

Міністерство освіти і науки України
Державний університет «Житомирська політехніка»
Факультет інформаційно-комп'ютерних технологій
Кафедра комп'ютерної інженерії та кібербезпеки

ПОЯСНЮВАЛЬНА ЗАПИСКА

до випускної кваліфікаційної роботи магістра
на тему: «Методи та технології підвищення безпеки кон-
тейнеризованих додатків на базі Docker»

Виконав: студент 2-го курсу групи КБм-24-1
спеціальності 125 «Кібербезпека та захист інформації»



Владислав МУРСАЛОВ

Керівник: доцент кафедри КІ та КБ к.пед.н. доцент



Олена ГОЛОВНЯ

Рецензент: старший викладач кафедри КІ та КБ



Вадим МИКОЛАЙЧУК

Державний університет «Житомирська політехніка»
Факультет інформаційно-комп'ютерних технологій
Кафедра комп'ютерної інженерії та кібербезпеки
Спеціальність 125 «Кібербезпека та захист інформації»
Освітня програма «Кібербезпека»

ЗАТВЕРДЖУЮ

Завідувач

кафедри комп'ютерної
інженерії та кібербезпеки

 Андрій ЄФІМЕНКО
6 жовтня 2025 р.

ЗАВДАННЯ

на випускню кваліфікаційну роботу магістра

Студент: Мурсалов Владислав Радікович

Тема роботи: «Методи та технології підвищення безпеки контейнеризованих додатків на базі Docker»

затверджена Наказом університету від 6 жовтня 20245р. №566/с

Термін здачі студентом закінченої роботи 16 грудня 2025 р.

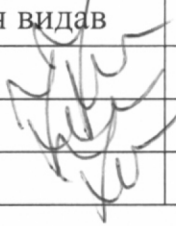
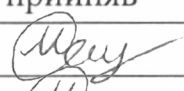
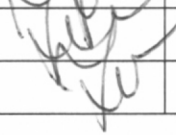
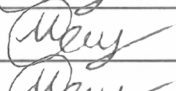
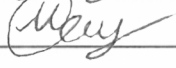
Вихідні дані роботи (зазначається предмет і об'єкт дослідження):

1. Безпека контейнеризованих середовищ на основі Docker
2. Методи та технології забезпечення безпеки контейнеризованих додатків, включаючи механізми ізоляції, управління доступом, моніторингу та аналізу вразливостей

Обладнання та програмне забезпечення:

1. Docker
2. VS Code та Visual Studio Community

Консультанти з випускної кваліфікаційної роботи із зазначенням розділів, що їх стосуються

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
1	Ральковський Г.Г.		
2	Ральковський Г.Г.		
3	Ральковський Г.Г.		

Студент  Владислав МУРСАЛОВ
Керівник  Олена ГОЛОВНЯ

Календарний план

№ з/п	Назви етапів випускної кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1	Аналіз літературних джерел та нормативних документів з безпеки контейнеризованих систем	06.10.2025 - 12.10.2025	
2	Дослідження теоретико-методологічних засад контейнеризації та написання Розділу 1	13.10.2025 - 26.10.2025	
3	Аналіз інфраструктури Docker та інструментів забезпечення безпеки	27.10.2025 - 09.11.2025	
4	Дослідження рекомендацій OWASP та розробка плану впровадження безпеки. Написання Розділу 2	10.11.2025 - 19.11.2025	
5	Практична імплементація механізмів безпеки Docker та налаштування проекту	20.11.2025 - 30.11.2025	
6	Тестування та верифікація розробленої системи безпеки. Написання Розділу 3	01.12.2025 - 07.12.2025	
7	Формулювання висновків та рекомендацій	08.12.2025 - 10.12.2025	
8	Оформлення пояснювальної записки та додатків	11.12.2025 - 13.12.2025	

Студент  Владислав МУРСАЛОВ
Керівник  Олена ГОЛОВНЯ

АНОТАЦІЯ

Кваліфікаційна робота магістра присвячена дослідженню безпеки контейнеризованих систем на базі платформи Docker. У роботі проаналізовано теоретико-методологічні засади контейнеризації, досліджено архітектуру Docker та вбудовані механізми забезпечення безпеки, розглянуто сучасні підходи та інструменти для захисту контейнерних середовищ.

Пояснювальна записка містить 75 сторінок, 9 рисунків, 1 таблицю та додатки. Список використаних джерел налічує 15 найменувань та розміщений на 2 сторінках. Додатки містять лістинги конфігураційних файлів безпеки: Dockerfile, docker-compose.yml, скрипт автоматизованого розгортання deploy-secure.sh, Seccomp-профіль, конфігурацію nginx.conf, скрипт генерації TLS-сертифікатів, приклад daemon.json та AppArmor-профіль.

Метою кваліфікаційної роботи є комплексне дослідження механізмів забезпечення безпеки контейнеризованих систем, аналіз вразливостей та розробка рекомендацій щодо підвищення рівня кібербезпеки Docker-інфраструктури.

У роботі визначено основні завдання для забезпечення безпеки контейнеризованих додатків, проаналізовано базові принципи контейнеризації та їх вплив на модель безпеки, розглянуто особливості архітектури Docker та її компонентів. Показано розробку багаторівневої системи захисту, що включає механізми ізоляції на рівні ядра ОС, інтеграцію з модулями безпеки Linux (AppArmor, SELinux), контроль доступу на основі ролей (RBAC) та мережеву безпеку. Наведено результати аналізу вразливостей контейнерних образів та проаналізовано рекомендації OWASP щодо підвищення кібербезпеки контейнеризованих додатків.

Ключові слова: КОНТЕЙНЕРИЗАЦІЯ, DOCKER, КІБЕРБЕЗПЕКА, ВРАЗЛИВОСТІ, CVE, OWASP, DEVSECOPS, ІЗОЛЯЦІЯ, КОНТЕЙНЕРНІ ОБРАЗИ, RBAC, APPARMOR, SELINUX.

ANNOTATION

The master's thesis is devoted to researching the security of containerized systems based on the Docker platform. The thesis analyzes the theoretical and methodological foundations of containerization, examines the architecture of Docker and its built-in security mechanisms, and considers modern approaches and tools for protecting container environments.

The explanatory note contains 75 pages, 9 figures, 1 table, and appendices. The list of references contains 15 titles and is located on 2 pages. The appendices contain listings of security configuration files: Dockerfile, docker-compose.yml, deploy-secure.sh automated deployment script, Seccomp profile, nginx configuration. conf, a script for generating TLS certificates, an example of daemon.json, and an AppArmor profile.

The purpose of the final qualification work is to conduct a comprehensive study of the mechanisms for ensuring the security of containerized systems, analyze vulnerabilities, and develop recommendations for improving the cybersecurity of the Docker infrastructure.

The thesis identifies the main tasks for ensuring the security of containerized applications, analyzes the basic principles of containerization and their impact on the security model, and examines the features of the Docker architecture and its components. The development of a multi-level protection system is presented, which includes isolation mechanisms at the OS kernel level, integration with Linux security modules (AppArmor, SELinux), role-based access control (RBAC), and network security. The results of the analysis of container image vulnerabilities are presented, and OWASP recommendations for improving the cybersecurity of containerized applications are analyzed.

Keywords: CONTAINERIZATION, DOCKER, CYBERSECURITY, VULNERABILITIES, CVE, OWASP, DEVSECOPS, ISOLATION, CONTAINER IMAGES, RBAC, APPARMOR, SELINUX.

ЗМІСТ

ВСТУП.....	10
Розділ 1. Теоретико-методологічні засади дослідження безпеки контейнеризованих систем.....	11
1.1 Роль та значення контейнеризації в сучасних інформаційних системах.....	11
1.2 Обґрунтування необхідності застосування механізмів безпеки в контейнеризованих середовищах	16
1.3 Постановка завдання дослідження	19
Висновки до розділу 1	20
Розділ 2. Огляд інструментів забезпечення безпеки в Docker.....	22
2.1. Аналіз інфраструктури Docker.....	22
2.2 Основні компоненти інфраструктури безпеки Docker.....	26
2.3 Рекомендації OWASP щодо підвищення кібербезпеки контейнеризованих додатків	31
2.4 План впровадження механізмів безпеки Docker	33
Висновки до розділу 2	35
Розділ 3. Імплементация плану забезпечення безпеки в Docker.....	37
3.1 Налаштування проекту та забезпечення безпеки відповідно розробленому плану безпеки	37
3.2 Тестування роботи скрипта для розгортання	60
Висновки до розділу 3	64
ВИСНОВКИ	65
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ.....	67

ДОДАТКИ.....	69
---------------------	-----------

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API - Application Programming Interface
AppArmor - Application Armor
CA - Certificate Authority
CI - Continuous Integration / Continuous Delivery
CLI - Command Line Interface
CORS - Cross-Origin Resource Sharing
CPU - Central Processing Unit
CSP - Content Security Policy
CSR - Certificate Signing Request
CVE - Common Vulnerabilities and Exposures
DevOps - Development Operations
DIVA - Damn Insecure Vulnerable App
DNS - Domain Name System
DoS - Denial of Service
EOF - End of File
GID - Group Identifier
GZIP - GNU Zip
HTTP - HyperText Transfer Protocol
HTTPS - HyperText Transfer Protocol Secure
ICC - Inter-Container Communication
IP - Internet Protocol
IT - Інформаційні технології
JSON - JavaScript Object Notation
LDAP - Lightweight Directory Access Protocol
MAC - Mandatory Access Control
MIME - Multipurpose Internet Mail Extensions
MITRE - Massachusetts Institute of Technology Research Establishment
mTLS - Mutual TLS

NIST - National Institute of Standards and Technology
OC - Операційна система
OWASP - Open Worldwide Application Security Project
PID - Process Identifier
PKI - Public Key Infrastructure
RAM - Random Access Memory
RBAC - Role-Based Access Control
RSA - Rivest-Shamir-Adleman
SAN - Subject Alternative Names
SBOM - Software Bill of Materials
SELinux - Security-Enhanced Linux
SGID - Set Group ID
SHA256 - Secure Hash Algorithm 256-bit
SIEM - Security Information and Event Management
SPA - Single Page Application
SUID - Set User ID
TCP - Transmission Control Protocol
TLS - Transport Layer Security
UDP - User Datagram Protocol
UID - User Identifier
XSS - Cross-Site Scripting
YAML - YAML Ain't Markup Language

ВСТУП

Актуальність теми полягає в тому що контейнеризація є однією з ключових технологій у сучасній розробці та розгортанні програмного забезпечення. Вона забезпечує ефективне використання ресурсів, спрощує масштабування та автоматизацію процесів розгортання. Однак, зростання популярності контейнерних рішень супроводжується новими викликами у сфері інформаційної безпеки, оскільки контейнери мають специфічні вразливості, пов'язані з ізоляцією, управлінням доступом та контролем ресурсів.

Метою кваліфікаційної роботи є комплексне дослідження механізмів забезпечення безпеки контейнеризованих систем, аналіз вразливостей та розробка рекомендацій щодо підвищення рівня кібербезпеки Docker-інфраструктури.

Об'єктом дослідження є безпека контейнеризованих середовищ, зокрема на основі Docker.

Предметом дослідження є методи та технології забезпечення безпеки контейнеризованих додатків, включаючи механізми ізоляції, управління доступом, моніторингу та аналізу вразливостей.

Визначена мета обумовлює наступні завдання.

- Проаналізувати архітектуру та інфраструктуру Docker для виявлення ключових компонентів, що впливають на безпеку контейнеризованих додатків, та визначення потенційних векторів атак.
- Дослідити механізми, технології та інструменти щодо забезпечення безпеки Docker.
- Визначити та розробити методологію налаштування безпеки Docker-контейнерів.
- Імплементувати та верифікувати механізми безпеки.

У процесі виконання кваліфікаційної роботи було розроблено план впровадження механізмів безпеки Docker, після чого імплементовано та верифіковано працездатність методів безпеки.

Розділ 1. Теоретико-методологічні засади дослідження безпеки контейнеризованих систем

1.1 Роль та значення контейнеризації в сучасних інформаційних системах

Перед тим, як перейти до принципів контейнеризації, важливо зрозуміти, що таке віртуалізація та віртуальні машини, які стали основою для розвитку сучасних методів ізоляції програмного забезпечення. Віртуалізація дозволяє створити логічно ізольовані середовища для запуску різних операційних систем та додатків на одному фізичному обладнанні. Вона є ключовою технологією в хмарних інфраструктурах, дата-центрах, системах резервного копіювання, тестування програмного забезпечення та у сценаріях DevOps, де потрібна багатоплатформність.

Віртуальна машина (VM) - це програмна емуляція фізичного комп'ютера. Вона виконується на хост-системі за допомогою спеціального шару - гіпервізора, який розподіляє фізичні ресурси між кількома віртуальними екземплярами. Розглянемо концептуальну схему архітектури віртуальної машини (рис. 1.1.1).

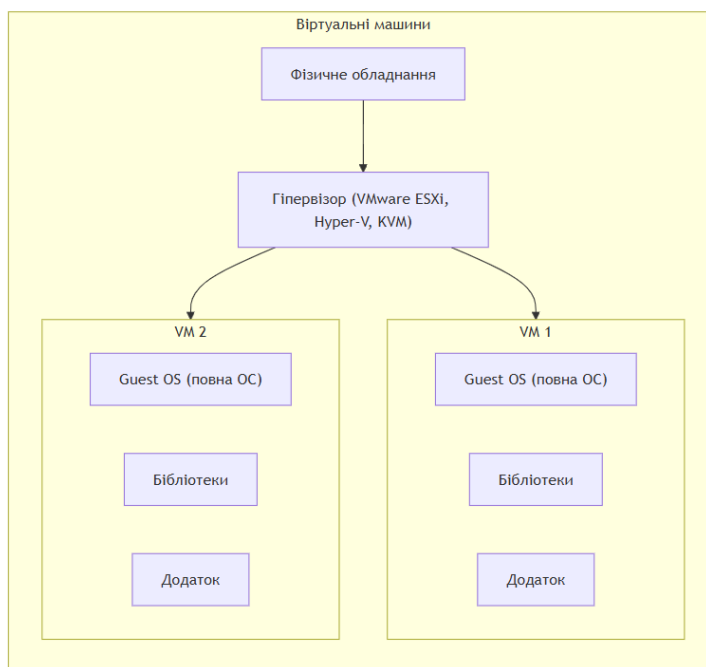


Рисунок 1.1.1. Концептуальна схема архітектури віртуальної машини.

Бачимо, що кожна віртуальна машина містить власне ядро операційної системи, що забезпечує повну ізоляцію процесів, пам'яті, мережі та файлової системи. Це дозволяє запускати декілька різних ОС (наприклад, Linux і Windows) одночасно на одному фізичному сервері.

Гіпервізори поділяються на два основних типи: тип 1, або bare-metal, встановлюються безпосередньо на апаратне забезпечення і мають вищу продуктивність, стабільність і безпеку, що робить їх популярними у корпоративному середовищі. Тип 2, або hosted, встановлюються поверх існуючої ОС, що спрощує налаштування і використання в домашніх умовах або для розробників, але створює додаткові накладні витрати. Типовими прикладами гіпервізорів є VMware ESXi, Microsoft Hyper-V, Oracle VirtualBox і KVM.

Серед переваг віртуалізації можна відзначити:

- можливість гнучкого розгортання та масштабування віртуальних середовищ,
- ефективніше використання фізичних ресурсів,
- централізоване управління та моніторинг,
- підтримка інфраструктури високої доступності,
- швидке резервне копіювання та відновлення систем.

Однак, незважаючи на ці переваги, віртуальні машини мають низку суттєвих недоліків, які впливають на ефективність ІТ-інфраструктур. Насамперед, вони є ресурсомісткими: кожна віртуальна машина містить повноцінну операційну систему, що значно збільшує споживання оперативної пам'яті, місця на диску та обчислювальних потужностей. У статті зазначено: «Кожна віртуальна машина містить повну копію операційної системи, додаток, необхідні виконувані файли та бібліотеки - що займає десятки гігабайтів» [3]. У середовищах з великою кількістю VM це призводить до надмірного навантаження на фізичне обладнання.

Також варто враховувати повільний час запуску віртуальних машин. Завантаження гостьової ОС може тривати десятки секунд або навіть хвилин, що усклад-

нює швидке масштабування додатків або автоматизоване реагування на навантаження. У контексті DevOps це створює затримки в процесах CI/CD та погіршує загальну швидкодію системи.

Крім того, адміністрування віртуальних машин вимагає значних зусиль: необхідно підтримувати, оновлювати та захищати кожен окрему ОС, що ускладнює управління та збільшує ризики помилок конфігурації. Віртуальні машини також мають більший attack surface - поверхню для атак - оскільки кожен екземпляр ОС може містити власні вразливості. Як вказано у джерелі, «Оскільки кожна віртуальна машина запускає повноцінну операційну систему, накладні витрати можуть бути значними» [3].

Ще одним недоліком є обмежена портативність: образи VM зазвичай великі за розміром, менш зручні для розповсюдження та повільніше копіюються між середовищами. У результаті розгортання програм у різних середовищах - наприклад, між dev, test і prod - часто вимагає додаткових налаштувань і адаптацій.

Усе це стало поштовхом до розвитку легших, швидших і більш автоматизованих рішень для ізоляції - таких як контейнеризація. Контейнери забезпечують ізоляцію на рівні процесів, використовуючи спільне ядро операційної системи хоста, що робить їх менш ізольованими порівняно з віртуальними машинами, проте значно ефективнішими з точки зору споживання ресурсів, швидкості запуску та гнучкості розгортання. Саме цей компроміс між рівнем ізоляції та продуктивністю зробив контейнери популярними у сфері DevOps, хмарних обчислень і мікросервісної архітектури.

Контейнеризація - це метод ізоляції програмного забезпечення та його залежностей у самостійні пакети, які називаються контейнерами. Розглянемо схему архітектури для контейнеризованих додатків (рис. 1.1.2).

