також слід запускати з їх кореневими файловими системами в режимі “лише для читання” . Цей підхід ізолює записи до спеціально визначених каталогів, які потім можна буде легше контролювати за допомогою вищезгаданих інструментів. Крім того, використання файлових систем “лише для читання” робить контейнери більш стійкими до компромісу, оскільки будь-які втручання ізольовані в цих конкретних місцях і можуть бути легко відокремлені від решти програми.

Контрзаходи спрямовані безпосередньо на мінімізацію уразливостей компонентів ОС хосту

Необхідно організаціям впроваджувати методи управління та інструменти для перевірки версій компонентів, наданих для управління базовою ОС і функціональності. Незважаючи на те, що ОС конкретного контейнера мають набагато більший мінімальний набір компонентів, ніж ОС загального призначення, вони все одно мають уразливості і все ще потребують виправлення. Організації повинні використовувати інструменти, надані постачальником ОС або іншими довіреними організаціями, щоб регулярно перевіряти та застосовувати оновлення до всіх програмних компонентів, що використовуються в ОС. ОС має бути в курсі не тільки оновлень безпеки, але й останніх оновлень компонентів, рекомендованих постачальником. Це особливо важливо для компонентів середовища виконання ядра та контейнера, оскільки нові версії цих компонентів часто додають додатковий захист і можливості, крім простого виправлення вразливостей. Деякі організації можуть вирішити просто перерозгорнути нові екземпляри ОС з необхідними оновленнями, а не оновлювати існуючі системи. Цей підхід також має місце бути, хоча він часто вимагає більш складних операційних практик.

Операційні системи хоста повинні працювати незмінно, без даних або стану, що зберігаються однозначно і постійно на хості, і без залежностей на рівні програми, наданих хостом. Натомість усі компоненти програми та залежності мають бути запаковані та розгорнуті в контейнерах. Це дозволяє працювати з хостом зі значно зменшеною поверхнею атаки. Крім того, він забезпечує більш надійний спосіб виявлення аномалій і зміни конфігурації.

Неналежні права доступу користувача

Хоча більшість розгортань контейнерів покладаються на оркестратори для розподілу завдань між хостами, організації все одно повинні переконатися, що вся автентифікація в ОС проходить належний аудит, відстежуються аномалії входу та реєструється будь-яка ескалація для виконання привілейованих операцій. Це дає змогу ідентифікувати аномальні шаблони доступу, такі як індивідуальний вхід на

хост безпосередньо та виконання привілейованих команд для маніпулювання контейнерами.

Підробка файлової системи хоста

Контейнери потрібно запускати з мінімальним набором необхідних дозволів файлової системи. Дуже рідко контейнери повинні монтувати локальні файлові системи на хост. Замість цього будь-які зміни файлів, мають бути внесені в томи зберігання, спеціально виділені для цієї мети. Ні в якому разі контейнери не повинні мати можливість монтувати конфіденційні каталоги в файлову систему хоста, особливо ті, що містять параметри конфігурації для операційної системи.
Організації повинні використовувати інструменти, які можуть контролювати, які каталоги монтуються контейнерами, і запобігати розгортанню контейнерів, які порушують ці політики.

Апаратні контрзаходи

Як стверджується в NIST SP 800-164 [18], безпека на основі програмного забезпечення регулярно зазнає поразки. NIST визначає вимоги до надійних обчислень у NIST SP 800-147 [19], 800-155 [20] і 800-164.

Для NIST «довірений» означає, що платформа веде себе так, як очікується: інвентаризація програмного забезпечення є точною, налаштування конфігурації та засоби контролю безпеки встановлені та працюють так, як вони повинні тощо. «Довірений» також означає, що відомо, що жодна стороння особа не втрутилася в програмне забезпечення або його конфігурацію на хостах. Корінь довіри до апаратного забезпечення не є унікальною концепцією контейнерів, але інструменти керування контейнерами та засоби безпеки можуть використовувати атестації для решти архітектури технології контейнерів, щоб гарантувати, що контейнери працюють у безпечному середовищі.

Наразі існує доступний спосіб забезпечити надійні обчислення:

1. Виміряти мікропрограмне забезпечення, програмне забезпечення та конфігураційні дані перед їх виконанням за допомогою кореня довіри для вимірювання (RTM).
2. Зберігати ці вимірювання в апаратному корені довіри, як-от модуль надійної платформи (TPM).
3. Перевірити відповідність поточних вимірювань та порівняти з очікуваним. Якщо все добре, то можна вважати, що платформі можна довіряти, що вона веде себе належним чином. Пристрої з підтримкою TPM можуть перевіряти цілісність машини під час процесу завантаження, дозволяючи механізмам захисту та виявлення функціонувати в апаратному забезпеченні, під час попереднього завантаження та в процесі безпечного завантаження. Ця ж гарантія довіри та цілісності може бути поширена за межі ОС і завантажувача на середовища виконання контейнера та програми. У випадках, коли технічна перевірка не надається, організації повинні розглядати вимоги довіри до обладнання в рамках своїх угод про обслуговування у хмарних провайдерах.

Зростаюча складність систем і глибоко вкорінений характер сучасних загроз означає, що безпека має поширюватися на всі компоненти контейнерної технології, починаючи з апаратного забезпечення та мікропрограм. Це формує розподілену модель надійних обчислень і забезпечує найбільш надійний і безпечний спосіб створення, запуску, оркестрування та керування контейнерами.

Довірена обчислювальна модель повинна починатися з вимірюваного/безпечного завантаження, яке забезпечує перевірену системну платформу, а також побудувати ланцюжок довіри, заснований на апаратному забезпеченні та розширений на завантажувачі, ядро ОС і компоненти ОС, щоб уможливити криптографічну перевірку механізмів завантаження, образи системи, середовища виконання контейнерів і образи контейнерів. Для контейнерних технологій ці методи наразі застосовні на рівнях апаратного забезпечення, гіпервізора та ОС хоста, щоб застосувати їх до конкретних компонентів контейнера.

На момент написання цієї статті NIST співпрацює з галузевими партнерами для створення еталонних архітектур на основі комерційних готових продуктів, які демонструють надійну обчислювальну модель для контейнерних середовищ.

2.2 Безпека Kubernetes

У період з жовтня 2020 року по лютий 2021 року дослідники Unit 42 [21] періодично сканували та аналізували незахищені кластери Kubernetes (також відомі як k8s) в Інтернеті. Кластери Kubernetes можна і потрібно налаштувати задля підвищення безпеки, але якщо вони не захищені, до цих кластерів може отримати анонімний доступ будь-хто, хто знає їхні IP-адреси, порти та API. Дослідники виявили 2100 незахищених кластерів Kubernetes, які складаються з 5300 вузлів, 31 340 ЦП і 75 270 модулів. У цих незахищених кластерах, якими керують організації в таких секторах, як електронна комерція, фінанси та охорона здоров'я, було помічено широкий спектр застосунків. Великі обчислювальні ресурси та велика кількість конфіденційних даних у програмах, таких як маркери API, облікові дані бази даних, вихідний код та персональна інформація (PII) – роблять ці незахищені кластери привабливими для супротивників. Найбільший кластер, з яким зіткнулися дослідники, мав понад 500 вузлів і 2000 активних капсул. На рисунку 2.3 показано порівняння розкритих ресурсів Kubernetes.

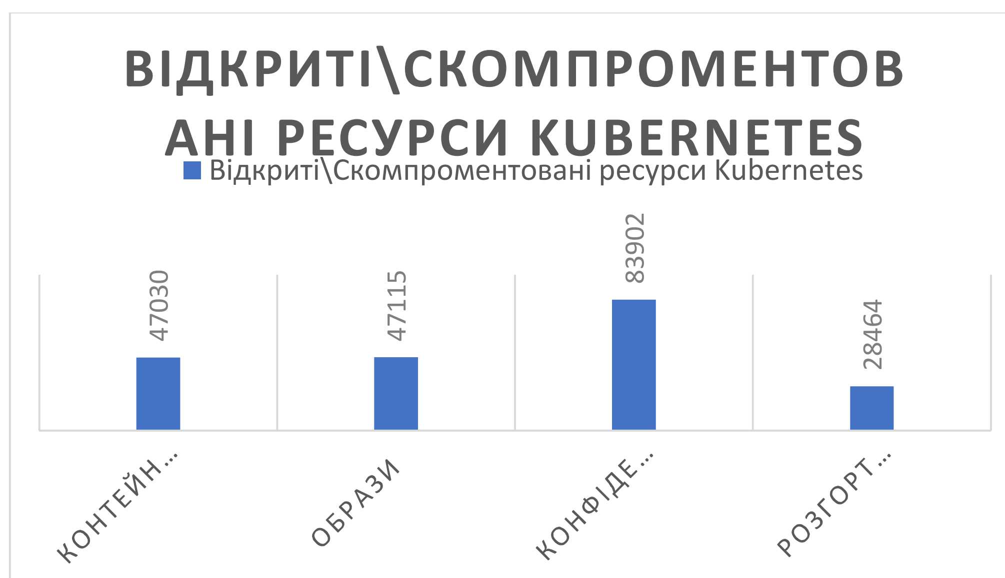


Рисунок 2.3 - Числові дані по кожному скомпроментованому ресурсу Kubernetes

На малюнку 2.4 показано API, до яких можна отримати анонімний доступ. Можна побачити, що різні сервери надають різні набори API. Наприклад, 18,3% відкритих серверів API дозволяють анонімно отримати доступ до /api/v1/nodes, але лише 13,6% відкритих серверів API дозволяють анонімно отримати доступ до /api/apps/v1/deployments. Незважаючи на це, отримання доступу до будь-якого з цих API може призвести до витоку конфіденційної інформації про програму або платформу.

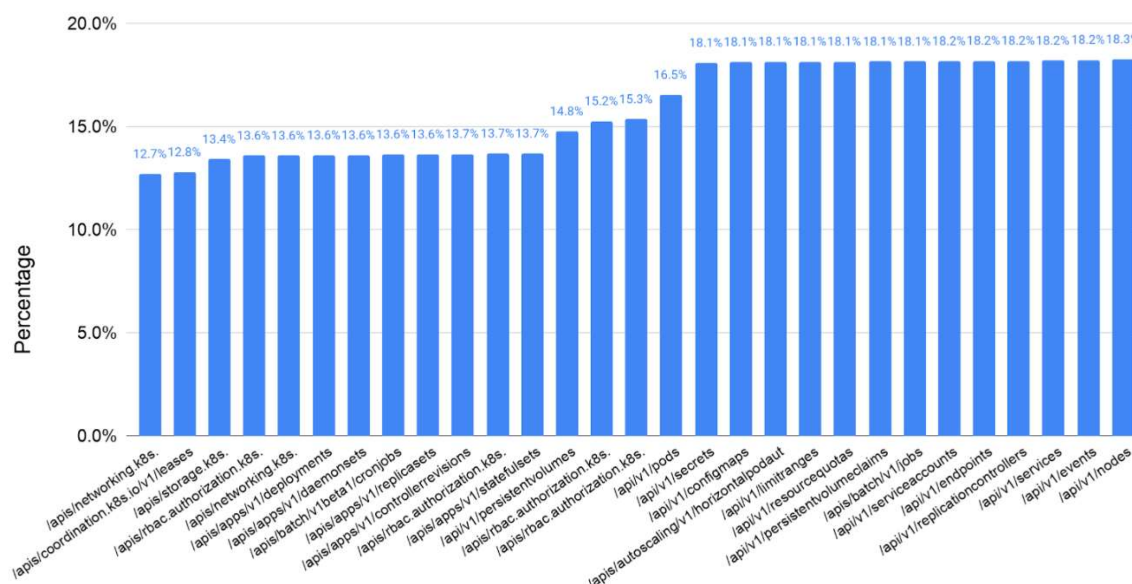


Рисунок 2.4 - API сервіси, що дозволяють анонімний (відкритий) доступ

На малюнку 2.5 показані версії визначених кластерів Kubernetes – v1.12, v1.13 і v1.16, що є найпоширенішими версіями, які забезпечують 50% розгортань. Цифра свідчить про те, що 50% кластерів не оновлювалися більше двох років. На момент написання цього матеріалу, версія 1.9 була найновішою доступною версією, яка принесла лише 2,2% розгортань. Тому до цих розгортань можна застосувати різні публічні експлоїти, що раніше уже були написані дослідниками безпеки.

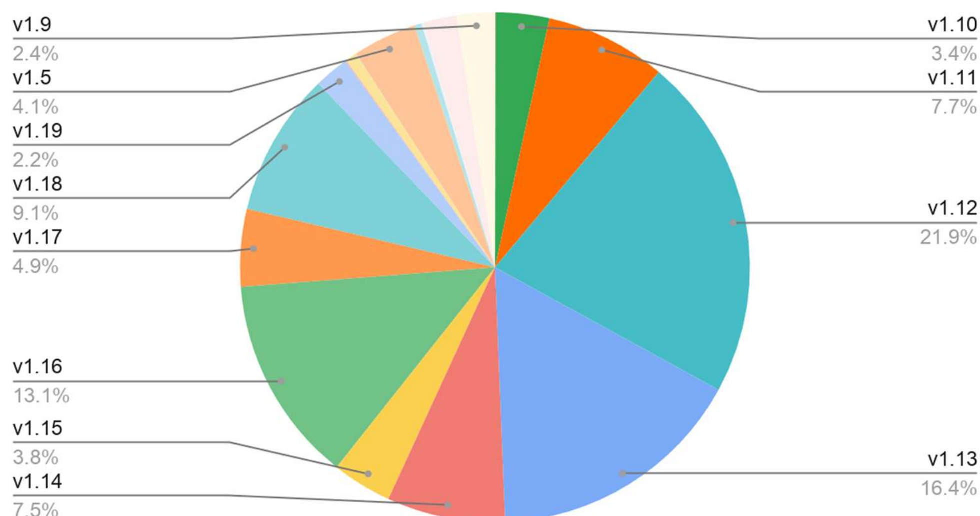


Рисунок 2.5 - Версії розгортань застосунків на момент дослідження

Завдяки швидкому впровадженню контейнерної технології та багатим обчислювальним ресурсам у цих кластерах Kubernetes, можна вважати, що незабаром більше зловмисників зверне свою увагу на цей невивчений простір. Незважаючи на це, прості передові методи, як сканування, встановлення корректних налаштувань та постійний моніторинг можуть ефективно запобігти та виявити більшість цих атак. Завдяки складному та динамічному характеру технології Kubernetes.

2.2.1 Основні вектори атак

Як і в будь-якій складній системі, в інфраструктурі кластера K8s є типові проблеми безпеки, з якими стикаються системні адміністратори. Нижче будуть описані найпоширеніші з них:

- Explosion of East-West Traffic. Суть цієї атаки полягає в тому, що контейнери можуть бути динамічно розгорнуті в декількох незалежних хмарах, що значно збільшує трафік обміну даними всередині логічного кластера. Віддалене розташування контейнерів може використовуватися зловмисниками, наприклад для реалізації DDoS-атак.
- Increased Attack Surface (збільшена площа атаки). Проблема ґрунтується в тому, що кожен контейнер може мати різну поверхню атаки та власні унікальні вразливості, які використовуються хакерами для подальшого злому. Наприклад, можна використовувати вразливість для Docker або системи авторизації AWS.
- Container compromise (компрометація контейнера). Суть атаки криється у використанні невірної конфігурації (security misconfiguration) для всіх контейнерів кластера, які опосередковано сприяють компрометації або включають вразливості додатків. До компрометацій контейнера відносяться маніпуляції внутрішньою комутацією, керуванням процесами або доступом до файлової системи. Це є найпоширенішим вектором атаки на Kubernetes.
- Unauthorized connections between pods (несанкціоновані з'єднання між подами всередині єдиного кластера). Скомпрометовані контейнери можуть з'єднуватися з іншими контейнерами на тому ж чи інших хостах, для того запустити будь-яку атаку. Незважаючи на те, що фільтрація на рівні L3 (ACL-листи) забезпечується мережевим обладнанням згідно з налаштованими правилами, деякі неавторизовані звернення можуть бути виявлені лише за допомогою фільтрації на сьомому рівні моделі OSI.

2.2.2 Моделювання загроз

Існує небагато аналізів загроз та їх загальні принципи моделювання, проте нещодавно організація NCC виділила у своєму звіті чотири основні групи загроз для кластера [22], що будуть розглянуті детальніше надалі в таблиці 2.1: зовнішні зловмисники, шкідливі контейнери, зламані користувачі, etcd.

