Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя
(повне найменування вищого навчального закладу)
Факультет комп'ютерно-інформаційних систем і програмної інженерії
(назва факультету)
Кафедра кібербезпеки
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(освітній рівень)

на тему: "Механізм безпечного оновлення контейнерів із застосуванням криптографії"

Виконав: студент VI курсу, групи СБм-61
Спеціальності:
125 Кібербезпека та захист інформації
(шифр і назва напряму підготовки, спеціальності)
Вівчарівський Назарій Ігорович
підпис (прізвище та ініціали)

Керівник Лечаченко Т. А.
підпис (прізвище та ініціали)

Нормоконтроль Стадник М. А.
підпис (прізвище та ініціали)

Завідувач кафедри Загородна Н.В.
підпис (прізвище та ініціали)

Рецензент
підпис (прізвище та ініціали)

м. Тернопіль – 2025

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра кібербезпеки
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Загородна Н.В.
(підпис) (прізвище та ініціали)
«__» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Магістр
(назва освітнього ступеня)

за спеціальністю 125 Кібербезпека та захист інформації
(шифр і назва спеціальності)

Студенту Вівчарівському Назарієві Ігоровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Механізм безпечного оновлення контейнерів із застосуванням криптографії

Керівник роботи Лечаченко Тарас Анатолійович,
доктор філософії, старший викладач кафедри КБ
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від « 24 » 11 2025 року № 4/7-1024

2. Термін подання студентом завершеної роботи 12.12.2025

3. Вихідні дані до роботи Git, Docker, GitHub Actions, GitHub Container Registry, Sigstore Cosign, Linux

4. Зміст роботи (перелік питань, які потрібно розробити)
Аналіз сучасних контейнерних технологій та загроз безпеці оновлень.
Дослідження криптографічних методів захисту контейнерних оновлень.
Огляд та порівняльний аналіз існуючих рішень для захищеного оновлення контейнерів.
Проєктування архітектури захищеного CI/CD-конвеєра з наскрізним ланцюжком довіри.
Реалізація, тестування та оцінка ефективності механізму безпечного оновлення контейнерів.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
Актуальність дослідження. Мета, об'єкт, предмет дослідження.
Наукова новизна та практичне значення.
Завдання дослідження.
Загрози безпеці контейнерних оновлень.
Криптографічні основи рішення.
Реалізація CI/CD (GitHub Actions).
Архітектура захищеного CI/CD. Ланцюжок довіри.
Механізм оновлення на Raspberry Pi. Результати та перевірка роботи.
Висновки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці	Осухівська Г.М., к.т.н., доцент		
Безпека в надзвичайних ситуаціях	Теслюк В.М., проректор з адміністративно-господарської роботи та будівництва		

7. Дата видачі завдання 19.09.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	20.09 – 22.09	Виконано
2.	Підбір джерел для аналізу в галузі дослідження	25.09 – 10.10	Виконано
3.	Опрацювання джерел в галузі дослідження	11.10 – 15.10	Виконано
4.	Налаштування тестового середовища	16.10 – 25.10	Виконано
5.	Оформлення розділу «Огляд предметної області»	25.10 – 05.11	Виконано
6.	Оформлення розділу «Архітектура захищеного CI/CD та ланцюжок довіри»	06.11 – 10.11	Виконано
7.	Оформлення розділу «Реалізація захищеного CI/CD-конвеєра для контейнерних систем»	11.11 – 25.11	Виконано
8.	Виконання завдання до підрозділу «Охорона праці та безпека в надзвичайних ситуаціях»	26.11-01.12	
9.	Оформлення кваліфікаційної роботи	02.12 – 10.12	Виконано
10.	Нормоконтроль	08.12 – 10.12	Виконано
11.	Перевірка на плагіат	12.12 – 14.12	Виконано
12.	Попередній захист кваліфікаційної роботи	15.12 – 20.12	Виконано
13.	Захист кваліфікаційної роботи	22.12.2025	

Студент _____ Вівчарівський. Н. І.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Лечаченко Т. А.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Механізм безпечного оновлення контейнерів із застосуванням криптографії // ОР «Магістр» // Вівчарівський Назарій Ігорович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра кібербезпеки, група СБм-61 // Тернопіль, 2025 // С. 74, рис. – 41, табл. – - , кресл. – 13, додат. – 1.

Ключові слова: Cosign, GitHub Actions, Docker, CI/CD, Sigstore, SLSA.