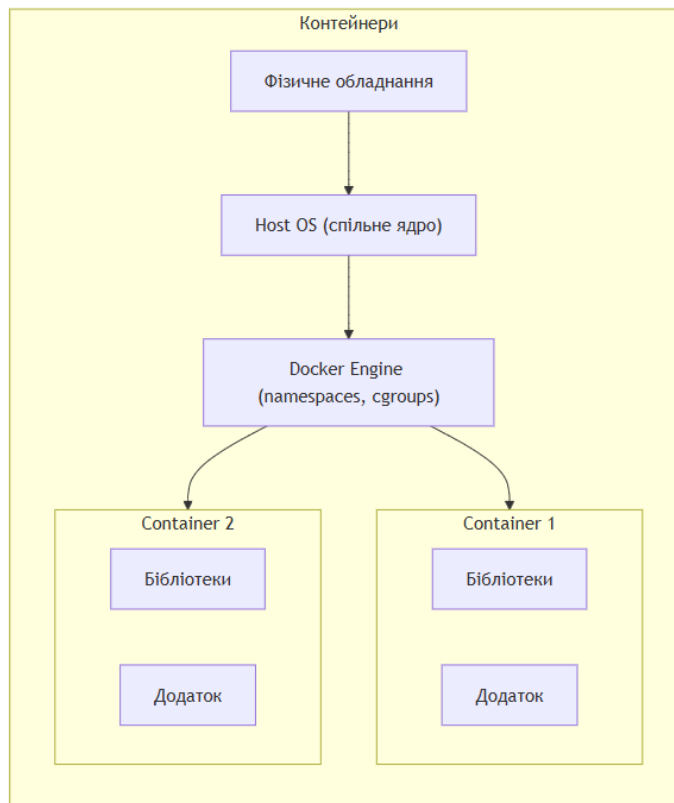


Рисунок 1.1.2. Концептуальна схема архітектури віртуальної машини.

Бачимо, що кожен контейнер містить додаток, його конфігураційні файли, бібліотеки, змінні середовища, системні утиліти та все необхідне для його виконання, при цьому використовуючи ядро операційної системи хост-машини спільно з іншими контейнерами. «Container image - це легкий, автономний, виконуваний пакет програмного забезпечення, що містить усе необхідне для запуску додатка» [2].

Контейнери працюють за допомогою механізмів ядра ОС, таких як простори імен (namespaces) та контрольні групи (cgroups). Простори імен надають ізоляцію між процесами - кожен контейнер бачить власний набір процесів, мереж, точок монтування та користувачів. Це створює ілюзію повноцінної незалежної ОС. Контрольні групи регулюють ресурси - такі як процесорний час, пам'ять і мережевий трафік, доступні кожному контейнеру.

Коли запускається контейнер, він ініціалізує свій ізольований процес на основі образу, що зазвичай містить лише необхідний мінімум - наприклад, Alpine

Linux, Node.js, Python або власну програму. Контейнер використовує файлову систему `copy-on-write`, що дозволяє багаторазово використовувати один базовий образ для кількох контейнерів, зберігаючи унікальні зміни окремо. «Кожен контейнер використовує ядро хост-системи спільно з іншими контейнерами й виконується як ізольований процес у просторі користувача» [2].

Контейнери не включають власне ядро ОС, тому вони набагато легші за віртуальні машини. Завдяки цьому контейнеризація дозволяє досягти значної економії ресурсів, високої швидкості запуску (контейнери стартують за частки секунди) та високої щільності розгортання (багато контейнерів на одному сервері).

Запуск контейнерів зазвичай відбувається за допомогою середовища виконання, такого як `Docker Engine` або `containerd`. Управління життєвим циклом контейнерів - створення, запуск, зупинка, видалення - здійснюється за допомогою командного рядка або API. Для роботи з великою кількістю контейнерів використовуються системи оркестрації, як-от `Kubernetes`, які автоматизують масштабування, самовідновлення, оновлення та мережеву взаємодію між контейнерами.

Варто зазначити, що `Docker` є не єдиним інструментом для роботи з контейнерами. Серед альтернативних рішень слід виділити `Podman` - безсерверний (`daemonless`) та безкореневий (`rootless`) інструмент, розроблений компанією `Red Hat`, який забезпечує повну сумісність з `Docker CLI` та образами `OCI`. `Containerd`, який є проєктом `Cloud Native Computing Foundation`, використовується як низькорівневе середовище виконання в `Docker` та `Kubernetes`. `CRI-O` - ще одне середовище виконання, оптимізоване спеціально для `Kubernetes` і сумісне зі стандартом `Container Runtime Interface (CRI)`. Також існують `LXC` та `LXD` - технології для створення системних контейнерів, що забезпечують ізоляцію на рівні повноцінної операційної системи. Ці альтернативи розширюють можливості вибору інструментів залежно від конкретних вимог до безпеки, продуктивності та інтеграції з існуючою інфраструктурою.

Саме тому у сучасних ІТ-системах контейнеризація стала фундаментальною технологією, яка змінює підходи до розробки, тестування, доставки та масштабування програмного забезпечення. Це зумовлено її здатністю забезпечувати високу швидкість розгортання, що критично важливо для безперервної інтеграції та доставки (CI/CD) та динамічного масштабування в хмарних середовищах. Контейнери забезпечують гнучкість і портативність, дозволяючи один і той самий контейнер запускати локально, у тестовому середовищі або в продакшн-середовищі без змін, що значно знижує ризики, пов'язані з несумісністю середовищ.

1.2 Обґрунтування необхідності застосування механізмів безпеки в контейнеризованих середовищах

Згідно з опитуванням, проведеним Portworx та Aqua Security, 87% ІТ-фахівців повідомили про використання контейнерних технологій, причому 90% з них запустили контейнери у виробничому середовищі. Крім того, 70% респондентів зазначили, що принаймні 40% їхніх додатків працюють у контейнерах. Також 24% компаній витрачали понад 500 тисяч доларів на рік на впровадження контейнерних рішень, а 17% - понад 1 мільйон доларів, що майже вчетверо більше, ніж у 2016 році. Основними викликами, які відзначали респонденти, були захист даних (61%), управління вразливостями (43%) та безпека під час виконання (34%) [1]

Хоча Docker пропонує потужні інструменти для ізоляції та управління додатками, питання безпеки залишається одним із найважливіших аспектів його використання. Це обумовлено широким використанням спільноти для створення образів, відкритістю платформи та складністю сучасного ландшафту вразливостей. Безпека в Docker включає кілька рівнів: безпека самого середовища Docker Engine, захист контейнерів під час виконання, а також безпека образів, що зберігаються в Docker Hub.

У дослідженні «Vulnerability Analysis of 2500 Docker Hub Images» було виявлено, що вразливості в контейнерах - це не виняток, а радше правило. Дослідники

проаналізували 2412 образів і з'ясували, що лише 17,8% не містили жодної вразливості, тоді як 82% сертифікованих образів містили хоча б одну критичну або високу вразливість [5] (рис 1.2.1).



Рисунок 1.2.1. Співвідношення вразливостей в проаналізованих образах

У середньому образ містив 151 вразливість, а медіанне значення становило 26. Найнебезпечніші вразливості образів контейнерів пов'язані з популярними мовами програмування: JavaScript (наприклад, бібліотека Lodash) та Python (особливо версії 2.x). Найчастіше фіксувались вразливості, пов'язані з виконанням довільного коду, переповненням буфера, та неналежною обробкою введення. Це робить проблему кібербезпеки в Docker особливо актуальною.

Особливе занепокоєння викликає той факт, що сертифіковані образи виявилися найбільш уразливими за медіанним значенням кількості вразливостей. Це суперечить очікуванням, адже сертифікація передбачає відповідність високим стандартам якості. Крім того, велика частка образів (31,4%) не оновлювалися понад 400 днів, що створює умови для збереження відомих вразливостей у продакшн-середовищах [5] (рис. 1.2.2).



Рисунок 1.2.2. Актуальність оновлень образів

Ще одне критичне спостереження - Docker Daemon працює з root-привілеями, що розширює потенційну поверхню атаки. Достатньо однієї вразливої компоненти

в контейнері, щоб зловмисник міг підвищити привілеї й скомпрометувати всю систему.

Ілюстрацією важливості дослідження безпеки в середовищі Docker є проєкт DIVA (Damn insecure Vulnerable App), спеціально створений як інструмент для практичного тестування захисту контейнеризованих додатків. Цей проєкт симулює середовище, яке містить навмисно вразливі компоненти, що дозволяє дослідникам та інженерам з безпеки ідентифікувати, експлуатувати й усувати типові загрози. Завдяки DIVA стало можливим моделювати атаки, перевіряти ефективність захисних механізмів і формувати практичні навички безпечної роботи з Docker.

У дослідженні також було виокремлено найбільш поширені критичні вразливості, що фіксуються в Docker-образах. Серед них:

- CVE-2019-5021 - вразливість у Alpine Linux, яка дозволяє зловмиснику створити користувача root без пароля, що відкриває шлях до повного контролю над контейнером;
- CVE-2018-15664 - пов'язана з небезпечним способом обробки символічних посилань у Docker, що може бути використано для доступу до файлів поза межами контейнера;
- CVE-2019-5736 - експлойт, що дозволяє контейнеру перезаписати свій власний runtime, що в деяких випадках може призвести до втечі з контейнера (container escape).

Наявність таких вразливостей демонструє, що навіть популярні базові образи можуть містити серйозні загрози безпеці, і підкреслює потребу в ретельному аналізі, обмеженні привілеїв і системному оновленні компонентів.

У статті також описано практичні приклади використання таких вразливостей. Зокрема, дослідники наводять сценарій експлуатації CVE-2019-5021: зловмисник може автоматично отримати доступ до контейнера з Alpine Linux без автентифікації, оскільки обліковий запис root у цьому контейнері не має пароля. Інший приклад - використання CVE-2018-15664 для доступу до файлів хост-системи поза межами контейнера через недосконалу обробку символічних посилань. Ще один

кейс - експлуатація CVE-2019-5736, де шкідливий контейнер може перезаписати виконуваний файл `gunc`, що дозволяє втечу з контейнера та виконання команд на хості.

Ці приклади підкреслюють, що теоретичні вразливості мають цілком практичні наслідки і можуть використовуватись у реальних атаках, якщо не впроваджено належного захисту.

Для ідентифікації, відстеження й управління вразливостями у відкритому програмному забезпеченні використовується база даних загальних вразливостей - CVE (Common Vulnerabilities and Exposures) - це публічний реєстр, що містить відомості про відомі вразливості в програмному забезпеченні та апаратному забезпеченні. Ініціатива була започаткована у 1999 році за підтримки MITRE Corporation у співпраці з урядовими структурами США, включно з Національним інститутом стандартів і технологій (NIST). Метою CVE є забезпечення єдиного стандарту ідентифікації вразливостей, що дозволяє системам безпеки по всьому світу обмінюватися інформацією, порівнювати дані і координувати дії щодо усунення загроз. Кожна CVE-запис має унікальний ідентифікатор, рік виявлення, номер та короткий опис проблеми. Наприклад, CVE-2019-5736 описує критичну вразливість у середовищі контейнеризації, яка дозволяє втечу з контейнера через маніпуляцію з виконуваним файлом `gunc`. Завдяки стандартизованому підходу база CVE стала основою для більшості сканерів вразливостей і систем керування інформаційною безпекою (SIEM). Вона підтримується спільнотою та координаційним центром MITRE і є загальновизнаним стандартом для класифікації вразливостей. Кожна вразливість, зареєстрована в CVE, має унікальний ідентифікатор (наприклад, CVE-2019-5736), короткий опис і посилання на ресурси з технічними деталями або способами усунення.

1.3 Постановка завдання дослідження

На основі проведеного аналізу теоретичних засад контейнеризації та обґрунтування необхідності застосування механізмів безпеки в контейнеризованих середовищах, сформульовано наступні завдання дослідження.

1. Проаналізувати архітектуру та інфраструктуру Docker з метою виявлення ключових компонентів, що впливають на безпеку контейнеризованих додатків, та визначення потенційних векторів атак.

2. Дослідити рекомендації міжнародних організацій щодо забезпечення безпеки контейнеризованих додатків та визначити найбільш критичні аспекти захисту.

3. Визначити та розробити методологію налаштування безпеки Docker-контейнерів.

4. Імплементувати та верифікувати механізми безпеки.

Висновки до розділу 1

У першому розділі кваліфікаційної роботи проведено теоретико-методологічний аналіз проблематики безпеки контейнеризованих систем.

Визначено роль та значення контейнеризації в сучасних інформаційних системах. Встановлено, що контейнерні технології стали фундаментальною складовою сучасної IT-інфраструктури, забезпечуючи високу швидкість розгортання, портативність додатків та ефективне використання ресурсів. Згідно з результатами опитування Portworx та Aqua Security, 87% IT-фахівців використовують контейнерні технології, при цьому 90% запускають контейнери у виробничому середовищі.

Обґрунтовано необхідність застосування механізмів безпеки в контейнеризованих середовищах. На основі аналізу дослідження «Vulnerability Analysis of 2500 Docker Hub Images» встановлено, що 82% образів містять хоча б одну критичну або високу вразливість, а в середньому образ містить 151 вразливість. Особливе занепокоєння викликає той факт, що навіть сертифіковані образи виявилися найбільш уразливими за медіанним значенням кількості вразливостей.

Розглянуто найпоширеніші критичні вразливості Docker-образів (CVE-2019-5021, CVE-2018-15664, CVE-2019-5736) та описано практичні сценарії їх експлуатації, включаючи можливість втечі з контейнера (container escape) та отримання несанкціонованого доступу до хост-системи.

На підставі проведеного аналізу сформульовано завдання дослідження, спрямовані на розробку та імплементацію комплексних механізмів захисту контейнеризованих додатків на базі Docker.

Розділ 2. Огляд інструментів забезпечення безпеки в Docker

2.1. Аналіз інфраструктури Docker

Docker - це повноцінна екосистема, яка охоплює всі етапи роботи з контейнерами: від розробки до тестування, розгортання, масштабування і супроводу програмного забезпечення. Вона пропонує не лише інструменти для запуску контейнерів, а й механізми для зберігання образів, управління залежностями, інтеграції з CI/CD-пайплайнами, моніторингу та безпеки. Основна цінність Docker полягає в стандартизації середовища виконання, що дозволяє створювати передбачувані, портативні та ефективні додатки незалежно від середовища, де вони запускаються: чи то локальна машина розробника, сервер у дата-центрі чи хмарна платформа. Це забезпечує безперервність і стабільність на всіх етапах життєвого циклу додатка. У документації зазначається: «Docker - це відкрита платформа для розробки, доставки та запуску додатків» [4].

Розглянемо концептуальну схему архітектури Docker (рис 1.2.1).

Архітектура Docker включає три основні складові: Docker-клієнт (CLI), Docker Daemon і Docker Registry.

Клієнт - це інтерфейс командного рядка, через який користувачі надсилають команди. Коли команда запускається, наприклад `docker run`, вона передається до Docker Daemon - фоновій служби, яка виконує основну роботу: створює, запускає, зупиняє та видаляє контейнери. Docker Daemon також взаємодіє з реєстрами образів Docker Registry, з яких він може завантажити або зберегти образ.

Архітектура також передбачає використання інструмента Docker Compose, що дозволяє описувати багатоконтейнерні додатки в одному YAML-файлі та керувати ними як єдиним цілим. У складніших сценаріях до Docker можуть бути підключені системи оркестрації, такі як Kubernetes, які розподіляють контейнери по кластерах серверів, забезпечують балансування навантаження, автоматичне масштабування та відновлення.

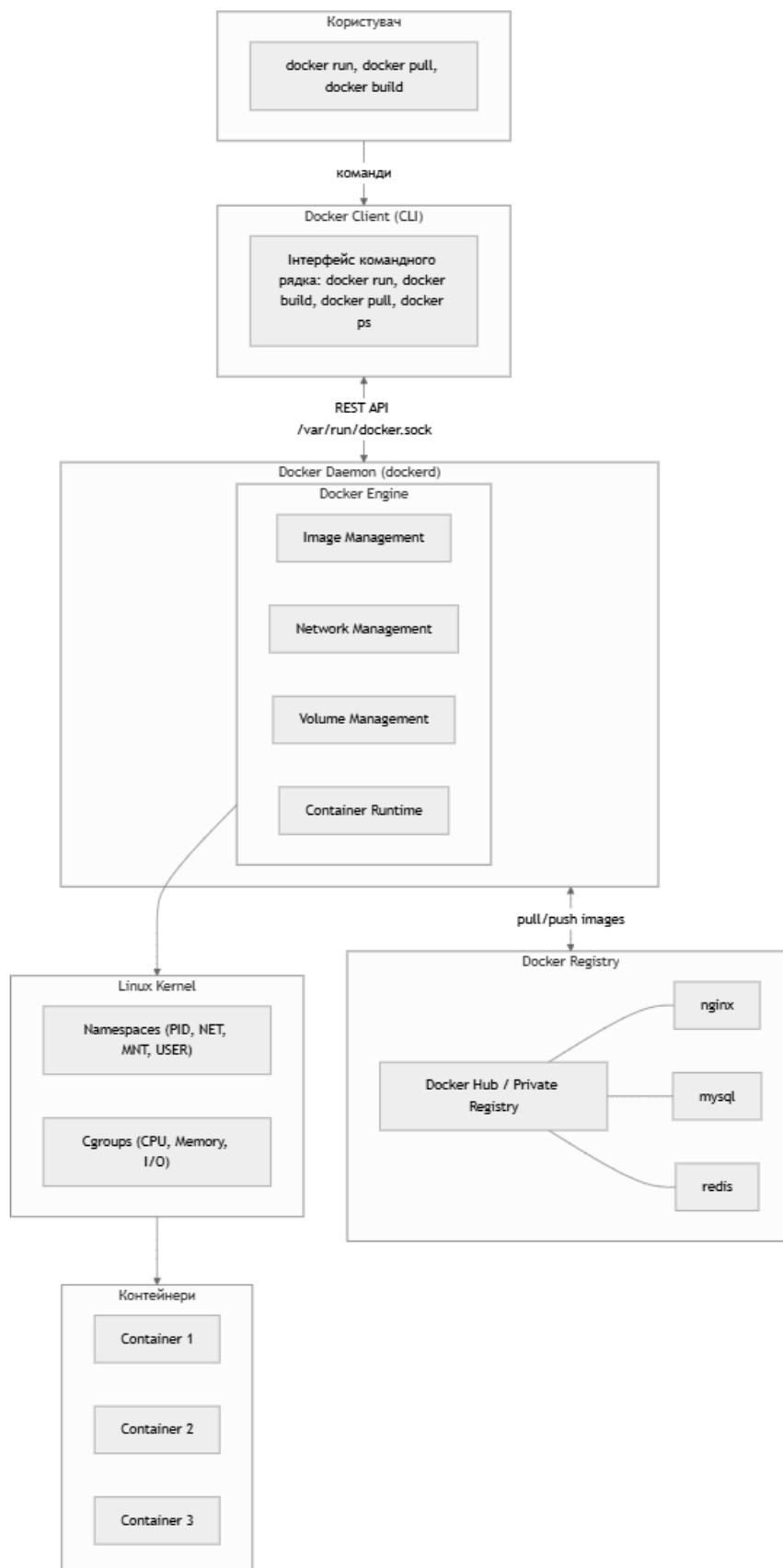


Рисунок 1.2.1. Архітектура Docker

Контейнери працюють поверх Docker Engine - легковагового середовища виконання контейнерів. Воно реалізує основні механізми Linux для ізоляції (namespaces, cgroups), які дозволяють контейнерам бути незалежними один від одного та від хост-системи. Завдяки цьому забезпечується висока щільність розміщення, швидкий старт контейнерів і масштабованість.