Таблиця 2.1 – Групи загроз для кластера

Дійові особи	Опис ризиків
Зовнішні зловмисники	Персони, які не мають доступу до кластера, лише можуть отримати доступ до запущених на ньому програм та/або портів керування через мережу.
Шкідливі контейнери	Зловмисник має доступ до одного контейнера і хотів би розширити свій доступ, щоб захопити весь кластер Підвищення привілеїв: через kubelet; через доступ до etcd; через токени служби, які розгорнуті на всіх контейнерах з високими привілейованими правами. Деякі реалізації Kubernetes мають маркери для API AWS/Azure, що збільшить вплив скомпрометованого одного контейнера
Скомпроментований користувач	Зловмисник має дійсні облікові дані для виконання команд проти Kubernetes API, а також доступ до мережі до порту
etcd	etcd зберігає стан кластера, включаючи такі елементи, як маркери служб, секрети та конфігурації служб. kubeadm буде прив’язувати etcd лише до інтерфейсу локального хосту. Зловмиснику потрібно отримати доступ до головного вузла, щоб отримати доступ до інтерфейсу API. Проблема з прив’язкою localhost, яка полягає в тому, що вона не дозволяє налаштувати кластеризовані etcd і якщо потрібно мати декілька баз даних etcd (надлишковість), необхідно дозволити зв'язок між сховищами даних

Межі зон безпеки (іноді звані межами довіри) між частинами системи означають, що набори прав доступу у цих частинах відрізняються. Іноді кордони задаються під час адміністрування — наприклад, у системах під керуванням Linux системний адміністратор може модифікувати межі зон безпеки, вказуючи, які групи файлів доступні користувачеві. Адміністратор робить це, змінюючи групи, у яких приймає участь користувач. Чим суворіший кордон зон безпеки між зловмисником та його метою (наприклад, даними користувачів), тим складніше йому досягти цієї мети.

У наведеній нижче таблиці було розглянуто та проаналізовано межі безпеки для Kubernetes [23], а на рис. 2.6 – наочно проілюстровано розташування цих меж.

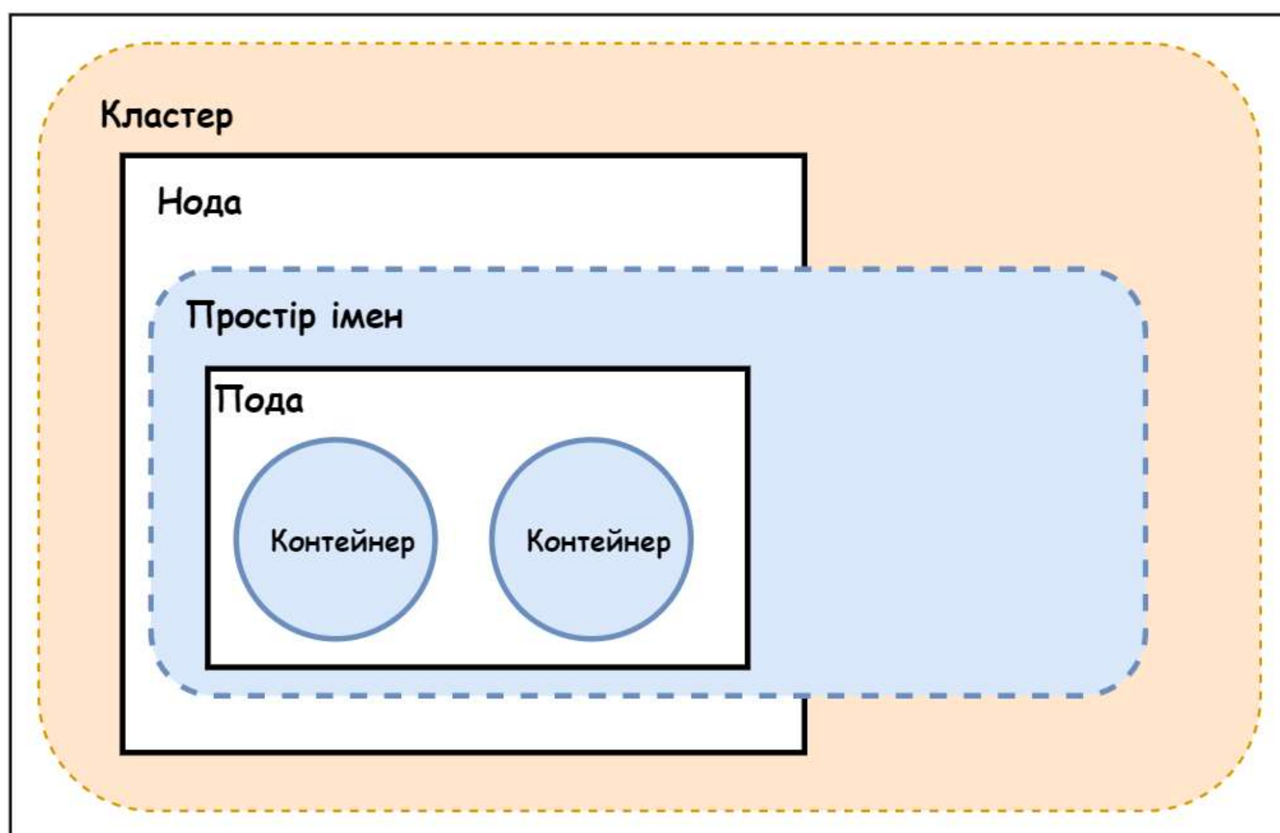


Рисунок 2.6 - Межі довіри Kubernetes

Таблиця 2.2 – Семантичний опис меж довіри

Кластер	Містить усі вузли, а також компоненти плоскості керування, забезпечуючи ізоляцію мережі та утворює блок верхнього рівня. Існує можливість для програмістів віддати перевагу

	різним кластерам для кожної команди та/або етапу (наприклад, розробка, підготовка, виробництво), щоб реалізувати багатоарендність на рівні простору імен або ізоляції на рівні вузлів.
Вузол	Віртуальна або звичайна машина в кластері, на якій розміщено декілька модулів і системних компонентів (наприклад, kubelet). Ці вузли обмежені щодо доступу до ресурсів, необхідних для виконання своїх завдань
Простір імен	Своєрідний віртуальний кластер, що містить кілька ресурсів (наприклад, служби та модулі). Простори імен є основною одиницею авторизації.
Под	Використовується для групування контейнерів із гарантією того, що всі контейнери в модулі заплановані працювати на одному вузлі. Він пропонує певний рівень ізоляції: можливо визначити контекст безпеки та застосувати його, а також вказати ізоляцію на рівні мережі.
Контейнер	Комбінація cgroups, просторів імен і файлових систем копіювання при записі, яка керує залежностями на рівні програми. Налаштувавши якість обслуговування модулів програми, існує можливість вплинути на поведінку під час виконання, але якщо не використовувати передові методи пісочниці під час виконання, контейнери зазвичай не забезпечують надійних гарантій ізоляції, крім гарантій безпеки на рівні ядра.

Тепер варто розглянути дослідження, що стосуються межі довіри, проте уже за методикою STRIDE. Наразі буде проаналізовано вектори за даною методикою у таблиці 2.3, а потім вже у подальшому побудовано та проаналізовано дерева атак у наступному розділі.

Таблиця 2.3 - Головні вектори атак

Сервісний маркер	За замовчуванням маркер служби автоматично монтується в кожен модуль. Якщо контейнер зламано, зловмиснику буде надано механізм експлуатації з використанням цих облікових даних. Сувора політика RBAC і відключення автоматичного підключення маркерів служби є ключовими вирішенням для запобігання проблем, що можуть виникнути тут.
Зламаний контейнер	Основний фокус уваги напрямлений на кластер, оскільки це забезпечує віддалену точку виконання для зловмисника. Крім згаданої вище атаки на маркери служби, інші вектори атаки включають доступ до мережі керування за замовчуванням для всіх запущених контейнерів.
Кінцеві точки мережі	Кожна кінцева точка Kubernetes має бути захищена від внутрішніх зловмисників, і якщо зловмиснику вдається зламати контейнер, він отримує доступ до кінцевих точок, якщо це дозволяє політика мережі pods. Тому тут потрібно спрямувати всі сили на захист pods і впровадження корректної політики щоло цього
Відмова в обслуговуванні	До версії 1.14 існувало досить мало контрзаходів спрямованих на стримання атак відмови в обслуговуванні. Тому необхідно користуватись новітнішими версіями Kubernetes.
Проблеми RBAC	Багато векторів атак покладаються на неправильну конфігурацію політик RBAC. Превентивна робота інженерів має полягати на автоматизовані інструменти для перевірки такої політики.

2.2.3 Методи забезпечення безпеки Kubernetes

Забезпечення безпеки Kubernetes умовно можна подати двома практичними підходами. Перший підхід – це налаштування опцій безпеки (security hardening) та використання найкращих практик (security best practices) на всіх ключових елементах K8s-інфраструктури. Другий підхід — застосування сторонніх утиліт забезпечення моніторингу, контролю та управління рівнем безпеки.

Нижче будуть наведені загальні рекомендації щодо налаштувань безпеки, що можуть бути застосовані в будь-якому додатку з Kubernetes:

1. Використання шифрування TLS. Воно має бути увімкнене для кожного інфраструктурного компонента K8s, що підтримує протокол TLS. Це гарантує захист від сніфінгу трафіку, форсування посвідчення ідентичності сервера та (у разі Mutual TLS) посвідчення ідентичності клієнта.
2. RBAC-орієнтована модель доступу та встановлення за замовченням мінімально можливі привілеї. Управління доступом на основі ролей (Role-based access control, RBAC) забезпечує необхідне управління політиками, з використанням яких користувачі отримують доступ до таких ресурсів, як простір імен.
3. Стороння автентифікація для API Server. Для допомоги видачі та відклику прав для співробітників підприємства існує централізація автентифікації та авторизації для всієї компанії, наприклад на базі LDAP та Single Sign On.
4. Інкапсуляція кластеру etcd за мережевим екраном. Кластер є критичним компонентом K8s, адже кластер etcd зберігає інформацію про стан Kubernetes та секрети доступу (токени, сертифікати). Саме тому etcd має бути захищений окремо від решти кластерів, а ще краще за Firewall та ізольованою VPC.
5. Систематична зміна ключів шифрування. Одна з найкращих практик у забезпеченні безпеки будь-якої IT-системи — регулярна зміна (за терміном та подією) ключів безпеки та сертифікатів. Вона дозволяє обмежити «радіус ураження» під час компрометації ключа доступу.

6. Регулярний статичний аналіз файлів YAML конфігурації. Конфіденційна інформація, що продекларована в форматі YAML, не повинна зберігатися у відкритому вигляді на подах. А конфіденційні конфігурації та секрети доступу (паролі) мають бути зашифровані за допомогою таких інструментів як git-crypt. Статичний аналіз конфігурації YAML може використовуватися для встановлення базових показників безпеки, причому проводити такий аналіз слід регулярно. Це допомагає забезпечити базові потреби системи в безпеці. Найчастіще саме помилки в даній конфігурації призводять до найпоширеніших атак на Kubernetes.
7. Обмеження запуску контейнерів під обліковим записом root. Найчастіше у контейнерів, які запускаються з правами суперкористувача, набагато більше прав, ніж вимагають їх робочі навантаження, що у разі компрометації допомагає атакуючим отримати ще більше можливостей.
8. Обов'язкові мережеві політики. Стандартно мережа Kubernetes дозволяє весь трафік між подами без будь-яких обмежень, що можна обмежити мережевою політикою NetworkPolicy.

2.3 Порівняння підходів забезпечення безпеки Kubernetes

Дотримання всіх раніше наведених рекомендацій дозволить забезпечити базові потреби у безпеці, проте процес ручного налаштування безпеки застосунку чи аудиту вже готового продукту потребує чимало робочого часу інженера з безпеки.

Згідно з цим звітом [24], понад 86% організацій для управління будь-якої частиною всіх своїх контейнерів використовують Kubernetes, проте забезпечення безпеки при цьому залишає бажати кращого. Більше половини цих організацій не зробили достатньо вкладень на захист контейнерів, що уповільнило впровадження Kubernetes або ж спричинило за собою серйозні інциденти безпеки, для вирішення і запобігання яких не було конкретної стратегії.

Крім базових функцій на зразок доступу на основі ролей в Kubernetes немає можливості захистити застосунок, тому доводиться вдаватися до додаткового

програмного забезпечення, таких як: моніторинг, автоматичне сканування та інші, а задля економії часу треба обрати правильну стратегію перевірки застосунку.

Для вибору найбільш ефективного підходу до виявлення вразливостей контейнера і застосунків, необхідно провести дослідження різних методик виявлення вразливостей. Для аналізу отриманих результатів сформувати кількісні характеристики ефективності кожної з методик.

2.3.1 Опис попередніх досліджень

Для того щоб переконатися у правильності вибору найкращого підходу до пошуку вразливостей, було проведені певні дослідження. Суть експерименту полягає в тому, щоб порівняти автоматичні засоби аудиту безпеки Kubernetes з подальшим аналізом результатів та ручну перевірку експертом за певний обмежений проміжок часу.

Поділятись вразливості будуть на 3 основні класи згідно з загальною системою оцінки вразливостей (CVSS) - High(високий), Medium(середній), Low(низький) [25].

Трьом експертам з кібербезпеки, чиї імена будуть замінені на “A”, “B” та “C” відповідно, був даний тестовий застосунок в Kubernetes, що має одну й ту саму кількість загроз, а саме - 10. Загрози в тестовому застосунку поділяються на: 3 - високого рівня; 5 середнього рівня; 2 - низького рівня небезпеки.

Перший експерт (A) користувався лише заданим застосунком з аудиту налаштувань та пасивним сканером вразливостей, що був написаний у ході досліджень, на базі яких розробляв план щодо подальшої ліквідації загроз.

Другому експерту (B) не можна було користуватись нічим крім власних знань та стандартних команд в командній строчці за потреби.

Останній (C) - використовував засоби автоматичного аудиту Kubernetes налаштувань та кластерів, і використовував ручну перевірку з подальшим аналізом та валідацією знайдених вразливостей.

У процесі експерименту також було виміряно час за який кожен експерт провів повну перевірку безпеки.

В якості критерія порівняння результатів експерименту розраховувався показник К (формула 2.1), що показує ефективність виявлення загроз, та залежить від кількості затраченого часу, коефіцієнту рівня небезпеки загроз, і має вигляд:

$$K = \frac{h}{t} \times k_h + \frac{m}{t} \times k_m + \frac{l}{t} \times k_l \tag{2.1}$$

де **h** - кількість вразливостей високого рівня небезпеки; **m** - кількість вразливостей середнього рівня небезпеки; **l** - кількість вразливостей низького рівня небезпеки; **t**- час виявлення; k_h , k_m , k_l [26] - коефіцієнти небезпечності для застосунку після виявлення - можливість виникнення загрози (ступінь доступності зловмисника до застосунку);

- рівень кваліфікації зловмисника;
- ступінь непереборності наслідків атаки.

Чисельні значення коефіцієнтів наведені табл. 2.4.

Таблиця 2.4 – Розраховані коефіцієнти небезпечності

	High	Medium	Low
k	0.24	0.216	0.12

2.3.2 Аналіз результатів

У ході досліджень, було виявлено та показано на табл. 2.5, що у співвідношенні виявлених ризиків та затраченого часу, лідирував експерт “С” з коефіцієнтом - 0.28. Другий результат показав експерт “А”, а останню позицію зайняв експерт “В”.

Таблиця 2.5 – Результати досліджень

	Результат	High	Medium	Low	Час(год)
A	0.23	1	2	2	4
B	0.17		2	2	4
C	0.28	1	3	2	4
Загальний	0.56	3	5	2	

Наочно результати проведеного експерименту можна побачити на рис. 2.7. Слід зазначити, що з плином часу (після 14 годин експерименту) майже всі вразливості були знайдені всіма експертами, проте найшвидше закінчили 2 експерти - "C" та "A" за 7 годин та 5 годин відповідно, останній результат був у експерта "B" - 14 годин.