У кваліфікаційній роботі магістра досліджено та реалізовано механізм безпечного оновлення контейнеризованих застосунків із використанням криптографічних методів перевірки автентичності та цілісності програмного забезпечення. Запропоноване рішення базується на побудові захищеного CI/CD-конвеєра з наскрізним ланцюжком довіри, що охоплює всі етапи життєвого циклу контейнерного застосунку, від формування релізу до автоматичного розгортання на кінцевому пристрої. У першому розділі виконано огляд контейнерних технологій і процесів безперервної інтеграції та доставки, а також проаналізовано загрози безпеці, пов'язані з атаками на ланцюжок постачання програмного забезпечення. Розглянуто криптографічні основи захищених оновлень і існуючі рішення для перевірки автентичності контейнерних образів. У другому розділі спроєктовано архітектуру захищеного CI/CD-конвеєра з використанням GitHub Actions, приватного контейнерного реєстру та технологій Sigstore Cosign, описано механізми keyless-підпису та формування атестацій походження збірки. У третьому розділі реалізовано прототип системи безпечного оновлення контейнерів на сервері Raspberry Pi з автоматизованою перевіркою цифрових підписів, атестацій походження та коректності версій перед розгортанням застосунків.

Результати тестування підтвердили ефективність запропонованого підходу та можливість його практичного використання в DevOps-, IoT- та edge-середовищах для підвищення рівня інформаційної безпеки.

ABSTRACT

Secure container update mechanism using cryptography // Thesis of educational level "Master"// Nazarii Vivcharivskyi // Ternopil Ivan Puluj National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Cybersecurity, group СБм-61 // Ternopil, 2025 // p. 74, figs. 41, tbls. - , drws. 13, apps. 1.

Keywords: Cosign, GitHub Actions, Docker, CI/CD, Sigstore, SLSA.

The master's thesis explores and implements a mechanism for securely updating containerized applications using cryptographic methods to verify the authenticity and integrity of software. The proposed solution is based on the construction of a secure CI/CD pipeline with an end-to-end chain of trust covering all stages of the container application lifecycle, from release formation to automatic deployment on the end device. The first chapter provides an overview of container technologies and continuous integration and delivery processes, as well as an analysis of security threats associated with attacks on the software supply chain. The cryptographic foundations of secure updates and existing solutions for authenticating container images are discussed. The second chapter designs the architecture of a secure CI/CD pipeline using GitHub Actions, a private container registry, and Sigstore Cosign technologies, and describes the mechanisms of keyless signing and the formation of build origin attestations. The third section implements a prototype of a secure container update system on a Raspberry Pi server with automated verification of digital signatures, attestations of origin, and version correctness before application deployment.

Testing results confirmed the effectiveness of the proposed approach and its practical applicability in DevOps, IoT, and edge environments to improve information security.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

CI	—	Continuous Integration
CD	—	Continuous Deployment
GHCR	—	GitHub Container Registry
IoT	—	Internet of Things
HTTPS	—	Hypertext Transfer Protocol Secure
CA	—	Certificate Authority
SHA	—	Secure Hash Algorithm
DCT	—	Docker Content Trust
SLSA	—	Supply-chain Levels for Software Artifacts
API	—	Application Programming Interface
ARM	—	Advanced RISC Machine
OPA	—	Open Policy Agent
JSON	—	JavaScript Object Notation
PAT	—	Personal Access Token
RSA	—	Rivest–Shamir–Adleman
WAN	—	Wide Area Network

ВСТУП

Актуальність теми. Сучасні програмні системи дедалі частіше розгортаються з використанням контейнерних технологій та автоматизованих CI/CD-процесів, що суттєво прискорює розробку й оновлення програмного забезпечення, але водночас підвищує ризики безпеки, пов’язані з атаками на ланцюжок постачання. Компрометація CI/CD-конвеєрів або підміна контейнерних образів може призвести до розгортання шкідливого коду в продуктивних середовищах. У зв’язку з активним використанням контейнеризації в хмарних, edge- та IoT-системах актуальним є впровадження механізмів криптографічної перевірки автентичності й цілісності оновлень. Тому дослідження та реалізація захищених CI/CD-конвеєрів із наскрізним ланцюжком довіри є важливим науково-практичним завданням.

Мета і задачі дослідження. Метою кваліфікаційної роботи є розробка та реалізація механізму безпечного оновлення контейнеризованих застосунків на основі захищеного CI/CD-конвеєра з використанням криптографічних методів перевірки автентичності та цілісності програмного забезпечення.

Для досягнення цієї мети необхідно вирішити такі задачі:

- провести аналіз сучасних контейнерних технологій та загроз безпеці, пов’язаних із процесом оновлення контейнерів;
- дослідити криптографічні механізми, що використовуються для забезпечення цілісності та автентичності програмних артефактів;
- проаналізувати існуючі рішення для захищеного оновлення контейнерів і побудови ланцюжка довіри;
- спроєктувати архітектуру захищеного CI/CD-конвеєра з використанням цифрових підписів та атестацій походження;
- реалізувати автоматизований процес збірки, підпису та публікації контейнерних образів;
- розробити механізм безпечного оновлення контейнерів на кінцевому пристрої з перевіркою підписів і захистом від rollback-атак;
- провести тестування розробленої системи та оцінити її ефективність.