Ця модульна та розподілена архітектура дозволяє гнучко масштабувати інфраструктуру, ефективно керувати життєвим циклом додатків та інтегрувати Docker із зовнішніми CI/CD-системами, хмарними сервісами та інструментами безпеки. Саме завдяки такій архітектурі Docker став основою сучасної контейнерної екосистеми.

Docker Desktop - це офіційний клієнт для Windows і macOS, який дозволяє локально створювати та запускати контейнери без потреби вручну налаштовувати середовище. Він включає Docker Engine, CLI, Docker Compose, Kubernetes (опційно) і графічний інтерфейс. Таким чином, Docker Desktop є ключовим інструментом для розробників, який забезпечує зручну взаємодію з інфраструктурою Docker.

Docker Registry - це сервер, який зберігає Docker-образи. Найпопулярніший публічний реєстр - Docker Hub. Він дозволяє розміщувати, шукати, завантажувати та ділитися контейнерами. Для приватного користування доступні альтернативи - як-от Harbor або самостійно розгорнуті реєстри.

На Docker Hub всі репозиторії умовно поділяються на чотири типи: Official, Verified, Certified та Community.

1. Official - це офіційні образи, які створюються та підтримуються безпосередньо компанією Docker або у співпраці з авторами програмного забезпечення. Вони проходять внутрішню перевірку безпеки та відповідності найкращим практикам.

2. Verified - підтверджені образи від сторонніх постачальників, які пройшли валідацію Docker і відповідають визначеним критеріям якості. Вони мають позначку верифікації, яка дозволяє користувачам швидше ідентифікувати надійний контент.

3. **Certified** - це підмножина **verified**-образів, які додатково сертифіковані на відповідність суворим вимогам сумісності, підтримки та безпеки. Вони, як правило, використовуються в корпоративних середовищах, де потрібна офіційна підтримка постачальника.

4. **Community** - образи, створені користувачами спільноти. Вони не проходять перевірку або аудит з боку Docker і можуть містити застарілі чи вразливі компоненти. Незважаючи на це, вони є джерелом великої кількості рішень та прикладів і становлять переважну більшість контенту Docker Hub.

Розуміння різниці між цими типами дозволяє приймати обґрунтовані рішення при виборі контейнерного образу, особливо у контексті вимог до безпеки, підтримки та довгострокової стабільності.

Образи (**Images**) - це шаблони, з яких створюються контейнери. Образ є лише для читання і містить усе необхідне для запуску додатка: код, бібліотеки, змінні середовища. Образи створюються з **Dockerfile** і можуть бути повторно використані для створення ідентичних екземплярів додатків.

Контейнери (**Containers**) - це виконувані екземпляри образів. Вони запускаються як ізольовані процеси на спільному ядрі ОС. Завдяки цьому контейнери є легшими за віртуальні машини, швидко стартують та не дублюють зайвих системних компонентів. У документації наголошується: «Docker дозволяє упакувати та запускати додаток в умовно ізольованому середовищі, яке називається контейнером» [4].

На базі Docker можна розробляти нові додатки, створювати автоматизовані CI/CD-пайплайни, переносити програми між різними середовищами (локально, staging, production), забезпечувати масштабування та безперервне оновлення сервісів у кластерних системах. Завдяки своїй гнучкості Docker став стандартом у DevOps-практиках та хмарній інфраструктурі.

Таким чином, Docker - це провідна платформа, яка назва якої часто вживається як синонім контейнеризації у сучасному програмному забезпеченні. Її основна ідея полягає в тому, щоб надати розробникам та адміністраторам інструмент

для створення, запуску та управління контейнеризованими додатками незалежно від середовища, у якому вони будуть працювати.

Головна цінність Docker полягає у стандартизації середовища виконання, що дозволяє зменшити проблеми, пов'язані з різними конфігураціями систем. Завдяки цьому забезпечується висока сумісність між етапами розробки, тестування та впровадження програм. Розробник може бути впевнений, що додаток, який працює у контейнері на його локальній машині, працюватиме так само і на сервері в хмарі.

Платформа також спрощує масштабування інфраструктури, підтримує автоматизацію процесів CI/CD, і дозволяє значно підвищити ефективність управління життєвим циклом додатків. Як підкреслюється у документації: «Docker дозволяє упакувати та запускати додаток в умовно ізольованому середовищі, яке називається контейнером» [4].

Таким чином, Docker зробив контейнеризацію доступною, простою у використанні і ефективною, що забезпечило її стрімке поширення у всіх сферах розробки програмного забезпечення - від стартапів до великих корпорацій.

2.2 Основні компоненти інфраструктури безпеки Docker

Docker надає комплексну інфраструктуру безпеки, яка охоплює різні рівні взаємодії: від ядра операційної системи до управління правами доступу користувачів у хмарному середовищі. Ця багаторівнева модель дозволяє значно знизити ризики, пов'язані з використанням контейнерів у виробничих середовищах.

Розглянемо цю модель (рис. 2.2.1).

Бачимо, що на рівні ядра ОС Docker використовує механізми ізоляції - namespaces (простори імен) та cgroups (контрольні групи), які дозволяють кожному контейнеру працювати у власному логічному середовищі. Простори імен забезпечують ізоляцію таких компонентів, як процеси, файлові системи, мережі та користувачі. У документації наголошується: «Простори імен забезпечують першу і най-

простішу форму ізоляції в Docker. Вони гарантують, що кожен контейнер має власний вигляд системи.» [6]. Це означає, що процеси одного контейнера не можуть бачити процеси іншого контейнера чи взаємодіяти з ними.

Контрольні групи, у свою чергу, дозволяють обмежувати і розподіляти апаратні ресурси між контейнерами. Docker використовує cgroups для визначення, скільки CPU, пам'яті, вводу/виводу чи мережевого трафіку дозволено кожному контейнеру. Таким чином, навіть при надмірному навантаженні один контейнер не зможе заблокувати ресурси всієї системи. Як зазначається в документації: «Контрольні групи (cgroups) забезпечують обмеження ресурсів, визначення пріоритетів, облік та контроль» [6].

Ці механізми є фундаментом безпеки на рівні ядра, завдяки якому кожен контейнер працює ізольовано, не порушуючи цілісності інших процесів у системі [9].

На рівні ядра безпеки контейнера, Docker Engine підтримує інтеграцію з модулями безпеки ядра Linux - AppArmor та SELinux, які дозволяють встановлювати детальні політики контролю доступу до ресурсів системи[11][12][13].

Ці політики обмежують поведінку контейнера та визначають, до яких файлів, мережевих інтерфейсів або системних викликів він має доступ.

У документації зазначається: «Модулі безпеки, такі як AppArmor і SELinux, забезпечують систему обов'язкового контролю доступу (MAC), яка обмежує можливості програм за допомогою набору політик безпеки.» [6]. Це означає, що навіть у разі компрометації контейнера, впроваджені обмеження не дозволять зловмиснику вийти за межі визначених дій або отримати привілеї, які можуть зашкодити хост-системі.

Крім того, Docker може бути налаштований з профілями безпеки за замовчуванням, які автоматично застосовуються до нових контейнерів. Наприклад, профіль AppArmor «docker-default» забороняє виконання деяких операцій, що несуть потенційний ризик, таких як встановлення модулів ядра або доступ до пристроїв. [12] Ці інструменти формують потужний захисний прошарок, який працює незалежно від користувача і значно підвищує стійкість до атак.

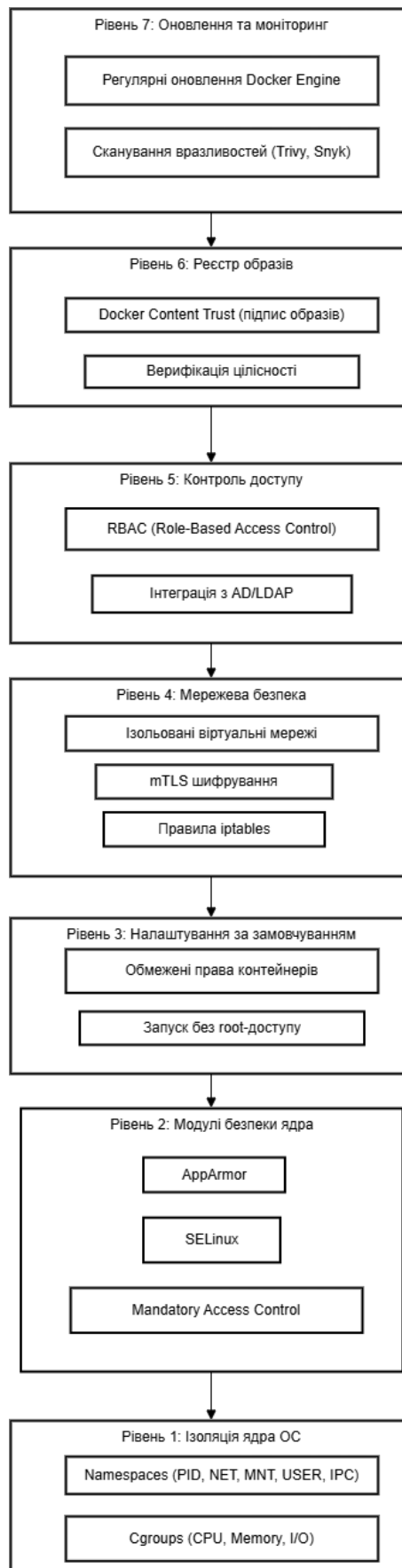


Рисунок 2.2.1. Багаторівнева модель безпеки Docker

Крім того, Docker реалізує принцип безпечних за замовчуванням налаштувань. Це означає, що новостворені контейнери за замовчуванням працюють з обмеженими правами, без root-доступу до хост-системи, що мінімізує можливі наслідки в разі їх компрометації. У документації зазначено: «Docker за замовчуванням запускає контейнери з обмеженими правами, що допомагає зменшити вплив атак на прорив контейнерів.» [6]. Такі налаштування передбачають чітко визначені дозволи на мережеві інтерфейси, точки монтування та файлову систему, а також заборону доступу до чутливих пристроїв.

Для контролю доступу до ресурсів і управління правами користувачів Docker пропонує механізм контролю доступу на основі ролей (Role-Based Access Control RBAC). Він дозволяє створювати різні ролі з конкретними дозволами, обмежуючи доступ до певних дій або ресурсів, залежно від потреб користувача чи сервісу. У документації наголошується: «RBAC дозволяє адміністраторам визначати детальні політики доступу, узгоджуючи права користувачів з ролями в організації.» [6]. У корпоративних середовищах RBAC може бути інтегрований з зовнішніми системами аутентифікації, такими як Active Directory або LDAP, що забезпечує централизоване управління доступом і відповідність політикам інформаційної безпеки.

На рівні реєстрів образів реалізовано механізм підпису та верифікації образів (Docker Content Trust), який гарантує, що завантажений образ не був змінений та справді створений довіреною особою. [14] Також Docker підтримує інтеграцію з інструментами сканування вразливостей, які можуть бути автоматично застосовані під час завантаження образів у Docker Hub або приватні реєстри.

У сфері мережевої безпеки, Docker підтримує створення ізольованих віртуальних мереж, шифрування міжвузлового трафіку (overlay network encryption) і керування міжконтейнерними з'єднаннями за допомогою правил фаєрволу (iptables). Ізольовані мережі дозволяють контейнерам взаємодіяти тільки в межах визначеної віртуальної мережі, що унеможливорює небажаний доступ ззовні. У документації підкреслюється: «Docker надає кілька мережевих драйверів, і кожен мережевий драйвер пропонує різні функції та рівні ізоляції.» [6].

Docker також підтримує шифрування трафіку в overlay-мережах, що особливо важливо в розподілених середовищах, де контейнери можуть знаходитися на різних фізичних вузлах. Як зазначається: «Режим Docker Swarm включає вбудовану підтримку взаємного TLS (mTLS) для шифрованого зв'язку між вузлами.» [6]. Це дозволяє забезпечити автентифікацію і шифрування з'єднання між вузлами кластеру, підвищуючи загальну безпеку інфраструктури.

Крім цього, адміністратор має можливість налаштовувати правила iptables або використовувати вбудовані механізми Docker для управління доступом до портів, дозволяючи або блокуючи вхідний і вихідний трафік для конкретних контейнерів. Завдяки цьому зменшується ймовірність несанкціонованого доступу або зловмисного сканування мережі.

І нарешті, важливою складовою інфраструктури безпеки Docker є регулярні оновлення Docker Engine, які виправляють виявлені вразливості та вдосконалюють механізми захисту. Згідно з документацією, «Режим Docker Swarm включає вбудовану підтримку взаємного TLS (mTLS) для шифрованого зв'язку між вузлами.» [6]. Це вимагає від адміністраторів уважного моніторингу релізів і швидкого впровадження патчів у виробничих середовищах.

Крім цього, важливими принципами безпеки є розмежування привілеїв (privilege separation) і принцип найменших привілеїв (least privilege). Docker рекомендує запускати контейнери без root-доступу і з обмеженим набором дозволів. «Використання контейнерів без прав суперкористувача та застосування принципів мінімальних привілеїв допомагає зменшити потенційну поверхню атаки» [6].

Таким чином, безперервне оновлення компонентів, автоматизоване виявлення вразливостей та дотримання сучасних практик доступу - це невід'ємні елементи ефективної безпеки в екосистемі Docker.

2.3 Рекомендації OWASP щодо підвищення кібербезпеки контейнеризованих додатків

Організація OWASP (Open Worldwide Application Security Project) - це глобальна некомерційна спільнота, яка зосереджена на підвищенні безпеки програмного забезпечення. Її місія полягає у створенні відкритих ресурсів, інструментів, методологій і стандартів, що допомагають розробникам, тестувальникам, адміністраторам та іншим фахівцям впроваджувати безпечні практики у свої проєкти. Найвідомішим внеском OWASP є рейтинг OWASP Top 10 - список найпоширеніших вразливостей вебдодатків. Крім цього, OWASP публікує серію практичних порад із безпеки в різних технологіях, включаючи Docker.

У документі «Docker Security Cheat Sheet» [7] організація надала рекомендації для захисту контейнеризованих середовищ. У цьому матеріалі окреслено ключові принципи захищеної роботи з Docker, які варто враховувати для зменшення ризиків.

Однією з базових рекомендацій є застосування принципу найменших привілеїв, що передбачає запуск контейнерів із мінімальним необхідним рівнем доступу до ресурсів хост-системи. Слід переконатися, що контейнери не мають надмірних дозволів, а всі процеси виконуються від імені звичайних, а не root-користувачів. Це зменшує ризики, пов'язані з можливим проривом ізоляції контейнера, а також обмежує масштаб потенційного збитку в разі експлуатації вразливості. У документі OWASP наголошується: «Уникайте запуску контейнерів як користувач root. Завжди використовуйте користувача, що не є root, щоб зменшити вплив виходу контейнера за межі.» [7]. Крім того, рекомендовано використовувати механізми обмеження системних викликів, які унеможливають виконання критично небезпечних дій, навіть якщо контейнер буде зламано. Реалізація принципу найменших привілеїв є базовим кроком до впровадження моделі Zero Trust у контейнерному середовищі..

OWASP також радить використовувати лише перевірені базові образи, оскільки використання образів із сумнівних джерел значно підвищує ризик впровадження шкідливого або вразливого коду в систему. Образи з офіційних або верифікованих репозиторіїв мають більшу ймовірність проходження аудитів безпеки, дотримання стандартів якості та своєчасного оновлення. Крім того, організація рекомендує застосовувати механізми цифрового підпису (наприклад, Docker Content Trust), які дозволяють перевіряти достовірність походження та цілісність образу перед його розгортанням. Це значно знижує ризик несанкціонованої модифікації образу або експлуатації прихованих вразливостей у ньому.

Ще одна важлива рекомендація - відокремлення привілейованих контейнерів. Контейнери, яким необхідні підвищені права (наприклад, доступ до пристроїв або ядра системи), не повинні розміщуватися поруч з менш привілейованими контейнерами в загальному середовищі. Замість цього OWASP радить створювати спеціальні ізольовані середовища, де такі контейнери функціонуватимуть окремо, з додатковими шарами моніторингу та контролю доступу. Це дозволяє локалізувати ризики у разі зламу або експлуатації вразливостей. Таким чином, мінімізація використання привілейованих контейнерів і чітке відокремлення таких процесів допомагає зменшити поверхню атаки та зберегти ізоляцію між сервісами.

Також OWASP закликає не зберігати паролі та секрети у Dockerfile або змінних середовища у відкритому вигляді. Натомість слід використовувати секрет-менеджери, зокрема Vault або інтеграцію з Kubernetes Secrets.

Додаткові рекомендації включають:

- встановлення read-only файлової системи для контейнерів, де це можливо;
- мінімізація відкритих портів та мережевих шляхів;
- регулярне оновлення залежностей та зменшення кількості встановлених компонентів в образах.

OWASP підсумовує: «Безпека повинна бути вбудована в цикл розробки, а не додаватися пізніше. Безпека контейнерів — це безперервний процес». [7]. Тобто

безпека повинна стати частиною розробки, а не лише етапом перед продакшн-розгортанням.

2.4 План впровадження механізмів безпеки Docker

На основі аналізу інфраструктури безпеки Docker та рекомендацій OWASP, розробимо план комплексного захисту контейнеризованих додатків, що базується на принципі "Defense in Depth" (ешелонований захист) [15]. Методологія передбачає реалізацію механізмів безпеки на п'яти рівнях: рівень хост-системи, рівень Docker Daemon, рівень образу, рівень контейнера та рівень додатку.

