КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

ФАКУЛЬТЕТ РАДІОФІЗИКИ, ЕЛЕКТРОНІКИ ТА КОМП'ЮТЕРНИХ СИСТЕМ

Кафедра комп'ютерної інженерії

До захисту допущено: «На правах рукопису»
Завідувач кафедри ________ Юрій Бойко
« _ » _________ 2023 р.

КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

на тему:

«ДОСЛІДЖЕННЯ ВЕКТОРІВ АТАК НА KUBERNETES ТА ПРОТИДІЯ ЇМ RESEARCHING ATTACK VECTORS ON KUBERNETES AND PREVENTING THEM»

Виконав:
студент 2-го курсу магістратури
денної форми навчання
спеціальності 123 Комп'ютерна інженерія
ОНП «__________________»
Валентин Щербань ________________

Науковий керівник:
кандидат технічних наук, асистент
Слюсар Євген Андрійович ________________

Рецензент:
____

Засвідчую, що у цій магістерській роботі
немає запозичень з праць інших авторів без
відповідних посилань
Студент _____________

Робота допущена до захисту в ЕК рішенням кафедри ___________________________________
від «__»_________ 2023 р., протокол № __.

Завідувач кафедри _____________,
кандидат фізико-математичних наук, доцент
Бойко Юрій Володимирович

(підпис)

Київ – 2023

РЕФЕРАТ

Дипломна робота магістра за об’ємом складає 53 сторінки, містить 9 рисунків, 4 додатки, використано 11 інформаційних джерел.

Об’єктом роботи є система система контейнерної оркестрації Kubernetes. **Метою роботи** є розробка рішення для забезпечення від інформаційної безпеки кластера Kubernetes.

Результати роботи: проведено аналіз архітектури системи Kubernetes та її основних об’єктів, проаналізовано можливі вектори атак на вказану систему і існуючі засоби для їх протидії, сформовано вимоги до оператора спостереження за станом безпеки кластера Kubernetes та проведено його розробку.

Ключові слова: засоби контейнерного оркестрування, кібератака, контейнеризація, API, под, оператор.

ЗМІСТ

РЕФЕРАТ 1
ЗМІСТ 2
СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ 4
ВСТУП 5
1. ТЕХНОЛОГІЯ КОНТЕЙНЕРИЗАЦІЇ. СИСТЕМА KUBERNETES 7
1.1 Технологія контейнеризації 7
1.2 Система Kubernetes 10
1.3 Архітектура кластера Kubernetes 10
1.4 Об’єкти кластера Kubernetes 12
2. АНАЛІЗ ВЕКТОРІВ АТАК НА KUBERNETES ТА ПРОТИДІЯ ЇМ 14
2.1 Вектори атак на кластер Kubernetes 14
2.2 Аналіз існуючих рішень забезпечення інформаційної безпеки кластера Kubernetes 20
2.2.1 Порівняння безпекових доповнень кластеру Kubernetes 22
2.3 Постановка задачі 23
3. СТВОРЕННЯ ОПЕРАТОРА ДЛЯ АНАЛІЗУ СТАНУ БЕЗПЕКИ КЛАСТЕРУ KUBERNETES 25
3.1 Оператор Kubernetes 25
3.2 Задання правил політик безпеки 28
3.2.1 Директива resources 28
3.2.2 Директива securityContext 30
3.3 Розгортання оператору в кластері Kubernetes 35

ВИСНОВКИ 38

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 39

ДОДАТКИ 41

Додаток А 41

Додаток Б 43

Додаток В 50

Додаток Г 52

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

Кібератака – це навмисна спроба здійснення шкідливих дій, направлених на працездатність комп'ютерних системи або безпеку даних, що в них наявні.

Кластер – набір комп'ютерів, що працюють разом і можуть розглядатися як єдина система.

API (абр. Application Programming Interface) – програмний інтерфейс застосунку.

CI/CD (абр. Continuous Integration/Continuous Deployment) - неперервна інтеграція/розгортання.

CRD (абр. Custom Resource Definition) - визначення користувацького ресурсу.

CNCF - Cloud Native Computing Foundation

RBAC - role-based access control.

k8s - Kubernetes.

ВСТУП

Kubernetes - система оркестрування контейнерів, що була створена компанією Google у 2014 році та передана організації Cloud Native Computing Foundation у 2015. З тих пір дана система набула широкого поширення та згідно з опитуванням 2021 року 96% опитаних організацій використовують або розглядають Kubernetes для розгортання своїх сервісів [1] порівняно з 83% у 2020 та 78% у 2019.

Також очевидним є те, що використання контейнерів для розгортання додатків стало стандартом для IT-індустрії з ростом частоти їх використання на 300% з 2016 року [2].

Використання технологій контейнеризації при розробці програмного забезпечення та його розгортання у робочому середовищі надає компаніям ряд переваг, серед яких можна виділити наступні:

- Короткі цикли нових релізів;
- Швидші виправлення програмних помилок;
- Висока гнучкість при розгортанні сервісів у гібридних середовищах (наприклад, при поєднанні сервісів хмарного провайдера та власних дата-центрів компанії).

Проте якщо питання інформаційної безпеки не є частиною розробки, то врешті-решт воно може нівелювати наявні переваги контейнеризації. Тому забезпечення безпеки є одним із найбільш гострих викликів при впровадженні контейнеризації для використання у робочому середовищі. За даними опитувань [3] 55% компаній зіштовхувались з сповільненням або перенесенням нових релізів у зв’язку з проблемами забезпечення безпеки контейнерів та, зокрема, Kubernetes у 2021 році. Також 93% опитаних компаній мали негативний досвід пов’язаний із безпековими інцидентами в контейнерних середовищах.

Таким чином недостатні знання про контейнери та Kubernetes, недостатнє використання інструментів аудиту інформаційної безпеки і неправильна

організація команд IT-безпеки, при якій аудит безпеки не встигає за швидкістю роботи команд розробки програмного забезпечення, призводять до того, що 31% респондентів повідомили, що вони зіштовхнулися з втратами прибутків або клієнтських даних із-за інцидентів безпеки.

До найбільш поширених причин виникнення інцидентів безпеки відносять помилки у конфігурації контейнерів і кластерів Kubernetes, а також несвоєчасне реагування на критичні вразливості.

У зв’язку з вищенаведеним постає питання у забезпеченні інформаційної безпеки при використання контейнеризації та, зокрема, Kubernetes. Саме дослідженню векторів можливих атак, а також протидію їм, присвячено дану роботу.

1. ТЕХНОЛОГІЯ КОНТЕЙНЕРИЗАЦІЇ. СИСТЕМА KUBERNETES

1.1 Технологія контейнеризації

Контейнеризація - це технологія, що дозволяє компонування програмного коду з усіма необхідними компонентами, такими як бібліотеки, фреймворки та інші залежності, таким чином, щоб вони були ізольовані у власному контейнері.

Контейнеризація є складовою частиною сучасної розробки програмного забезпечення та його розгортання, а також вона все частіше використовується для виконання хмарних обчислень, забезпечуючи гнучкість та масштабованість додатків.

Контейнери є схожими за призначенням до віртуальних машин, але на відміну від останніх вони ділять спільне ядро з хостовим комп’ютером, що дозволяє їм мати меншу швидкість запуску та вимкнення, а також краще керувати ресурсами комп’ютера.

Контейнеризація застосунків надає наступні переваги:

- Переносимість. Розробники програмного забезпечення використовують контейнеризацію для розгортання додатків у різних середовищах без переписування програмного коду. Вони створюють програму один раз і розгортають її на декількох операційних системах. Наприклад, вони запускають одні й ті ж контейнери в операційних системах Linux і Windows. Розробники також оновлюють застарілий код додатків до сучасних версій, використовуючи контейнери для розгортання.
- Масштабованість. Контейнери - це легкі програмні компоненти, які працюють ефективно. Наприклад, віртуальна машина може запустити контейнерний додаток швидше, оскільки їй не потрібно завантажувати операційну систему. Тому розробники програмного забезпечення можуть легко розгортати кілька контейнерів для різних додатків на

одній машині. Контейнерний кластер використовує обчислювальні ресурси однієї спільної операційної системи, але один контейнер не заважає роботі інших контейнерів.