Об’єкт дослідження. Процес безпечного оновлення програмного забезпечення в контейнерних системах із використанням автоматизованих CI/CD-процесів.

Предмет дослідження. Методи та засоби забезпечення автентичності, цілісності та довіри до контейнерних образів у процесі автоматизованого оновлення на основі криптографічних механізмів.

Наукова новизна одержаних результатів кваліфікаційної роботи. Наукова новизна роботи полягає в удосконаленні підходів до захисту ланцюжка постачання програмного забезпечення шляхом побудови захищеного CI/CD-конвеєра з наскрізним ланцюжком довіри. Запропонований підхід поєднує використання keyless-криптографічних підписів, атестацій походження збірки та автоматизованої верифікації оновлень на кінцевому вузлі, що дозволяє мінімізувати ризики компрометації програмних артефактів без зберігання довготривалих секретних ключів.

Практичне значення одержаних результатів. Практичне значення результатів полягає у можливості застосування розробленого механізму в реальних DevOps- та IoT-системах для безпечного автоматичного оновлення контейнеризованих застосунків. Запропоноване рішення може бути використане в CI/CD-процесах, edge- та embedded-середовищах, а також слугувати основою для подальшого розвитку систем захисту від атак на ланцюжок постачання програмного забезпечення.

Апробація результатів магістерської роботи. Основні результати дослідження були представлені на XIII науково-технічній конференції «Інформаційні моделі, системи та технології» (ТНТУ, Тернопіль, Україна, 17-18 грудня 2025 р).

Публікації. Основні результати кваліфікаційної роботи опубліковано у працях конференції (див. Додаток А).

РОЗДІЛ 1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Сучасні системи контейнерного розгортання та безпека оновлень

Контейнеризація стала невід'ємною частиною сучасної розробки і розгортання застосунків. Docker забезпечує створення ізольованих контейнерів, що пакують застосунок з усіма залежностями, а Kubernetes дозволяє масштабовано оркеструвати запуск і оновлення цих контейнерів у кластері. Станом на 2024 рік кількість завантажених контейнерних образів перевищила 15 мільярдів, проте близько 75% образів містять вразливості високого або критичного рівня [1]. Це підкреслює важливість своєчасного оновлення контейнерів для усунення відомих вразливостей. Зволікання з оновленнями призводить до накопичення відомих дефектів безпеки у контейнерних службах.

На рисунку 1.1 показано типову архітектуру контейнерного середовища, що включає контейнерний реєстр, вузли кластера Kubernetes, pod-и з контейнерами та механізм оркестрації.