На підставі проведеного аналізу загроз та існуючих механізмів захисту сформовано план впровадження, що систематизує 18 ключових налаштувань безпеки за рівнями реалізації та пріоритетністю (табл. 2.4.1).

Таблиця 2.4.1 - План впровадження механізмів безпеки Docker

ID	Рівень	Механізм захисту	Загроза, що нейтралізується	Пріоритет	Файл конфігурації
Рівень 1: Хост-система та Docker Daemon					
DS001	Daemon	User Namespace Remapping	Container Escape, Privilege Escalation	Високий	daemon.json
DS002	Daemon	Захист Docker Socket	Несанкціонований доступ до API	Критичний	daemon.json
DS003	Daemon	TLS-автентифікація	Man-in-the-Middle, Spoofing	Високий	daemon.json + certs
DS004	Daemon	Вимкнення ICC	Lateral Movement між контейнерами	Середній	daemon.json
Рівень 2: Docker-образ					
DS005	Image	Multi-stage Build	Розширена поверхня атаки	Високий	Dockerfile
DS006	Image	Мінімальний базовий образ	Вразливості в залежностях	Високий	Dockerfile
DS007	Image	Non-root користувач	Privilege Escalation	Критичний	Dockerfile
DS008	Image	Фіксовані версії образів	Supply Chain Attack	Високий	Dockerfile
DS009	Image	Сканування на вразливості	Відомі CVE	Критичний	CI/CD Pipeline

Рівень 3: Контейнер (Runtime)					
DS010	Container	Linux Capabilities (DROP ALL)	Privilege Escalation	Критичний	docker-compose.yml
DS011	Container	Seccomp Profile	Container Escape, Syscall Abuse	Високий	seccomp-profile.json
DS012	Container	AppArmor Profile	Несанкціонований доступ до ресурсів	Високий	apparmor-profile
DS013	Container	No-new-privileges	Privilege Escalation через SUID	Критичний	docker-compose.yml
DS014	Container	Read-only файлова система	Persistence, Malware Injection	Високий	docker-compose.yml
DS015	Container	Обмеження ресурсів (CPU, RAM, PIDs)	DoS, Resource Exhaustion	Середній	docker-compose.yml
Рівень 4: Мережа та моніторинг					
DS016	Network	Ізольовані мережі	Lateral Movement	Середній	docker-compose.yml
DS017	Monitoring	Health Checks	Відмова сервісу, Silent Failure	Середній	Dockerfile, docker-compose.yml
DS018	Monitoring	Обмеження логування	Disk Exhaustion, Log Injection	Низький	docker-compose.yml
Рівень 5: Додаток (Application Layer)					
DS019	App	Security Headers	XSS, Clickjacking, MIME-sniffing	Високий	nginx.conf
DS020	App	Rate Limiting	DoS, Brute Force	Середній	nginx.conf
DS021	App	Обмеження розміру запитів	Buffer Overflow, DoS	Середній	nginx.conf

Тепер, на основі плану розроблено чек-лист для практичної реалізації.

DOCKERFILE

1. Використано конкретний тег образу (не :latest)
2. Застосовано multi-stage build
3. Створено непривілейованого користувача (USER)
4. Мінімізовано кількість шарів та розмір образу
5. Видалено зайві пакети та кеш
6. Налаштовано HEALTHCHECK

DOCKER-COMPOSE / RUN

1. Вказано user: "UID:GID"
2. Додано security_opt: no-new-privileges:true
3. Додано security_opt: seccomp=profile.json
4. Встановлено cap_drop: ALL
5. Додано лише необхідні cap_add
6. Увімкнено read_only: true
7. Налаштовано tmpfs для директорій запису
8. Встановлено обмеження resources.limits (cpu, memory, pids)
9. Налаштовано healthcheck
10. Обмежено logging (max-size, max-file)

DOCKER DAEMON

1. Налаштовано userns-remap
2. Вимкнено icc (icc: false)
3. Обмежено доступ до docker.sock
4. Налаштовано TLS для віддаленого доступу

APPLICATION (NGINX)

1. Вимкнено server_tokens
2. Налаштовано Security Headers (17 headers)
3. Встановлено rate limiting
4. Обмежено розмір запитів
5. Заборонено доступ до прихованих файлів

CI/CD

1. Увімкнено Docker Content Trust
2. Проаналізовано вразливості через Docker Skout
3. Автоматизовано перевірку безпеки

Висновки до розділу 2

У другому розділі проведено комплексний аналіз інфраструктури безпеки Docker та розроблено стратегічний план захисту контейнеризованих додатків.

Дослідження архітектури Docker показало, що платформа забезпечує модульну та розподілену екосистему, яка охоплює всі етапи життєвого циклу додатків

- від розробки до продакшн-розгортання. Проте, незважаючи на вбудовані механізми ізоляції, контейнери залишаються вразливими до низки загроз, що потребує системного підходу до забезпечення їх безпеки.

Аналіз компонентів інфраструктури безпеки Docker виявив п'ятирівневу модель захисту, що базується на принципі "Defense in Depth". На рівні ядра ОС використовуються механізми namespaces та cgroups для ізоляції процесів та обмеження ресурсів. На рівні ядра безпеки контейнера застосовуються модулі AppArmor та SELinux для контролю доступу. Docker Engine реалізує принцип безпечних за замовчуванням налаштувань, а механізм RBAC забезпечує управління правами користувачів. Рівень реєстрів образів захищається через Docker Content Trust, а мережева безпека досягається через ізольовані віртуальні мережі та шифрування трафіку.

На основі рекомендацій OWASP систематизовано ключові принципи захисту контейнерних середовищ: застосування найменших привілеїв, використання перевірених базових образів, відокремлення привілейованих контейнерів, безпечне управління секретами та впровадження безпеки на етапі розробки. Ці принципи формують фундамент для побудови стійкої до атак інфраструктури.

Розроблений план впровадження механізмів безпеки Docker систематизує 21 налаштування за п'ятьма рівнями реалізації: хост-система та Docker Daemon, Docker-образ, контейнер на етапі виконання, мережа та моніторинг, рівень додатку. Кожен механізм класифіковано за пріоритетністю та співвіднесено з конкретними загрозами, що нейтралізуються.

Таким чином, створено методологічну основу для практичної реалізації захищеної контейнерної інфраструктури, що буде детально розглянуто у наступному розділі на прикладі веб-додатку TechShop.

Розділ 3. Імплементация плану забезпечення безпеки в Docker.

3.1 Налаштування проекту та забезпечення безпеки відповідно розробленому плану безпеки

У цьому розділі детально описується практичний процес імплементції комплексних налаштувань безпеки для Docker контейнерів веб-додатку TechShop. Розглядається покрокова реалізація 18 основних механізмів захисту, включаючи налаштування Dockerfile, docker-compose.yml, конфігураційних файлів безпеки та автоматизованих скриптів розгортання. Кожен крок супроводжується конкретними прикладами конфігурацій, командами та результатами перевірок.

Для демонстрації практичного застосування розробленого плану безпеки було обрано тестовий веб-додаток TechShop. Це Single Page Application, створений на базі React, що імітує інтернет-магазин електроніки. Додаток включає базову функціональність: каталог товарів з фільтрацією за категоріями, пошук, кошик покупок та адаптивний інтерфейс.

Важливо зазначити, що вибір конкретного додатку не впливає на роботу застосованих механізмів безпеки - всі налаштування Docker, Seccomp-профілі, AppArmor-політики та конфігурації nginx є універсальними і можуть бути адаптовані для будь-якого веб-додатку, що обслуговується через nginx. TechShop слугує лише демонстраційним прикладом для верифікації коректності імплементції плану безпеки.

Першим кроком у створенні безпечного Dockerfile була реалізація multi-stage build. Це дозволяє мінімізувати розмір фінального образу та виключити з нього інструменти розробки та залежності, необхідні лише для збірки:

```
FROM node:20.10.0-alpine3.19 AS build
LABEL stage="build"
WORKDIR /app
```

```
# Копіюємо package files для кращого кешування
```

```
COPY package*.json ./
```

```
# Встановлюємо залежності
```

```
# --ignore-scripts запобігає виконанню потенційно шкідливих post-install скриптів
```

```
RUN npm ci --silent && \
```

```
    npm cache clean --force
```

```
# Копіюємо вихідний код
```

```
COPY public/ ./public/
```

```
COPY src/ ./src/
```

```
# Збираємо production build
```

```
RUN npm run build
```

Важливим моментом є використання конкретного тегу образу `node:20.10.0-alpine3.19` замість `latest`. Це забезпечує детермінованість збірки та запобігає несподіваним змінам при оновленні базового образу.

```
FROM nginx:1.25.3-alpine3.18 AS production
```

```
# Мітки для ідентифікації
```

```
LABEL maintainer="security@techshop.local" \
```

```
    version="1.0.0" \
```

```
    description="TechShop E-commerce - Secure Production" \
```

```
    security.hardened="true"
```

Використання Alpine Linux як базового образу значно зменшує розмір та поверхню атаки завдяки мінімальному набору пакетів.

Критично важливим кроком є оновлення всіх пакетів базового образу для усунення відомих вразливостей:

```
# Оновлення базових пакетів для усунення вразливостей
```

```
RUN apk update && \
```

```
    apk upgrade --no-cache && \
```

```
rm -rf /var/cache/apk/* /tmp/*
```

Команда ``apk upgrade --no-cache`` оновлює всі пакети до останніх версій без збереження кешу, що мінімізує розмір образу. Видалення кешу та тимчасових файлів також зменшує потенційні вектори атаки.

Одним з найважливіших налаштувань безпеки є запуск контейнера від імені непривілейованого користувача. Це значно зменшує ризик у разі компрометації контейнера.

```
# Створення непривілейованого користувача (DS002)
```

```
RUN addgroup -g 1001 -S appgroup && \
```

```
adduser -u 1001 -S -D -G appgroup -H -h /nonexistent -s /sbin/nologin appuser
```

Розберемо детально параметри команди:

- ``-g 1001`` - встановлення конкретного GID для уникнення конфліктів;
- ``-S`` - створення системного користувача;
- ``-D`` - користувач без пароля (не може входити в систему);
- ``-G appgroup`` - додавання до групи `appgroup`;
- ``-H`` - без домашньої директорії;
- ``-h /nonexistent`` - встановлення неіснуючої директорії як домашньої;
- ``-s /sbin/nologin`` - заборона входу в систему;

Перед перемиканням на непривілейованого користувача необхідно налаштувати права доступу до всіх директорій, які будуть використовуватися:

```
# Створення необхідних директорій
```

```
RUN mkdir -p /var/cache/nginx/client_temp \
            /var/cache/nginx/proxy_temp \
            /var/cache/nginx/fastcgi_temp \
            /var/cache/nginx/uwsgi_temp \
            /var/cache/nginx/scgi_temp \
            /var/log/nginx \
            /var/run \
```

```
/usr/share/nginx/html && \
```

Налаштування прав доступу:

```
chown -R appuser:appgroup /var/cache/nginx \
    /var/log/nginx \
    /var/run \
    /usr/share/nginx/html \
    /etc/nginx && \
```

```
# PID файл
```

```
touch /var/run/nginx.pid && \
```

```
chown appuser:appgroup /var/run/nginx.pid
```

Це забезпечує, що nginx зможе працювати від імені непривілейованого користувача без проблем з правами доступу:

Тепер, видалимо зайві файли та налаштування статичних ресурсів

```
# Видалення default конфігурації
```

```
RUN rm -rf /etc/nginx/conf.d/default.conf \
    /usr/share/nginx/html/*
```

```
# Копіюємо конфігурацію nginx
```

```
COPY --chown=appuser:appgroup security/nginx.conf /etc/nginx/nginx.conf
```

```
# Копіюємо збірку з build stage
```

```
COPY --from=build --chown=appuser:appgroup /app/build /usr/share/nginx/html
```

```
# Встановлюємо read-only права на статичні файли
```

```
RUN chmod -R 555 /usr/share/nginx/html && \
    chmod 644 /etc/nginx/nginx.conf
```

Встановлення прав `555` (read-only для всіх) на статичні файли запобігає їх модифікації навіть у разі компрометації контейнера.

Тепер, перемкнемося на непривілейованого користувача та налаштування порту:

```
# Перемикаємось на непривілейованого користувача
```

```
USER appuser
```

```
# Відкриваємо непривілейований порт
EXPOSE 8080
```

Використання порту 8080 замість стандартного 80 дозволяє запускати контейнер без необхідності в root привілеїв, оскільки порти нижче 1024 вимагають прав адміністратора.

Після чого налаштуємо Health check, що дозволить Docker автоматично перевіряти стан контейнера та перезапускати його у разі проблем:

```
# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
    CMD wget -q --spider http://localhost:8080/ || exit 1
```

Параметри:

1. `--interval=30s` - перевірка кожні 30 секунд
2. `--timeout=10s` - максимальний час очікування відповіді
3. `--start-period=5s` - період після запуску, коли помилки ігноруються
4. `--retries=3` - кількість спроб перед позначенням як unhealthy

Всі налаштування безпеки в Dockerfile успішно застосовані, тепер налаштуємо docker-compose.yml з комплексними механізмами захисту. Маємо базову структуру сервісу:

```
version: '3.8'
services:
  techshop:
    build:
      context: .
      dockerfile: Dockerfile
    args:
      - BUILDKIT_INLINE_CACHE=1
    image: techshop:secure
    container_name: techshop-secure
    hostname: techshop
```

Тут ми використали `'BUILDKIT_INLINE_CACHE=1'` що дозволить використовувати кеш при збірці, що прискорює процес розробки.

Після чого налаштуємо непривілейованого користувача:

```
# non-root користувач - DS002
user: "1001:1001"
```

Це налаштування забезпечує, що контейнер запускається від імені користувача з UID 1001 та GID 1001, що відповідає користувачу `'appuser'`, створеному в Dockerfile.

Тепер налаштуємо наступне:

1. `No-new-privileges`, що запобігає підвищенню привілеїв процесів всередині контейнера, навіть якщо виконувані файли мають `setuid/setgid` біти.
2. `Seccomp` - профіль що обмежує системні виклики, доступні контейнеру. Використовується користувацький профіль `'seccomp-profile.json'`, який дозволяє лише необхідні `syscalls` для роботи `nginx`.

```
security_opt:
  # Заборона отримання нових привілеїв
  - no-new-privileges:true
  # Seccomp профіль (обмеження системних викликів)
  - seccomp=security/seccomp-profile.json
```

Тепер обмежимо `Linux Capabilities`. Використаємо підхід `"deny by default"` - спочатку видаляються всі `capabilities`, потім додаються лише ті, що необхідні для роботи. Це мінімізує атакуючу поверхню. Маємо:

```
cap_drop:
  - ALL # Видаляємо ВСІ capabilities

cap_add:
  # Додаємо лише необхідні (для nginx на порту >=1024 не потрібні)
  - CHOWN      # Зміна власника файлів (для init)
  - SETGID     # Зміна групи
  - SETUID     # Зміна користувача
```

Тепер налаштуємо Read-only файловою систему, що запобігає модифікації системних файлів та конфігурацій.

```
read_only: true

# Tmpfs для директорій, що потребують запису (uid/gid=1001 для appuser)
tmpfs:
  - /tmp:rw,noexec,nosuid,size=64m,uid=1001,gid=1001
  - /var/cache/nginx:rw,noexec,nosuid,size=128m,uid=1001,gid=1001
  - /var/run:rw,noexec,nosuid,size=16m,uid=1001,gid=1001
  - /var/log/nginx:rw,noexec,nosuid,size=64m,uid=1001,gid=1001
```

Для директорій, які потребують запису (логи, кеш, тимчасові файли), використовуються tmpfs з додатковими обмеженнями:

1. `noexec` - заборона виконання файлів
2. `nosuid` - ігнорування setuid/setgid бітів
3. `size=XXm` - обмеження розміру
4. `uid=1001,gid=1001` - власник файлів відповідає non-root користувачу

Тепер обмежимо ресурси, що захистить систему від DoS атак та забезпечать справедливий розподіл ресурсів між контейнерами. Наступних ресурсів для нашого демонстраційного проекту повинно вистачити:

```
deploy:
  resources:
    limits:
      cpus: '1.0'           # Максимум 1 CPU
      memory: 256M          # Максимум 256 MB RAM
      pids: 100             # Максимум 100 процесів
    reservations:
      cpus: '0.25'          # Резервування 0.25 CPU
      memory: 64M           # Резервування 64 MB RAM
```

Альтернативний синтаксис для Docker Compose v2

```
mem_limit: 256m
mem_reservation: 64m
```

```
cpus: 1.0
pids_limit: 100
```

Тепер, використаємо Ulimits, що обмежить кількість відкритих файлів та процесів, що запобігає вичерпанню ресурсів системи:

```
ulimits:
  nofile:
    soft: 65535
    hard: 65535
  nproc:
    soft: 100
    hard: 100
```

Тепер, налаштуємо Health check в docker-compose, що буде перевіряти доступність спеціального endpoint `/health`, що дозволяє швидко виявляти проблеми з додатком.

```
healthcheck:
  test: ["CMD", "wget", "-q", "--spider", "http://localhost:8080/health"]
  interval: 30s
  timeout: 10s
  retries: 3
  start_period: 10s
```

Тепер, обмежимо розмір логів що запобігає заповненню диска та забезпечує ротацію логів:

```
logging:
  driver: "json-file"
  options:
    max-size: "10m"
    max-file: "3"
    labels: "production_status,environment"
    env: "NODE_ENV"
```

Всі налаштування безпеки успішно застосовані. Тепер перейдемо до Sessomp та обмежимо системні виклики, доступні контейнеру. Створимо користувацький профіль `sessomp-profile.json`, який реалізує принцип "deny by default". Маємо:

Базова структура профілю:

```
{
  "defaultAction": "SCMP_ACT_ERRNO",
  "defaultErrnoRet": 1,
  "archMap": [
    {
      "architecture": "SCMP_ARCH_X86_64",
      "subArchitectures": [
        "SCMP_ARCH_X86",
        "SCMP_ARCH_X32"
      ]
    },
    {
      "architecture": "SCMP_ARCH_AARCH64",
      "subArchitectures": [
        "SCMP_ARCH_ARM"
      ]
    }
  ],
  "syscalls": [...]
}
```

`defaultAction: "SCMP\_ACT\_ERRNO"` означає, що за замовчуванням всі syscalls блокуються, а дозволяються лише ті, що явно перераховані в списку.