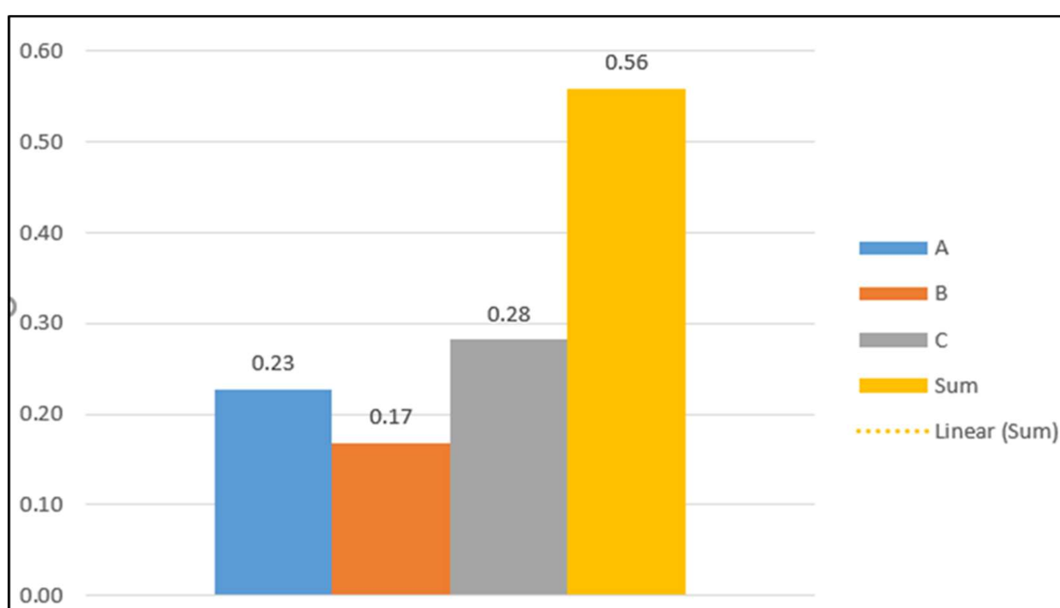


Рисунок 2.7 - Порівняльний графік

Висновки до розділу 2

У даному розділі було проаналізовано основні аспекти безпеки Kubernetes та Docker, у результаті чого були визначені основні вектори атак, групи загроз та моделі загроз. Також було проведено експеримент серед експертів щодо вибору методу сканування, який показав, що комбінація мануального та автоматичного сканування дозволяє найефективніше знаходити загрози за короткий проміжок часу. Тому було

вирішено рухатись у напрямку розробки програмного забезпечення, яке мінімізувало роботу інженера з безпеки, для уникнення помилок людського фактору, та яке б одночасно проводило пасивний аудит безпеки досліджуваних технологій за короткий проміжок часу, а потім все знайдене буде провалідовано мануально.

3 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ПОСТАНОВКА ЗАВДАННЯ

За все існування Kubernetes, було чимало різноманітних спроб створити застосунки для автоматичного сканування. Надалі у цьому розділі буде розглянуті вже готові програмні забезпечення для проведення сканування програм, проаналізовані їхні переваги та недоліки, а також після цього буде сформована задача, яка стоїть перед цією роботою.

3.1 Сканер kube-hunter

Kube-hunter — це інструмент, який можна використовувати для сканування та захисту кластерів Kubernetes, та який розроблений компанією Aqua Security [27]. Інструмент був розроблений для підвищення обізнаності та видимості щодо проблем безпеки в середовищах Kubernetes. Tesla була однією з компаній, яка залишила сотні консолей адміністрування Kubernetes доступними через Інтернет без заходів безпеки або захисту паролем. Проте сканування та моніторинг змін у контейнерах може допомогти уникнути подібних ситуацій.

Kube-hunter відкриває інтерактивний сеанс, де можливо запустити команди та вибрати необхідний параметр сканування. kube-hunter перевіряє домен або діапазон адрес для відкритих портів, пов’язаних з Kubernetes, і перевіряє будь-які проблеми з конфігурацією, які залишають кластер відкритим для зловмисників. Наразі інструмент підтримує три типи сканування:

- Віддалене сканування - щоб вказати віддалені машини для скану
- Внутрішнє сканування- щоб вказати внутрішнє сканування
- Сканування мережі- щоб вказати певний CIDR для сканування

Інструмент також надає пропозиції щодо усунення широкого кола вразливостей, а його вихідний код, написаний на Python, доступний на GitHub. Найпопулярнішими послугами даного сканера є:

Kubelet Container Logs Hunter – отримує журнали з випадкового контейнера

Kubelet Run Hunter – виконує uname всередині випадкового контейнера

Build Date Hunter - перевіряє, коли проксі відкрито, витягує дату збірки kubernetes

Kube-hunter може працювати як контейнер на будь-якій машині за межами кластера — проте компанія попереджає, що вона призначена для роботи лише на кластерах, якими ви володієте. Потрібно лише вказати доменне ім’я або IP-адресу кластера, і kube-hunter проведе пошук в домені або діапазоні адрес відкритих портів, пов’язаних із Kubernetes.

Він також може працювати на машині в кластері. Ще один варіант — запустити його як модуль у кластері, де він може повідомляти про ризик зараження, якщо один із модулів програми буде зламаний.

Дана програма може потенційно виконувати операції зміни стану на кластері, що спонукало компанію попередити користувачів про те, що вони повинні бути дуже обережними при його використанні. Наприклад, буде намагатися записати щось в etcd, виконати “uname -a” у випадковому pod'і та прочитати /etc/shadow з pod'a, куди примонтовано каталог /var/log. У активному стані Kube-hunter використовуватиме знайдені вразливості для пошуку подальших уразливостей. Основна відмінність між звичайним і активним скануванням полягає в тому, що звичайне сканування ніколи не змінить стан кластера, тоді як активне потенційно може виконувати операції зміни стану над кластером, що може бути шкідливим для розробленого програмного забезпечення. Приклад виводу сканування досліджуваного застосунку наведений на рис. 3.1.

```
$ docker run -it --rm --network host aquasec/kube-hunter --token
Choose one of the options below:
1. Remote scanning      (scans one or more specific IPs or DNS names)
2. Interface scanning   (scans subnets on all local network interfaces)
3. IP range scanning    (scans a given IP range)
Your choice: 1
Remotes (separated by a ','): 6
~ Started

Report will be available at:
+------------------------------------------------------------------------------
-------------------------------------------------------------------------+
| https://kube-hunter.aquasec.com/report.html?token=
                                                                                       |
+------------------------------------------------------------------------------
-------------------------------------------------------------------------+
~ Discovering Open Kubernetes Services...

|
| Etcd:
|   type: open service
|   service: Etcd
|_  location: 6:2379
|
| Kubelet API (readonly):
|   type: open service
|   service: Kubelet API (readonly)
|_  location: 6:10255
|
| Kubelet API:
|   type: open service
|   service: Kubelet API
```

Confidential data

(а)

```
Vulnerabilities
+--------------------+----------------------+----------------------+----------------------+----------------------+
| LOCATION           | CATEGORY             | VULNERABILITY        | DESCRIPTION          | EVIDENCE             |
+--------------------+----------------------+----------------------+----------------------+----------------------+
|            6:10250 | Remote Code          | Anonymous            | The kubelet is       |                      |
|                    | Execution            | Authentication       | misconfigured,       |                      |
|                    |                      |                      | potentially allowing |                      |
|                    |                      |                      | secure access to all |                      |
|                    |                      |                      | requests on the      |                      |
|                    |                      |                      | kubelet, without the |                      |
|                    |                      |                      | need to authenticate |                      |
+--------------------+----------------------+----------------------+----------------------+----------------------+
|            6:10255 | Privilege Escalation | Arbitrary Access To  | Api Server not       | v1.15.0              |
|                    |                      | Cluster Scoped       | patched for          |                      |
|                    |                      | Resources            | CVE-2019-11247. API  |                      |
|                    |                      |                      | server allows access |                      |
|                    |                      |                      | to custom resources  |                      |
|                    |                      |                      | via wrong scope      |                      |
+--------------------+----------------------+----------------------+----------------------+----------------------+
|            6:10255 | Information          | K8s Version          | The kubernetes       | v1.15.0              |
|                    | Disclosure           | Disclosure           | version could be     |                      |
|                    |                      |                      | obtained from the    |                      |
|                    |                      |                      | /metrics endpoint on |                      |
|                    |                      |                      | the Kubelet          |                      |
+--------------------+----------------------+----------------------+----------------------+----------------------+
|            6:10255 | Information          | Exposed Pods         | An attacker could    | count: 9             |
|                    | Disclosure           |                      | view sensitive       |                      |
|                    |                      |                      | information about    |                      |
|                    |                      |                      | pods that are bound  |                      |
|                    |                      |                      | to a Node using the  |                      |
|                    |                      |                      | /pods endpoint       |                      |
+--------------------+----------------------+----------------------+----------------------+----------------------+
```

(б)

Рисунок 3.1 (а,б) – Результати роботи сканеру Kube-hunter

Переваги даного рішення :

Хоча дана утиліта дещо агресивно поводиться з мережею і програмою уцілому, проте вона є досить результативною для невеликої програми, яку потрібно просканувати на безпечність. Проте якщо казати про найкращі практики

впровадження безпеки, то варто зазначити, що набагато краще застосовувати комбіновані пасивні сканування.

Також встановлення програми є досить зручним, потрібно лише запуститу одну команду – «pip3 install kube-hunter»

3.2 Сканер Chechov

Основні відомості.У 2019 році творці Checkov [28] взялися за місію створити зручний для розробників спосіб захисту шаблонів інфраструктури як коду (IaC). Коли понад 20% шаблонів Terraform і понад 40% шаблонів CloudFormation в хмарі є неправильно налаштованими, це велике завдання. Неправильні конфігурації в шаблонах можуть призвести до реальних ризиків у бізнесі, наприклад, порти SSH, відкриті для світу, або ж незашифровані бази даних. Ось чому команда Bridgecrew (зараз входить до Prisma Cloud від Palo Alto Networks) задумалася про те, як зробити кращим виявлення проблем до того, як шаблони IaC будуть уже остаточно визначені — під час розробки або на етапі збірки.

Checkov вже охоплює інфраструктуру як сканування коду безпеки CloudFormation для AWS, Azure і GCP, що дозволяє знайти в різноманітній інфраструктурі помилки в конфігурації, такі як публічно відкритих ресурсів, а також допомагає не допускати певних помилок при розміщенні програми в хмарі.

Завдяки даній програмі можна виявити проблеми, такі як контейнери з надмірними привілеями, погані методи життєвого циклу зображення, неправильна конфігурація QoS та перевірки працездатності тощо. Пошук і запобігання цих проблем у рамках кожної збірки мінімізує ризик компрометації робочих навантажень і заощаджує час на усунення проблем у виробництві.

Checkov визначає всі несправні ресурси, включаючи файли і рядки коду, що дозволяє вирішувати проблеми заздалегідь, перш ніж відбудеться розгортання у кластері. Крім того, існує можливість запустити Checkov за допомогою GitHub або в CI/CD, щоб заблокувати розгортання проблем безпеки.

Приклад роботи продемонстровано на малюнку 3.2

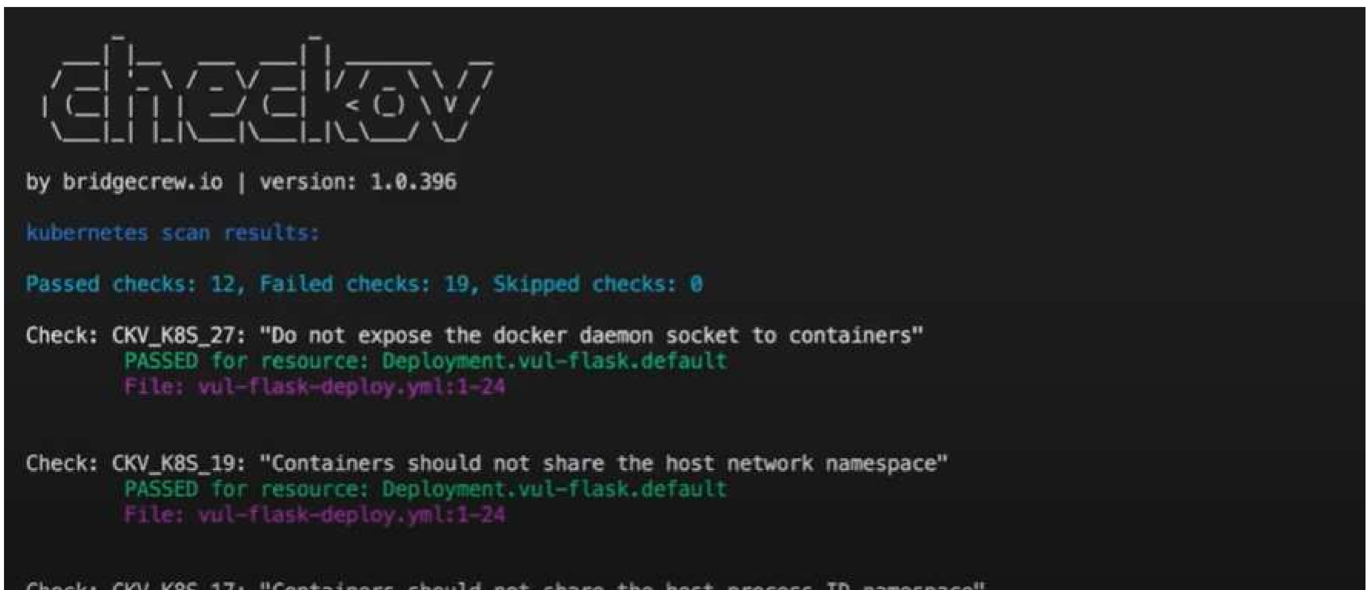


Рисунок 3.2 – Приклад роботи сканера Chechov

Переваги даного рішення :

- Відкритий вихідний код і простий у використанні
- Більше 500 вбудованих політик безпеки
- Найкращі методи відповідності для AWS, Azure і Google Cloud
- Підтримує декілька форматів виводу – CLI, JUnit XML, JSON
- Інтегроване сканування у CI/CD
- Запускає сканування вхідної папки, яка містить файли Terraform і Cloudformation

3.3 Недоліки існуючих рішень

Існуючі рішення є гарним прикладом автоматизації процесів, якими вимушено займались раніше команди інженерів безпеки. Проте варто звернути певну увагу на недоліки даних рішень. У Kube-hunter є один великий недолік, який заважає бути йому «швейцарським ножем» у світі Kubernetes безпека – він є більш агресивним і не рекомендується запускати його в фінальних версіях, і нарешті, так як утиліта дуже агресивно сканує мережі, на це можуть реагувати netcan-детектори провайдерів. Тому

як рішення для невеликого проекту, який не має коштів на утримання високоякісних кадрів, що займаються безпекою застосунку, є непоганим, проте варто враховувати вище наведений ризик, що може вщент поламати цей застосунок, при неправильній конфігурації, або ж при недостатній увазі при його роботі. Також варто зазначити, що усі сканери мають не зручний вивід результату та не користуються уніфікованою методологією.

Що стосується рішення на ім'я Checkov, то тут варто помітити, що воно забезпечує сканування коду та файлів, які стосуються інтеграції застосунку в хмари. Тому дуже важко назвати його сканування уніфікованим і воно не охоплює повністю всю програму, тому може випустити дещо з уваги.

3.4 Аналіз гілок досліджуваного застосунку

Для того щоб зрозуміти як саме впливає сканування конфігурацій, налаштувань та версій використаного програмного забезпечення у Kubernetes на загальну картину загроз, потрібно повернутися до моделей загроз, а саме до дерев рішень, що були побудовані на аналізі векторів атак. Для створення цієї роботи було використано два підходи.

Підхід знизу вгору

Спочатку був здійснений підхід знизу вгору, який розглядав шляхи, як достягти конкретної мети. Цей підхід показує точки входу по всій платформі Kubernetes з метою досягнення поставленої мети – повній компроментації. Надалі детально описано буде кожен листовий вузол за методикою STRIDE, що надає гарні стратегічні дані про загрози та підходи для підтвердження того, який підхід був обраний у даній роботі.

Відмова в обслуговуванні

Заважаючи функціонуванню системи, вичерпуються ресурси всередині кластера. Ця мета розглядається як з точки зору зовнішнього зловмисника, так і з боку

зловмисника, який отримує доступ до контейнера з можливостями віддаленого виконання.

Виконання шкідливого коду

Метою цього дерева атаки є виконання шкідливого коду в контейнері.

Встановалена стійкість (експлоїту)

Підтримуючи стабільність у скомпроментованій системі Kubernetes, дозволяє зберігати свій стан, без певного виявлення у системі як порушника

Сценарний підхід

Другий підхід ґрунтується на сценарному погляді, який визначає вектори атаки, відкриті для зловмисника в певному сценарії. Цей підхід використовує більшу частину деталей першого підходу, але в більш реалістичній формі, яку можна використовувати для фокусування на більш поширених векторах атаки.