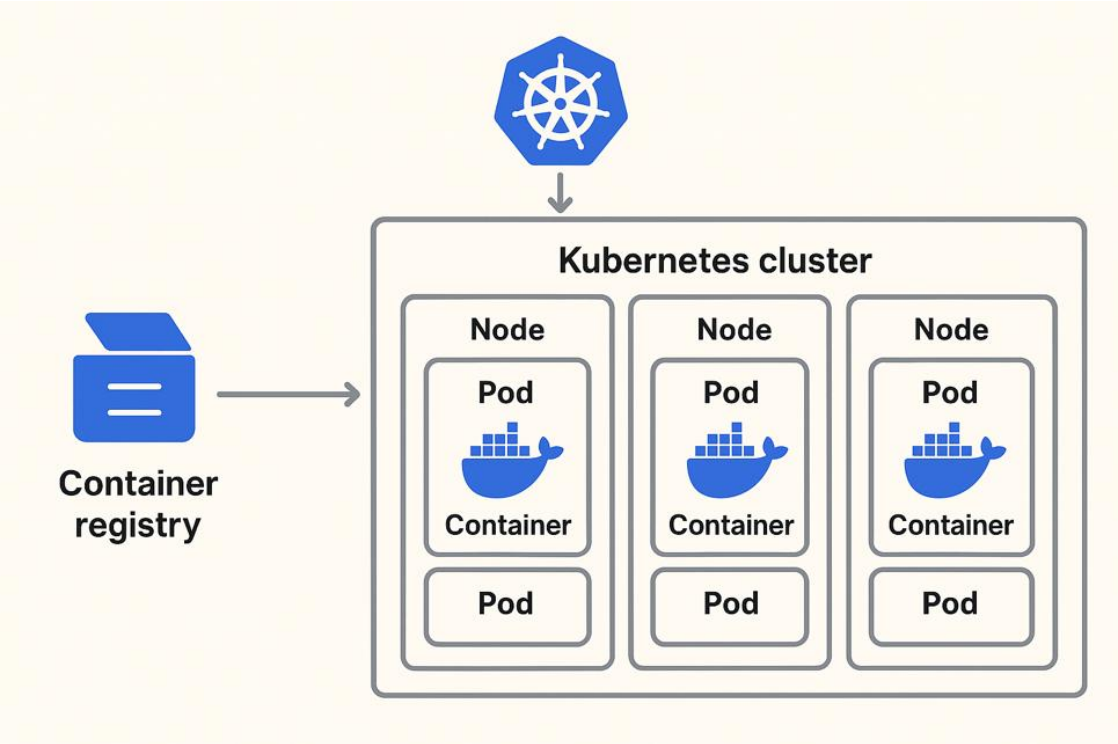

Рисунок 1.1 – Загальна схема контейнерної інфраструктури Docker та Kubernetes

Контейнерні образи зберігаються в реєстрі та завантажуються в кластер під час розгортання або оновлення сервісів.

Втім, процес оновлення також має свої ризики. Новий образ, отриманий із ненадійного джерела або змінений зловмисниками, може стати троянським конем у інфраструктурі. Дослідження показують, що приблизно 87% базових образів містять критичні вразливості, і використання застарілих або неперевірених образів із публічних реєстрів (наприклад, Docker Hub) може призвести до впровадження шкідливого коду або бекдорів [1]. Відомі випадки, коли зловмисники розповсюджували шкідливі контейнери через Docker Hub, вбудовуючи майнери криптовалюти чи крадії облікових даних у образи, які нічого не підозрюючі користувачі завантажували у свої системи [2]. Інший інцидент 2021 року продемонстрував ризики неправильної конфігурації: через відкритий Docker API зловмисники встановили майнер на незахищені хост-системи контейнерів [1].

Контейнерні оркестратори на кшталт Kubernetes автоматизують розгортання та rolling-update контейнерів, але за замовчуванням не перевіряють автентичність образів. Це означає, що якщо в реєстр потрапить скомпрометований образ під очікуваною міткою, кластер його завантажить. Був зафіксований випадок, коли витік даних в Kubernetes-кластері стався через запуск підробленого контейнера: хоча у конвеєрі CI/CD проводилося сканування, зловмисному образу вдалося пройти в внутрішній реєстр і на кластер, оскільки політики перевіряли лише назву образу, а не його вміст. Лише впровадження політики довіри до вмісту (підписування та верифікації контейнерних образів) могло гарантувати, що в продакшн потрапляють саме ті збірки, які пройшли перевірку безпеки [3]. Таким чином, атаки на ланцюжок постачання програмного забезпечення (supply chain) стали реальною загрозою для контейнерних середовищ [1], і питання «чи можна довіряти цьому образу?» стало ключовим для безпеки контейнерів.

Для захисту від описаних загроз необхідно забезпечити довірене оновлення контейнерів. По-перше, трафік між клієнтом Docker і реєстром має бути

шифрований (HTTPS/TLS), щоб виключити можливість атаки «людина посередині» (MitM) при передачі образів [17-19]. Якщо образи передаються незашифрованим каналом, атакувальник може перехопити та підмінити дані на льоту [4]. По-друге, і головне – слід гарантувати цілісність та походження самих образів. Для цього в Docker запроваджена функція Content Trust – механізм криптографічного підписування образів. З увімкненою довірою до вмісту Docker-клієнт перевіряє, що всі завантажувані образи мають дійсний цифровий підпис від довіреного видавця [1]. Іншими словами, до виконання допускаються лише підписані та криптографічно перевірені образи, що підтверджує їх автентичність і цілісність [1]. Це суттєво ускладнює зловмисникам можливість непомітно підмінити образ або вставити шкідливий код при оновленні контейнера. Таким чином, сучасні системи контейнерного розгортання, доповнені механізмами криптографічного підпису, дозволяють реалізувати безпечне оновлення контейнерів, зберігаючи баланс між оперативністю деплоїв та довірою до їхнього вмісту.