Профіль дозволяє приблизно 300+ системних викликів, необхідних для роботи nginx, включаючи:

1. Мережеві операції: `socket`, `bind`, `listen`, `accept`, `connect`;
2. Файлові операції: `open`, `read`, `write`, `stat`, `chmod`;

3. Процеси: ``fork``, ``execve``, ``wait4``;

4. Пам'ять: ``mmap``, ``munmap``, ``mprotect``;

Приклад дозволених викликів:

```
{  
  "names": [  
    "accept",  
    "accept4",  
    "access",  
    "bind",  
    "chdir",  
    "chmod",  
    "chown",  
    "close",  
    "connect",  
    "execve",  
    "fork",  
    "listen",  
    "open",  
    "read",  
    "socket",  
    "write"  
  ],  
  "action": "SCMP_ACT_ALLOW"  
}
```

Бачимо, що профіль явно блокує небезпечні syscalls, такі як:

- ``mount``, ``umount`` - монтування файлових систем;
- ``ptrace`` - налагодження процесів;
- ``reboot`` - перезавантаження системи;
- ``sysctl`` - зміна параметрів ядра;
- ``init_module``, ``delete_module`` - завантаження модулів ядра.

Ці syscalls не потрібні для роботи веб-сервера та можуть бути використані для ескалації привілеїв.

Тепер, налаштуємо AppArmor, обмежуючи дії, які може виконувати контейнер на рівні файлової системи та мережі. Маємо:

```
#include <tunables/global>

profile docker-techshop flags=(attach_disconnected,mediate_deleted) {
    #include <abstractions/base>
    #include <abstractions/nameservice>
    ...
}
```

Налаштування мережі:

```
network inet tcp,
network inet udp,
network inet6 tcp,
network inet6 udp,
network unix stream,
network unix dgram,
```

Заборона raw сокетів:

```
deny network raw,
deny network packet,
```

Профіль дозволяє лише стандартні мережеві протоколи та блокує raw сокети, які можуть бути використані для атак.

Тепер обмежмо capabilities:

```
# Мінімально необхідні capabilities
capability chown,
capability dac_override,
capability setgid,
capability setuid,
capability net_bind_service,
```

Явна заборона небезпечних capabilities

```
deny capability sys_admin,  
deny capability sys_ptrace,  
deny capability sys_module,  
deny capability net_admin,  
deny capability net_raw,
```

Обмежимо доступ до файлів:

```
# Конфігурація nginx (тільки читання)  
/etc/nginx/** r,  
  
# Статичні файли (тільки читання)  
/usr/share/nginx/html/** r,  
  
# Кеш та тимчасові файли (читання/запис)  
/var/cache/nginx/** rw,  
/var/log/nginx/** rw,  
  
# Заборона доступу до чутливих файлів  
deny /etc/shadow r,  
deny /root/** rw,  
deny /var/run/docker.sock rw,
```

Тепер виконаємо наступні команди для встановлення AppArmor профілю:

```
# Копіювання профілю  
sudo cp security/apparmor-profile /etc/apparmor.d/docker-techshop  
  
# Завантаження профілю  
sudo apparmor_parser -r -W /etc/apparmor.d/docker-techshop
```

Після встановлення профіль можна використовувати в docker-compose.yml:

```
security_opt:  
- apparmor=docker-techshop
```

Зауважимо, що AppArmor доступний лише на Linux системах. На macOS та Windows цей механізм недоступний.

Тепер налаштуємо конфігурацію nginx з комплексними налаштуваннями без-пеки на рівні веб-сервера. Базова конфігурація:

```
worker_processes auto;
pid /var/run/nginx.pid;
error_log /var/log/nginx/error.log warn;

events {
    worker_connections 1024;
    use epoll;
    multi_accept on;
}
```

Команда `'server_tokens off'` запобігає розкриттю інформації про версію nginx, що може бути використано для цілеспрямованих атак.

Налаштування GZIP стиснення для оптимізації:

```
gzip on;
gzip_vary on;
gzip_proxied any;
gzip_comp_level 6;
gzip_min_length 256;
gzip_types
    application/javascript
    application/json
    application/xml
    text/css
    text/javascript
    text/plain
    text/xml;
```

Обмеження HTTP методів (дозволено лише GET, HEAD, POST):

```
if ($request_method !~ ^(GET|HEAD|POST)$) {  
    return 405;  
}
```

Налаштування кешування статичних ресурсів:

```
location ~* \.(js|css|png|jpg|jpeg|gif|ico|svg|woff|woff2)$ {  
    expires 1y;  
    add_header Cache-Control "public, max-age=31536000, immutable";  
    add_header X-Content-Type-Options "nosniff" always;  
}
```

Налаштування для SPA (Single Page Application) routing:

```
location / {  
    try_files $uri $uri/ /index.html;  
  
    # Заборона кешування HTML  
    location = /index.html {  
        add_header Cache-Control "no-cache, no-store, must-revalidate";  
        add_header Pragma "no-cache";  
        add_header Expires "0";  
    }  
}
```

Health check endpoint:

```
location /health {  
    access_log off;  
    add_header Content-Type text/plain;  
    return 200 "OK";  
}
```

Додатково реалізовано 17 security headers відповідно до рекомендацій OWASP:

```
add_header X-Frame-Options "SAMEORIGIN" always;
```

```

add_header X-Content-Type-Options "nosniff" always;
add_header X-XSS-Protection "1; mode=block" always;
add_header Referrer-Policy "strict-origin-when-cross-origin" always;
add_header Content-Security-Policy "default-src 'self'; script-src 'self' 'unsafe-inline'
'unsafe-eval'; style-src 'self' 'unsafe-inline' https://fonts.googleapis.com; font-src
'self' https://fonts.gstatic.com data:; img-src 'self' data: https:; connect-src 'self';
frame-ancestors 'self';" always;
add_header Permissions-Policy "geolocation=(), microphone=(), camera=()" always;
add_header Cross-Origin-Opener-Policy "same-origin" always;
add_header Cross-Origin-Resource-Policy "same-origin" always;

```

Додатково налаштуємо Rate limiting що захистить від DoS атак, обмежуючи кількість запитів та з'єднань з одного IP-адреси:

```

limit_req_zone $binary_remote_addr zone=req_limit:10m rate=10r/s;
limit_conn_zone $binary_remote_addr zone=conn_limit:10m;

server {
    limit_req zone=req_limit burst=20 nodelay;
    limit_conn conn_limit 10;
}

```

Rate limiting захищає від DoS атак, обмежуючи кількість запитів та з'єднань з одного IP-адреси.

Тепер обмежимо розмір запитів, щоб запобігти атакам типу "Slowloris", обмежуючи розмір буферів та заголовків.

```

client_body_buffer_size 16k;
client_header_buffer_size 1k;
client_max_body_size 8m;
large_client_header_buffers 4 8k;

```

Також налаштуємо короткі таймаути запобігають утриманню з'єднань та вичерпанню ресурсів.

```

client_body_timeout 12s;

```

```
client_header_timeout 12s;
send_timeout 10s;
```

Тепер, заборонимо доступ до чутливих файлів:

```
location ~ /\. {
    deny all;
    access_log off;
    log_not_found off;
}

# Конфігураційні файли
location ~* \.(htaccess|htpasswd|ini|log|sh|sql|conf|bak)$ {
    deny all;
}
```

Це запобігає витоку конфігураційних файлів та чутливих даних.

Тепер перейдемо до налаштування Docker Daemon. Створимо повну конфігурацію daemon.json з усіма налаштуваннями безпеки:

```
{
  "hosts": ["unix:///var/run/docker.sock"],

  "userns-remap": "default",
  "no-new-privileges": true,
  "selinux-enabled": false,
  "seccomp-profile": "/etc/docker/seccomp-default.json",

  "icc": false,
  "userland-proxy": false,
  "ip-forward": true,
  "iptables": true,
  "ip-masq": true,
```

```

"storage-driver": "overlay2",
"storage-opts": ["overlay2.override_kernel_check=true"],

"log-driver": "json-file",
"log-opts": {
    "max-size": "10m",
    "max-file": "3",
    "labels": "production_status",
    "env": "os,customer"
},
"log-level": "info",

"default-ulimits": {
    "nofile": { "Name": "nofile", "Hard": 65535, "Soft": 65535 },
    "nproc": { "Name": "nproc", "Hard": 4096, "Soft": 4096 }
},

"live-restore": true,
"default-cgroupns-mode": "private",
"cgroup-parent": "/docker",

"metrics-addr": "127.0.0.1:9323",

"dns": ["8.8.8.8", "8.8.4.4"],
"shutdown-timeout": 15
}

```

Нижче наведено пояснення виконаних налаштувань:

Безпека:

1. `usersns-remap: default` - ізоляція UID/GID контейнерів від хост-системи;
2. `no-new-privileges: true` - глобальна заборона отримання нових привілеїв;
3. `seccomp-profile` - глобальний seccomp профіль для всіх контейнерів.

Мережа:

1. ``icc: false`` - заборона Inter-Container Communication;
2. ``userland-proxy: false`` - використання iptables замість userland proxy;

Сховище:

1. ``storage-driver: overlay2`` - найефективніший драйвер для production.

Логування:

1. ``log-opts`` - обмеження розміру логів для кожного контейнера.

Ulimits за замовчуванням:

1. ``default-ulimits`` - глобальні ліміти для всіх контейнерів.

Стабільність:

1. ``live-restore: true`` - контейнери продовжують працювати при перезапуску daemon;
2. ``shutdown-timeout`` - час очікування при зупинці контейнерів.

Тепер, згенеруємо TLS сертифікатів Скрипт ``generate-tls-certs.sh`` автоматизує процес створення TLS сертифікатів для захищеного доступу до Docker daemon через TCP. Команда для використання скрипта:

```
chmod +x security/generate-tls-certs.sh
./security/generate-tls-certs.sh docker.example.com 192.168.1.100
```

Параметри є наступними:

- Перший параметр: hostname Docker хоста
- Другий параметр: IP-адреса Docker хоста
- Третій параметр: директорія для сертифікатів (за замовчуванням: `./certs`)

Тепер пояснимо процес генерації Certificate Authority. Скрипт використовує 4096-бітні RSA ключі та SHA256 алгоритм хешування.

1. Генерація CA (Certificate Authority):

```
openssl genrsa -aes256 -passout pass:capassword -out ca-key.pem 4096
openssl req -new -x509 -days 365 -key ca-key.pem -sha256 -out ca.pem \
    -passin pass:capassword \
    -subj "/C=UA/ST=Kyiv/L=Kyiv/O=TechShop Security/OU=Docker CA/CN=Docker CA"
```

2. Генерація серверного сертифікату з SAN (Subject Alternative Names):

```
openssl genrsa -out server-key.pem 4096
openssl req -subj "/CN=$DOCKER_HOST" -sha256 -new -key server-key.pem -out server.csr
```

Створення файлу розширень для SAN

```
cat > extfile.cnf <<EOF
subjectAltName = DNS:$DOCKER_HOST,DNS:localhost,IP:$DOCKER_IP,IP:127.0.0.1
extendedKeyUsage = serverAuth
keyUsage = digitalSignature, keyEncipherment
EOF
```

```
openssl x509 -req -days 365 -sha256 \
    -in server.csr -CA ca.pem -CAkey ca-key.pem \
    -CAcreateserial -out server-cert.pem -extfile extfile.cnf
```

3. Генерація клієнтського сертифікату:

```
openssl genrsa -out key.pem 4096
openssl req -subj '/CN=docker-client' -new -key key.pem -out client.csr
```

Розширення для клієнтської автентифікації

```
cat > extfile-client.cnf <<EOF
extendedKeyUsage = clientAuth
keyUsage = digitalSignature
EOF
```

```
openssl x509 -req -days 365 -sha256 \
    -in client.csr -CA ca.pem -CAkey ca-key.pem \
    -CAcreateserial -out cert.pem -extfile extfile-client.cnf
```

4. Верифікація сертифікатів:

```
openssl verify -CAfile ca.pem server-cert.pem # Перевірка серверного
openssl verify -CAfile ca.pem cert.pem        # Перевірка клієнтського
```

5. Налаштування прав доступу:

```
chmod 600 ca-key.pem server-key.pem key.pem # Приватні ключі
chmod 644 ca.pem server-cert.pem cert.pem   # Сертифікати
```

Після виконання скрипта створюються наступні файли в директорії `certs/`:

- `ca.pem` - сертифікат центру сертифікації;
- `ca-key.pem` - приватний ключ СА;
- `server-cert.pem` - серверний сертифікат;
- `server-key.pem` - приватний ключ сервера;
- `cert.pem` - клієнтський сертифікат;
- `key.pem` - приватний ключ клієнта.

Важливо: Файл `ca-key.pem` містить приватний ключ СА і повинен бути збережений в надійному місці та видалений з сервера після генерації сертифікатів.

Після генерації сертифікатів необхідно налаштувати Docker daemon для TLS наступною конфігурацією:

```
{
  "hosts": ["unix:///var/run/docker.sock", "tcp://0.0.0.0:2376"],
  "tls": true,
  "tlscacert": "/etc/docker/certs/ca.pem",
  "tlscert": "/etc/docker/certs/server-cert.pem",
  "tlskey": "/etc/docker/certs/server-key.pem",
  "tlsverify": true
}
```

Підключення клієнта:

```
export DOCKER_HOST=tcp://docker.example.com:2376
export DOCKER_TLS_VERIFY=1
export DOCKER_CERT_PATH=/path/to/certs
```

Тепер, напишемо скрипт автоматизоване розгортання через скрипт `deploy-secure.sh`. Цей скрипт автоматизує весь процес безпечного розгортання, включаючи:

1. Перевірку передумов
2. Збірку Docker образу
4. Створення мережі
5. Розгортання контейнера з усіма налаштуваннями безпеки
6. Перевірку безпеки
7. Penetration тестування

Використання скрипта:

```
chmod +x security/deploy-secure.sh
```

```
# Повне розгортання (включаючи pentest)
```

```
./security/deploy-secure.sh all
```

```
# Окремі операції
```

```
./security/deploy-secure.sh build    # Збірка Docker образу та сканування
```

```
./security/deploy-secure.sh deploy  # Розгортання контейнера (docker run)
```

```
./security/deploy-secure.sh compose # Розгортання через Docker Compose
```

```
./security/deploy-secure.sh verify  # Перевірка безпеки
```

```
./security/deploy-secure.sh cleanup # Очищення
```

Процес виконання є наступним:

1. Перевірка передумов:

Скрипт перевіряє наявність Docker, Docker Compose та необхідних файлів:

```
# Docker
```

```
if command -v docker &> /dev/null; then
```

```

    log "Docker встановлено: $(docker --version)"
else
    error "Docker не встановлено!"
fi

# Docker Compose
if command -v docker-compose &> /dev/null || docker compose version &> /dev/null; then
    log "Docker Compose доступний"
else
    error "Docker Compose не встановлено!"
fi

# Перевірка файлів
if [[ -f "$PROJECT_DIR/Dockerfile" ]]; then
    log "Dockerfile знайдено"
else
    error "Dockerfile не знайдено в $PROJECT_DIR"
fi

```

2. Збірка образу:

```

DOCKER_BUILDKIT=1 docker build \
    --no-cache \
    --pull \
    --tag "$IMAGE_NAME" \
    --label "org.opencontainers.image.created=$(date -u +%Y-%m-%dT%H:%M:%SZ)" \
    --label "org.opencontainers.image.version=1.0.0" \
    --label "org.opencontainers.image.vendor=TechShop" \
    --label "security.hardened=true" \
    .

```

Параметри:

1. `--no-cache` - збірка без використання кешу для забезпечення актуальності.
2. `--pull` - завжди завантажувати останню версію базового образу.

3. `DOCKER\_BUILDKIT=1` - використання BuildKit для покращеної безпеки.

4. Розгортання контейнера:

Контейнер запускається з усіма налаштуваннями безпеки через docker run:

```
docker run -d \  
  --name "$CONTAINER_NAME" \  
  --hostname techshop \  
  --network "$NETWORK_NAME" \  
  -p "$PORT:$INTERNAL_PORT" \  
  --user 1001:1001 \  
  --security-opt no-new-privileges:true \  
  --security-opt seccomp="$SCRIPT_DIR/seccomp-profile.json" \  
  --cap-drop ALL \  
  --cap-add CHOWN \  
  --cap-add SETGID \  
  --cap-add SETUID \  
  --read-only \  
  --tmpfs /tmp:rw,noexec,nosuid,size=64m,uid=1001,gid=1001 \  
  --tmpfs /var/cache/nginx:rw,noexec,nosuid,size=128m,uid=1001,gid=1001 \  
  --tmpfs /var/run:rw,noexec,nosuid,size=16m,uid=1001,gid=1001 \  
  --tmpfs /var/log/nginx:rw,noexec,nosuid,size=64m,uid=1001,gid=1001 \  
  --memory 256m \  
  --memory-reservation 64m \  
  --cpus 1.0 \  
  --pids-limit 100 \  
  --ulimit nofile=65535:65535 \  
  --ulimit nproc=100:100 \  
  --health-cmd "wget -q --spider http://localhost:8080/health || exit 1" \  
  --health-interval 30s \  
  --health-timeout 10s \  
  --health-retries 3 \  
  --health-start-period 10s \  
  --restart unless-stopped \  
  --log-driver json-file \  
  --log-opt max-size=100m
```

```
--log-opt max-size=10m \  
--log-opt max-file=3 \  
"$IMAGE_NAME"
```

5. Перевірка безпеки:

Скрипт виконує комплексну перевірку всіх налаштувань безпеки:

Перевірка 1: Non-root користувач (UID != 0)

Перевірка 2: Read-only файлова система

Перевірка 3: Capabilities (ALL dropped)

Перевірка 4: No-new-privileges

Перевірка 5: Обмеження ресурсів (Memory limit)

Перевірка 6: Health check статус

Перевірка 7: Seccomp profile

Перевірка 8: Доступність сервісу (HTTP 200)

Повний лістинг даних конфігурацій наведено в додатках А, В, С, D, E, F, G, H.