Порушене застосування призводить до закріплення в контейнері

Цей сценарій описує потенційні вектори атаки, які відкриваються для зловмисника після того, як він експлуатує програму, що працює в контейнері. Це призведе до віддаленого виконання коду в контейнері через механізми програмного або оболонкового доступу. Це ключовий сценарій і в центрі уваги низка заходів пом’якшення на платформі Kubernetes, щоб обмежити можливості зловмисника в такому сценарії.

Зловмисник у мережі

Цей сценарій зосереджено на внутрішньому зловмиснику з доступом до мереж, де розміщено кластер Kubernetes. Це, ймовірно, буде більш привілейованим користувачем, але без прямого доступа до кластера.

Всі побудовані дерева рішень будуть наведені у Додатку А, проте варто взяти до уваги початкові стадії гілок даного дерева (рис.3.3), які показують, що через існуючі проблеми у конфігурації або у налаштувані безпекових політик або забуті критичні даних можна просуватися далі до кінцевої мети. Тобто, якщо провести початково правильний аналіз та провести налаштування Kubernetes правильним чино, то можна уникнути цих помилок.

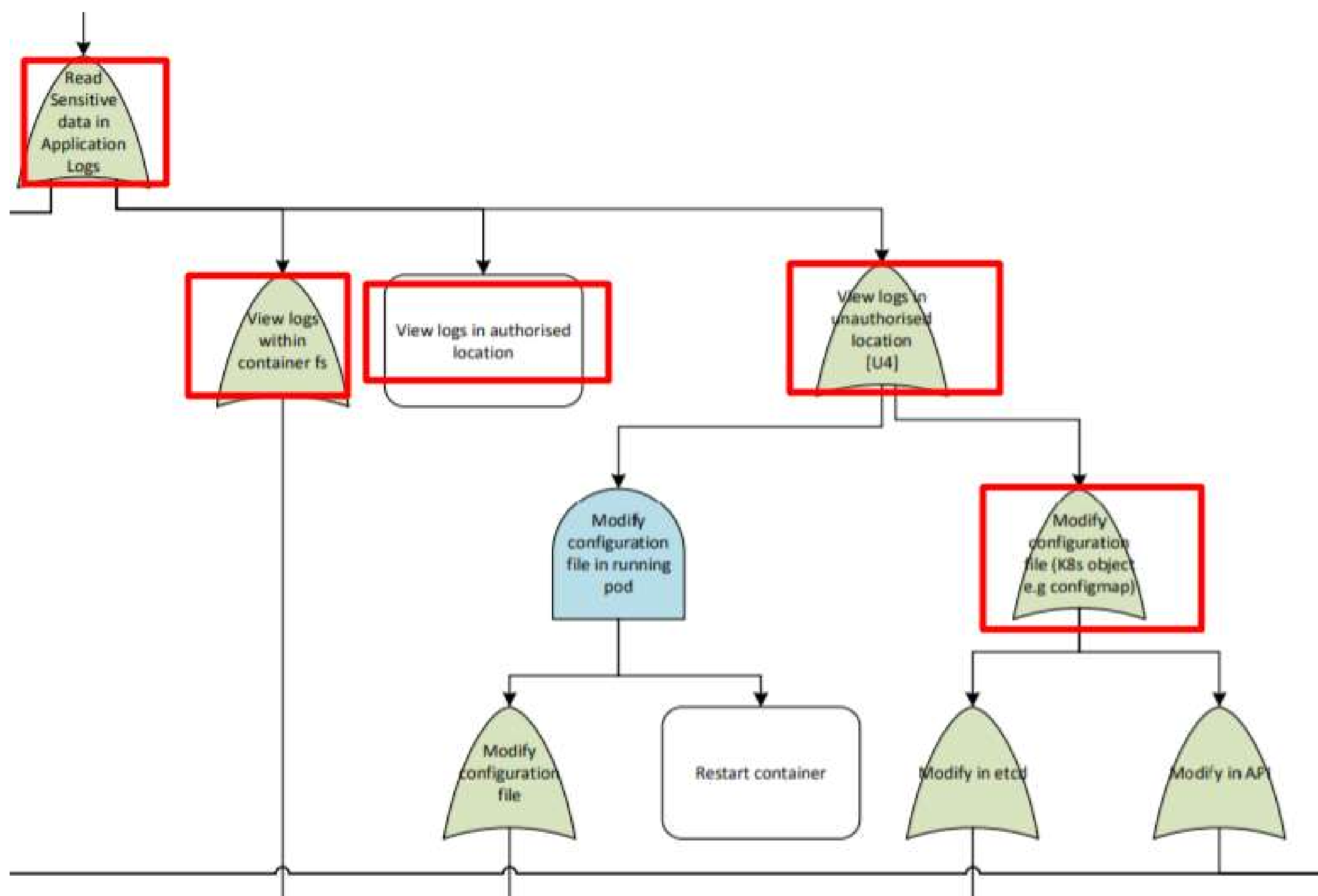


Рисунок 3.3 – Частина дерева рішень

3.5 Постановка задачі

Метою дослідження є аналіз ризиків пов'язаних з використанням Kubernetes та розробка варіантів їх усунення чи зменшення. Для цього необхідно спроектувати засіб, який зможе допомогти пошуку вразливостей у застосунку. Для досягнення поставленої задачі в магістерській дисертації виокремлено наступні завдання:

- Провести аналіз архітектури Docker та Kubernetes;
- Описати основні вектори атак на дані технології та провести первинний аналіз;
- Семантично описати моделі загроз;
- Спроектувати дерева рішень на аналізі векторів атак;

- Проаналізувати, як можна прибрати більшість гілок дерева рішень шляхом розробки застосунку для аудиту;
- Обрати технології та середовище виконання для розробленого програмного забезпечення аудиту Kubernetes;
- Спроектувати роботу застосунку та розробити його;
- Провести тестування на обраному вразливому середовищі розробленого програмного забезпечення;
- Зробити висновки, щодо отриманий результатів.

Створений продукт має відповідати наступним вимогам:

- Простота у використанні;
- Інтуїтивний веб-інтерфейс, доступний користувачу, в даному випадку – інженеру з забезпечення безпеки;
- Охоплення тестуванням усього вразливого застосунку;
- Мінімальні технічні зусилля користувача;
- Рекомендації щодо знайдених вразливостей.

Висновки до розділу 3

У даному розділі було розглянуто існуючі рішення та побудовано дерева рішень, а потім проведено аналіз гілок досліджуваних технологій. Основними недодіками існуючих рішень є те, що деякі з них не можуть охопити скануванням весь застосунок, для перевірки, або ж поводять себе досить агресивно по відношенню до нього, що не дозволяє проводити тестування на більш великих додатках та на пізніх версіях. До того ж, при використанні даних аналізаторів безпеки Kubernetes, їх не можна лишати без людської уваги, адже існує вірогідність завдати продукту невипраної шкоди. Одним із ключових факторів забезпечення мінімізації гілок дерев рішень для моделі загроз є розроблене програмне забезпечення, що дозволяє

проводити статичний аналіз конфігурацій та загальних налаштувань безпеки. За результатами проведеного аналізу визначено задачі, що повинні бути розв'язані, для того щоб побудувати програмне забезпечення для аудиту безпеки Kubernetes, та вимоги, яки має відповідати застосунок.

4 ОПИС РОБОТИ РОЗРОБЛЕНОГО РІШЕННЯ ДЛЯ СИСТЕМ ОРКЕСТРАЦІЇ KUBERNETES

У даному розділі буде описано розроблене рішення для систем оркестрації Kubernetes, початково проаналізувавши потрібні технології для розробки, описано його функціональність та проаналізовано результати роботи.

4.1 Огляд технологій розробки системи

Для розробки програмного застосунку слід обрати застосунок, на якому будуть проводитись сканування та мови програмування, що задовольнать потреби.

4.1.1 CIS Benchmarck

Центр інтернет-безпеки (CIS) є некомерційною організацією, що розробляє вдласті контрольні показники і рекомендації, що дозволяють організаціям поліпшувати свої застосунки забезпечення безпеки, а також налаштовувати власні програмні продукди безпечним чином. Ця ініціатива спрямована на створення базових рівниів конфігурації безпеки систем, які зазвичай зустрічаються в усіх організацях. Він спирається на досвід фахівців з кібербезпеки та IT з уряду, бізнесу та наукових кіл з усього світу. Для розробки стандартів і передового досвіду, включаючи контрольні показники CIS, елементи керування та посилені зображення, вони дотримуються моделі прийняття рішень консенсусу.

Тобто вони розробляють певні стандарти та чек-лісти для полегшення роботи інженерам з безпеки та уніфікацію процедури перевірки або забезпечення безпеки. Для завантаження доступні декілька десятків гадйлайнів з безпечного налаштування різноманітних систем: Windows, Linux, OSX, MySQL, Cisco та багатьох інших.

Benchmark CIS – це базові лінії конфігурації та найкращі методи безпечного налаштування системи. Кожна з рекомендацій посилається на один або кілька засобів контролю CIS, які були розроблені, щоб допомогти організаціям покращити свої можливості кіберзахисту. Елементи керування CIS відповідають багатьом встановленим стандартам і нормативним базам, включаючи NIST Cybersecurity Framework (CSF) і NIST SP 800-53, серію стандартів ISO 27000, PCI DSS, HIPAA та інші.

Кожен контрольний показник проходить два етапи консенсусної перевірки. Перше відбувається під час початкової розробки, коли експерти збираються для обговорення, створення та тестування робочих проектів, поки не досягнуть консенсусу щодо еталону. Під час другого етапу, після публікації контрольного показника, команда консенсусу розглядає відгуки від інтернет-спільноти на предмет включення до тесту.

Тести CIS забезпечують два рівні параметрів безпеки:

- Рівень 1 рекомендує основні основні вимоги безпеки, які можуть бути налаштовані в будь-якій системі і повинні спричиняти незначні переривання обслуговування або знижувати функціональність.
- Рівень 2 рекомендує налаштування безпеки для середовищ, що вимагають більшої безпеки, що може призвести до деякого зниження функціональності.

Одним з продуктів даної організації також є образи CIS Hardened — це безпечно налаштовані образи віртуальних машин на основі тестів CIS, закріплених за профілем CIS рівня 1 або рівня 2. Загартовування – це процес, який допомагає захистити від несанкціонованого доступу, відмови в обслуговуванні та інших кіберзагроз, обмежуючи потенційні слабкі місця, які роблять системи вразливими для кібератак. Остання версія CIS Kubernetes Benchmark v1.5.1 надає рекомендації щодо конфігурацій безпеки для Kubernetes версій 1.15 і новіших версій. CIS Kubernetes Benchmark призначений для реалізацій, які керують як площиною

керування, яка включає etcd, сервер API, контролером і планувальником, так і площиною даних, яка складається з одного або кількох вузлів або екземплярів EC2.

Тому розроблене програмне забезпечення магістерської дисертації буде опиратись на дані, інформацію та рекомендації саме CIS Kubernetes Benchmark v1.5.1, оскільки це є загально прийнятим набором правил, якими керуються DevSecOps інженери. Так як даний гайдлайн містить в собі сотні практичних рекомендацій, які вміщуються в 250 сторінках, то буде дещо складно їх реалізовувати вручну, не припускаючись при цьому помилок. Саме тому розроблений застосунок допомагає автоматизувати перевірку конфігурацій та образів з мінімальними втручаннями в це інженера.

4.1.2 Мова програмування Python

Python як мова програмування зараз дуже популярна, і за останні кілька років її ріст було неймовірним. Наразі дана мова домінує в галузі кібербезпеки і існує вже довгий час. Python підтримує синтаксис, простий і зрозумілий і має широкий спектр бібліотек, що робить його універсальною мовою програмування.

Існує ряд факторів, які роблять Python для кібербезпеки кращою мовою для професіоналів з кібербезпеки. Нижче наведено фактори:

1. Легко вивчати, розуміти та впроваджувати: Python досить простий у навчанні, проектуванні та реалізації, що робить його найбільш зручною мовою програмування.
2. Python — це безкоштовна мова програмування з відкритим вихідним кодом. Вона розроблена як повна безкоштовна мова програмування з відкритим вихідним кодом, що робить її доступною для будь-кого для подальшого її вдосконалення. Існує чимало інтерпретаторів та середови розробки для даної мови, які є також безкоштовними.

3. Спрощене налагодження коду. Python вимагає мінімального кодування через наявність кількох вбудованих бібліотек для виконання певної функціональності, що полегшує програмісту процес тестування коду на наявність помилок, водночас зменшуючи ризик мовних проблем.

4. Широкий спектр бібліотек та зручні для користувача структури даних. Python включає в себе великі бібліотеки, які підтримують широкі функціональні можливості та функції, такі як операції з рядками, протоколи та інструменти веб-сервісів. Це обмежує довжину коду для написання, тим самим зменшуючи кількість помилок. Python також охоплює структури даних, такі як вбудовані списки та словники.

5. Python забезпечує швидкість та продуктивність. Основні першопричини швидкості та продуктивності цієї мови полягають у розширеному контролі та можливостях обробки тексту, які надаються користувачам. Досвідчені професіонали з кібербезпеки вважають, що Python реалізує його як безпроблемний код.

6. Підтримка автоматичного керування пам’яттю. Python підтримує вбудовані функції керування пам’яттю автоматично за проектом. Програмістам полегшється робота такими факторами, як розбиття на розділи, виділення пам’яті та кешування.

Python — це також мова сценаріїв загального призначення на стороні сервера, реалізована тисячами проектів безпеки. Деякі функції, які підтримуються за допомогою базового програмування Python, не потребують сторонніх інструментів, приклади яких наведено нижче:

- Симуляція атак
- Сканування портів
- Відбитки пальців веб-сервера
- Доступ до поштових серверів
- Генерація навантаження та тестування сайту
- Сканування бездротової мережі
- Передача трафіку в мережі

- Системи виявлення та запобігання вторгненням

Python з точки зору кібербезпеки показує себе як чудовий інструмент автоматизації, адже щоденні завдання спеціаліста з безпеки передбачають написання ефективних скриптів для автоматизації завдань. Python рекомендується через його простий синтаксис і широкий спектр підтримуваних бібліотек з великою функціональністю. Процес автоматизації значно скорочує не тільки час, але й кількість помилок.

Крім того у даній мові є бібліотека та функціонал Regex. Regex відноситься до регулярних виразів, і цей інструмент дозволяє шукати за певними патернами інформацію, для подальшої її обробки та використання. Він надає можливість витягувати інформацію з файлів журналів під час дослідження або вилучення інформації з Інтернету та навіть допоможе знайти IP-адреси, щоб виявити хакерську діяльність, об'єднавши регулярні вирази з іншими бібліотеками Python для кібербезпеки.

Тестування на проникнення – це процес спроби зламати систему, мережу або програмне забезпечення для перевірки безпеки. Це дуже важливе напрямок застосування Python в кібербезпеці. Для досягнення цього ефекту професіоналам необхідно створювати свої сценарії та інструменти, проте не сильно заглиблюватись у процес розробки, саме через простоту даної мови, вона є така популярна в сфері кібербезпеки.

Також Python надає можливість використовувати такі необхідні інструменти як перехоплення трафіку, сканування портів, виконання команд безпосередньо в командній строці системи, програмувати сокети та навіть вилучення геолокації, за яку відповідає модуль «pygeoip», що отримує місцезнаходження IP-адреси в режимі реального часу за допомогою API Google і Python.

Отже, Python стала надзвичайно корисною у кібербезпеці, оскільки вона підтримує та виконує безліч функцій кібербезпеки, таких як аналіз шкідливих

програм, сканування та функції тестування на проникнення, тощо. Тому для автоматизації процесів пошуку конфігураційних файлів та всих тих, які потребують перевірки, була обрана саме ця мова програмування.