- Відмовостійкість. Команди розробників програмного забезпечення використовують контейнери для створення відмовостійких додатків. Вони використовують кілька контейнерів для запуску мікросервісів у хмарі. Оскільки мікросервіси в контейнерах працюють в ізольованому користувацькому просторі, один несправний контейнер не впливає на роботу інших контейнерів. Це підвищує відмовостійкість і доступність програми.
- Гнучкість. Контейнерні програми працюють в ізольованих обчислювальних середовищах. Розробники програмного забезпечення можуть усувати несправності та змінювати код програми, не втручаючись в операційну систему, апаратне забезпечення або інші сервіси програми. Завдяки контейнерній моделі вони можуть скоротити цикл випуску програмного забезпечення та швидко працювати над оновленнями.

Контейнерний рушій - це програмний компонент відповідальний за створення та керування контейнерами на хостовій операційній системі шляхом взаємодії з ядром даної системи.

Контейнерний рушій надає API, який дозволяє розробникам створювати контейнери та керувати ними, вказуючи такі деталі, як образ, який буде використовуватися, мережеву конфігурацію контейнера та необхідні обсяги сховища. Контейнерний рушій відповідає за створення та керування цими контейнерами, гарантуючи, що вони мають необхідні ресурси та дозволи для запуску вказаних додатків.

Існує кілька популярних контейнерних рушіїв, зокрема Docker, Containerd та CRI-O. Docker є найвідомішим і найпоширенішим рушієм, що надає простий і

зручний інтерфейс для створення та керування контейнерами. Containerd і CRI-O забезпечують більш мінімалістичний і низько рівневий підхід до керування контейнерами.

Складовими можливостями ядра Linux, що є складовими компонентами для технології віртуалізації є наступні:

- Простори імен
- Контрольні групи (cgroups)
- UnionFS

Простори імен дозволяють створити певну обгортку над системними ресурсами (мережа, ідентифікатори процесів тощо). Завдяки цьому процес з певним простором імен може мати наче ізольовану частину системних ресурсів. Таким чином в декількох різних просторах імен можуть існувати ресурси з однаковими ідентифікаторами, хоча для хостової системи ці ідентифікатори будуть відрізнятися.

Контрольні групи дозволяють організовувати процеси в ієрархічній моделі та виділяти системні ресурси в даній ієрархії контрольовано. Таким чином для кожного контейнеру можна встановити ліміти використання системних ресурсів (кількість оперативної пам’яті, використання CPU).

Об’єднана файлова система (UnionFS) дозволяє директоріям та файлам окремих файлових систем (рівнів) накладатися одна на одну, створюючи нову віртуальну файлову систему.

Образи, що використовуються в сучасних контейнерних рушіях, складаються з декількох рівнів. При старті нового контейнера ці рівні об’єднуються, створюючи нову read-only файлову систему. На верхньому рівні цієї системи контейнер отримує read-write рівень визначений лише для даного контейнеру.

1.2 Система Kubernetes

Система Kubernetes - система оркестрування контейнерів із відкритим вихідним кодом, призначенням якої є автоматизація розгортання, масштабування та управління контейнеризованими додатками [4].

Kubernetes дозволяє розробникам легко розгортати та керувати контейнерними додатками, шляхом створення гнучкої, масштабованої та високодоступної інфраструктури.

Kubernetes використовує декларативний підхід управління інфраструктурою, що означає, що користувачі визначають бажаний стан свого додатку, а Kubernetes гарантує, що фактичний стан додатку відповідає бажаному.

1.3 Архітектура кластера Kubernetes

Kubernetes можна складається з одного чи декількох мастер-вузлів під назвою Control Plane та одного та більше робочих вузлів.

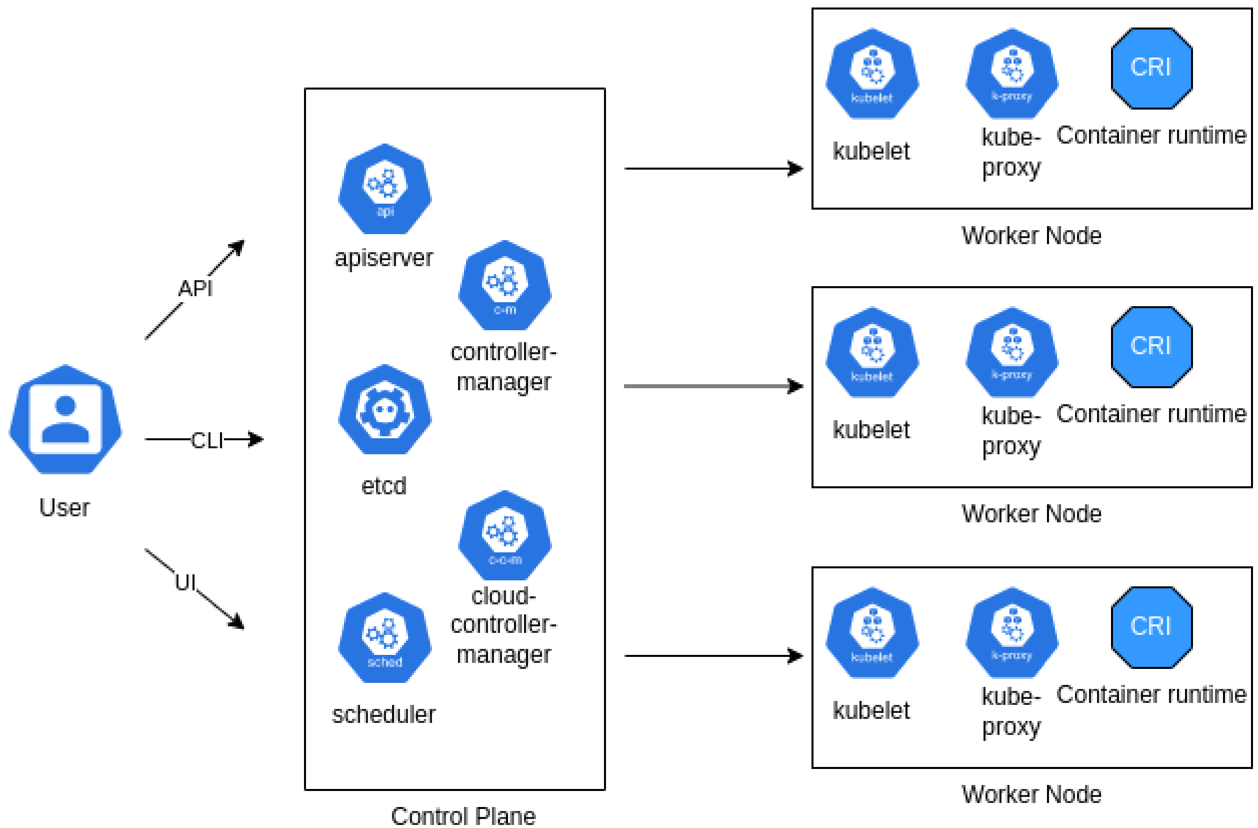

Рис. 2.1.1. Архітектура кластера Kubernetes

Control Plane у свою чергу складається з наступних компонентів [5]:

1. kube-apiserver - фронтенд-сервер, задачею якого є обробка запитів до API;
2. etcd - високо-доступне розподілене сховище типу ключ-значення, в якому зберігається вся інформація про кластер Kubernetes (наприклад, існуючі вузли, поди, сервіси тощо);
3. kube-scheduler - компонент, що слідкує за появою нових подів та вирішує, на яких вузлах їх слід запускати;
4. Kube-controller-manager - компонент, що відповідає за запуск контролерів (наприклад, контролер вузлів або Deployments);

5. cloud-controller-manager - компонент, що реалізує специфічну логіку для хмарного провайдера і дозволяє керувати ресурсами даного провайдера.