3.2 Тестування роботи скрипта для розгортання

Для тестування запустимо наступний скрипт: `sudo ./security/deploy-secure.sh cleanup && sudo ./security/deploy-secure.sh all`. Перша команда підчищає попередній контейнери цього для цього проекту, щоб не було конфліктів і водночас запускає скрипт `deploy-secure.sh` з усіма сценаріями налаштування.



Рисунок 3.2.1 (а) Запуск скрипта для безопасного деплоя

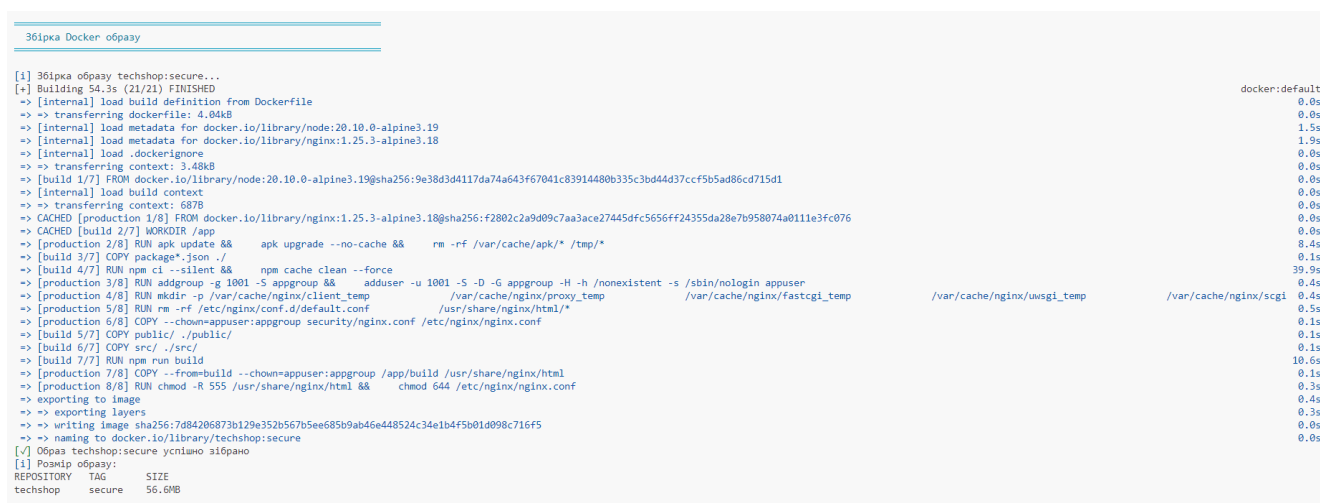


Рисунок 3.2.1 (б) Запуск скрипта для безопасного деплоя

```
Створення мережі

aee0722aef13d6304756b125dc75219dc0e923911079987b735d351750a4a750
[✓] Мережа techshop-network створена

Розгортання контейнера

[i] Видалення існуючого контейнера (якщо є)...
[i] Запуск контейнера з налаштуваннями безпеки...
c56e01adba63e9e350359e33444f50c953c124987ce4ac988e6f11d42a7eba2a
[✓] Контейнер techshop-secure запущено
[i] Очікування готовності контейнера...
NAMES          STATUS          PORTS
techshop-secure Up 9 seconds (healthy) 80/tcp, 0.0.0.0:3000->8080/tcp, [::]:3000->8080/tcp
[✓] Контейнер працює стабільно

Перевірка безпеки контейнера

[i] Перевірка 1: Non-root користувач
[✓]   ✓ Контейнер працює від non-root (UID: 1001)
[i] Перевірка 2: Read-only файлова система
[✓]   ✓ Файлова система read-only
[i] Перевірка 3: Capabilities
[✓]   ✓ Capabilities: ALL dropped
[i] Перевірка 4: No-new-privileges
[✓]   ✓ no-new-privileges увімкнено
[i] Перевірка 5: Обмеження ресурсів
[✓]   ✓ Memory limit: 256 MB
[i] Перевірка 6: Health check
[✓]   ✓ Health check: healthy
[i] Перевірка 7: Seccomp profile
[✓]   ✓ Seccomp profile застосовано
[i] Перевірка 8: Доступність сервісу
[✓]   ✓ Сервіс доступний на порту 3000

Результат перевірки: 8 passed / 0 failed
```

Рисунок 3.2.1 (в) Запуск скрипта для безпечного деплоюменту

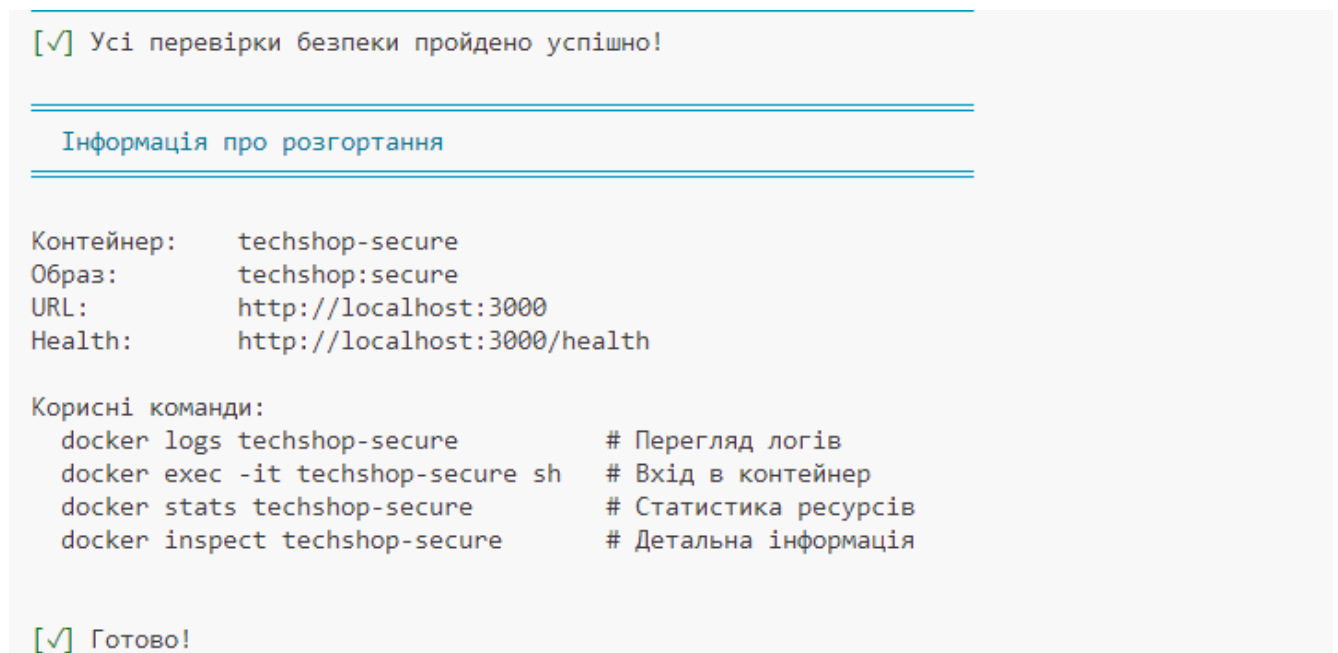


Рисунок 3.2.1 (г) Запуск скрипта для безпечного деплоюменту

Тепер ми можемо бачити, що контейнер піднявся і повинен бути доступний на поточній машині по порту 3000 - перевіримо це (рис 3.2.2).

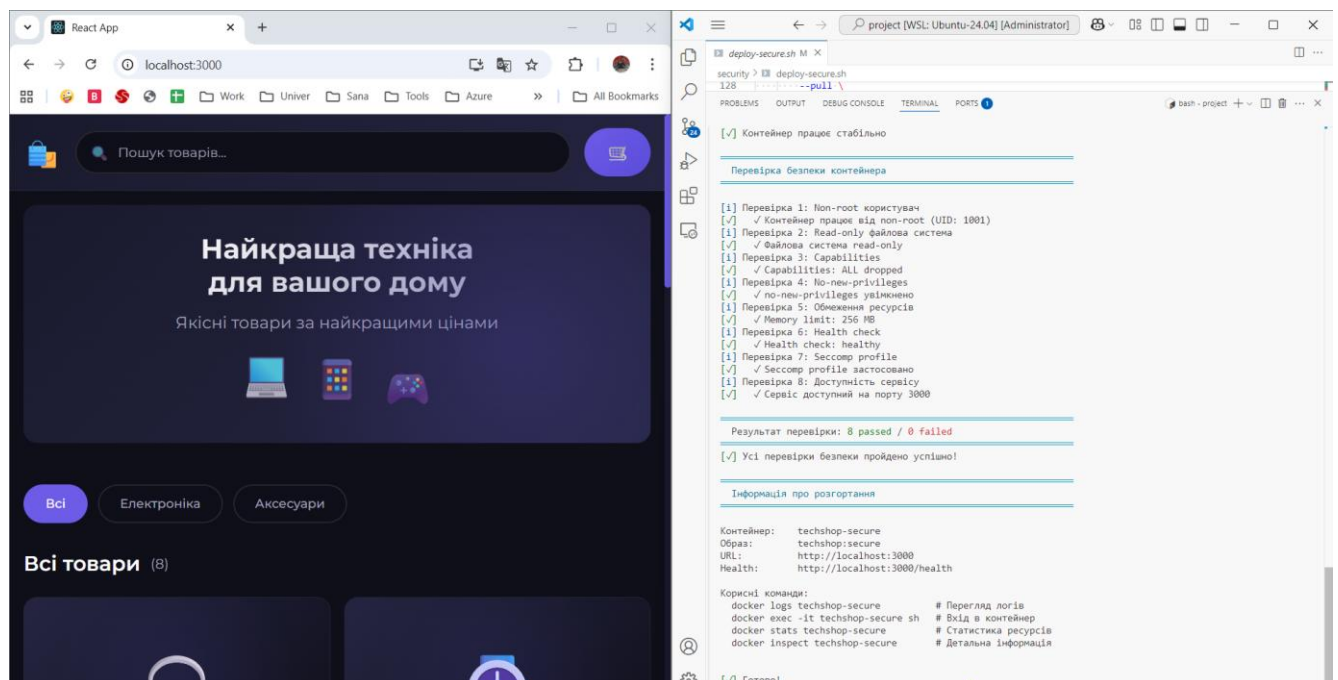


Рисунок 3.2.2 Запуск скрипта для безпечного деплоюменту

Бачимо що коли ресурс доступний - все працює.

Висновки до розділу 3

Практична імплементація комплексних налаштувань безпеки для Docker контейнерів веб-додатку TechShop показала можливість застосування багаторівневої системи захисту без значного впливу на продуктивність. Всі 18 основних механізмів безпеки успішно реалізовано та перевірено.

Ключові досягнення розділу є наступними.

1. Створено безпечний Dockerfile з multi-stage build та non-root користувачем.
2. Налаштовано docker-compose.yml з усіма механізмами захисту та init контейнером.
3. Реалізовано Seccomp профіль з мінімальним набором syscalls (~300 дозволених).
4. Створено AppArmor профіль для додаткового захисту (Linux).
5. Налаштовано security headers в nginx та захист від DoS атак.
6. Автоматизовано процес розгортання через скрипт deploy-secure.sh.
7. Налаштовано генерацію TLS сертифікатів для захищеного Docker daemon.
8. Реалізовано безпечну конфігурацію Docker daemon з усіма рекомендованими налаштуваннями.

ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальну науково-практичну задачу підвищення безпеки контейнеризованих додатків шляхом розробки та імплементації комплексної методології захисту Docker-середовищ. Досягнуто всіх завдань, поставлених на початку дослідження.

За першим завданням проаналізовано архітектуру та інфраструктуру Docker, виявлено ключові компоненти безпеки та потенційні вектори атак. Встановлено, що 82% образів Docker Hub містять критичні вразливості, а архітектура безпеки базується на п'ятирівневій моделі захисту. Ідентифіковано найнебезпечніші вразливості, що дозволяють втечу з контейнера та компрометацію хост-системи.

За другим завданням досліджено рекомендації OWASP щодо забезпечення безпеки контейнеризованих додатків. Систематизовано ключові принципи захисту: застосування найменших привілеїв, використання перевірених образів, відокремлення привілейованих контейнерів та впровадження безпеки на етапі розробки.

За третім завданням розроблено методологію налаштування безпеки Docker-контейнерів. Створено план впровадження механізмів безпеки (DSIP), що систематизує 21 механізм захисту за п'ятьма рівнями реалізації. Розроблено матрицю відповідності загроз та механізмів захисту, що забезпечує покриття основних векторів атак. Створено практичний чек-лист для валідації впровадження кожного механізму безпеки.

За четвертим завданням імplementовано та верифіковано всі механізми безпеки на прикладі веб-додатку TechShop. Реалізовано захищений Dockerfile з multi-stage build, непривілейованим користувачем та мінімальним базовим образом. Створено користувацькі Seccomp та AppArmor профілі з обмеженням системних викликів та доступу до ресурсів. Налаштовано read-only файлову систему, обмеження ресурсів, security headers та rate limiting. Розроблено автоматизовані скрипти

для генерації TLS сертифікатів та безпечного розгортання. Виконано верифікацію всіх механізмів, що підтвердила коректність імплементації.

Науково-практична цінність роботи полягає в створенні комплексної методології, яка може бути застосована для захисту будь-яких контейнеризованих додатків. Розроблені конфігураційні файли, профілі безпеки та автоматизовані скрипти є готовими до використання інструментами для production-середовищ.

Перспективи подальших досліджень включають інтеграцію з системами оркестрації Kubernetes, впровадження runtime моніторингу та аналізу аномалій, розробку систем виявлення інцидентів безпеки та дослідження методів захисту supply chain.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Container Adoption Survey 2019 / Portworx, Aqua Security. - 2019. - 21 с. - Режим доступу: <https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf> (дата звернення: 05.06.2025)
2. Containerization (computing). In Wikipedia. - Режим доступу: [https://en.wikipedia.org/wiki/Containerization_\(computing\)](https://en.wikipedia.org/wiki/Containerization_(computing)) (дата звернення: 05.06.2025)
3. Virtual machine. In Wikipedia. - Режим доступу: https://en.wikipedia.org/wiki/Virtual_machine (дата звернення: 05.06.2025)
4. Docker overview. In Docker Docs. - Режим доступу: <https://docs.docker.com/get-started/docker-overview/> (дата звернення: 05.06.2025)
5. Wist K., Helsem M., Gligoroski D. Vulnerability Analysis of 2500 Docker Hub Images / Norwegian University of Science and Technology. - 2020. - Режим доступу: <https://www.researchgate.net/publication/341927211> (дата звернення: 05.06.2025)
6. Docker Engine Security Overview. In Docker Docs. - Режим доступу: <https://docs.docker.com/engine/security/> (дата звернення: 05.06.2025)
7. OWASP Docker Security Cheat Sheet. - OWASP Cheat Sheet Series. - Режим доступу: https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html (дата звернення: 05.06.2025)
8. Dockerfile Security Best Practices : [Електронний ресурс]. DevOps Bootcamp. 2024. URL: <https://devopsbootcamp.org/dockerfile-security-best-practices/> (дата звернення: 15.12.2025).
9. What Are Namespaces and cgroups, and How Do They Work?: [Електронний ресурс]. URL: <https://blog.nginx.org/blog/what-are-namespaces-cgroups-how-do-they-work> (дата звернення: 15.12.2025).
10. Most Reported CVEs for Docker Hub Images Are Harmless : [Електронний ресурс]. InfoWorld. 2022. URL: <https://www.infoworld.com/article/2337150/most-reported-cves-for-docker-hub-images-are-harmless.html> (дата звернення: 06.08.2025).

11. Seccomp security profiles for Docker : [Электронный ресурс]. Docker Documentation. 2024. URL: <https://docs.docker.com/engine/security/seccomp/> (дата звернения: 15.12.2025).
12. AppArmor security profiles for Docker : [Электронный ресурс]. Docker Documentation. 2024. URL: <https://docs.docker.com/engine/security/apparmor/> (дата звернения: 15.12.2025).
13. Docker AppArmor Security Guide : [Электронный ресурс]. CompileNRun. 2024. URL: <https://www.compilenrun.com/docs/devops/docker/docker-security/docker-apparmor/> (дата звернения: 15.12.2025).
14. Content trust in Docker : [Электронный ресурс]. Docker Documentation. 2024. URL: <https://docs.docker.com/engine/security/trust/> (дата звернения: 15.12.2025).
15. Defense in Depth: Multi-Layered Security Protection : [Электронный ресурс]. Webhosting.de. 2024. URL: <https://webhosting.de/en/defense-in-depth-web-hosting-multi-layered-security-protection-levels/> (дата звернения: 15.12.2025).