1.2 Криптографічні алгоритми та механізми підпису

Основою механізму безпечного оновлення є використання перевірених методів криптографії для забезпечення конфіденційності, цілісності та автентичності даних. Існують два фундаментальних підходи до шифрування даних: симетричне та асиметричне шифрування. Симетричне шифрування використовує один і той самий секретний ключ як для шифрування, так і для розшифрування інформації. Натомість асиметричне шифрування оперує парою пов’язаних ключів – відкритим (публічним) і закритим (приватним) [5]. Приватний ключ зберігається в таємниці й використовується для шифрування або підпису, тоді як публічний ключ можна вільно розповсюджувати для розшифрування або перевірки підпису. Завдяки цьому асиметрична криптографія забезпечує можливість підтвердження авторства та цілісності даних без передачі секретів – це свого роду цифрове рукостискання, яке надзвичайно складно підробити [6]. Симетричні алгоритми (наприклад, AES)

зазвичай швидші та підходять для шифрування великих обсягів даних, тоді як асиметричні (RSA, ECC) забезпечують вищий рівень безпеки для обміну ключами і створення цифрових підписів. Обидва підходи часто комбінуються: симетричне шифрування використовується для захисту вмісту (конфіденційність), а асиметричне – для безпечної передачі ключів та підтвердження автентичності (наприклад, у протоколі TLS симетричний сеансовий ключ передається з шифруванням відкритим ключем).

На рисунку 1.2 схема ілюструє принцип асиметричного шифрування, при якому для шифрування або перевірки підпису використовується відкритий ключ, а для створення підпису або розшифрування – приватний ключ, що зберігається в таємниці.

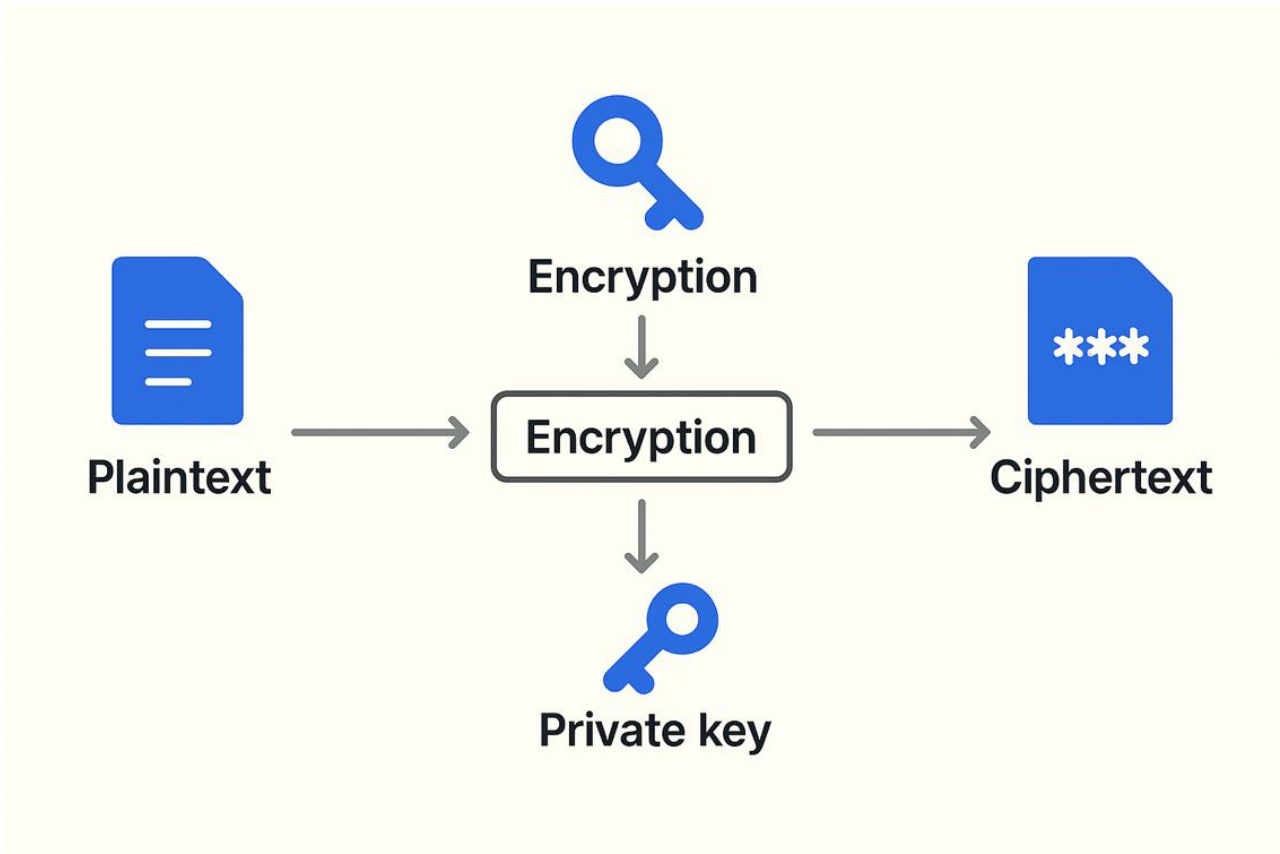

Рисунок 1.2 – Узагальнена схема процесу шифрування з використанням криптографічного ключа