4.1.3 Мова програмування Go

Google Go (також відомий як Golang) продовжує бути однією з популярних мов програмного забезпечення, що дозволяє розробникам швидко розробляти якісний код. Його походження можна простежити з того часу, коли інженери Google вирішили створити просту у використанні мову програмування, яка б усунула повільність і незграбність розробки програмного забезпечення, зробивши процес більш продуктивним і масштабованим. Результатомцієї роботи було скомпільоване рішення, яке забезпечує велику багатопотоковість, паралельність і продуктивність під великим об'ємом розробки. Сьогодні Go є найшвидшою зростаючою мовою з точки зору впровадження на ринку. Варто зазначити декілька особливих переваг, такі як:

1. Простота коду. Синтаксис Go відносно невеликий і залишився більш-менш незмінним з плином часу і більше того, існує лише один стандартний формат коду (генерується інструментом fmt). Відсутність змін у синтаксисі Go також допомагає розробникам, адже тому код не надто складний, і немає необхідності вивчати нові парадигми чи синтаксис. Це робить його прозорим, легким для вивчення та розуміння. Якщо ви використовуєте нову версію Go, майже всі програми, написані на старішій версії мови програмування, компілюватимуться та запускатимуться без будь-яких змін.
2. Широкі інструменти програмування

Як проект із відкритим кодом, Go забезпечує легкий доступ до необхідних інструментів розробки, таких як:

- GoLand від JetBrains — це кросплатформна IDE з такими функціями, як виявлення помилок «на льоту» з пропозиціями щодо виправлення, рефакторинг з однокроковою скасуванням, інтелектуальне завершення коду, виявлення мертвого коду, підказки щодо документації та інтеграція git тощо.
- GoClipse - IDE Eclipse з підтримкою мови програмування Go.
- LiteIDE – це рішення з відкритим вихідним кодом та кросплатформенним рішенням включає настроювані команди збірки, редактор коду та керування, а також широку підтримку Go.
- Zeus IDE — це IDE для платформи Windows, що полегшує роботу розробників завдяки gocode для автозаповнення, документації та навігації по коду, а також таким інструментам, як gofmt і goimports для автоматичного форматування коду.
- VSCode - редактор коду з широкою підтримкою Go, що забезпечує, наприклад, навігацію по мовному коду, пошук символів, збіг у дужках і фрагменти.
- VIM-go – цей плагін для Vim, який регулярно оновлюється, забезпечує компіляцію пакетів Go, підсвічування та підсвічування синтаксису, а також інтегровану підтримку Delve.

3. Мова програмування Go швидка. Загальна ідея Go полягає в тому, що це швидка мова програмування, навіть у порівняння з раніше описаною швидкодією Python, Go явно є лідером у даній характеристиці.

4. Парадигма паралельності. Якщо програма підтримує парадигму паралельності, це означає, що вона може обробляти кілька завдань/дій (здавалося б) одночасно в періоди, що збігаються. Go ізначально підтримує парадигму паралельності і дозволяє багатопоточність, багатообробку та асинхронність. Планувальник часу виконання Golang допомагає керувати всіма діями (горутинами), які створюються і потребують часу процесора. У цьому також допомагають канали - ці введені канали

синхронізують горутини і роблять спілкування між ними більш ефективним і плавним.

Це не єдині функції Go, які охоче використовують інженери – керування залежностями, потужність відображення під час виконання, збір сміття та компоненти також є помітними перевагами.

Kubernetes, як система оркестрування контейнерів із відкритим кодом для керування контейнерними робочими навантаженнями та сервісами була створена Google. І безсумнівно тому він написаний мовою Go. Рішення використовувати цю мову програмування було не наслідком «родинних зв'язків», а завдяки швидким інструментам Go, високоякісним бібліотекам і збиранню сміття.

Мова програмування Go поки не настільки популярна, як Python, але вона дала про себе знати в світі розробників. Його мінімальний і простий синтаксис, інноваційні підпрограми замість типових потоків і широкий спектр інструментів переконали навіть такі великі імена, як American Express, Cloudflare, Facebook, Microsoft і Netflix. Цей список показує, скільки компаній з усього світу використовують мову програмування Go. «Код, який росте з благодаттю», також зростає завдяки спільноті та потужним інструментам. Це робить мову програмування Go конкурентоспроможним і гідним уваги вибором для цієї магістерської дисертації. Тому основна частина, що взаємодіє з Kubernetes та проводить сканування конфігураційних файлів, написана на цій мові програмування.

4.1.4 Kubernetes-goat

Так як заборонено сканування застосунків, що написані не власноруч, або нема дозволу для їх перевірки, то дослідники, що вивчають безпеку Kubernetes, створили власний плацдарм для відточування навичок Kubernetes-Goat [29]. Тут в одному місці можна відтворити всі відомі атаки на Kubernetes, такі як:

Конфіденційні ключі в кодових базах. Розробники дуже часто передають конфіденційну інформацію в системи контролю версій і не видаляють її перед комітом.

Експлуатація DIND(docker-in-docker). Більшість систем CI/CD, які використовують Docker та створюють контейнери, у конвеєрі використовують те, що називається DIND (docker-in-docker). Тут у цьому сценарії користувач намагатиметься використати даний функціонал на свою користь та отримати доступ до хост-системи.

SSRF у світі K8S. Уразливість SSRF (Server Side Request Forgery) стала основним видом атак для хмарних середовищ. У цьому сценарії користувач може використовувати уразливість програми, що призводить до SSRF, щоб отримати доступ до метаданих хмарного екземпляра, а також до інформації метаданих внутрішніх служб.

Вихід із контейнера для доступу до хост-системи. Більшість програмного забезпечення моніторингу, відстеження та налагодження вимагає для роботи додаткові привілеї та можливості. Тут у цьому сценарії користувач може використати модуль із додатковими можливостями та привілеями, включаючи HostPath, що дозволяє йому отримати доступ до хост-системи та надати конфігурацію рівня вузла, щоб отримати повну компроментацію кластера.

Атака на приватний реєстр. Реєстр контейнерів — це місце, куди передаються всі зображення контейнерів. У більшості випадків кожна організація має власний приватний реєстр. Також іноді він виявляється неправильно налаштованим, загальнодоступним/відкритим. З іншого боку, розробники припускають, що це лише внутрішній приватний реєстр, і в кінцевому підсумку зберігають всю конфіденційну інформацію всередині зображень контейнера.

Відкриті служби NodePort. Якщо хтось із користувачів відкрив будь-який сервіс із кластером Kubernetes за допомогою NodePort, то це означає, що на вузлах, на яких працюють кластери Kubernetes, не ввімкнено брандмауер/безпеку мережі. Тому можна там побачити деякі неавтентифіковані та неавторизовані служби.

Helm v2 tiller для PwN кластера. Helm є менеджером пакетів для Kubernetes. Це як apt-get для ubuntu. У цьому сценарії користувач може використати старішу версію helm (версія 2), налаштування RBAC служби tiller за замовчуванням, щоб отримати доступ до завершеного кластера.

Аналіз контейнера для майнера криптовалют. Майнінг криптовалют став популярним завдяки цій сучасній інфраструктурі. Особливо такі середовища, як Kubernetes, є легкою мішенню, оскільки ви можете не подивитися, на чому саме побудований образ контейнера і що він робить за допомогою активного моніторингу. У цьому сценарії користувач після знаходження, аналізує та ідентифікує майнер криптовалют.

Обхід просторів імен Kubernetes. За замовчуванням Kubernetes використовує плоску мережеву схему, що означає, що будь-який модуль/служба, який є в кластері, може спілкуватися з іншими. Простіри імен у кластері за замовчуванням не мають жодних обмежень безпеки мережі, тому будь-хто в будь-якому просторі імен може спілкуватися з іншим простором імен.

Отримання інформації про середовище. Кожне середовище в Kubernetes має багато інформації, якою можна поділитися. Деякі з ключових місць в середовищі містять в собі відкриті та загальнодоступні секрети, apikey, конфігурації, сервіси, та інші.

DoS ресурсів пам’яті/процесора. Якщо в маніфестах Kubernetes немає специфікації ресурсів і не застосовуються діапазони обмежень для контейнерів, то зловмисник може споживати всі ресурси, на яких запущено модуль/розгортання, позбавляти інших ресурсів і відтворювати атаку DoS для середовища.

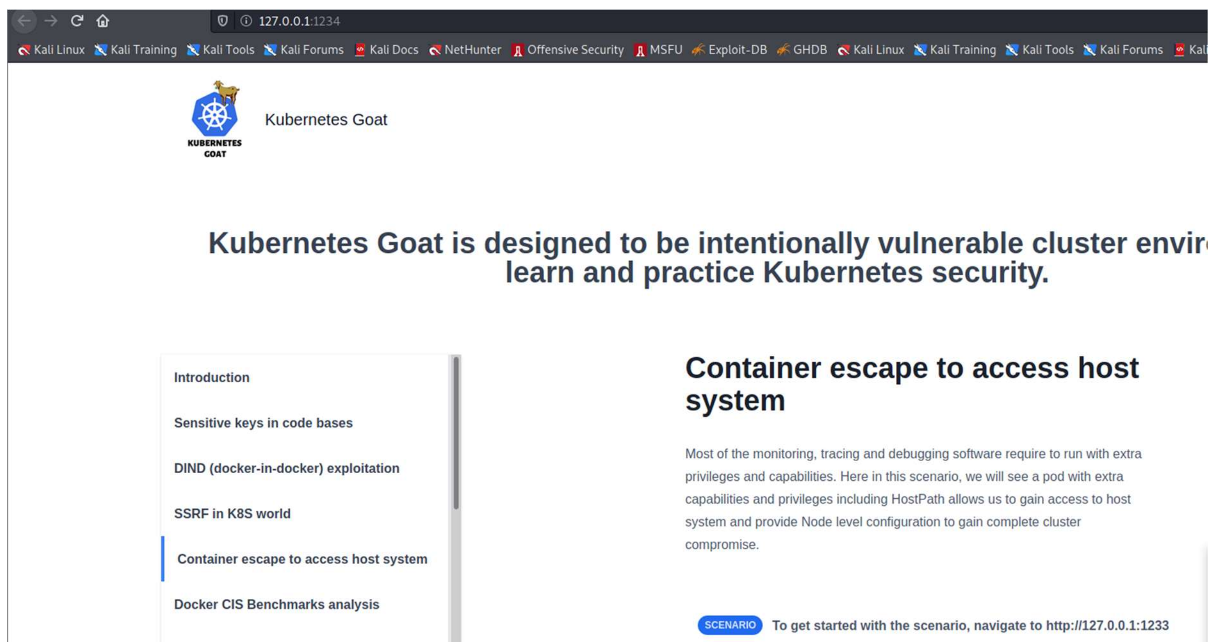
З усього вищенаписаного, можна зробити висновки, що даний відкритий ресурс є чудовим місцем для аналізу безпеки Kubernetes, проведення аудиту, адже вже чітко відомі місця атак і їхню кількість. А тому Kubernetes-goat вартий свого застосування у даній магістерській дисертації.

Запуск застосунку є дуже легким та здійснюється лише за допомогою 1ї команди (рис. 4.1), проте варто зазначити, що перед цим потрібно ще запустити minikube для корректної роботи програми.

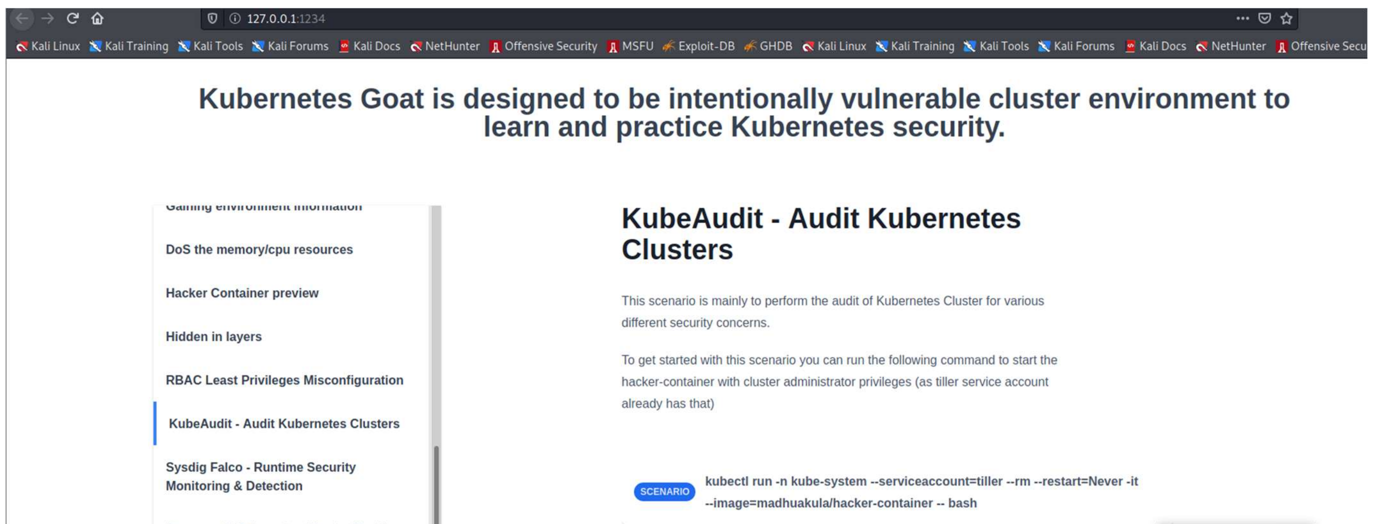
```
┌──(kali㉿kali)-[~/kubernetes-goat]
└─$ minikube start
😄  minikube v1.24.0 on Debian kali-rolling
✨  Using the docker driver based on existing profile
👍  Starting control plane node minikube in cluster minikube
🚜  Pulling base image ...
🔄  Restarting existing docker container for "minikube" ...
🐳  Preparing Kubernetes v1.22.3 on Docker 20.10.8 ...
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: default-storageclass, storage-provisioner
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" n
amespace by default

┌──(kali㉿kali)-[~/kubernetes-goat]
└─$ bash access-kubernetes-goat.sh
kubectl setup looks good.
Creating port forward for all the Kubernetes Goat resources to locally. We wi
ll be using 1230 to 1236 ports locally!
Visit http://127.0.0.1:1234 to get started with your Kuberenetes Goat hacking
!
```

Рисунок 4.1 - Налаштування Kuber-goat

Дане програмне забезпечення також має графічний інтерфейс (рис.4.2), який знаходиться за адресою http://127.0.0.1:1234, де можна знайти сторінки, що ж вразливими до тої чи іншої вищезгаданої атаки.



(a)



(б)

Рисунок 4.2 (а, б) – Приклади запущеного застосунку для тестування на віртуальній машині

4.2 Опис функціональності застосунку

У ході роботи над магістерською дисертацією було розроблено інструмент для перевірки вразливостей і тестування за стандартами CIS Docker\Kubernetes, який дозволяє користувачам даного застосунку отримати точну й швидку оцінку ризиків своїх кластерів Kubernetes. Дане програмне забезпечення сканує всі зображення, які використовуються в кластері Kubernetes, включаючи зображення модулів програм і системних модулів, а також конфігураційні файли застосунку. Він не сканує всі реєстри образів і не вимагає попередньої інтеграції з конвеєрами CI/CD.

Це інструмент можна налаштувати так, що він дозволяє користувачам визначати обсяг сканування (цільові простори імен), швидкість та рівень уразливостей, які цікавлять. За допомогою написаного коду на мові Python (рис. 4.3) з підключеними бібліотеками, що дозволяють виконувати і відображати код у цільовій системі, було досягнуто мінімального втручання кінцевого користувача продукту, адже достатньо лише запустити скріпт за допомогою 1 команди і отримати бажаний результат.

scr.py

```
1 #!/usr/bin/env python
2 import os
3 import subprocess
4 from subprocess import call
5 from termcolor import colored
6 def shell_command ():
7         call('echo "Detecting all configuration files ..........."', shell=True)
8         os.system('find /home/kali/kubernetes-goat -type f -name "*.yaml"')
9         print(colored('[scan-preparing]', 'green'), colored('Writed to out.txt file, please check it!', 'blue'))
10        os.system('find /home/kali/kubernetes-goat -type f -name "*.yaml" > out.txt')
11        print(colored('[scan-preparing]', 'green'), colored('Modifying for starting scanning.......', 'blue'))
12 def execution(new):
13     data = new.readlines()
14     for n,line in enumerate(data):
15         if line.startswith("line"):
16            data[n] = "\n"+line.rstrip()
17         else:
18            data[n]=line.rstrip()
19     cmd = ' '.join(data)
20     print(cmd)
21     os.system(cmd)
22
23 if __name__ == '__main__':
24     shell_command()
25     yaml_files = list(open('out.txt'))
26     for i in range(len(yaml_files)):
27         yaml_files[i] = yaml_files[i].strip('\n')
28         prefix = '-f '
29         suf=' '
30     new=open('dest.txt', 'r+')
31     old=open('out.txt', 'r')
32     with old as src:
33         with new as dest:
```

Рисунок 4.3 – Частина основної програмної частини

Даний скріпт запускає виконання сканеру на заданому застосунку, шукає в ньому усі файли для аналізу, які записує у окремий файл, та виводить результат роботи у консоль.