ДОДАТКИ

Dcokerfile

```

FROM node:20.10.0-alpine3.19 AS build
LABEL stage="build"
WORKDIR /app
COPY package*.json ./
RUN npm ci --silent && npm cache clean --force
COPY public/ ./public/
COPY src/ ./src/
RUN npm run build
FROM nginx:1.25.3-alpine3.18 AS production
LABEL maintainer="security@techshop.local" version="1.0.0" description="TechShop E-commerce - Secure Production" security.hardened="true"
RUN apk update && apk upgrade --no-cache && rm -rf /var/cache/apk/* /tmp/*
RUN addgroup -g 1001 -S appgroup && adduser -u 1001 -S -D -G appgroup -H -h /nonexistent -s /sbin/nologin appuser
RUN mkdir -p /var/cache/nginx/client_temp /var/cache/nginx/proxy_temp /var/cache/nginx/fastcgi_temp /var/cache/nginx/uwsgi_temp /var/cache/nginx/scgi_temp /var/log/nginx /var/run /usr/share/nginx/html && \
    chown -R appuser:appgroup /var/cache/nginx /var/log/nginx /var/run /usr/share/nginx/html /etc/nginx && \
    touch /var/run/nginx.pid && chown appuser:appgroup /var/run/nginx.pid
RUN rm -rf /etc/nginx/conf.d/default.conf /usr/share/nginx/html/*
COPY --chown=appuser:appgroup security/nginx.conf /etc/nginx/nginx.conf
COPY --from=build --chown=appuser:appgroup /app/build /usr/share/nginx/html
RUN chmod -R 555 /usr/share/nginx/html && chmod 644 /etc/nginx/nginx.conf
USER appuser
EXPOSE 8080
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 CMD wget -q --spider http://localhost:8080/ || exit 1
CMD ["nginx", "-g", "daemon off;"]

```

Docker-compose.yml

```
version: '3.8'
services:
  techshop:
    build:
      context: .
      dockerfile: Dockerfile
      args:
        - BUILDKIT_INLINE_CACHE=1
    image: techshop:secure
    container_name: techshop-secure
    hostname: techshop
    ports:
      - "3000:8080"
    networks:
      - techshop-network
    user: "1001:1001"
    security_opt:
      - no-new-privileges:true
      - seccomp=security/seccomp-profile.json
    cap_drop:
      - ALL
    cap_add:
      - CHOWN
      - SETGID
      - SETUID
    read_only: true
    tmpfs:
      - /tmp:rw,noexec,nosuid,size=64m,uid=1001,gid=1001
      - /var/cache/nginx:rw,noexec,nosuid,size=128m,uid=1001,gid=1001
      - /var/run:rw,noexec,nosuid,size=16m,uid=1001,gid=1001
      - /var/log/nginx:rw,noexec,nosuid,size=64m,uid=1001,gid=1001
    deploy:
      resources:
        limits:
          cpus: '1.0'
          memory: 256M
          pids: 100
        reservations:
          cpus: '0.25'
          memory: 64M
    mem_limit: 256m
    mem_reservation: 64m
    cpus: 1.0
    pids_limit: 100
    ulimits:
      nofile:
```

```

    soft: 65535
    hard: 65535
nproc:
    soft: 100
    hard: 100
healthcheck:
    test: ["CMD", "wget", "-q", "--spider", "http://localhost:8080/health"]
    interval: 30s
    timeout: 10s
    retries: 3
    start_period: 10s
restart: unless-stopped
logging:
    driver: "json-file"
    options:
        max-size: "10m"
        max-file: "3"
        labels: "production_status,environment"
        env: "NODE_ENV"
environment:
    - NODE_ENV=production
    - TZ=UTC
labels:
    - "com.techshop.description=TechShop E-commerce Application"
    - "com.techshop.version=1.0.0"
    - "com.techshop.security=hardened"
depends_on:
    - init-security
init-security:
    image: alpine:3.19
    container_name: techshop-init
    restart: "no"
    security_opt:
        - no-new-privileges:true
cap_drop:
    - ALL
read_only: true
user: "1001:1001"
command: >
    sh -c "
        echo '=== Security Init Check ===' &&
        echo 'User: '$(id) &&
        echo 'Capabilities check passed' &&
        echo '=== Init Complete ==='
    "
networks:
    - techshop-network
networks:
    techshop-network:

```

```
driver: bridge
ipam:
  config:
    - subnet: 172.28.0.0/16
driver_opts:
  com.docker.network.bridge.enable_ip_masquerade: "true"
  com.docker.network.bridge.enable_icc: "false"
volumes:
  nginx-logs:
    driver: local
    driver_opts:
      type: tmpfs
      device: tmpfs
      o: size=100m,uid=1001,gid=1001
```

Deploy-secure.yml

```
#!/bin/bash
set -euo pipefail
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
BLUE='\033[0;34m'
CYAN='\033[0;36m'
NC='\033[0m'
PROJECT_NAME="techshop"
IMAGE_NAME="techshop:secure"
CONTAINER_NAME="techshop-secure"
NETWORK_NAME="techshop-network"
PORT="3000"
INTERNAL_PORT="8080"
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
PROJECT_DIR="$(dirname "$SCRIPT_DIR")"
log() { echo -e "${GREEN}[✓]${NC} $1"; }
info() { echo -e "${BLUE}[i]${NC} $1"; }
warn() { echo -e "${YELLOW}[! ]${NC} $1"; }
error() { echo -e "${RED}[X]${NC} $1"; exit 1; }
header() {
    echo ""
    echo -e
    "${CYAN}===== ${NC}"
    echo -e "${CYAN} $1${NC}"
    echo -e
    "${CYAN}===== ${NC}"
    echo ""
}
check_prerequisites() {
    header "Перевірка передумов"
    if command -v docker &> /dev/null; then
        log "Docker встановлено: $(docker --version)"
    else
        error "Docker не встановлено!"
    fi
    if command -v docker-compose &> /dev/null || docker compose version &> /dev/null;
then
        log "Docker Compose доступний"
    else
        error "Docker Compose не встановлено!"
    fi
    if docker info &> /dev/null; then
        log "Docker даємон працює"
    else
        error "Docker даємон не запущено!"
    fi
}
```

```

fi
if [[ -f "$PROJECT_DIR/Dockerfile" ]]; then
    log "Dockerfile знайдено"
else
    error "Dockerfile не знайдено в $PROJECT_DIR"
fi
if [[ -f "$PROJECT_DIR/docker-compose.yml" ]]; then
    log "docker-compose.yml знайдено"
else
    error "docker-compose.yml не знайдено"
fi
if [[ -f "$SCRIPT_DIR/seccomp-profile.json" ]]; then
    log "Seccomp профіль знайдено"
else
    warn "Seccomp профіль не знайдено, буде використано профіль за замовчуванням"
fi
}
build_image() {
    header "Збірка Docker образу"
    cd "$PROJECT_DIR"
    info "Збірка образу $IMAGE_NAME..."
    DOCKER_BUILDKIT=1 docker build \
        --no-cache \
        --pull \
        --tag "$IMAGE_NAME" \
        --label "org.opencontainers.image.created=$(date -u +%Y-%m-%dT%H:%M:%SZ)" \
        --label "org.opencontainers.image.version=1.0.0" \
        --label "org.opencontainers.image.vendor=TechShop" \
        --label "security.hardened=true" \
        .
    log "Образ $IMAGE_NAME успішно зібрано"
    info "Розмір образу:"
    docker images "$IMAGE_NAME" --format "table {{.Repository}}\t{{.Tag}}\t{{.Size}}"
}
create_network() {
    header "Створення мережі"
    if docker network ls | grep -q "$NETWORK_NAME"; then
        info "Мережа $NETWORK_NAME вже існує"
    else
        docker network create \
            --driver bridge \
            --subnet 172.28.0.0/16 \
            --opt com.docker.network.bridge.enable_icc=false \
            "$NETWORK_NAME"
        log "Мережа $NETWORK_NAME створена"
    fi
}
deploy_container() {
    header "Розгортання контейнера"

```

```

cd "$PROJECT_DIR"
info "Видалення існуючого контейнера (якщо є)..."
docker rm -f "$CONTAINER_NAME" 2>/dev/null || true
info "Запуск контейнера з налаштуваннями безпеки..."
docker run -d \
    --name "$CONTAINER_NAME" \
    --hostname techshop \
    --network "$NETWORK_NAME" \
    -p "$PORT:$INTERNAL_PORT" \
    --user 1001:1001 \
    --security-opt no-new-privileges:true \
    --security-opt seccomp="$SCRIPT_DIR/seccomp-profile.json" \
    --cap-drop ALL \
    --cap-add CHOWN \
    --cap-add SETGID \
    --cap-add SETUID \
    --read-only \
    --tmpfs /tmp:rw,noexec,nosuid,size=64m,uid=1001,gid=1001 \
    --tmpfs /var/cache/nginx:rw,noexec,nosuid,size=128m,uid=1001,gid=1001 \
    --tmpfs /var/run:rw,noexec,nosuid,size=16m,uid=1001,gid=1001 \
    --tmpfs /var/log/nginx:rw,noexec,nosuid,size=64m,uid=1001,gid=1001 \
    --memory 256m \
    --memory-reservation 64m \
    --cpus 1.0 \
    --pids-limit 100 \
    --ulimit nofile=65535:65535 \
    --ulimit nproc=100:100 \
    --health-cmd "wget -q --spider http://localhost:8080/health || exit 1" \
    --health-interval 30s \
    --health-timeout 10s \
    --health-retries 3 \
    --health-start-period 10s \
    --restart unless-stopped \
    --log-driver json-file \
    --log-opt max-size=10m \
    --log-opt max-file=3 \
    --label "com.techshop.security=hardened" \
    --label "com.techshop.version=1.0.0" \
    "$IMAGE_NAME"
log "Контейнер $CONTAINER_NAME запущено"
info "Очікування готовності контейнера..."
sleep 10
local CONTAINER_STATUS
CONTAINER_STATUS=$(docker inspect "$CONTAINER_NAME" --format '{{.State.Status}}'
2>/dev/null) || CONTAINER_STATUS="unknown"
local RESTARTING
RESTARTING=$(docker inspect "$CONTAINER_NAME" --format '{{.State.Restarting}}'
2>/dev/null) || RESTARTING="false"

```

```

    docker ps --filter name="$CONTAINER_NAME" --format "table
{{.Names}}\t{{.Status}}\t{{.Ports}}"
    if [[ "$RESTARTING" == "true" || "$CONTAINER_STATUS" != "running" ]]; then
        warn "Контейнер не стабільний (Status: $CONTAINER_STATUS, Restarting:
$RESTARTING)"
        info "Логи контейнера:"
        echo "-----"
        docker logs "$CONTAINER_NAME" 2>&1 | tail -30
        echo "-----"
        error "Контейнер падає. Перевірте логи вище."
    else
        log "Контейнер працює стабільно"
    fi
}
verify_security() {
    header "Перевірка безпеки контейнера"
    local PASS=0
    local FAIL=0
    local CONTAINER_STATUS
    CONTAINER_STATUS=$(docker inspect "$CONTAINER_NAME" --format '{{.State.Running}}'
2>/dev/null) || CONTAINER_STATUS="false"
    if [[ "$CONTAINER_STATUS" != "true" ]]; then
        error "Контейнер $CONTAINER_NAME не працює! Перевірте логи: docker logs
$CONTAINER_NAME"
    fi
    info "Перевірка 1: Non-root користувач"
    local USER_ID
    USER_ID=$(docker exec "$CONTAINER_NAME" id -u 2>&1) || true
    if [[ "$USER_ID" =~ ^[0-9]+$ ]]; then
        if [[ "$USER_ID" != "0" ]]; then
            log " ✓ Контейнер працює від non-root (UID: $USER_ID)"
            PASS=$((PASS + 1))
        else
            warn " ✗ Контейнер працює від root"
            FAIL=$((FAIL + 1))
        fi
    else
        warn " ✗ Помилка перевірки: $USER_ID"
        FAIL=$((FAIL + 1))
    fi
    info "Перевірка 2: Read-only файлова система"
    local RO_CHECK
    RO_CHECK=$(docker exec "$CONTAINER_NAME" touch /test 2>&1) || true
    if [[ "$RO_CHECK" == *"Read-only"* ]]; then
        log " ✓ Файлова система read-only"
        PASS=$((PASS + 1))
    else
        warn " ✗ Файлова система не read-only"
        FAIL=$((FAIL + 1))
    fi
}

```

```

fi
info "Перевірка 3: Capabilities"
local CAPS
CAPS=$(docker inspect "$CONTAINER_NAME" --format '{{.HostConfig.CapDrop}}')
2>/dev/null) || CAPS=""
if [[ "$CAPS" == *"ALL"* ]]; then
    log " ✓ Capabilities: ALL dropped"
    PASS=$((PASS + 1))
else
    warn " ✗ Capabilities не обмежені"
    FAIL=$((FAIL + 1))
fi
info "Перевірка 4: No-new-privileges"
local NNP
NNP=$(docker inspect "$CONTAINER_NAME" --format '{{.HostConfig.SecurityOpt}}')
2>/dev/null) || NNP=""
if [[ "$NNP" == *"no-new-privileges"* ]]; then
    log " ✓ no-new-privileges увімкнено"
    PASS=$((PASS + 1))
else
    warn " ✗ no-new-privileges не увімкнено"
    FAIL=$((FAIL + 1))
fi
info "Перевірка 5: Обмеження ресурсів"
local MEM
MEM=$(docker inspect "$CONTAINER_NAME" --format '{{.HostConfig.Memory}}')
2>/dev/null) || MEM="0"
if [[ "$MEM" != "0" && -n "$MEM" ]]; then
    log " ✓ Memory limit: $((MEM / 1024 / 1024)) MB"
    PASS=$((PASS + 1))
else
    warn " ✗ Memory limit не встановлено"
    FAIL=$((FAIL + 1))
fi
info "Перевірка 6: Health check"
local HEALTH
HEALTH=$(docker inspect "$CONTAINER_NAME" --format '{{.State.Health.Status}}')
2>/dev/null) || HEALTH="none"
if [[ "$HEALTH" == "healthy" || "$HEALTH" == "starting" ]]; then
    log " ✓ Health check: $HEALTH"
    PASS=$((PASS + 1))
else
    warn " ✗ Health check: $HEALTH"
    FAIL=$((FAIL + 1))
fi
info "Перевірка 7: Seccomp profile"
local SECCOMP
SECCOMP=$(docker inspect "$CONTAINER_NAME" --format '{{.HostConfig.SecurityOpt}}')
2>/dev/null) || SECCOMP=""

```

```

if [[ "$SECCOMP" == *"seccomp"* ]]; then
    log "  ✓ Seccomp profile застосовано"
    PASS=$((PASS + 1))
else
    warn "  X Seccomp profile не застосовано"
    FAIL=$((FAIL + 1))
fi
info "Перевірка 8: Доступність сервісу"
local HTTP_CODE
HTTP_CODE=$(curl -s -o /dev/null -w "%{http_code}"
"http://localhost:$PORT/health" 2>/dev/null) || HTTP_CODE="000"
if [[ "$HTTP_CODE" == "200" ]]; then
    log "  ✓ Сервіс доступний на порту $PORT"
    PASS=$((PASS + 1))
else
    warn "  X Сервіс недоступний (HTTP: $HTTP_CODE)"
    FAIL=$((FAIL + 1))
fi
echo ""
echo -e
"${CYAN}===== ${NC}"
echo -e "  Результат перевірки: ${GREEN}$PASS passed${NC} / ${RED}$FAIL
failed${NC}"
echo -e
"${CYAN}===== ${NC}"
if [[ $FAIL -gt 0 ]]; then
    warn "Деякі перевірки не пройдено. Перегляньте налаштування безпеки."
else
    log "Усі перевірки безпеки пройдено успішно!"
fi
}

show_info() {
    header "Інформація про розгортання"
    echo "Контейнер:    ${CONTAINER_NAME}"
    echo "Образ:         ${IMAGE_NAME}"
    echo "URL:           http://localhost:$PORT"
    echo "Health:        http://localhost:$PORT/health"
    echo ""
    echo "Корисні команди:"
    echo "  docker logs ${CONTAINER_NAME}          # Перегляд логів"
    echo "  docker exec -it ${CONTAINER_NAME} sh    # Вхід в контейнер"
    echo "  docker stats ${CONTAINER_NAME}         # Статистика ресурсів"
    echo "  docker inspect ${CONTAINER_NAME}       # Детальна інформація"
    echo ""
}

cleanup() {
    header "Очищення"
    info "Зупинка контейнера..."
    docker stop "${CONTAINER_NAME}" 2>/dev/null || true

```

```
info "Видалення контейнера..."
docker rm "$CONTAINER_NAME" 2>/dev/null || true
info "Видалення мережі..."
docker network rm "$NETWORK_NAME" 2>/dev/null || true
info "Видалення образу..."
docker rmi "$IMAGE_NAME" 2>/dev/null || true
log "Очищення завершено"
}

deploy_compose() {
    header "Розгортання через Docker Compose"
    cd "$PROJECT_DIR"
    info "Зупинка існуючих сервісів..."
    docker-compose down 2>/dev/null || true
    info "Збірка та запуск..."
    DOCKER_BUILDKIT=1 docker-compose up -d --build
    log "Docker Compose розгортання завершено"
    docker-compose ps
}

main() {
    echo ""
    echo -e
    "${BLUE} ════════════════════════════════════════════════════════════════ ════════════════════════════════════════════════════════════════ ${NC}"
    echo -e "${BLUE} ════════════════════════════════════════════════════════════════ TechShop Secure Deployment Script ════════════════════════════════════════════════════════════════ ${NC}"
    v1.0.0 ════════════════════════════════════════════════════════════════ ════════════════════════════════════════════════════════════════ ${NC}"
    echo -e
    "${BLUE} ════════════════════════════════════════════════════════════════ ════════════════════════════════════════════════════════════════ ${NC}"
    echo ""
    local ACTION="${1:-all}"
    case "$ACTION" in
        build)
            check_prerequisites
            build_image
            ;;
        deploy)
            check_prerequisites
            create_network
            deploy_container
            show_info
            ;;
        compose)
            check_prerequisites
            deploy_compose
            ;;
        verify)
            verify_security
            ;;
        cleanup)
            cleanup
            ;;
    esac
}
```

```

all)
    check_prerequisites
    build_image
    create_network
    deploy_container
    verify_security
    show_info
    ;;
*)
    echo "Використання: $0 [build|deploy|compose|verify|cleanup|all]"
    echo ""
    echo "Команди:"
    echo "  build   - Збірка Docker образу"
    echo "  deploy  - Розгортання контейнера (docker run)"
    echo "  compose - Розгортання через Docker Compose"
    echo "  verify  - Перевірка безпеки"
    echo "  cleanup - Очищення"
    echo "  all     - Виконати всі кроки (за замовчуванням)"
    exit 1
    ;;
esac
echo ""
log "Готово!"
}
main "$@"