Навантаження від користувачів обробляється робочими нодами, що складаються з таких частин [5]:

1. kubelet - процес відповідальний за запуск контейнерного двигуна та моніторинг запущених подів;
2. kube-proxy - мережевий проксі, задачею якого є маршрутизація між подами на різних вузлах, а також між подами та мережею Інтернет;
3. контейнерний двигун - програмне забезпечення відповідальне за запуск контейнерів. Kubernetes підтримує такі двигуни як containerd, CRI-O та будь-які інші, що імплементують Kubernetes Container Runtime Interface.

1.4 Об'єкти кластера Kubernetes

Об'єкти кластеру Kubernetes є базовими будівельними блоками, що використовуються для опису бажаного стану інфраструктури. Керування об'єктами здійснюється шляхом взаємодії з Kubernetes API, за допомогою якого можна створювати, змінювати та видаляти певні ресурси.

До основних об'єктів Kubernetes можна віднести наступні:

1. Под (англ. Pods) - один або група контейнерів, що розділяють спільне сховище та мережеві ресурси; найменша одиниця в кластері Kubernetes для розгортання.
2. Сервіси - об'єкти, виступають своєрідним веб-проксі та надають стабільну IP-адресу або DNS-ім'я для набору подів, що дозволяє іншим частинам програми звертатися до них.
3. Розгортання (англ. Deployments) - контролер, що слідкує за актуальним станом вказаних об'єктів і змінює його відповідно до

задекларованого; використовуються для оновлення додатків, а також поверненням додатків до попередніх версій.

4. Набори реплік (англ. Replica Sets) - об’єкт кластеру Kubernetes, задачею якого є забезпечення постійної наявності вказаної кількості запущених ідентичних подів.
5. Конфігураційні мапи та секрети (англ. ConfigMaps, Secrets) - об’єкти, призначенням яких є збереження конфігураційних даних та конфіденційної інформації (наприклад, паролю або токену доступу), які є необхідними для роботи окремих подів.
6. Простори імен (англ. Namespaces) - об’єкти, призначенням яких є організація та логічне групування об'єктів Kubernetes у кластері, що дозволяє проводити ізоляцію ресурсів всередині одного кластеру.

Існують і інші об'єкти Kubernetes, кожен з яких має власний набір властивостей та функцій.

2. АНАЛІЗ ВЕКТОРІВ АТАК НА KUBERNETES ТА ПРОТИДІЯ ЇМ

2.1 Вектори атак на кластер Kubernetes

Фреймворк MITRE ATT&CK® - це база знань відомих тактик і технік, що використовуються в кібератаках. Починаючи з покриття операційних систем Windows і Linux матриці MITRE ATT&CK покривають різні етапи кібератак і відомі методи, що використовуються в них. Такі матриці допомагають організаціям розуміти можливі вектори атак в їхніх системах та приймати рішення щодо їх захисту. Фреймворк MITRE ATT&CK включає в себе наступні етапи кібератак:

- Первинний доступ
- Виконання коду
- Закріплення у системі
- Підвищення привілеїв
- Обхід систем захисту
- Отримання облікових даних
- Дослідження системи для пошуку можливостей подальшого проведення атаки
- Подальше просування
- Прямий вплив на систему

Для отримання первинного доступу до кластеру Kubernetes нападник може використовувати скомпрометовані ресурси розгорнуті у кластері. До таких ресурсів можна віднести хмарні облікові дані, які дозволяють доступ до керування кластером. Також можлива компрометація образів контейнерів, що використовуються у Kubernetes-кластері. Такі образи можуть бути завантажені з публічних або приватних репозиторіїв (наприклад, Docker Hub, Amazon ECR) і містити в собі зловмисний код. Ще одним можливим способом отримання доступу до кластеру може бути отримання нападником kubeconfig файлу, який містить

інформацію про кластер, в тому числі й облікові дані. Не можна забувати і про вразливості застосунків перед експлойтами, що теж може стати причиною отримання несанкціонованого доступу до кластеру.

Друга тактика описує як нападник може виконати зловмисний код всередині контейнерів, що належать кластеру. Таким чином загрозу становить можливість несанкціонованого виконання команди kubectl exec для виконання коду в контейнері або виконання команд зсередини поду за допомогою bash/cmd при наявності достатніх прав доступу. Ще одним варіантом неправомірного виконання коду в кластері є його виконання через розгортання контейнерів зловмисником. Вони можуть бути розгорнуті як под або контролер в кластері (наприклад, Deployment, DaemonSet тощо). Зловмисники можуть також використовувати допоміжний контейнер (sidecar) [6], який ділить сховище та мережеві ресурси з іншими контейнерами у поді, задля виконання зловмисного коду та приховування своєї активності.

Тактика закріплення у системі описує техніки, за допомогою яких нападник може забезпечити собі присутність у системі, навіть якщо він більше не зможе отримати доступ до кластеру як раніше. Цього можна досягти, використовуючи k8s-контролери, як DaemonSets, Deployment або CronJob. Дані контролери будуть гарантувати, що у кластері завжди будуть наявні скомпрометовані контейнери. Загрозу також можуть становити hostPath-томи з правами на запис, так як нападник може з їхньою допомогою забезпечити собі доступ до worker-хоста.

Тактика з підвищення привілеїв включає в себе техніки, що дозволяють отримувати додаткові привілеї в системі. У цьому зловмиснику можуть допомогти привілейовані контейнери, роль кластер-адміністратора або hostPath-томи з правами на запис.

Техніки обходу захисту включають в себе видалення лог-повідомлень контейнеру, логів подій Kubernetes, DNS-тунелювання, використання проксі-

серверу. Вищевказані дії допомагають приховати сліди перебування зловмисника у системі.

Тактика доступу до облікових даних об'єднує в собі техніки, метою яких є отримання чутливих даних: секрети додатків, паролі, токени доступу тощо. Отримавши такі дані, нападник може отримати доступ до додатків, хмарних ресурсів, а також ресурсів Kubernetes-кластеру. Нападник може отримати доступ до чутливих даних, наприклад, при неправильній конфігурації RBAC. Таким чином він може використовувати сервісний акаунт поду з підвищеними привілеями доступу до Kubernetes API-серверу для отримання Kubernetes Secrets. Також серед поширених причин компрометації різноманітних секретів є некоректна конфігурація застосунків, наприклад, при передачі чутливих даних застосунку за допомогою змінних оточення в конфігураційних файлах - в такому випадку нападник може їх отримати, використовуючи запит до API-серверу.

Техніки етапу дослідження необхідні для знаходження нових шляхів розвитку атак. Нападнику в цьому може допомогти незахищений доступ до API-серверу, kubelet API або Kubernetes дашборду. За замовчуванням Kubernetes не фільтрує трафік між подами, тому якщо нападник отримує доступ до одного поду, то надалі він може сканувати мережу, наприклад, за допомогою nmap, задля знаходження нових ресурсів для майбутніх атак.

Тактика подальшого просування покликана провести нападника далі через систему експлуатуючи знайдені вразливості. Техніки даного включають у себе техніки первинного доступу, підвищення привілеїв та доступу до облікових даних.

Врешті-решт, коли нападник встановлює контроль над необхідними ресурсами, він може переходити до тактики впливу на кластер Kubernetes, знищуючи/шифруючи дані або застосовуючи різноманітні атаки (наприклад, DoS, resource hijacking тощо).

В таблиці 2.1.1 наведено MITRE ATT&CK матрицю для кібератак на кластер Kubernetes.