Спочатку забираються конфігураційні дані, які потрапляють надалі у препроцесор, що виокремлює кожну конфігурацію для подальшого аналізу для кожного файлу окремо. Після закінчення збору, відбувається придушення збірника файлів та розпочинається робота основного застосунку – сканеру, що забирає правильні конфігурації зі сховища та співставляє з тестовими, виносить рішення щодо праильності кожної з них, виокремлює лише небезпечні конфігурації, а правильні надалі не застосовуются при аналізі. Далі відбувається сортування даних конфігурацій за рівнем їх небезпечності по кожному файлу, додавання ознаки

небезпечності та у подальшому – додаткової інформації, інформації, що стосується їх усуненню до них.

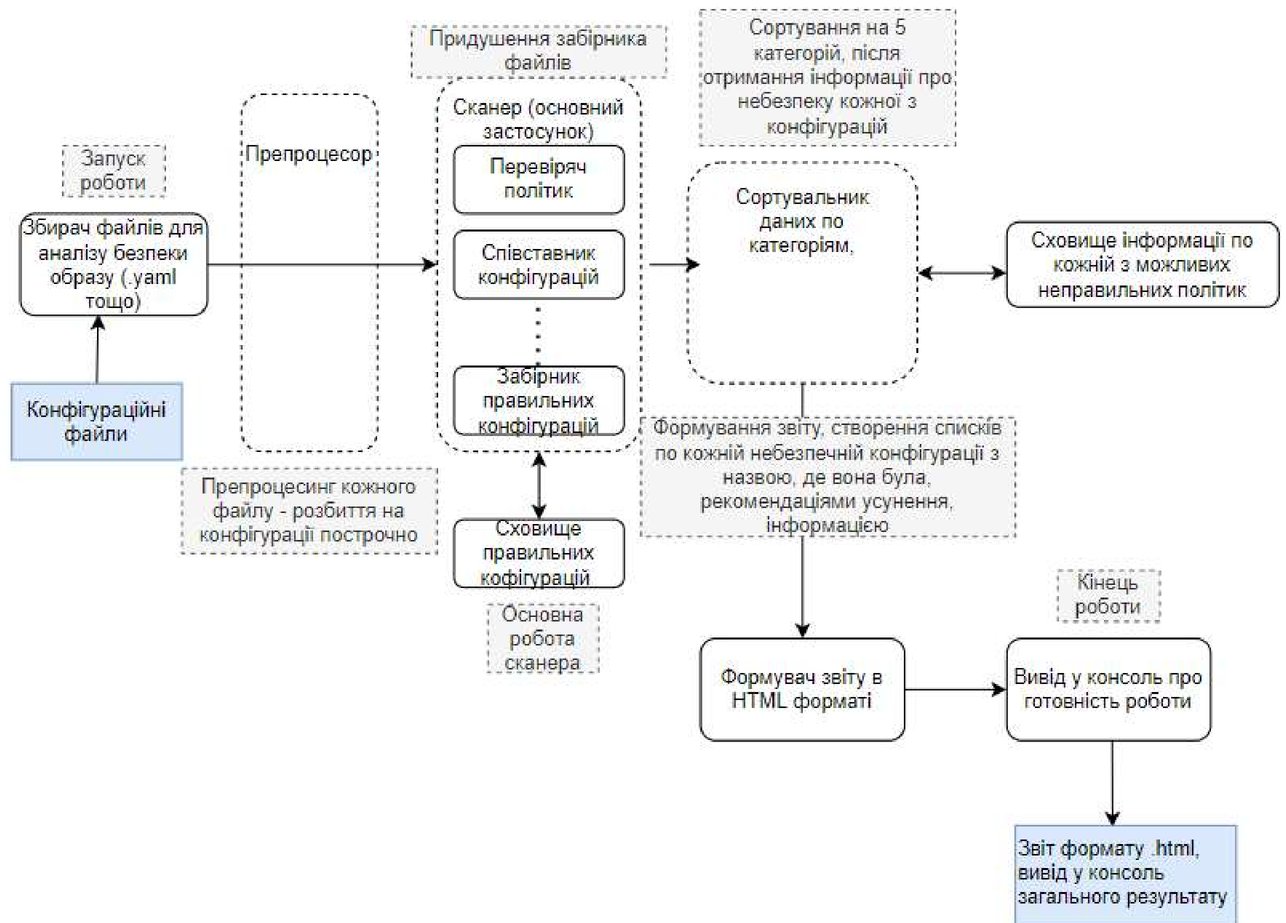


Рисунок 4.4 - Dataflow діаграма

Також розроблений інструмент надає графічний інтерфейс користувачу (рис 4.5), який дозволяє отримати звіт в форматі HTML, за допомогою додаткових підключених бібліотек та стилів, по кожній загрозі, де можна побачити основні знайдені проблеми\вразливості застосунку та проранжувати їх по класу вразливостей; рекомендації щодо усунення та опис самої проблеми; кодований індентифікатор, що позначає проблему, для того щоб було легко орієнтуватися. Вся інформація про опис, рекомендації та посилання, які стосуються вразливого місця, були взяті з CIS Kubernetes Benchmark. У кінці роботи застосунку можна побачити вивід у консолі про готовність та загальну кількість знайдених вразливостей.

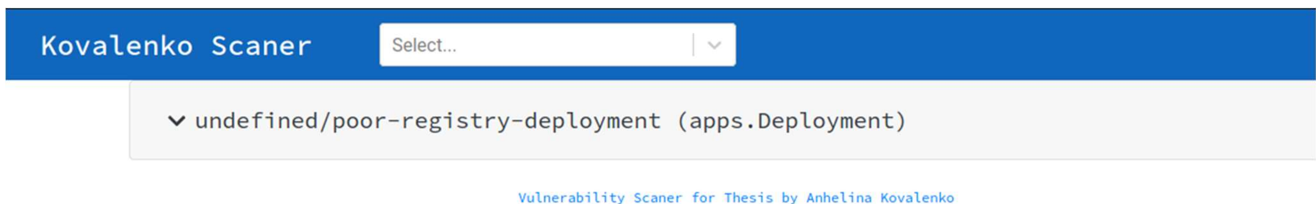


Рисунок 4.5 - Приклад звіту для скану 1ї конфігурації

Для того щоб протестувати наше розроблене програмне забезпечння, було виконано наступні дії, що передували цьому.

- Налаштовано кластер Kubernetes і kubeconfig (~/.kube/config) для цільового кластера.
- Встановлено kubectl — це функція Kubernetes cli, яка може робити багато речей. Однією з таких речей є можливість виконати перевірку сухого запуску для файлу yaml.
- Налаштовано та встановлено Helmv2. Helm допомагає в керуванні застосунками Kubernetes — визначати, встановлювати та оновлювати програми Kubernetes як діаграми керування.
- Використано Datree, що сканує маніфести та діаграми Helm Kubernetes на дійсність та дотримання встановлених політик.
- Для валідації файлів конфігурації за допомогою спеціальних правил, зазначених у YAML було встановлено та використано Config Lint.
- Налаштовано та встановлено minikube, що є зручним інструментом, який застосовується в компаніях для локальних експериментів з Kubernetes та є інструментом із відкритим вихідним кодом, який допомагає налаштувати одновузловий кластер Kubernetes на локальному комп'ютері.

4.3 Результати роботи застосунку

Для тестування інструменту забезпечення аудиту Kubenetes було обрано за ціль вищезгаданий застосунок Kubernetes-goat та розгорнуто локально на віртуальній машині разом з попередньо налаштованим Docker, minikube та встановлено всі необхідні для роботи бібліотеки.

Після переконання у коректності роботи застосунку, було визначено його місцезнаходження у директорії та запущено скріпт на Python для отримання результатів сканування. У ході роботи можна побачити наглядно результати у консолі локальної машини, де було розгорнуто програму (рис. 4.6).

```
┌──(kali㉿kali)-[~/kubernetes-goat]
└─$ python3 scr.py
Detecting all configuration files ..........
/home/kali/kubernetes-goat/infrastructure/helm-tiller/pwnchart/Chart.yaml
/home/kali/kubernetes-goat/infrastructure/helm-tiller/pwnchart/templates/clusterrole.yaml
/home/kali/kubernetes-goat/infrastructure/helm-tiller/pwnchart/templates/clusterrolebinding.yaml
/home/kali/kubernetes-goat/infrastructure/helm-tiller/pwnchart/values.yaml
/home/kali/kubernetes-goat/scenarios/metadata-db/Chart.yaml
/home/kali/kubernetes-goat/scenarios/metadata-db/templates/service.yaml
/home/kali/kubernetes-goat/scenarios/metadata-db/templates/ingress.yaml
/home/kali/kubernetes-goat/scenarios/metadata-db/templates/deployment.yaml
/home/kali/kubernetes-goat/scenarios/metadata-db/templates/tests/test-connection.yaml
/home/kali/kubernetes-goat/scenarios/metadata-db/values.yaml
/home/kali/kubernetes-goat/scenarios/cache-store/deployment.yaml
/home/kali/kubernetes-goat/scenarios/health-check/deployment.yaml
/home/kali/kubernetes-goat/scenarios/health-check/deployment-kind.yaml
/home/kali/kubernetes-goat/scenarios/helm2-rbac/setup.yaml
/home/kali/kubernetes-goat/scenarios/hunger-check/deployment.yaml
/home/kali/kubernetes-goat/scenarios/batch-check/job.yaml
/home/kali/kubernetes-goat/scenarios/hidden-in-layers/deployment.yaml
/home/kali/kubernetes-goat/scenarios/internal-proxy/deployment.yaml
/home/kali/kubernetes-goat/scenarios/kubernetes-goat-home/deployment.yaml
/home/kali/kubernetes-goat/scenarios/system-monitor/deployment.yaml
/home/kali/kubernetes-goat/scenarios/docker-bench-security/deployment.yaml
/home/kali/kubernetes-goat/scenarios/build-code/deployment.yaml
/home/kali/kubernetes-goat/scenarios/poor-registry/deployment.yaml
/home/kali/kubernetes-goat/scenarios/kube-bench-security/master-job.yaml
/home/kali/kubernetes-goat/scenarios/kube-bench-security/node-job.yaml
/home/kali/kubernetes-goat/kind-setup/kind-cluster-setup.yaml
[scan-preparing] Writed to out.txt file, please check it!
[scan-preparing] Modifying for starting scanning.......
[scan-preparing] .................executing command ...........
```

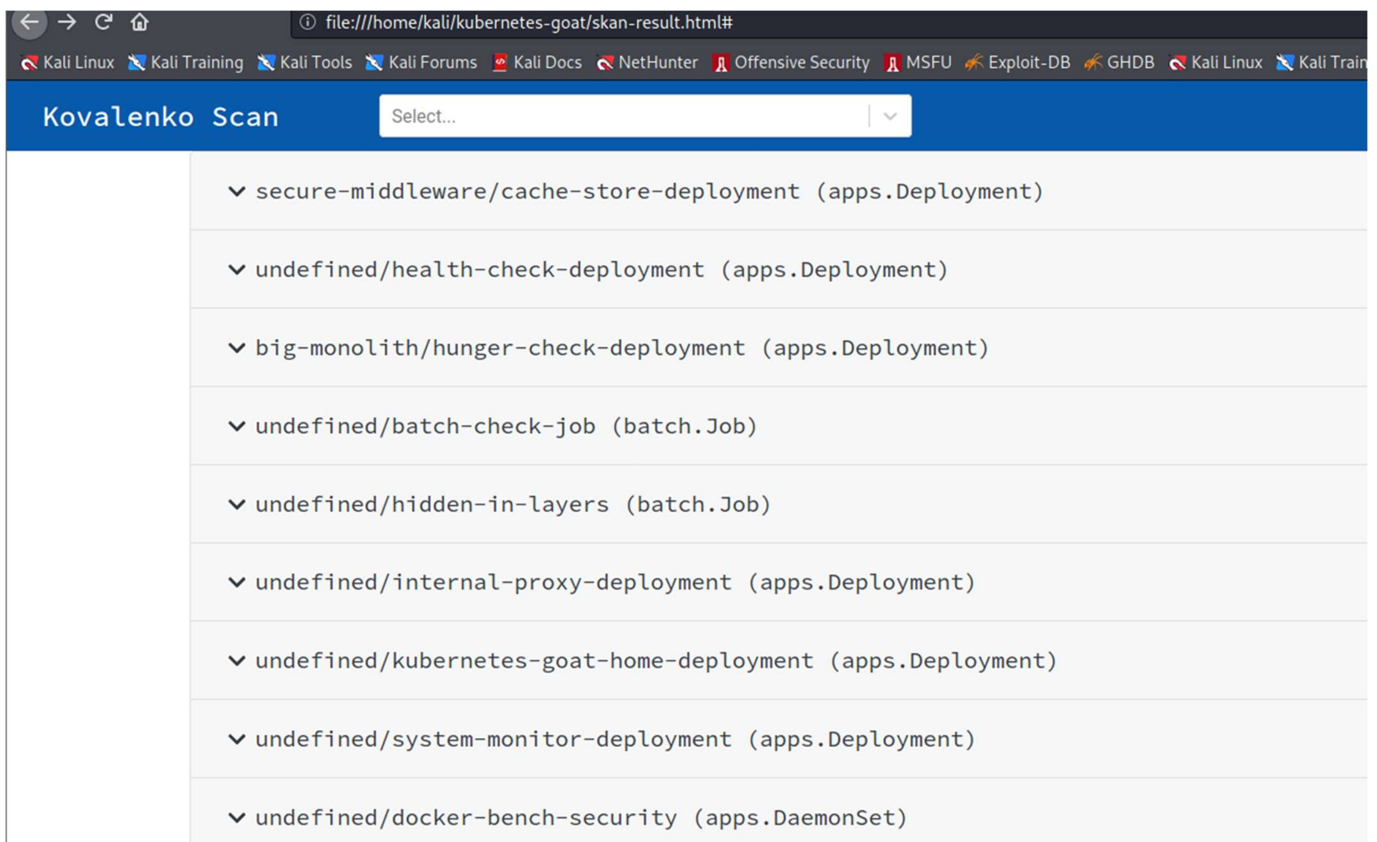
(a)

```
Analyzing resources from '26' files/directories.
Loaded '14' objects
Ops Conformance | Workload Readiness & Liveness
Ops Conformance | Workload Capacity Planning
Workload Software Supply Chain | Image Registry Whitelist
Ingress Controllers & Services | Ingress Security & Hardening Configuration
Ingress Controllers & Services | Ingress Controller (nginx)
Ingress Controllers & Services | Service Resource Checks
Pod Security | Workload Hardening
Secret Hunting | Find Secrets in ConfigMaps
Secret Hunting | Find Secrets in Pod Environment Variables
Admission Controllers | Validating Admission Controllers
Admission Controllers | Mutating Admission Controllers
Generating report (html) and saving as 'skan-result.html'
Summary:
Critical .... 13
High ........ 56
Medium ...... 68
Low ......... 0
Pass ........ 87
```

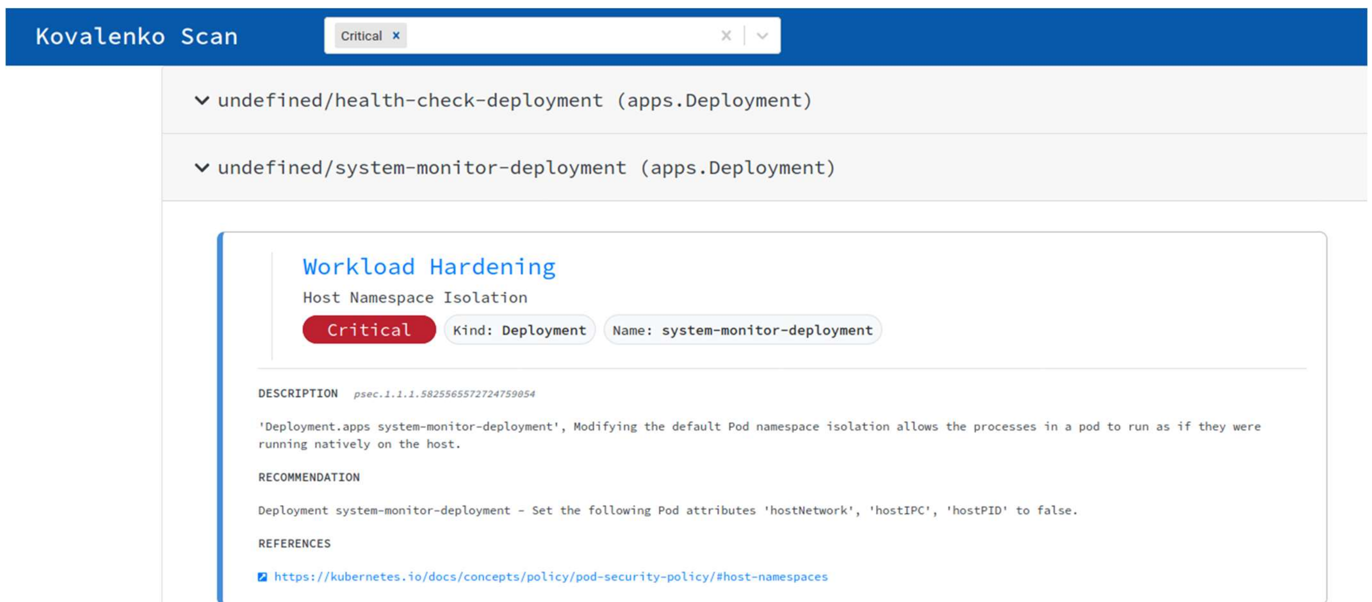
(б)

Рисунок 4.6 (а,б) – Результати роботи інструменту сканування в консолі

Також було згенеровано звіт у файлі scan-result.html (рис. 4.7), що дає змогу інженерові наглядно побачити весь перелік проблем застосунку та отримати рекомандації, щодо їх усунення.