Для контролю цілісності даних застосовуються криптографічні геш-функції. Геш-функція обчислює унікальний «відбиток» довільного блоку даних фіксованої довжини. Важливо, що найменша зміна вхідних даних призводить до

цілковито іншого значення гешу. Це широко використовується для перевірки цілісності: розрахувавши геш отриманого файлу і порівнявши з еталонним значенням, можна впевнитися, що файл не було змінено. Сучасні алгоритми гешування (SHA-256, SHA-3 тощо) стійкі до підбору: практично неможливо знайти інші дані, що дають той самий геш. На рисунку 1.3 показано, як контейнерний образ хешується за допомогою криптографічної геш-функції (наприклад, SHA-256), після чого геш шифрується приватним ключем розробника, утворюючи цифровий підпис.

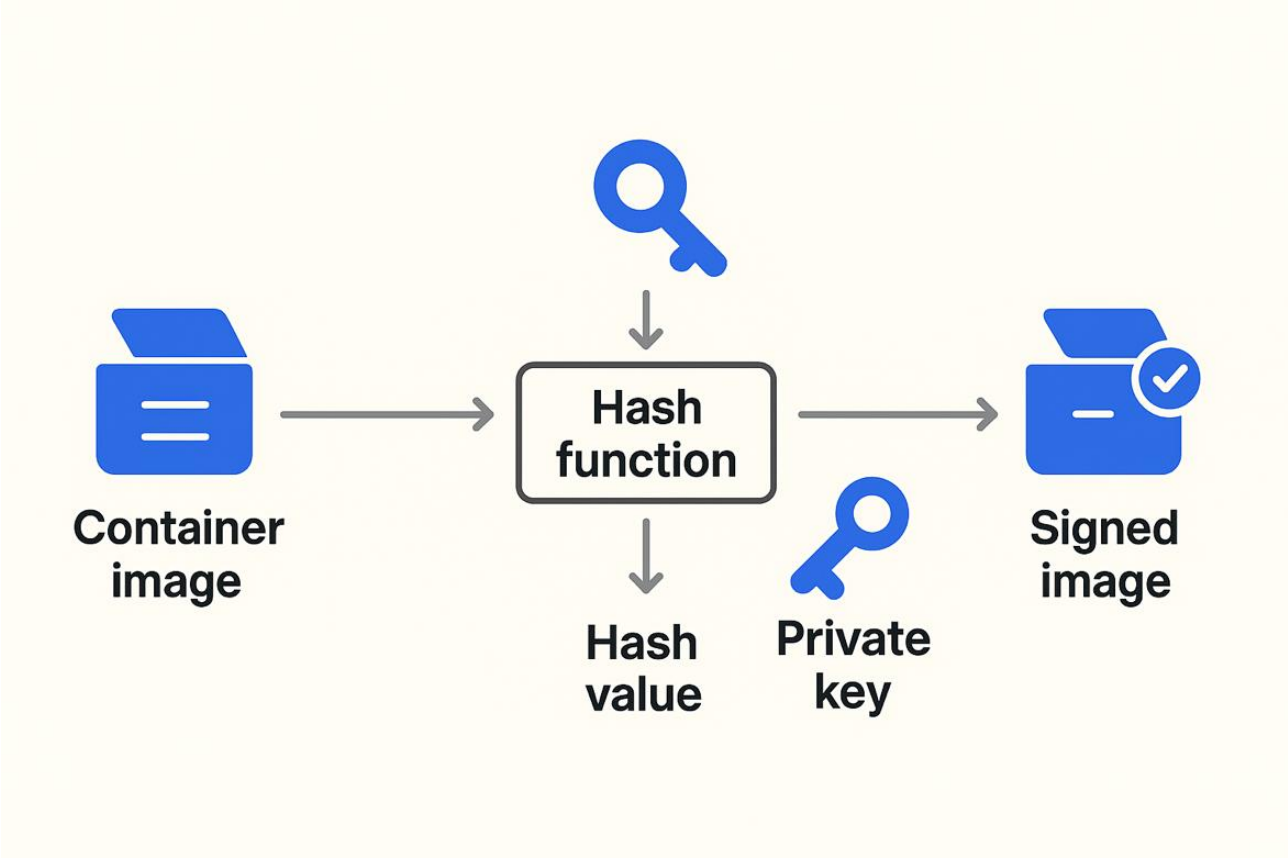

Рисунок 1.3 – Процес створення цифрового підпису образу контейнера

Під час оновлення контейнера система перевіряє підпис за допомогою публічного ключа, що дозволяє гарантувати цілісність та автентичність образу.