Первинний доступ	Виконання коду	Закріплення у системі	Підвищення привілеїв	Обхід захисту	Доступ до облікових даних	Дослідження	Подальше просування	Вплив
Використання хмарних облікових даних	exec усередині контейнера	Backdoor-контейнер	Привілегійований контейнер	Зачистка логів контейнеру	Доступ до k8s секретів	Доступ до k8s API-серверу	Доступ до хмарних ресурсів	Знищення даних
Скомпрометовані образи контейнерів	Bash/cmd усередині контейнера	Монтований шлях хосту в контейнері з правами на запис	Прив’язка ролі адміністратора кластеру	Видалення k8s подій	Дані в конфігураційних файлах додатків	Доступ до kubelet API	Доступ до сервісного акаунт контейнеру	Викрадання (hijacking) ресурсів
Вразливості застосунку	Створення допо	k8s CronJob	Доступ до хмарних	Підключення з проксі-	Доступ до сервісного	Доступ до k8s дашборду	Внутрішня мережа	DoS-атака додатку

Первинний доступ	Виконання коду	Закріплення у системі	Підвищення привілеїв	Обхід захисту	Доступ до облікових даних	Дослідження	Подальше просування	Вплив
	міжного контейнера		ресурсів	серверу	акаунта контейнеру		кластеру	
kubeconfig файл	Експлойт застосунку (RCE)	Статичні поди	hostPath монтування до поду	Схожість назв подів/контейнерів	Компрометація контролеру доступу	Сканування мережі	Монтований шлях хосту в контейнері з правами на запис	DoS-атака на планувальник нод

Первинний доступ	Виконання коду	Закріплення у системі	Підвищення привілеїв	Обхід захисту	Доступ до облікових даних	Дослідження	Подальше просування	Вплив
	SSH-сервіс у контейнері	Допоміжний контейнер	Компрометація контролеру доступу	DNS (DNS tunneling)		Компрометація k8s оператору	Доступ до k8s дашборду	DoS-атака на service discovery
		Переписування хуків життєв	Компрометація k8s оператору	Підміна контролеру доступ		Доступ до файлової систем	Доступ до k8s оператору	

Первинний доступ	Виконання коду	Закріплення у системі	Підвищення привілеїв	Обхід захисту	Доступ до облікових даних	Дослідження	Подальше просування	Вплив
		ого циклу контейнера		у або API-серверу		и хоста		
		Переписування liveness probes	Втеча з контейнеру					
		K3d ботнет						

Табл. 2.1.1. MITRE ATT&CK матриця для Kubernetes-кластеру [7]

2.2 Аналіз існуючих рішень забезпечення інформаційної безпеки кластера Kubernetes

Існують різноманітні інструменти для допомоги в забезпеченні інформаційної безпеки кластеру Kubernetes. Задачею таких інструментів є впровадження додаткових політик безпеки та організація моніторингу за станом безпеки кластеру. Такі інструменти можна поділити на наступні категорії:

- Вбудовані інструменти безпеки системи Kubernetes;
- Інструменти контейнерного сканування;

- Доповнення моніторингу стану інформаційної безпеки IT-інфраструктури та, зокрема, кластеру Kubernetes;
- Інструменти безпеки мережі;
- Інструменти управління секретами;
- Інструменти ведення журналів і моніторингу.

Kubernetes надає декілька вбудованих засобів безпеки, таких як контроль доступу на основі ролей (RBAC), мережеві політики безпеки та політики безпеки для подів. Ці засоби можна використовувати для забезпечення дотримання політик безпеки та обмеження доступу до важливих ресурсів.

Задачею інструменти сканування контейнерів є перевірка образів контейнерів на наявність вразливостей і нормативних вимог перед їх розгортанням у кластері. Прикладом таких інструментів є Snyk Container, Aqua Security, статичний аналізатор Clair та інші.

Задачею доповнень для моніторингу стану безпеки кластеру Kubernetes є перевірка політик безпеки у реальному часі та сповіщення про підозрілу активність у кластері. Прикладами таких систем є kube-bench, Gatekeeper та більш складні, орієнтовані на комплексну перевірку стану безпеки хмарної інфраструктури: Falco, Sysdig Secure.

Інструменти мережевої безпеки, такі як Calico і Cilium, можна використовувати для забезпечення дотримання мережевих політик безпеки і контролю трафіку між подами.

Інструменти управління секретами, такі як HashiCorp Vault і Kubernetes Secrets Controller, призначені для безпечного управління і розподілу конфіденційної інформації, такої як паролі і токени API, серед додатків, що працюють в кластері.

Інструменти ведення журналів і моніторингу, такі як Elasticsearch, Fluentd, Kibana, Prometheus, Grafana тощо, використовуються для збору та аналізу

журналів і метрик кластера, що дозволяє виявляти інциденти безпеки і реагувати на них.

Вибір інструментів забезпечення безпеки кластеру Kubernetes залежить від конкретних політик безпеки додатків та організації, яка їх використовує.

З огляду на вищесказане можна зробити висновок, що забезпечення безпеки кластеру Kubernetes є комплексним завданням і потребує використання різноманітних інструментів для успішної протидії різноманітним векторам атак. Проте в даній випускній роботі магістра пропонується розробка інструменту протидії атакам на кластер Kubernetes, що є саме доповненням для моніторингу стану безпеки кластеру. Таке рішення є доволі гнучким в налаштуванні політик безпеки і може бути просто та успішно інтегровано в уже існуючі Kubernetes-інфраструктури.

2.2.1 Порівняння безпекових доповнень кластеру Kubernetes

Доповнення Kube-bench та Gatekeeper - це два популярні інструменти з відкритим вихідним кодом, що використовуються для забезпечення безпеки Kubernetes.

Kube-bench - це інструмент, розроблений організацією CIS (Center for Internet Security), який виконує автоматичну перевірку безпеки відповідно до Kubernetes Security Benchmark - набору практик і рекомендацій щодо захисту кластера Kubernetes, розроблений у співпраці з спільнотою Kubernetes. Kube-bench може допомогти виявити ризики безпеки та неправильні конфігурації в кластері Kubernetes, виконуючи перевірки за допомогою тесту, який охоплює такі області, як автентифікація, мережеві політики та безпека виконання контейнерів. Kube-bench простий у використанні і може бути запущений як окрмий двійковий файл або як контейнер всередині кластера.

До переваг Kube-bench належать наступні:

- Аудит кластеру Kubernetes відповідно до правил CIS (Center of Internet Security);

- Є можливість розширення набору тестів через YAML-маніфести.

Недоліками можна визначити такі особливості:

- Необхідність ручного виконання команди для сканування кластеру;
- Неможливість перевірки стану кластеру в режимі реального часу;
- Відсутність інтеграції с системами моніторингу.

Gatekeeper - це доповнення для слідкування за політиками безпеки кластеру Kubernetes, розроблений спільнотою розробників з відкритим вихідним кодом. Він надає декларативний фреймворк для визначення та застосування політик безпеки для об'єктів Kubernetes, таких як поди та сервіси. Gatekeeper використовує Open Policy Agent (OPA) в якості механізму політик і дозволяє адміністраторам визначати політики за допомогою спеціалізованої мови Rego. Політики можуть бути визначені для перевірки різних властивостей об'єктів Kubernetes, таких як мітки, анотації та образи контейнерів.

Перевагами Gatekeeper є:

- Використання нативних Kubernetes CRDs);
- Можливість задавати власні правила безпеки;
- Широка бібліотека шаблонів політик безпеки.

Недоліками системи Gatekeeper є такі:

- Використання власної мови Rego для задання політик безпеки;
- Не слідкування за мутацією об'єктів Kubernetes.

2.3 Постановка задачі

Проаналізувавши існуючі інструменти забезпечення інформаційної безпеки кластера Kubernetes було прийнято рішення розробки власного доповнення для перевірки політик безпеки кластеру Kubernetes з наступними вимогами:

- Система моніторингу стану кластера повинна бути створена у вигляді оператора Kubernetes;

- Рішення повинно надавати можливість задання правил безпеки без прив’язки до специфічних мов шаблонів, програмування тощо;
- Система повинна повідомляти про проблеми при створенні нових або мутації існуючих об’єктів у кластері Kubernetes;
- Система повинна надавати можливість інтеграції з системами моніторингу.

3. СТВОРЕННЯ ОПЕРАТОРА ДЛЯ АНАЛІЗУ СТАНУ БЕЗПЕКИ КЛАСТЕРУ KUBERNETES

3.1 Оператор Kubernetes

Система Kubernetes надає змогу створювати власні ресурси, користуючись при цьому вбудованими можливостями: самовідновленням, узгодженням, а також розширенням цих можливостей для відповідності потребам певного застосунку, обслуговування тощо. Таке розширення можливостей можливе завдяки операторам Kubernetes.