(а)



(б)

Рисунок 4.7 (а, б) - Згенерований звіт у форматі HTML

У ході роботи було отримано результати, що були попередньо проранжовані на класи та провалідовані з мого боку (перевірено вразливе місце на хибно-позитивні значенння):

- Critical класу – 13 вразливостей в конфігурації;
- High – 56 вразливостей в конфігурації;
- Medium - 68 вразливостей в конфігурації;
- Low – 0.

Після цього була проведена мануальна перевірка, що дала змогу оцінити, які з 20 сценаріїв атак можуть бути знайдені лише за допомогою використання інструменту. Дана перевірка показала, що 14 з 20 сценаріїв, які були попередньо вказані в документації Kubernetes-goat, можна попередити за допомогою налаштування коректрих безпекових конфігурацій. Час роботи розробленого інструменту був мінімальним – 36с, а додаткова валідація і мануальна перевірка заняли лише 3 години.

Висновки до розділу 4

Отже, у даному розділі було описана розробка програмного забезпечення для проведення пасивного аудиту безпеки Kubernetes, на основі попередніх досліджень з моделювання дерев атак та загроз, вибору методу сканування та обрання використаних технологій\застосунків. За тестовий застосунок було обрано Kubernetes-goat, що є загальнодоступним та безкоштовним, а мовою для розробки було обрано Go, Python, що чудово підійшли для розроблення. У результаті було виявлено певну кількість загроз в конфігураціях, провалідовано та проаналізовано, що 14 з 20 сценаріїв покриває сканування даним застосунком, а отже задовольняє поставлені перед ним задачі.

5 МАРКЕТИНГОВИЙ АНАЛІЗ СТАРТАП-ПРОЕКТУ

За данними дослідницьких організацій більшість стартапі провалюються через неправильне визначення оцінки ринку з самого початку, а іншими проблемами провалу є брак інвестицій або ж капіталу. Тому першочергові дослідження дозволяють проаналізувати потреби майбутньої аудиторії, рішення конкурентів та іх недоліки, а також визначити можливу прибутковість проекту.

Загалом виділяють чотири головні етапи аналізу стартапу:

1. Маркетинговий аналіз стартапу

На цьому етапі необхідно описати ідею стартапу, його мету та те, як його можна використовувати. Також необхідно здійснити огляд вже існуючих рішень і визначити їх переваги та недоліки. Відкриваючи новий бізнес, обов’язково потрібно знати про будь-які компанії, які можуть конкурувати. Незалежно від того, чи це прямий чи непрямий конкурент, знання того, як їхній бізнес може загрожувати вашому, допоможе краще просувати власний продукт і залишатися конкурентоспроможним серед спільної аудиторії. Під час маркетингового аналізу значну увагу приділяють аналізу ринку та методам виведення товару на ринок.

2. Організація стартапу

Цей етап передбачає проведення аналізу щодо необхідних ресурсів як матеріальних, так і нематеріальних. Проводяться розрахунки необхідних витрат, що будуть на початковому етапі. Перш ніж почнеться розробка стартапу, потрібно 60 буде оплатити рахунки. Розуміння витрат допомагає успішно запустити розробку. Розрахунок стартових витрат допомагає оцінити прибуток, зробити аналіз беззбитковості, брати безпечні позики, залучити інвесторів. Інвестори та позикодавці порівнюють очікувані витрати з прогнозованим доходом та визначають потенціал бізнесу для отримання прибутку. Також складають приблизний календарний план робіт.

3. Фінансово-економічний аналіз проекту та оцінка ризиків

Під час фінансово-економічного аналізу проекту проводять розрахунки основних економічних показників, визначають інвестиційну привабливість проекту. Під час цього етапу також визначаються потенційні ризики та шляхи їх попередження. Це може включати такі речі, як нові конкуренти, зміни в законодавстві, фінансові ризики та практично все інше, що потенційно може поставити під загрозу майбутнє проекту.

4. Комерціалізація проекту

Даний етап передбачає побудову стратегії щодо пошуку інвесторів для стартапу та визначає цільову групу інвесторів. Швидкий пошук інвесторів не є простим завданням. Більшість венчурних капіталістів отримують десятки запитів щодня і просто не встигають зустрітися з усіма. Тому цей процес є дуже важливим та потребує попередньої підготовки.

5.1 Опис ідеї проекту

Мета роботи полягає у тому, щоб розробити систему, що базується на основі моделей загроз та дерев атак та вирішує проблему нестачі автоматизації для сканування налаштувань та політик безпеки Kuberntetes. На основі даного рішення було створено веб інтерфейс для звіту для зручного використання кінцевим користувачем. Сформуємо зміст ідеї та розглянемо детально основні напрямки застосування та вигоду для користувачів (табл. 5.1). Розглянемо застосунки, що вже створені для сканування Kubenetes. Експерти стверджують, що аналіз ринку життєво важливий на самому ранньому етапі створення стартапу. Вузьке розуміння ділового середовища або відсутність даних про конкурентні переваги є одними з найпоширеніших помилок у презентаціях для інвесторів. Отже, якщо для того, щоб залучити інвестора, найбільше уваги слід приділити вивченню ринку під час створення бізнес-плану. Отримані дані повинні бути нещодавніми, інформативними та точними. Проведемо ретельний аналіз продуктів конкурентів та визначимо сильні і слабкі сторони (табл. 5.2).

Таблиця 5.1 – Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигода для користувача
Розробка системи автоматичного аудиту безпеки Kubernetes	- Використання DevSecOps інженерами для оцінки захищеності застосунку; - Використання програмістами, що хочуть налаштувати коректно кластери та образи	- Зменшення часових витрат на встановлення факту використання небезпечних конфігурацій; - Отримання звіту в комфортній формі; - Отримання одночасно з описом прогалини рекомендацій щодо її усунення.

Таблиця 5.2 - Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№ п/п	Техніко-економічні характеристики ідеї	(потенційні) товари/концепції конкурентів		W (слабка сторона)	N(нейтральна сторона)	S(сильна сторона)
		Мій проект	Конкурент			
1.	Сканування Kubernetes	+	+	Не покриття інших сценаріїв атаки, таких як DDOS.	Виявлення прогалин у політиках та конфігураціях.	Вибір найсучасніших методик сканування.
2.	Швидкодія пропонованого методу	+	-	-	-	Виконання мінімальних дій користувача, запуск лише 1

						програмного файлу
3.	Веб інтерфейс	+	-	-	-	Врахований зручний веб-інтерфейс для звіту.
4.	Охоплення сканування м	+	-	-	Охоплює більшість конфігураційних файлів, які можна знайти в проекті.	-

5.2 Технологічний аудит ідеї проекту

Розглянемо та проаналізуємо технології, що необхідні для реалізації ідеї стартапу. Зокрема визначимо наявність та доступність технологій (табл. 5.3)

Таблиця 5.3 – Технологічні можливості здійснення ідеї

Ідея	Наявність технологій	Технології та реалізації	Доступність технологій
Веб-інтерфейс	+	JavaScript, NodeJS, Express, NestJS, Angular	+
Обробка даних	Необхідна попередня розробка	Необхідно обробляти дані перед аналізом	Необхідна розробка
Тестові застосуни	+	Kubernetes-bench, Kuber-bench, simple-Kuber	+

5.3 Аналіз ринкових можливостей запуску стартап-проекту

При першому дослідженні ринку важливо з’ясувати, які компанії працюють у вашій ніші. І не менш важливо зрозуміти, як ці компанії потрапили до списку найкращих гравців галузі та яких маркетингових стратегій вони дотримуються. Також потрібно дослідити характеристики та уподобання цільової аудиторії, регіональні можливості, пропозиції конкурентів та загальний динамічний розвиток ринку. Здається очевидним досліджувати компанії, які тільки починають, що працюють з подібним бюджетом, і не можуть похвалитися надзвичайною впізнаваністю бренду. Засновники сподіваються, що це допоможе їм перевершити найближчих суперників у тактиці і, в якийсь момент, це справді так. Однак така тактика не сприяє довгостроковій стратегії. Для стійкого зростання необхідний інший підхід. Стартапам може бути вигідніше досліджувати нові горизонти, аналізуючи найкращі практики успішного бізнесу. Проведемо аналіз впровадження та виведення даного стартап-проекту на ринок (табл. 5.4)

Таблиця 5.4 – Характеристика потенційного ринку для проекту

№ п/п	Показники стану ринку	Характеристика
1 .	Кількість основних гравців	Немає
2.	Загальний обсяг продаж грн/ум.од	100000
3.	Динаміка ринку	Зростає
4.	Наявність обмежень для виходу	Немає
5.	Специфічні умови для сертифікацій чи дотримання стандартів	Немає
6.	Середня норма рентабельності на ринку, %	>= 70%

З результатів дослідження випливає що ринок відкритий та зараз зростає, не було знайдено продуктизованих конкурентів, через що ринок можна вважати інвестиційно привабливим.

Перед початком запуску стартапу необхідно встановити, хто є потенційними клієнти та як їх залучити. Цільовий ринок – це не всі, хто знаходиться у районі вашого бізнесу. Він складається лише з людей, які добре підходять для запропонованого продукту. Потрібно точно визначити, хто ваші клієнти і де їх 64 знайти. Ця розбивка називається сегментацією цільового ринку. визначити цільову аудиторію стартап-проекту (табл. 5.5).

Таблиця 5.5 – Потенційні клієнти продукту

№ п/п	Потреба, яка формує ринок	Цільова аудиторія	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
1.	Відсутність програмних засобів для пасивного сканування налаштувань та політик безпеки Kubernetes з генерацією звіту	1. Підприємства які займаються захистом від кібератак. 2. Підприємства які зазнали атак на Kubernetes. 3. Підприємства, що використовують досліджувані технології або розробляють ПЗ.	Перша група займається поширенням послуг кібербезпеки і хочуть отримати грошову вигоду. Друга хоче уникнути можливих втрат від повторення атак. Третя група це люди які хочуть вберегти себе від можливих атак на розроблене ПЗ.	Зручний інтерфейс, мінімальне втручання особи в процес сканування, зібраний воєдино звіт.

Можливості – це, як правило, зовнішні фактори організації, які мають велику ймовірність досягнення успіху. Можливості завжди супроводжуються загрозами . Це перешкоджаючі фактори, що неможливо контролювати. Єдиний спосіб протистояти

цьому – розробити надійний план дій на випадок непередбачених ситуацій, щоб ефективно боротися з ними, коли вони трапляються. Можна скористатися можливостями та захиститися від загроз, але не можливо їх змінити. Зробимо аналіз ринкового середовища, а саме визначимо фактори основних загроз, що можуть вплинути на продаж розробленої системи (табл. 5.6). Також визначимо фактори можливостей (табл. 5.7), які можуть вплинути на розроблений продукт під час виходу його на ринок.

Таблиця 5.6 – Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція
1.	Недостатнє фінансування	Відсутність коштів	Пошук нових інвесторів, розширення клієнтської бази
2.	Відсутність попиту	Використання звичних мануальних технік виявлення шкідливи конфігурацій та політик безпеки.	Рекламна компанія про важливість автоматичного сканування та мінімізацію втручання людини в даний процес, а також про важливість економії часу інженера з кібербезпеки.

Таблиця 5.7 – Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція
1.	Відсутність альтернатив	В Україні немає аналогів такого програмного продукту	Можливість стати монополістом в даній галузі в Україні

2.	Монетизація додатку	Впровадження контекстної та таргетованої реклами у додаток або ж обмеження кількості сканувань	Отримання додаткового прибутку

Проведемо ступеневий аналіз конкуренції на ринку (табл. 5.8). Даний аналіз допомагає проаналізувати особливості конкурентного середовища. На базі цього можно побудувати основну тактику залучення користувачів.

Таблиця 5.8 – Ступеневий аналіз ринкової конкуренції

Особливість конкурентного середовища	У чому саме проявляється особливість	Можливі дії компані щоб залишатися конкурентноспроможними
1. Вказати тип конкуренції - олігополія	Відсутні конкуренти на ринку в Україні.	Можливість завоювати частку ринку за рахунок вузької спеціалізації.
2. За рівнем конкурентної боротьби -міжнаціональна	Надавання послуг на території України і в інших країнах.	Ці програми не мають поширення в Україні; вдосконалення методик сканування, розширення бази сканування, збільшення функціоналу додатку.
3. За галуззю - міжгалузева	Можливість особистого використання та в компаніях різного напрямку діяльності.	Пошук нових клієнтів серед населення та промислових підприємств.
4. За видами товарів - товарно-видова	Товари одного виду.	Розробити якісне ПЗ та створити активну рекламну кампанію.

5. За характером конкурентних переваг - нецінова	Програмне забезпечення буде більш зручним, а також буде характеризуватися мінімізацією часу інженера.	Необхідно досягти повного обсягу перевірок образів та конфігурацій, за рахунок розширення функціоналу.
6. За інтенсивностю -не марочна	Фірма ще не набула достатньої популярності.	Створення дійсно якісного продукту для завоювання ринку.

Аналіз конкурентів за Портером – найкорисніший інструмент для власників та менеджерів, щоб бути на крок попереду конкурентів на складних ринках (табл. 5.9). Це аналітична модель, яка допомагає маркетологам та керівникам підприємств подивитися на “баланс сил” на ринку між різними організаціями на глобальному рівні та проаналізувати привабливість та потенційну прибутковість галузевого сектору.

Таблиця 5.9 - Аналіз галузевої конкуренції за М. Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти в галузі	Постачальники	Клієнти	Товари замінники
	Немає	Патенти на програмні продукти	IT компанії, які займаються кібербезпекою та розробкою	Компанії та особи які використовують Kubernetes	Відсутні
Висновки	Не інтенсивна боротьба за ринок	Можливість виходу на ринок висока, за відсутності прямих конкурентів	Не диктують умови роботи ринку	Зацікавлені в зменшенні часового та фізичного навантаження інженерів з безпеки	Обмежень на ринку через товари-замінники немає

Проаналізуємо та розглянемо фактори конкурентоспроможності (табл. 5.10). Наведемо чинники, що роблять даний фактор значущим для порівняння конкурентів та порівняємо з конкурентами (табл. 5.11).

Таблиця 5.10 - Обґрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтування
1.	Якість продукту	Якісний продукт може швидше зайти на ринок та закріпитися на ньому.
2.	Наявність прямих конкуретів	Аналогічних продуктів з спеціалізацією на Kubernetes немає в вільному продажі.
3.	Легкість у користуванні	Інтерфейс повинен бути інтуїтивно зрозумілим для того, щоб закріпитися на ринку.
4.	Масштабованість продукту	Продукт потребує масштабованості.
5.	Задоволення потреб клієнтів	Клієнти не мають широкого вибору застосунків пасивного аудиту заданих технологій.

Таблиця 5.11 – Порівняльний аналіз сторін проекту

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з запропонованим товаром						
			–3	–2	–1	0	+1	+2	+3
1	Ціна	15				+			
2	Легкість у користуванні	18			+				
3	Задоволення потреб клієнтів	18		+					
4	Висока точність	20			+				
5	Висока якість продукту	18				+			

Аналіз SWOT (сильних, слабких сторін, можливостей та загроз) – це основа, яка використовується для оцінки конкурентних позицій компанії та розробки стратегічного планування. SWOT-аналіз оцінює внутрішні та зовнішні фактори, а також поточний та майбутній потенціал. SWOT-аналіз допомагає максимально використати сильні сторони, з найкращою перевагою для організації (табл. 5.12). Він також може зменшити шанси на невдачу, так як виявляє те, чого не вистачає стартапу, і передбачає небезпеки, які в іншому випадку застали б зненацька. На основі проведеного SWOT-аналізу розроблено альтернативи ринкової поведінки (перелік можливих заходів) для виведення проекту на ринок та орієнтовний оптимальний час їх ринкової реалізації, дані зібрано у Таблиці 5.13.