Комбінуючи гешування з асиметричною криптографією, отримуємо механізм цифрового підпису. Цифровий підпис слугує аналогом власноручного підпису в електронному світі і забезпечує автентифікацію відправника та цілісність даних. Схема створення підпису така: для файлу (наприклад, образу контейнера) обчислюється криптографічний геш (наприклад, SHA-256), що

однозначно представляє його вміст [4]. Далі цей геш підписується приватним ключем відправника за допомогою алгоритму цифрового підпису – результатом є власне цифровий підпис [4]. Підпис зазвичай зберігається окремо від даних (наприклад, у реєстрі контейнерів поруч з образом) або вкладається як метадані. Для перевірки підпису приймач (Docker-клієнт чи Kubernetes-кластер) повторно гешує отриманий образ і розшифровує цифровий підпис за допомогою публічного ключа відправника. В результаті розшифрування має бути відновлено оригінальне значення гешу, згенероване при підписанні. Якщо обчислений геш образу співпадає з тим, що міститься у підписі, це підтверджує справжність та цілісність образу – він не був змінений і дійсно підписаний довіреним видавцем [4]. У разі будь-яких розбіжностей перевірка провалюється, і такий образ відхиляється як підроблений або пошкоджений.

Варто зазначити, що для управління довіреними ключами використовується інфраструктура відкритих ключів (PKI). Публічні ключі підписантів можуть підтверджуватися сертифікатами, які видають довірені сертифікаційні центри (Certificate Authority). Така багато-рівнева модель довіри схожа на механізм TLS-сертифікатів у веб-браузерах [7]. У випадку контейнерів це означає, що розробник або компанія-автор образу може мати власний сертифікат підпису, довірений організацією-споживачем. Таким чином, за допомогою криптографічних алгоритмів шифрування, гешування та цифрового підпису будується надійний механізм перевірки оновлень: навіть перебуваючи у відкритому середовищі (наприклад, публічному реєстрі Docker Hub), підписаний образ не може бути непомітно змінений або підроблений без втрати дійсності підпису.

1.3 Існуючі рішення для захищеного оновлення контейнерів

Концепція безпечних оновлень контейнерів реалізована у низці сучасних інструментів та продуктів. Першим значним кроком у цій галузі стало впровадження Docker Content Trust (DCT). Docker Content Trust – це вбудована функція Docker, що надає можливість криптографічно перевіряти походження та

цілісність контейнерних образів перед їх завантаженням або запуском [8]. DCT використовує інфраструктуру відкритих ключів і цифрові підписи: видавець підписує образ приватним ключем, а Docker-демон на стороні споживача перевіряє цей підпис за допомогою відповідного публічного ключа на спеціальному сервері довіри (Notary) [8]. Якщо підпис відсутній або недійсний, Docker відмовиться завантажувати такий образ. За увімкненої довіри до вмісту (через змінну DOCKER_CONTENT_TRUST=1) будь-яка спроба виконати непідписаний образ завершується помилкою – тобто, гарантується, що в кластері працюватимуть тільки довірені образи [8]. Це суттєво знижує ризик запуску підроблених або скомпрометованих контейнерів у продакшні. Внутрішньо Docker Content Trust побудований на основі проекту Notary v1 і стандартизованого протоколу The Update Framework (TUF) [8]. TUF забезпечує багаторівневу ієрархію ключів (кореневі, делеговані тощо), що дозволяє мінімізувати наслідки компрометації окремих ключів та гарантує свіжість оновлень (захист від атак повтору) [8]. Завдяки цьому, навіть у разі викрадення ключа підпису окремого репозиторію, є механізми відкликання довіри та перевидачі сертифікатів без компрометації всієї системи.

За час, що минув з появи Docker Content Trust (вперше представлений у Docker 1.8 у 2015 році), екосистема засобів підписування контейнерів розвивалася. Останніми роками спільнота перейшла до нових рішень, оскільки Notary v1 перестав активно підтримуватися, а відсоток використання DCT виявився незначним (менше 0,05% завантажень з Docker Hub в 2025 році) [9]. Вже у 2025 році компанія Docker оголосила про поетапне виведення Docker Content Trust з експлуатації та перехід до сучасніших механізмів підпису образів [9]. Зокрема, розробникам рекомендується мігрувати на відкриті рішення на кшталт Sigstore/Cosign або проєкт Notation (Notary v2) для підписування та перевірки контейнерів [9].