Оператор Kubernetes - це метод пакування, розгортання та керування додатком Kubernetes за допомогою визначень користувацьких ресурсів (CRD). Оператори дозволяють керувати складними додатками більш ефективно і послідовно, інкапсулюючи знання про програму в контролер, який потім може автоматизувати багато операційних завдань, які в іншому випадку вимагали б ручного втручання. Використовуючи оператори, розробники та команди DevOps можуть підвищити надійність, масштабованість та безпеку своїх додатків, що працюють на Kubernetes.

Рис. 3.1.1 Діаграма оператору Kubernetes [8]

Для роботи оператора необхідні наступні компоненти:

1. Застосунок або інфраструктура для керування;

2. Доменно-специфічна мова, що дозволяє користувачеві вказувати бажаний стан застосунку;
3. Контролер, що постійно працює:
   - Зчитує свій стан.
   - Виконує автоматичні дії щодо застосунку.
   - Оновлює свій стан.

Оператори Kubernetes можуть бути створені за допомогою різноманітних мов програмування та відповідних бібліотек, наприклад Python, Java, Rust, Golang тощо. Оператор для моніторингу безпеки кластеру Kubernetes в даній роботі реалізовано мовою Golang за допомогою фреймворку Kubebuilder.

Мова Golang - компільована мова програмування з відкритим вихідним кодом, розроблена компанією Google. Такий вибір був зумовлений, тим що система оркестрування Kubernetes написана цією мовою, тому й саме для неї доступна найбільш розширена версія бібліотеки для взаємодії з Kubernetes API.

Фреймворк Kubebuilder - це програмний каркас для створення програмних додатків, що взаємодіють з Kubernetes APIs використовуючи спеціальні визначення ресурсів (CRDs) кластеру Kubernetes [9]. Призначенням вказаного фреймворку є збільшення швидкості та зменшення складності для створення і публікації Kubernetes API-застосунків мовою Go.

На рисунку 3.1.2 наведено блок-схему алгоритму роботи оператора з перевірки стану безпеки кластера Kubernetes.

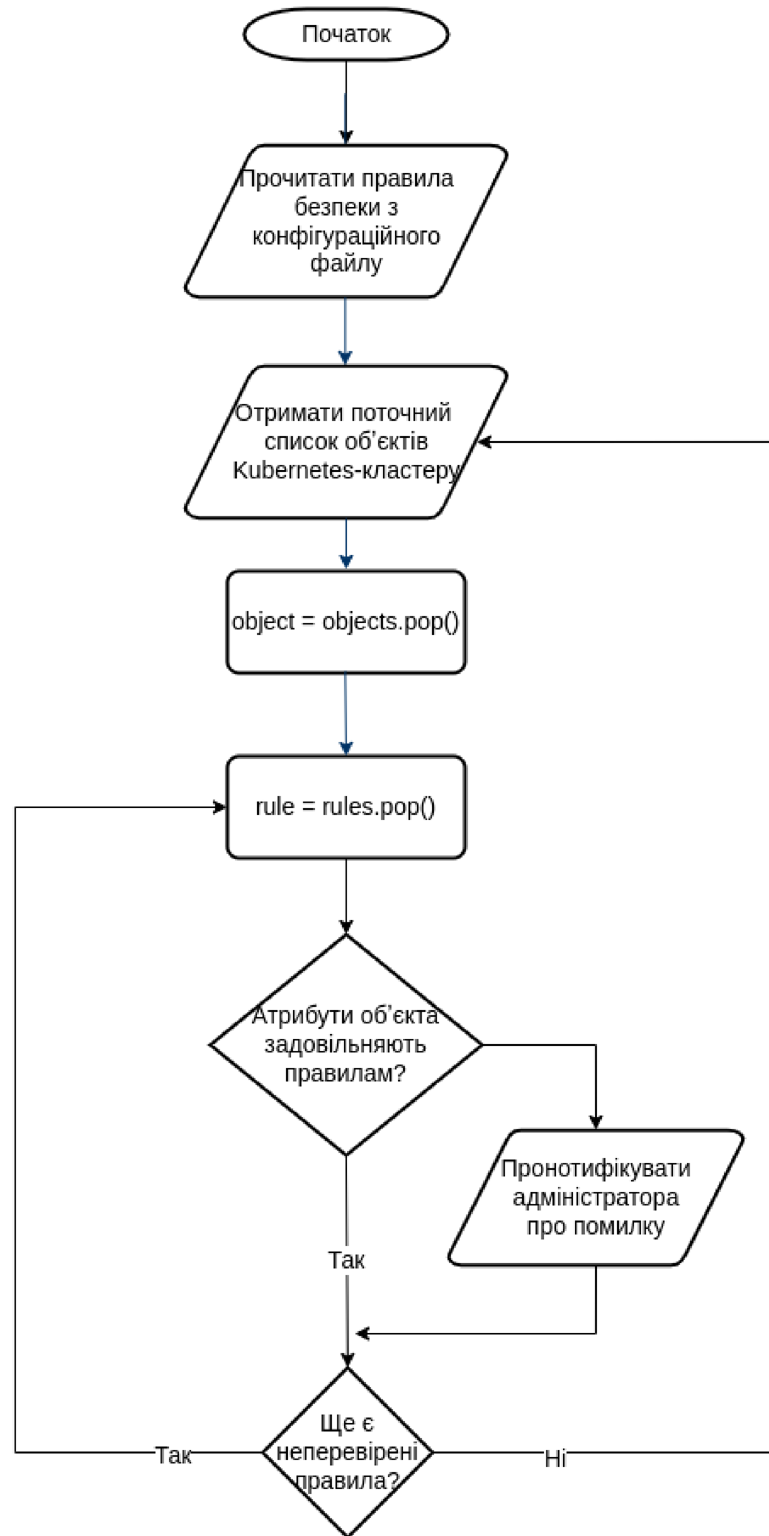

Рис. 3.1.2 Блок-схема алгоритму роботи оператора

3.2 Задання правил політик безпеки

Для задання політик безпеки кластеру Kubernetes було використано мову YAML. YAML - це мова серіалізації даних. Її часто використовують для конфігураційних файлів і обміну даними між системами, особливо в контексті хмарних обчислень, DevOps і контейнеризації.

Файли YAML складаються з серії пар ключ-значення, де ключі відокремлюються від значень двокрапкою і пробілом. Кожна пара ключ-значення відокремлюється символом нового рядка. Відступи використовуються для позначення ієрархії даних, і вкладені елементи мають більший відступ, ніж їхні батьківські елементи.

YAML розроблена так, щоб її було легко читати і писати людям, а також легко аналізувати і генерувати машинам. Вона часто використовується для конфігураційних файлів у таких системах, як Kubernetes, Docker та Ansible, де її застосовують для визначення конфігурацій розгортання, служб та інших деталей, пов'язаних з інфраструктурою.

Приклад задання політик безпеки для оператору для моніторингу стану безпеки кластера Kubernetes наведено у додатку Г. За допомогою ключа directive вказується назва властивості, що потребує перевірки у новостворених подів, а ключі exists та setTo перевіряють наявність цієї директиви в маніфесті об’єкту та її відповідність заданому значенню. Також є підтримка рекурсивних властивостей завдяки ключу children.

3.2.1 Директива resources

Директива resources у маніфесті поду використовується для вказання кількості ресурсів центрального процесору та оперативної пам’яті, що повинні бути виділені контейнеру або контейнерам, які запущені у поді. Директива resources має дві вкладені директиви: limits і requests.

Директива requests вказує мінімальну необхідну кількість ресурсів центрального процесору та пам'яті, які необхідні для запуску пода. Система Kubernetes використовує вказані дані для вибору вузлів з достатньою кількістю ресурсів, на яких буде запущено под. Якщо такого вузол є недоступним на даний момент, под буде в стані очікування на запуск поки не з'явиться достатньо вільних ресурсів.