Таблиця 5.12 – Порівняльний аналіз сторін проекту

Сильні сторони: – Собівартість; – Зручність і простота користування; – Відсутність прямих конкурентів на ринку; – Висока точність результатів; –Застосування у компаніях із різних галузей; – Висока якість продукту.	Слабкі сторони: -Залежність від маркетингу, -Надто вузька спеціалізація.
Можливості: – Відсутність аналогів; – Монетизація додатку; – Можливості виходу на міжнародний ринок;	Загрози: – Відсутність попиту; – Недостатнє фінансування;

Таблиця 5.13 – Альтернативи для виходу на ринок

№ п/п	Альтернатива – орієнтовний перелік заходів поведінки	Ймовірність отримання ресурсів	Строки реалізації
1.	Укладання угоди з ІТ компанією	Висока	до 10 місяців
2.	Ефективна реклама, що зацікавить клієнтів	Середня	до 3 місяців
3.	Презентування товару на виставках, конференціях	Середня	до 2 місяців
4.	Додання нового функціоналу	Мала	до 8 місяців

Обрано альтернативу “Укладання угоди з ІТ компанією”, оскільки за строками реалізації та ймовірністю отримання ресурсів вона є найбільш оптимізованою.

5.4 Розроблення ринкової стратегії проекту

Наявність ринкової стратегії є ключем до успіху будь-якого бізнесу. Маркетинг не є самостійною, одноразовою діяльністю. Він складається з кількох різних компонентів, необхідних на кожному етапі діяльності компанії – від задовго до того, як продаж навіть почнеться, до довгого після. Хоча ринкова стратегія може вимагати регулярних коригувань або доопрацювань, вона надає шаблон, з чого почати, і полегшує перегляд подібних або покращених результатів кожної кампанії без необхідності повністю починати з початку. Визначення стратегії охоплення ринку передує саме розробленню ринкової стратегії стартап-проекту. Оберемо цільові групи

потенційних споживачів та проведемо їх аналіз (табл. 5.14), а потім визначимо базову стратегію розвитку (табл. 5.15).

Таблиця 5.14 – Опис цільових груп клієнтів

№ п / п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах груп	Інтенсив-ність конкуренції в секторі	Простота входу
1.	ІТ компанії які займаються кібербезпекою	Готові	Високий	Низька/немає	Просто
2.	Фізичні особи та компанії які використовують Kubernetes	Готові	Середній	Низька/немає	Просто

Таблиця 5.15 - Визначення базової стратегії розвитку

Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
Зосередження на одному сегменті	Стратегія концентрованого маркетингу	Великі перспективи та визначні якості наявного рішення	Стратегія спеціалізацій

Після цього необхідно визначити стратегію конкурентної поведінки (табл. 5.16), а також стратегію позиціонування (табл. 5.17).

Таблиця 5.16 - Визначення базової стратегії конкурентної поведінки

Чи є проект «першопроходьцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Стратегія конкурентної поведінки
Ні	Компанія буде шукати нових споживачів	Так , функціональні можливості	Стратегія лідера

Таблиця 5.17 – Визначення стратегії позиціонування

Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових)
Зручність у використанні, висока точність результатів, швидкодія	Стратегія спеціалізації	Виявлення загроз без ризиків пошкодження проекту, зручний інтерфейс веб-додатку	Якість, простота використання, вітчизняний продукт. Швидкість. Забезпечення аудиту безпеки Kubernetes.

5.5 Розроблення маркетингової програми стартап-проекту

Маркетингова стратегія допомагає створювати товари та послуги з найбільшими шансами на отримання прибутку. Це тому, що маркетингова стратегія починається з дослідження ринку, беручи до уваги оптимального цільового клієнта, те, що роблять конкуренти та які тенденції можуть бути на горизонті. Використовуючи цю інформацію, необхідно визначити як можна відрізнити свій

товар чи послугу від конкурентів. Визначимо ключові переваги товару (табл. 5.18) та опишемо три рівня моделі товару (табл. 5.19).

Таблиця 5.18 – Визначення ключових переваг концепції потенційного товару

№	Потреби	Вигоди які пропонує товар	Ключові переваги над конкуурентами
1.	Виявлення прогалин безпеки Kubenetes	Точність рещультатів, покриття більшості ризиків	Баланс точності та швидкості сканування через мінімізацію втручання користувача
2.	Генерація звіту	Зібрання основних відомостей в зручному форматі	Швидкодія, надані рекомендації, зручність використання

Таблиця 5.19 - Опис 3 рівнів моделі товару

<table>
<tr><td>Рівні</td><td colspan="3">Сутність та складові</td></tr>
<tr><td>Товар за задумом</td><td colspan="3">Система статичного аналізу конфігурацій та політик безпеки Kubernetes з функціоналом генерування звіту в HTML-форматі</td></tr>
<tr><td rowspan="5">Товар у реальному виконанні</td><td>Властивості</td><td>М/Нм</td><td>Вр/Тх /Тл/Е/Ор</td></tr>
<tr><td>Отримання даних конфігурацій</td><td>М</td><td>Тх, Тл, Е</td></tr>
<tr><td>Підготовка даних для виявлення небезпечних конфігурацій та політик</td><td>М</td><td>Тх</td></tr>
<tr><td>Виявлення небезпечних налаштувань шляхоом порівняння їх зі стантадртом</td><td>М</td><td>Тх, Е</td></tr>
<tr><td colspan="3">Якість: вимоги до створення застосунку продиктовані користувачами</td></tr>
<tr><td></td><td colspan="3">До продажу: початок рекламної кампанії, підписання контрактів</td></tr>
</table>

Товар із підкріпленням	Після продажу: продовження рекламної компанії
За рахунок чого потенційний товар буде захищено від копіювання: патент, захист інтелектуальної власності	

Таблиця 5.20 - Визначення меж встановлення ціни

Рівень цін на товари - замінники	Рівень цін на товари - аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі
Розробка додатку – мінімум £5000	-	Високий	2000-16000 у.о.

Таблиця 5.21 - Формування системи збуту

Специфіка закупівельної поведінки цільових груп	Функції збуту, які має виконувати постачальник	Глибина каналів збуту	Оптимальнна система збуту
Власна система збуту	Усні та письмові консультації	Продаж користувачем, нульовий рівень	Прямий збут без посередників

Таблиця 5.22 - Концепція маркетингових комунікацій

Специфіка поведінки цільових клієнтів	Канали комінукацій якими користуються клієнти	Ключові пропозиції для позиціонування	Завдання рекламного повідомлення
Консервативна поведінка	Корпоративна пошта та телефонні дзвінки	Гнучке налаштування, спеціалізація на Kubernetes, післяпродажна підтримка	Висвітлення унікальних характеристик послуги

Висновки до розділу 5

У даному розділі проведено маркетинговий аналіз стартап-проекту. Зокрема проведено SWOT-аналіз, як результат визначено сильні та слабкі сторони, можливості та загрози проекту. Проаналізовані ринкові можливості запуску стартапу, за допомогою аналізу потенційного ринку встановлено, що ринок є привабливим. Це зумовлено відсутністю конкурентів, динамікою ринку, що зростає та високою рентабельністю. Потенційними клієнтами стартапу є звичайні користувачі, банківські установи та страхові компанії. Спільною потребою усіх груп користувачів є мінімізація втручання інженера безпеки в процес перевірки. Проведено ступеневий аналіз конкуренції на ринку, визначені особливості конкурентів. Також проведено аналіз конкуренції за Портером та розглянуто фактори конкурентоспроможності. На базі цього побудовано основну тактику залучення користувачів. На початковому етапі вирішено обрати як цільову групу людей, що хочуть купити дану систему. У подальшому можливе розширення функціоналу додатку, а також розширення ринку інтеграції. Визначено базову стратегію розвитку, стратегію конкурентної поведінки та стратегію позиціонування. Стратегією охоплення ринку обрано стратегію концентрованого маркетингу, а альтернативою розвитку проекту – зосередження на одному сегменті, адже на початковому етапі відсутня велика кількість ресурсів. Також визначені можливі способи монетизації додатку та встановлено ціну на товар. Розроблено концепцію маркетингових комунікацій. Маркетинговий аналіз стартапу встановив, що розробка та впровадження проекту є доцільною.

ВИСНОВКИ

У даній магістерській дисертації було розглянуто проблеми безпеки та проведено дослідження щодо методів та засобів забезпечення аудиту безпеки Kubernetes, зокрема використання графічних та семантичних методів зображення вразливостей.

Проведено аналіз архітектури Kubernetes та Docker, для подальшого визначення векторів загроз, груп загроз та побудови моделі загроз. Також було досліджено наявні системи, визначені їх недоліки та перваги. Головими недоліками існуючих рішень була відсутність комфортного графічного зображення отриманого звіту, щодо знайдених вразливостей, активні методи сканування, які не завжди є прийнятними в тому чи іншому застосунках, а також мале охоплення конфігураційних файлів. За результатами проведеного аналізу було визначено задачі, що стоять перед магістерською дисертацією, а також критерії, яким має відповідати розроблений інструмент.

У ході пркптичного дослідження було проаналізовано основні вектори атак та можливі загрози на контейнери та на власне Kubernetes. Після побудови дерев рішень та проведення дослідження щодо мінімізації їх гілок було виявлено, що правильне налаштування політик безпеки та конфігурацій Kubernetes, допомагає запобігти більшості з атак. Було проведено експеримент серед експертів щодо вибору методу сканування, який показав, що комбінація мануального та автоматичного сканування дозволяє найефективніше знаходити загрози за короткий проміжок часу. Тому було розроблене програмне забезпечення, яке мінімізувало роботу інженера з безпеки, для уникнення помилок людського фактору, та яке проводило пасивний аудит безпеки досліджуваних технологій за короткий проміжок часу, а потім провалідовано мануально. Наявне рішення також має функціонал генерування звіту, який у містить в собі всі інформацію про всі програлини у безпеці застосунку, а також практичні рекомендації, щодо їх усунення. Тому усі вимоги, що були поставлені перед

інструментом були виконані. У результаті аналізу тестового застосунку Kubernetes-goat було отримано, що усі 14 з 20 сценаріїв атак можна попередити за допомогою початкового коректного налаштування конфігурацій та політик безпеки.

Також наприкінці було проведено маркетинговий аналіз стартап-проекту, який показав, що потенційними клієнтами стартапу є IT компанії, що займаються кібербезпекою, або компаніх, що використовують дані технології у своїх цілях. Їхніми потребами виявились вимоги щодо якості інструменту, широти покриття та зручність у використанні. А ще було проведено SWOT-аналіз, як результат визначено сильні та слабкі сторони, можливості та загрози проекту. У висновку маркетинговий аналіз стартапу встановив, що розробка та впровадження застосунку є доцільною.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Microsoft. Developer tools, technical documentation and coding examples | Microsoft Docs. [Веб-сайт]. URL: https://docs.microsoft.com/ru-ru/virtualization/windowscontainers/about/containers-vs-vm\ (дата звернення: 09.11.2021).
2. Райс Л. Безопасность контейнеров: фундаментальный подход к защите контейнеризированных приложений : підручник. Пітер : Пітер, 2021. 224 с.
3. Docker Hub. *Docker Hub*. [Веб-сайт]. URL: https://hub.docker.com/ (date of access: 09.11.2021).
4. Маркелов А.А. Введение в технологии контейнеров и Kubernetes. – М.: ДМК Пресс, 2019. – 194 с.
5. Cloud Container Adoption In The Enterprise. How Tech Leaders Are Overcoming Barriers To Container Adoption At Scale. *FORRESTER*. 2020. P. 1–6.
6. Марта Родрігез. Container-based cluster orchestration systems: A taxonomy and future directions // Wiley. 2019.№ 5. C. 1-10
7. Лукша М. Kubernetes in Action. USA : MANNING SHELTER ISLAND, 2019. 673 p
8. Cloud Foundry. URL: https://www.cloudfoundry.org/ (дата звернення: 02.11.2021).
9. Container Orchestration | Enterprise-Ready Kubernetes by Kublr. *Kublr*. URL: https://kublr.com/ (date of access: 09.11.2021).
10. Лукша М. Kubernetes in Action. USA : MANNING SHELTER ISLAND, 2019. 673 p.
11. Capital One: [Веб-сайт]. URL: https://www.capitalone.com/tech/cloud-container-adoption-report/?gclid=EAIaIQobChMI1qCZ0O_86gIVD_DACh2slgiCEAAYASAAEgK31_D_BwE (дата звернення: 10.11.2021).
12. Сайфан Дж. Осваиваем Kubernetes. Оркестрация контейнерных архитектур. 2-ге вид. Москва : Питер, 2019. 400 с.
13. Райс Л., Хаузенблас М. Kubernetes Security Operating Kubernetes Clusters and Applications Safely. USA : O`Reilly, 2018. 85 p. 1. Маркелов А. А. Введение в технологии контейнеров и Kubernetes: книга. Москва: ДМК Пресс, 2019. 194 с.

14. What You Need to Know About the RunC Container Escape Vulnerability - The New Stack. *The New Stack*. [Веб-сайт]. URL: https://oreil.ly/cFSaJ (дата звернення: 09.10.2021).
15. CIS Docker Benchmark, Center for Internet Security (CIS). [Веб-сайт]. URL: https://www.cisecurity.org/benchmark/docker/.
16. Security Enhanced Linux (SELinux), [Веб-сайт]. URL: https://selinuxproject.org/page/Main_Page
17. AppArmor, [Веб-сайт]. URL: http://wiki.apparmor.net/index.php/Main_Page
18. NIST Special Publication (SP) 800-123, Guide to General Server Security, National Institute of Standards and Technology, Gaithersburg, Maryland, July 2008, 53pp. https://doi.org/10.6028/NIST.SP.800-123
19. NIST Special Publication (SP) 800-164, Guidelines on Hardware-Rooted Security in Mobile Devices (Draft), National Institute of Standards and Technology, Gaithersburg, Maryland, October 2012, 33pp. https://csrc.nist.gov/publications/detail/sp/800-164/draft.
20. NIST Special Publication (SP) 800-147, BIOS Protection Guidelines, National Institute of Standards and Technology, Gaithersburg, Maryland, April 2011, 26pp. https://doi.org/10.6028/NIST.SP.800-147.
21. Unsecured Kubernetes Instances. *Palto Network*. [Веб-сайт]. URL: https://unit42.paloaltonetworks.com/unsecured-kubernetes-instances/ (дата звернення: 02.11.2021).
22. NCC Group Newsroom. Mynewsdesk. [Веб-сайт]. URL: https://www.nccgroup.com/uk/about-us/newsroom-and-events/blogs/2017/november/kubernetes-security-consider-your-threat-model/ (дата звернення: 08.11.2021).
23. K8S Threat Model - CloudSecDocs. *Welcome to CloudSecDocs - CloudSecDocs*. [Веб-сайт]. URL: https://cloudsecdocs.com/container_security/theory/threats/k8s_threat_model/ (дата звернення: 03.11.2021).

24. The State of Kubernetes Security 2021: [Веб-сайт]. URL: https://security.stackrox.com/state-of-containers-and-kubernetes-security-report-winter-2020.html?Source=Website&LSource=Website (дата звернення: 10.10.2021). 2.
25. OWASP: [Веб-сайт]. URL: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology (дата звернення: 10.10.2021).
26. Вихорів С. В. Классификация угроз информационной безопасности // Сетевые атаки и системы информационной безопасности. Москва. С. 6. 2001 р.
27. Kube-hunter by Aqua. *Kube-hunter by Aqua*. URL: https://kube-hunter.aquasec.com/ (дата звернення: 15.11.2021).
28. Checkov 2.0: Context-Aware Security Scanning for Infrastructure as Code - The New Stack. *The New Stack*. URL: https://thenewstack.io/checkov-2-0-context-aware-security-scanning-for-infrastructure-as-code/ (дата звернення: 15.11.2021).
29. Kubernetes Goat is a "Vulnerable by Design" Kubernetes Cluster. Designed to be an intentionally vulnerable cluster environment to learn and practice Kubernetes security. *GitHub*. [Веб-сайт]. URL: https://github.com/madhuakula/kubernetes-goat (дата звернення: 04.10.2021).

ДОДАТОК А ДЕРЕВО РІШЕНЬ «ЗДОБУТТЯ КОНФІДЕНЦІЙНИХ ДАНИХ»