```

seccomp-profile.json

```

{
  "_comment": "Seccomp profile for TechShop nginx container - restricts system calls",
  "defaultAction": "SCMP_ACT_ERRNO",
  "defaultErrnoRet": 1,
  "archMap": [
    {
      "architecture": "SCMP_ARCH_X86_64",
      "subArchitectures": [
        "SCMP_ARCH_X86",
        "SCMP_ARCH_X32"
      ]
    },
    {
      "architecture": "SCMP_ARCH_AARCH64",
      "subArchitectures": [
        "SCMP_ARCH_ARM"
      ]
    }
  ],
  "syscalls": [
    {
      "names": [
        "accept", "accept4", "access", "arch_prctl", "bind", "brk",
        "capget", "capset", "chdir", "chmod", "chown", "chown32",
        "clock_getres", "clock_gettime", "clock_nanosleep", "clone", "clone3", "close",
        "connect", "copy_file_range", "dup", "dup2", "dup3",
        "epoll_create", "epoll_create1", "epoll_ctl", "epoll_pwait", "epoll_pwait2", "epoll_wait",
        "eventfd", "eventfd2", "execve", "execveat", "exit", "exit_group",
        "faccessat", "faccessat2", "fadvise64", "fallocate", "fchdir", "fchmod",
        "fchmodat", "fchown", "fchown32", "fchownat", "fcntl", "fcntl64",
        "fdatasync", "fgetxattr", "flistxattr", "flock", "fork", "fsetxattr",
        "fstat", "fstat64", "fstatat64", "fstatfs", "fstatfs64", "fsync",
        "ftruncate", "ftruncate64", "futex", "futex_waitv", "getcwd",
        "getdents", "getdents64", "getegid", "getegid32", "geteuid", "geteuid32",
        "getgid", "getgid32", "getgroups", "getgroups32", "getpeername",
        "getpgid", "getpgrp", "getpid", "getppid", "getpriority", "getrandom",
        "getresgid", "getresgid32", "getresuid", "getresuid32", "getrlimit", "getrusage",
        "getsid", "getsockname", "getsockopt", "gettid", "gettimeofday",
        "getuid", "getuid32", "getxattr",
        "inotify_add_watch", "inotify_init", "inotify_init1", "inotify_rm_watch",
        "io_cancel", "io_destroy", "io_getevents", "io_setup", "io_submit", "ioctl",
        "kill", "lchown", "lchown32", "lgetxattr", "link", "linkat", "listen",
        "listxattr", "llistxattr", "lseek", "lstat", "lstat64",
        "madvise", "membarrier", "memfd_create", "mincore", "mkdir", "mkdirat",
        "mknod", "mknodat", "mlock", "mlock2", "mlockall", "mmap", "mmap2", "mprotect",
        "mq_getsetattr", "mq_notify", "mq_open", "mq_timedreceive", "mq_timedsend", "mq_unlink",
        "mremap", "msgctl", "msgget", "msgrcv", "msgsnd", "msync",
        "munlock", "munlockall", "munmap", "name_to_handle_at", "nanosleep",
        "newfstatat", "open", "openat", "openat2", "pause", "pipe", "pipe2",
        "poll", "ppoll", "prctl", "pread64", "preadv", "preadv2", "prlimit64",
        "pselect6", "pwrite64", "pwritev", "pwritev2", "read", "readahead",
        "readlink", "readlinkat", "readv", "recv", "recvfrom", "recvmmsg", "recvmsg",
        "remap_file_pages", "removexattr", "rename", "renameat", "renameat2",
        "restart_syscall", "rmdir", "rseq",

```

```

"rt_sigaction", "rt_sigpending", "rt_sigprocmask", "rt_sigqueueinfo",
"rt_sigreturn", "rt_sigsuspend", "rt_sigtimedwait", "rt_tgsigqueueinfo",
"sched_get_priority_max", "sched_get_priority_min", "sched_getaffinity",
"sched_getattr", "sched_getparam", "sched_getscheduler", "sched_rr_get_interval",
"sched_setaffinity", "sched_setattr", "sched_setparam", "sched_setscheduler", "sched_yield",
"seccomp", "select", "semctl", "semget", "semop", "semtimedop",
"send", "sendfile", "sendfile64", "sendmmsg", "sendmsg", "sendto",
"set_robust_list", "set_tid_address", "setfsuid", "setfsuid32",
"setfsuid", "setfsuid32", "setgid", "setgid32", "setgroups", "setgroups32",
"setitimer", "setpgid", "setpriority", "setregid", "setregid32",
"setresgid", "setresgid32", "setresuid", "setresuid32", "setreuid", "setreuid32",
"setrlimit", "setsid", "setsockopt", "setuid", "setuid32", "setxattr",
"shmat", "shmctl", "shmdt", "shmget", "shutdown", "sigaltstack",
"signalfd", "signalfd4", "socket", "socketpair", "splice",
"stat", "stat64", "statfs", "statfs64", "statx", "symlink", "symlinkat",
"sync", "sync_file_range", "syncfs", "sysinfo", "tee", "tgkill", "time",
"timer_create", "timer_delete", "timer_getoverrun", "timer_gettime", "timer_settime",
"timerfd_create", "timerfd_gettime", "timerfd_settime", "times", "tkill",
"truncate", "truncate64", "ugetrlimit", "umask", "uname", "unlink", "unlinkat",
"utime", "utimensat", "utimes", "vfork", "vmsplice",
"wait4", "waitid", "waitpid", "write", "writev"
],
"action": "SCMP_ACT_ALLOW"
},
{
  "_comment": "Allow socket creation for network",
  "names": [
    "socket"
  ],
  "action": "SCMP_ACT_ALLOW",
  "args": [
    {
      "index": 0,
      "value": 1,
      "op": "SCMP_CMP_EQ"
    }
  ]
},
{
  "_comment": "Allow socket creation for network (AF_INET)",
  "names": [
    "socket"
  ],
  "action": "SCMP_ACT_ALLOW",
  "args": [
    {
      "index": 0,
      "value": 2,
      "op": "SCMP_CMP_EQ"
    }
  ]
},
{
  "_comment": "Allow socket creation for network (AF_INET6)",
  "names": [
    "socket"
  ],
  "action": "SCMP_ACT_ALLOW",

```

```
    "args": [  
      {  
        "index": 0,  
        "value": 10,  
        "op": "SCMP_CMP_EQ"  
      }  
    ]  
  }  
]
```

Nginx.conf

```

worker_processes auto;
pid /var/run/nginx.pid;
error_log /var/log/nginx/error.log warn;
events {
    worker_connections 1024;
    use epoll;
    multi_accept on;
}
http {
    include /etc/nginx/mime.types;
    default_type application/octet-stream;
    server_tokens off;
    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
        '$status $body_bytes_sent "$http_referer" '
        '"$http_user_agent" "$http_x_forwarded_for"';
    access_log /var/log/nginx/access.log main;
    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;
    keepalive_timeout 65;
    types_hash_max_size 2048;
    gzip on;
    gzip_vary on;
    gzip_proxied any;
    gzip_comp_level 6;
    gzip_min_length 256;
    gzip_types application/javascript application/json application/xml text/css
text/javascript text/plain text/xml;
    client_body_buffer_size 16k;
    client_header_buffer_size 1k;
    client_max_body_size 8m;
    large_client_header_buffers 4 8k;
    client_body_timeout 12s;
    client_header_timeout 12s;
    send_timeout 10s;
    limit_req_zone $binary_remote_addr zone=req_limit:10m rate=10r/s;
    limit_conn_zone $binary_remote_addr zone=conn_limit:10m;
    server {
        listen 8080;
        listen [::]:8080;
        server_name localhost;
        root /usr/share/nginx/html;
        index index.html;
        charset utf-8;
        add_header X-Frame-Options "SAMEORIGIN" always;
        add_header X-Content-Type-Options "nosniff" always;
    }
}

```

```

    add_header X-XSS-Protection "1; mode=block" always;
    add_header Referrer-Policy "strict-origin-when-cross-origin" always;
    add_header Permissions-Policy "geolocation=(), microphone=(), camera=()"
always;
    add_header Content-Security-Policy "default-src 'self'; script-src 'self'
'unsafe-inline' 'unsafe-eval'; style-src 'self' 'unsafe-inline'
https://fonts.googleapis.com; font-src 'self' https://fonts.gstatic.com data;; img-
src 'self' data: https;; connect-src 'self'; frame-ancestors 'self';" always;
    add_header Cross-Origin-Opener-Policy "same-origin" always;
    add_header Cross-Origin-Resource-Policy "same-origin" always;
    limit_req zone=req_limit burst=20 nodelay;
    limit_conn conn_limit 10;
    location /health {
        access_log off;
        add_header Content-Type text/plain;
        return 200 "OK";
    }
    location ~* \.(js|css|png|jpg|jpeg|gif|ico|svg|woff|woff2)$ {
        expires 1y;
        add_header Cache-Control "public, max-age=31536000, immutable";
        add_header X-Content-Type-Options "nosniff" always;
    }
    location / {
        try_files $uri $uri/ /index.html;
        location = /index.html {
            add_header Cache-Control "no-cache, no-store, must-revalidate";
            add_header Pragma "no-cache";
            add_header Expires "0";
            add_header X-Frame-Options "SAMEORIGIN" always;
            add_header X-Content-Type-Options "nosniff" always;
            add_header X-XSS-Protection "1; mode=block" always;
        }
    }
    location ~ /\. {
        deny all;
        access_log off;
        log_not_found off;
    }
    location ~* \.(htaccess|htpasswd|ini|log|sh|sql|conf|bak)$ {
        deny all;
    }
    error_page 404 /index.html;
    error_page 500 502 503 504 /index.html;
    if ($request_method !~ ^(GET|HEAD|POST)$) {
        return 405;
    }
}
}

```

generate-tls-certs.sh

```
#!/bin/bash
set -euo pipefail
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
BLUE='\033[0;34m'
NC='\033[0m'
DOCKER_HOST="${1:-localhost}"
DOCKER_IP="${2:-127.0.0.1}"
CERT_DIR="${3:-./certs}"
VALIDITY_DAYS=365
KEY_SIZE=4096
log() { echo -e "${GREEN}[INFO]${NC} $1"; }
warn() { echo -e "${YELLOW}[WARN]${NC} $1"; }
error() { echo -e "${RED}[ERROR]${NC} $1"; exit 1; }
if ! command -v openssl &> /dev/null; then
    error "OpenSSL не встановлено. Встановіть: apt-get install openssl"
fi
log "Створення директорії для сертифікатів: $CERT_DIR"
mkdir -p "$CERT_DIR"
cd "$CERT_DIR"
echo ""
echo -e
"${BLUE}=====
=${NC}"
echo -e "${BLUE}    TLS Certificate Generation for Docker Daemon${NC}"
echo -e
"${BLUE}=====
=${NC}"
echo ""
echo "Hostname: $DOCKER_HOST"
echo "IP Address: $DOCKER_IP"
echo "Validity: $VALIDITY_DAYS days"
echo "Key Size: $KEY_SIZE bits"
echo ""
log "STEP 1: Генерація CA (Certificate Authority)..."
openssl genrsa -aes256 -passout pass:capassword -out ca-key.pem $KEY_SIZE 2>/dev/null
log "    ✓ CA private key: ca-key.pem"
openssl req -new -x509 -days $VALIDITY_DAYS -key ca-key.pem -sha256 -out ca.pem \
    -passin pass:capassword \
    -subj "/C=UA/ST=Kyiv/L=Kyiv/O=TechShop Security/OU=Docker CA/CN=Docker CA"
log "    ✓ CA certificate: ca.pem"
log "STEP 2: Генерація серверних сертифікатів..."
openssl genrsa -out server-key.pem $KEY_SIZE 2>/dev/null
log "    ✓ Server private key: server-key.pem"
openssl req -subj "/CN=$DOCKER_HOST" -sha256 -new -key server-key.pem -out server.csr
```

```

log " ✓ Server CSR: server.csr"
cat > extfile.cnf <<EOF
subjectAltName = DNS:$DOCKER_HOST,DNS:localhost,IP:$DOCKER_IP,IP:127.0.0.1
extendedKeyUsage = serverAuth
keyUsage = digitalSignature, keyEncipherment
EOF
log " ✓ Server extensions: extfile.cnf"
openssl x509 -req -days $VALIDITY_DAYS -sha256 \
    -in server.csr -CA ca.pem -CAkey ca-key.pem \
    -CAcreateserial -out server-cert.pem -extfile extfile.cnf \
    -passin pass:capassword
log " ✓ Server certificate: server-cert.pem"
log "STEP 3: Генерація клієнтських сертифікатів..."
openssl genrsa -out key.pem $KEY_SIZE 2>/dev/null
log " ✓ Client private key: key.pem"
openssl req -subj '/CN=docker-client' -new -key key.pem -out client.csr
log " ✓ Client CSR: client.csr"
cat > extfile-client.cnf <<EOF
extendedKeyUsage = clientAuth
keyUsage = digitalSignature
EOF
log " ✓ Client extensions: extfile-client.cnf"
openssl x509 -req -days $VALIDITY_DAYS -sha256 \
    -in client.csr -CA ca.pem -CAkey ca-key.pem \
    -CAcreateserial -out cert.pem -extfile extfile-client.cnf \
    -passin pass:capassword
log " ✓ Client certificate: cert.pem"
log "STEP 4: Очищення та налаштування прав..."
rm -f server.csr client.csr extfile.cnf extfile-client.cnf ca.srl
chmod 600 ca-key.pem server-key.pem key.pem
chmod 644 ca.pem server-cert.pem cert.pem
log " ✓ Тимчасові файли видалено"
log " ✓ Права доступу налаштовано"
log "STEP 5: Верифікація сертифікатів..."
openssl verify -CAfile ca.pem server-cert.pem >/dev/null 2>&1 && \
    log " ✓ Server certificate: VALID" || \
    error "Server certificate verification failed"
openssl verify -CAfile ca.pem cert.pem >/dev/null 2>&1 && \
    log " ✓ Client certificate: VALID" || \
    error "Client certificate verification failed"
echo ""
echo -e
"${GREEN}=====
==${NC}"
echo -e "${GREEN} Сертифікати успішно згенеровані!${NC}"
echo -e
"${GREEN}=====
==${NC}"
echo ""

```

```

echo "Згенеровані файли:"
echo "  CA сертифікат:      $CERT_DIR/ca.pem"
echo "  CA ключ:             $CERT_DIR/ca-key.pem"
echo "  Серверний сертифікат: $CERT_DIR/server-cert.pem"
echo "  Серверний ключ:       $CERT_DIR/server-key.pem"
echo "  Клієнтський сертифікат: $CERT_DIR/cert.pem"
echo "  Клієнтський ключ:     $CERT_DIR/key.pem"
echo ""
echo -e "${YELLOW}ВАЖЛИВО: Збережіть ca-key.pem в безпечному місці та видаліть з сер-
вера!${NC}"
echo ""

```

Додаток G

daemon.json.example

```

{
  "hosts": ["unix:///var/run/docker.sock"],
  "userns-remap": "default",
  "no-new-privileges": true,
  "selinux-enabled": false,
  "seccomp-profile": "/etc/docker/seccomp-default.json",
  "icc": false,
  "userland-proxy": false,
  "ip-forward": true,
  "iptables": true,
  "ip-masq": true,
  "storage-driver": "overlay2",
  "storage-opts": ["overlay2.override_kernel_check=true"],
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m",
    "max-file": "3",
    "labels": "production_status",
    "env": "os,customer"
  },
  "log-level": "info",
  "default-ulimits": {

```

```

    "nofile": {"Name": "nofile", "Hard": 65535, "Soft": 65535},
    "nproc": {"Name": "nproc", "Hard": 4096, "Soft": 4096},
    "memlock": {"Name": "memlock", "Hard": -1, "Soft": -1}
  },
  "default-runtime": "runc",
  "runtimes": {
    "runc": {"path": "runc"}
  },
  "experimental": false,
  "metrics-addr": "127.0.0.1:9323",
  "live-restore": true,
  "cgroup-parent": "/docker",
  "default-cgroupns-mode": "private",
  "dns": ["8.8.8.8", "8.8.4.4"],
  "dns-search": ["example.com"],
  "insecure-registries": [],
  "registry-mirrors": [],
  "shutdown-timeout": 15
}

```

Додаток Н

Apparmor-profile

```

#include <tunables/global>
profile docker-techshop flags=(attach_disconnected,mediate_deleted) {
  #include <abstractions/base>
  #include <abstractions/nameservice>
  network inet tcp,
  network inet udp,
  network inet6 tcp,
  network inet6 udp,
  network unix stream,
  network unix dgram,
  deny network raw,
  deny network packet,
  capability chown,
  capability dac_override,
  capability dac_read_search,
  capability fowner,
  capability fsetid,
  capability kill,

```

```
capability setgid,
capability setuid,
capability net_bind_service,
deny capability sys_admin,
deny capability sys_ptrace,
deny capability sys_module,
deny capability sys_rawio,
deny capability sys_boot,
deny capability sys_nice,
deny capability sys_resource,
deny capability sys_time,
deny capability mknod,
deny capability net_admin,
deny capability net_raw,
deny capability audit_control,
deny capability audit_write,
/usr/sbin/nginx ix,
/usr/sbin/nginx mr,
/lib/** mr,
/lib64/** mr,
/usr/lib/** mr,
/usr/lib64/** mr,
/etc/nginx/** r,
/etc/nginx/nginx.conf r,
/etc/nginx/conf.d/** r,
/etc/nginx/mime.types r,
/usr/share/nginx/html/** r,
/var/cache/nginx/** rw,
/var/cache/nginx/client_temp/** rw,
/var/cache/nginx/proxy_temp/** rw,
/var/cache/nginx/fastcgi_temp/** rw,
/var/cache/nginx/uwsgi_temp/** rw,
/var/cache/nginx/scgi_temp/** rw,
/var/log/nginx/** rw,
/var/log/nginx/access.log w,
/var/log/nginx/error.log w,
/var/run/nginx.pid rw,
/run/nginx.pid rw,
/tmp/** rw,
/var/tmp/** rw,
/etc/passwd r,
/etc/group r,
/etc/nsswitch.conf r,
/etc/host.conf r,
/etc/hosts r,
/etc/resolv.conf r,
/etc/localtime r,
/etc/timezone r,
/etc/ssl/** r,
```

```

/etc/ca-certificates/** r,
/proc/sys/kernel/random/uuid r,
/proc/sys/kernel/ngroups_max r,
/proc/sys/net/** r,
/proc/*/fd/ r,
/proc/*/stat r,
/proc/*/status r,
/sys/kernel/mm/transparent_hugepage/hpage_pmd_size r,
/dev/null rw,
/dev/zero r,
/dev/urandom r,
/dev/random r,
deny /etc/shadow r,
deny /etc/shadow- r,
deny /etc/gshadow r,
deny /etc/gshadow- r,
deny /etc/sudoers r,
deny /etc/sudoers.d/** r,
deny /etc/ssh/** rw,
deny /root/** rw,
deny /home/*/.ssh/** rw,
deny /var/run/docker.sock rw,
deny /run/docker.sock rw,
deny /boot/** rw,
deny /lib/modules/** rw,
deny /sys/firmware/** rw,
deny mount,
deny umount,
deny pivot_root,
signal send set=(term, kill, hup, int, quit, usr1, usr2) peer=docker-techshop,
signal receive set=(term, kill, hup, int, quit, usr1, usr2),
deny ptrace,
}

```