За допомогою атрибуту limits вказується максимальна кількість ресурсів центрального процесору та пам'яті, що дозволено поду до використання. Використовуючи дану директиву Kubernetes запобігає використанню контейнерами завеликого об'єму обчислювальних ресурсів.

Забезпечення перевірки оператором спостереження за станом безпеки кластера Kubernetes надає наступні переваги:

- Ізоляція ресурсів: завдяки вказанню обмежень на ресурси адміністратор системи Kubernetes може впевнитись у тому, що контейнери, запущені на одному робочому вузлі, не будуть конкурувати за наявні обчислювальні ресурси, тому унеможливлюється виникнення ситуації, при якій певний з них буде споживати усі наявні потужності;
- Захист від DoS-атак: якщо навантаження на под раптово зростає і контейнери починають споживати більше обчислювальних ресурсів, ніж вказано у лімітах, то система Kubernetes знищить такий контейнер і вивільнить ресурси для інших контейнерів.

На рисунку 3.2.1.1 наведено лістинг маніфесту поду, котрий демонструє коректний спосіб задання директиви resources.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: nginx
      resources:
        requests:
          memory: "64Mi"
          cpu: "250m"
        limits:
          memory: "128Mi"
          cpu: "500m"

```

Рис. 3.2.1.1 Приклад задання директиви resources

3.2.2 Директива securityContext

Директива securityContext використовується у маніфесті поду для вказання налаштувань, що відносяться безпосередньо до безпеки контейнерів.

При відсутності або недостатньо строгому налаштуванні вказаної директиви виникає збільшений ризик компрометації поду, в той час як її коректне налаштування дозволяє попередити атаки типу "втеча з контейнеру". Директива securityContext включає в себе наступні налаштування:

- allowPrivilegeEscalation
- privileged
- hostPID, hostIPC
- hostNetwork
- allowedHostPaths
- readOnlyRootFilesystem
- runAsUser/runAsGroup/fsGroup/supplementalGroup
- SELinux
- AppArmor

- seccomp

Назва налаштування securityContext	Рекомендоване значення
allowPrivilegeEscalation	false
privileged	false
hostPID, hostIPC	false
hostNetwork	false
allowedHostPaths	Використання випадкової назви (наприклад, “/foo”)
readOnlyRootFilesystem	true
runAsUser	MustRunAsNonRoot
runAsGroup	Ненульове значення
fsGroup	Ненульове значення
supplementalGroup	Ненульове значення
SELinux	Додавання в середовищах, що підтримують SELinux
AppArmor	Додавання в середовищах, що підтримують AppArmor
seccomp	Додавання в середовищах, що підтримують seccomp

Табл. 3.2.2.1. Налаштування директиви securityContext [10]

Найчастіше процеси всередині контейнеру не повинні бути запущені від імені суперкористувача на відміну від повноцінної операційної системи Linux. Гарною практикою є застосування принципу “найменших привілеїв”. Таким чином програма повинно мати доступ виключно до тієї інформації і ресурсів, які потрібні їй для виконання своїх завдань. Такий підхід зменшує можливу шкоду, якщо зловмиснику вдалось використати вразливість у програмі і він намагається отримати конфіденційну інформацію або виконати зловмисний код.

Зважаючи на вищевказане, постає необхідність у забороні запуску контейнерів з правами суперкористувача. Для цієї задачі використовуються директиви runAsUser та runAsNonRootUser.

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: my-pod
5 spec:
6   securityContext:
7     runAsUser: 1000
8     runAsNonRoot: true
9   containers:
10  - name: my-container
11    image: nginx
12
```

Рис. 3.2.2.1 Приклад використання директиви runAsUser та runAsNonRootUser

Значенням директиви runAsUser є ідентифікатор користувача (UID), котрий повинен використовуватись у контейнері. Якщо вказане налаштування встановлено в маніфесті поду, то воно перезапише значення користувача, що вказане в образі контейнера. Для підвищення рівня безпеки рекомендовано встановлювати різні UID для різних контейнерів, якщо їм не потрібно ділити спільне сховище даних.

Директива runAsNonRootUser призначена для заборони запуску контейнера від імені суперкористувача. При запуску контейнера з встановленим налаштуванням runAsNonRootUser у true, система Kubernetes перевірить чи не відбувається запуск контейнера від імені суперкористувача. Якщо такий запуск відбувається, то система оркестрування відмовиться створювати такий ресурс і виведе повідомлення про помилку для адміністратора.

Директиви privileged і allowPrivilegeEscalation призначені для обмеження рівня привілеїв, що мають контейнеру всередині поду.

Директива privileged, встановлена у значення true, вимикає всі простори імен (окрім process) та безпекові модулі Linux (наприклад, SeLinux, AppArmor тощо), надає контейнеру всі привілеї, дає доступ до всіх хостових пристроїв через директиву /dev. Таким чином привілейовані поди є альтернативою запуска контейнера з правами суперкористувача і створюють потенційні безпекові ризики. Для уникнення вищевказаних проблем, директива privileged повинна встановлення у значення false.

Аналогічно рекомендовано встановлювати директиву allowPrivilegeEscalation у значення false. Таке налаштування попереджає отримання контейнером додаткових привілеїв у порівнянні з тими, що були надані їм при старті, наприклад через використання виконуваних файлів з бітом setuid або setgid.

Проте при використанні контейнерів можуть виникати ситуації, коли потрібні підвищений рівень прав для виконання певних операцій. В таких випадках доцільним є використання директиви capabilities.

Дана директива контролює доступність для поду привілеїв ядра Linux - гнучкої системи дозволів, що контролює доступ до виконання різних адміністративних операцій [11]. Прикладами таких привілеїв є наступні:

- CAP_SETPCAP - дозволяє процесу встановлювати привілеї іншого процесу (дочірнього) або модифікувати власні;
- CAP_NET_ADMIN - керування мережевими інтерфейсами;
- CAP_CHOWN - внесення змін до власників файлів;
- CAP_KILL - надсилання сигналів до процесів;
- CAP_MAC_ADMIN - модифікація MAC-налаштувань;
- CAP_SYS_ADMIN - виконання великої кількості адміністративних операцій;

- CAP_NET_BIND_SERVICE - використання привілейованих мережевиї сокетів (з порядковим номером до 1024);
- CAP_SYS_TIME - модифікація системного годинника.

Директива securityContext.capabilites дозволяє модифікувати Linux-привілеї для поду. Вона має дві власні директиви drop та add для видалення і додавання привілеїв відповідно. Гарною практикою є видалення усіх привілеїв і додавання тільки тих, які потрібні контейнеру для виконання необхідних йому операцій.

На рисунку 3.2.2.2 наведено прикладу маніфесту поду з зворотнім проксі Nginx, що має безпечну конфігурацію з дотриманням вищевказаних рекомендацій щодо налаштування securityContext.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secure-nginx
spec:
  containers:
  - name: nginx
    image: nginx:latest
    resources:
      limits:
        cpu: 500m
        memory: 512Mi
      requests:
        cpu: 100m
        memory: 128Mi
    securityContext:
      runAsNonRoot: true
      allowPrivilegeEscalation: false
      capabilities:
        drop:
        - ALL
        add:
        - NET_ADMIN
    ports:
    - containerPort: 80
      protocol: TCP
```

Рис. 3.2.2.2 Маніфест поду Nginx

3.3 Розгортання оператору в кластері Kubernetes

Для розгортання оператора в кластері Kubernetes використовується утиліта командного рядка kubectl, призначенням котрої є керування кластером: створення та видалення ресурсів, отримання інформації про стан кластеру та його об’єктів, створення та оновлення конфігураційних файлів тощо.

Лістинг коду оператора наведено у додатках А та Б, правила політик безпеки для перевірки стану кластера наведено у додатку Г.

Для демонстрації роботи оператора використано некоректно сконфігурований ресурс Deployment, наведений у додатку В.

Спочатку до кластеру Kubernetes було додано оператор (рис. 3.2), після чого також було додано ресурс типу Deployment (рис. 3.3).

```
❯ kubectl apply -f config/samples/

secmonitor.secmonitor.volek.security.io/secmonitor-sample created
```

Рис. 3.2. Розгортання оператора Secmonitor у кластері Kubernetes.

```
❯ kubectl apply -f hello-config-env/k8s/
configmap/demo-config created
deployment.apps/demo created
```

Рис. 3.3. Розгортання Deployment у кластері Kubernetes.

```
1.6835529731122503e+09	INFO	Pod found	{"controller": "secmonitor", "controllerGroup": "secmonitor.volek.securi
ty.io", "controllerKind": "Secmonitor", "Secmonitor": {"name":"secmonitor-sample","namespace":"default"}, "namespace": "
default", "name": "secmonitor-sample", "reconcileID": "d7592e68-ecbb-401f-a58a-b76066383784", "name": "kube-apiserver-mi
nikube"}
1.6835529731122687e+09	INFO	RunAsNonRoot has to be enabled	{"controller": "secmonitor", "controllerGroup": "secmoni
tor.volek.security.io", "controllerKind": "Secmonitor", "Secmonitor": {"name":"secmonitor-sample","namespace":"default"}
, "namespace": "default", "name": "secmonitor-sample", "reconcileID": "d7592e68-ecbb-401f-a58a-b76066383784", "Pod": "ku
be-apiserver-minikube"}
1.683552973112284e+09	INFO	Pod has to run with specified RunAsUser and RunAsGroup settings {"controller": "secmonit
or", "controllerGroup": "secmonitor.volek.security.io", "controllerKind": "Secmonitor", "Secmonitor": {"name":"secmonito
r-sample","namespace":"default"}, "namespace": "default", "name": "secmonitor-sample", "reconcileID": "d7592e68-ecbb-401
f-a58a-b76066383784"}
1.6835529731122992e+09	ERROR	pod is insecure {"controller": "secmonitor", "controllerGroup": "secmonitor.volek.securi
ty.io", "controllerKind": "Secmonitor", "Secmonitor": {"name":"secmonitor-sample","namespace":"default"}, "namespace": "
default", "name": "secmonitor-sample", "reconcileID": "d7592e68-ecbb-401f-a58a-b76066383784", "Name": "kube-apiserver-mi
nikube", "error": "pod is insecure"}
github.com/volekkkkk/secoperator/controllers.(*SecmonitorReconciler).Reconcile
	/home/volek/go/src/github.com/volekkkkk/secoperator/controllers/secmonitor_controller.go:76
sigs.k8s.io/controller-runtime/pkg/internal/controller.(*Controller).Reconcile
	/home/volek/go/pkg/mod/sigs.k8s.io/controller-runtime@v0.13.1/pkg/internal/controller/controller.go:121
sigs.k8s.io/controller-runtime/pkg/internal/controller.(*Controller).reconcileHandler
	/home/volek/go/pkg/mod/sigs.k8s.io/controller-runtime@v0.13.1/pkg/internal/controller/controller.go:320
```

Рис. 3.4. Демонстрація роботи оператора Secmonitor.

В результаті від оператора було отримано повідомлення з інформацією про наявність у системі ресурсу, налаштування безпеки котрого не збігаються із вказаними у правилах безпеки (рис. 3.4): не увімкнено директиву RunAsNonRoot, а також є можливість запуску поду від імені суперкористувача, адже директиви RunAsUser і RunAsGroup не налаштовано. Тому под визначено як небезпечний ресурс і в журнал виведено повідомлення з інформацією про помилку.

В подальшому дані журналу можуть бути достатньо просто інтегровані в існуючу систему моніторингу, що дозволить повідомляти адміністраторів кластеру Kubernetes про потенційні загрози і швидко їх нівелювати.

ВИСНОВКИ

У ході виконання дипломної роботи магістра було розглянуто проблему забезпечення інформаційної безпеки кластеру Kubernetes. Було проведено аналіз архітектури системи контейнерної оркестрації та її основних об’єктів.

Аналіз векторів атак на кластер Kubernetes продемонстрував можливі проблеми конфігурації, які потребують уваги при розгортанні контейнеризованих додатків. Також було проведено аналіз існуючих рішень для забезпечення комплексного захисту системи Kubernetes і прийнято рішення з розробки доповнення для моніторингу стану безпеки кластеру. Порівняння існуючих безпекових доповнень дозволило сформувати вимоги до рішення, що було розроблено в дипломній роботі магістра.

У результаті було створено і перевірено роботу оператора для моніторингу стану безпеки кластера Kubernetes, що повністю задовольняє визначеним у роботі вимогам:

- надання можливості задання правил безпеки без прив’язки до специфічних мов шаблонів, програмування тощо;
- аналіз наявних проблем при створенні нових або мутації існуючих об’єктів у кластері;
- наявність можливості інтеграції з системами моніторингу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. CNCF Annual Survey 2021 [Онлайн ресурс] https://www.cncf.io/reports/cncf-annual-survey-2021/?utm_source=thenewstack&utm_medium=website&utm_content=inline-mention&utm_campaign=platform (Дата звернення: 28.11.2022).
2. CNCF Annual Survey 2020 [Онлайн ресурс] https://www.cncf.io/wp-content/uploads/2020/12/CNCF_Survey_Report_2020.pdf (Дата звернення: 28.11.2022).
3. 2022 State of Kubernetes security report [Онлайн ресурс] https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-overview (Дата звернення: 02.12.2022).
4. Kubernetes Documentation [Онлайн ресурс] https://kubernetes.io/docs (Дата звернення: 23.03.2023).
5. Justin Domingus, John Arundel “Cloud Native DevOps with Kubernetes” (O’Reilly). 2022.
6. Secure containerized environments with updated threat matrix for Kubernetes [Онлайн ресурс] https://www.microsoft.com/en-us/security/blog/2021/03/23/secure-containerized-environments-with-updated-threat-matrix-for-kubernetes/ (Дата звернення: 13.02.2023).
7. Andrew Martin, Michael Hausenblas “Hacking Kubernetes” (O’Reilly). 2022.
8. CNCF Operator White Paper [Онлайн ресурс] https://github.com/cncf/tag-app-delivery/blob/eece8f7307f2970f46f100f51932db106db46968/operator-wg/whitepaper/Operator-WhitePaper_v1-0.md#operator-design-pattern (Дата звернення: 14.01.2023)

9. Kubebuilder Documentation [Онлайн ресурс]. URL: https://book.kubebuilder.io/introduction.html (Дата звернення: 14.01.2023)
10. Kubernetes Hardening Guide [Онлайн ресурс]. URL: https://media.defense.gov/2022/Aug/29/2003066362/-1/-1/0/CTR_KUBERNETES_HARDENING_GUIDANCE_1.2_20220829.PDF (Дата звернення: 15.04.2023)
11. Michael Kerrisk "The Linux Programming Interface: A Linux and UNIX System Programming Handbook" (No Starch Press). 2010.

ДОДАТКИ

Додаток А

Програмний код CRD оператору моніторингу безпеки подів.

```
package v1

import (
	// batchv1 "k8s.io/api/batch/v1"
	metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

// EDIT THIS FILE!  THIS IS SCAFFOLDING FOR YOU TO OWN!
// NOTE: json tags are required.  Any new fields you add must have json tags for the fields to be serialized.

// SecmonitorSpec defines the desired state of Secmonitor
type SecmonitorSpec struct {
	// INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
	// Important: Run "make" to regenerate code after modifying this file

	PodName string `json:"podName,omitempty"`
}

// SecmonitorStatus defines the observed state of Secmonitor
type SecmonitorStatus struct {
	// INSERT ADDITIONAL STATUS FIELD - define observed state of cluster
	// Important: Run "make" to regenerate code after modifying this file
```

```
	PodIsSafe bool `json:"podIsSafe,omitempty"`
}

//+kubebuilder:object:root=true
//+kubebuilder:subresource:status

// Secmonitor is the Schema for the secmonitors API
type Secmonitor struct {
	metav1.TypeMeta   `json:",inline"`
	metav1.ObjectMeta `json:"metadata,omitempty"`

	Spec   SecmonitorSpec   `json:"spec,omitempty"`
	Status SecmonitorStatus `json:"status,omitempty"`
}

//+kubebuilder:object:root=true

// SecmonitorList contains a list of Secmonitor
type SecmonitorList struct {
	metav1.TypeMeta `json:",inline"`
	metav1.ListMeta `json:"metadata,omitempty"`
	Items           []Secmonitor `json:"items"`
}

func init() {
	SchemeBuilder.Register(&Secmonitor{}, &SecmonitorList{})}
```

Додаток Б

Програмний код контролеру оператора моніторингу безпеки подів.

```
package controllers

import (
	"context"
	"errors"
	"fmt"
	"gopkg.in/yaml.v3"
	"k8s.io/apimachinery/pkg/runtime"
	"os"
	ctrl "sigs.k8s.io/controller-runtime"
	"sigs.k8s.io/controller-runtime/pkg/client"
	"sigs.k8s.io/controller-runtime/pkg/log"

	secmonitorv1 "github.com/volekkkkk/secoperator/api/v1"

	corev1 "k8s.io/api/core/v1"
	"k8s.io/apimachinery/pkg/fields" // Required for Watching
	"k8s.io/apimachinery/pkg/types"  // Required for Watching

	// "sigs.k8s.io/controller-runtime/pkg/builder"   // Required for Watching
	"sigs.k8s.io/controller-runtime/pkg/handler" // Required for Watching
	// "sigs.k8s.io/controller-runtime/pkg/predicate" // Required for Watching
	"sigs.k8s.io/controller-runtime/pkg/reconcile" // Required for Watching
	"sigs.k8s.io/controller-runtime/pkg/source"    // Required for Watching
)
```

```
const (
	podField = "Pod"
)

type SecmonitorReconciler struct {
	client.Client
	Scheme *runtime.Scheme
}

var Rules []Rule

//+kubebuilder:rbac:groups=secmonitor.volek.security.io,resources=secmonitors,verbs=
get;list;watch;create;update;patch;delete
//+kubebuilder:rbac:groups=secmonitor.volek.security.io,resources=secmonitors/status,v
erbs=get;update;patch
//+kubebuilder:rbac:groups=secmonitor.volek.security.io,resources=secmonitors/finalize
rs,verbs=update
//+kubebuilder:rbac:groups="",resources=pods,verbs=get;list;watch

func (r *SecmonitorReconciler) Reconcile(ctx context.Context, req ctrl.Request)
(ctrl.Result, error) {
	log := log.FromContext(ctx)
	log.Info("Reconciling Secmonitor resource")

	// Get the Secmonitor resource that triggered the reconciliation request
	var secmonitor secmonitorv1.Secmonitor
	if err := r.Get(ctx, req.NamespacedName, &secmonitor); err != nil {
```

```
log.Error(err, "unable to fetch Secmonitor")
return ctrl.Result{}, client.IgnoreNotFound(err)
}

var podList corev1.PodList
var podsAreSafe bool

if err := r.List(ctx, &podList); err != nil {
	log.Error(err, "unable to list Pods")
} else {
	for _, item := range podList.Items {
		log.Info("Pod found", "name", item.GetName())
		if !r.podIsSecured(ctx, item) {
			log.Error(errors.New("pod is insecure"), "pod is insecure",
"Name", item.GetName())
			podsAreSafe = false
		}
	}
}

secmonitor.Status.PodIsSafe = podsAreSafe
if err := r.Status().Update(ctx, &secmonitor); err != nil {
	log.Error(err, "unable to update secmonitor status", "podsAreSafe",
podsAreSafe)
	return ctrl.Result{}, err
}
log.Info("secmonitor status updated", "status", podsAreSafe)
log.Info("secmonitor custom resource reconciled")
```

```
	return ctrl.Result{}, nil
}

func (r *SecmonitorReconciler) podIsSecured(ctx context.Context, pod corev1.Pod)
bool {
	log := log.FromContext(ctx)
	isSecured := true

	securityCtx := pod.Spec.SecurityContext
	if securityCtx == nil {
		log.Info("SecurityContext has to be provided to Pod resource")
		isSecured = false
	} else {
		if securityCtx.RunAsNonRoot == nil || !*securityCtx.RunAsNonRoot {
			log.Info("RunAsNonRoot has to be enabled", "Pod", pod.Name)
			isSecured = false
		}
		if (securityCtx.RunAsUser == nil || *securityCtx.RunAsUser == 0) &&
(securityCtx.RunAsGroup == nil || *securityCtx.RunAsGroup == 0) {
			log.Info("Pod has to run with specified RunAsUser and RunAsGroup
settings")
			isSecured = false
		}
	}
	return isSecured
}
```

```
func (r *SecmonitorReconciler) InitRules(ctx context.Context) error {
	log := log.FromContext(ctx)

	f, err := os.ReadFile("rules.yaml")
	if err != nil {
		log.Error(err, "Error reading rules.yaml")
		return err
	}

	if err := yaml.Unmarshal(f, &Rules); err != nil {
		log.Error(err, "Error unmarshalling rules")
		return err
	}

	fmt.Printf("%+v\n", Rules)
	return nil
}

func (r *SecmonitorReconciler) SetupWithManager(mgr ctrl.Manager) error {
	if err := mgr.GetFieldIndexer().IndexField(context.Background(),
&secmonitorv1.Secmonitor{}, podField, func(rawObj client.Object) []string {
		secmonitor := rawObj.(*secmonitorv1.Secmonitor)
		if secmonitor.Spec.PodName == "" {
			return nil
		}
		return []string{secmonitor.Spec.PodName}
	}); err != nil {
		return err
```

```
	}

	return ctrl.NewControllerManagedBy(mgr).
		For(&secmonitorv1.Secmonitor{}).
		Watches(
			&source.Kind{Type: &corev1.Pod{}},

	handler.EnqueueRequestsFromMapFunc(r.findObjectsForSecmonitor),
		).
		Complete(r)
}

func (r *SecmonitorReconciler) findObjectsForSecmonitor(obj client.Object)
[]reconcile.Request {
	ctx := context.Background()
	log := log.FromContext(ctx)
	log.Info("we are in findObjectsForPod func", "pod", obj.GetName()) // DEBUG

	r.InitRules(ctx)

	attachedSecmonitors := &secmonitorv1.SecmonitorList{}
	listOps := &client.ListOptions{
		FieldSelector: fields.OneTermEqualSelector(podField, obj.GetName()),
		Namespace:    obj.GetNamespace(),
	}
	if err := r.List(context.TODO(), attachedSecmonitors, listOps); err != nil {
		log.Error(err, "unable to list secmonitor custom resources")
		return []reconcile.Request{}
```

```
	}

	requests := make([]reconcile.Request, len(attachedSecmonitors.Items))
	for i, item := range attachedSecmonitors.Items {
		requests[i] = reconcile.Request{
			NamespacedName: types.NamespacedName{
				Name:      item.GetName(),
				Namespace: item.GetNamespace(),
			},
		}

		log.Info("pod linked to a secmonitor issued an event", "name", obj.GetName())
	}
	return requests }
```

Додаток В

Маніфест тестового Deployment та ConfigMap.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
    spec:
      containers:
        - name:  demo
          image: cloudnatived/demo:hello-config-env
          ports:
            - containerPort: 8888
          env:
            - name: GREETING
              valueFrom:
                configMapKeyRef:
                  name: demo-config
                  key: greeting
```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo-config
data:
  greeting: Hola
```

Додаток Г

```yaml
rules:
  - directive: "resources"
    exists: true

  - directive: "securityContext"
    exists: true
    children:
      - directive: "allowPrivilegeEscalation"
        setTo: false

      - directive: "runAsUser"
        exists: true

      - directive: "runAsGroup"
        exists: true

      - directive: "runAsNonRoot"
        exists: true

      - directive: "fsGroup"
        exists: true
```