Sigstore – це відносно нова ініціатива, запущена в співпраці Google, Red Hat та спільноти Linux Foundation, яка націлена спростити процес підпису артефактів програмного забезпечення. До складу Sigstore входить утиліта Cosign, що спеціалізується на підписуванні контейнерних образів. Cosign дозволяє

генерувати ключі підпису або навіть використовувати тимчасові «безключові» підписи, інтегруючись з сервісами OIDC для отримання сертифікатів підписанта. Підпис Cosign зберігається в реєстрі контейнерів як окремий об’єкт (OCI artifact), пов’язаний з відповідним образом. При налаштуванні Kubernetes-кластера можна задіяти політики перевірки підписів (наприклад, за допомогою спеціальних admission controller’ів або інтеграції з Kyverno чи Connaisseur), щоб кластер автоматично відхиляв розгортання непідписаних або недовірених образів [3]. Вже зараз багато хмарних реєстрів підтримують Sigstore/Cosign: зокрема, Harbor (популярний корпоративний реєстр) з версії 2.5 додав підтримку Cosign поряд із Notary, що дозволяє користувачам підписувати образи та реплікувати підписи разом з образами між репозиторіями [10]. Таким чином, відкриті стандарти підписування OCI-образів стають загальноприйнятими, а інструменти на зразок Cosign швидко набувають популярності як прості та ефективні рішення для гарантування довіри до контейнерів. На рисунку 1.4 показано процес оновлення контейнерів, у якому перед розгортанням нового образу виконується перевірка цифрового підпису.

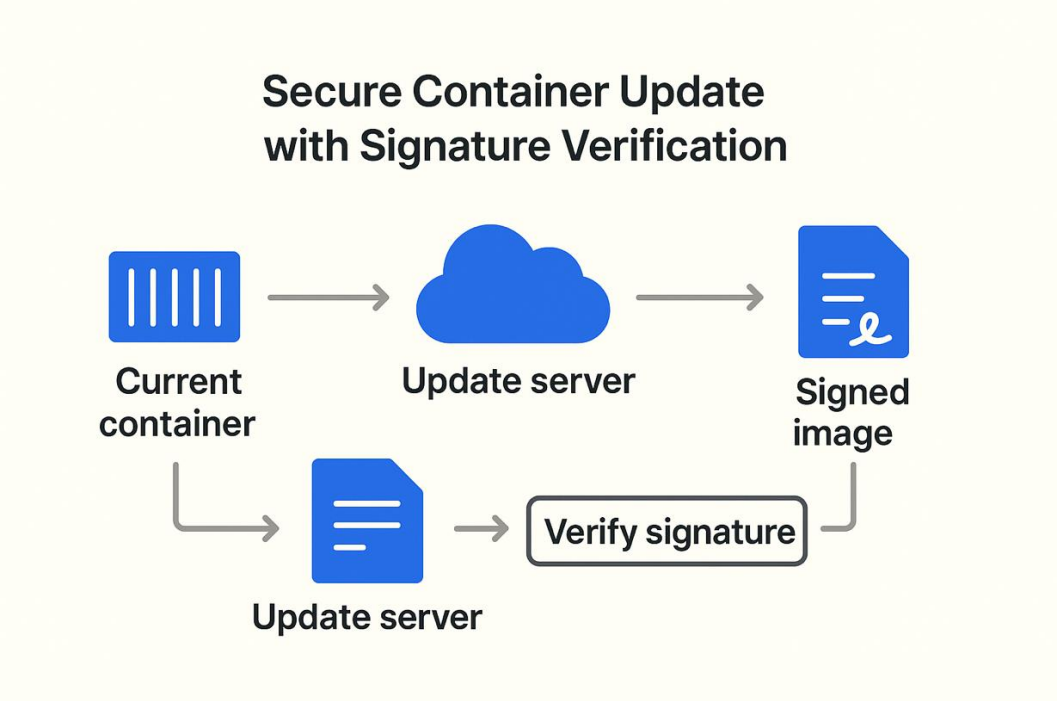

Рисунок 1.4 – Схема безпечного оновлення контейнерів із застосуванням криптографічної перевірки

У разі відсутності або недійсності підпису оновлення блокується, що запобігає впровадженню скомпрометованих контейнерів у систему.

Варто згадати й інші підходи. Компанія Red Hat у своїх контейнерних рішеннях (Podman, Red Hat Quay) використовує механізм підпису образів на основі GPG: підписи зберігаються у спеціальних сховищах (наприклад, файли в репозиторії або всередині OCI-реєстру), а верифікація здійснюється клієнтськими утилітами при завантаженні. Хоча реалізація може відрізнятися, мета залишається спільною – впевнитися, що жоден контейнерний образ не потрапить у систему без криптографічного підтвердження його цілісності та джерела [6].

Отже, на ринку зараз існує декілька рішень для безпечного оновлення контейнерів із застосуванням криптографії. Docker Content Trust проклав шлях, інтегрувавши надійне підписування в екосистему Docker. Його наступники – проекти Notary v2/Notation та Sigstore/Cosign – розвивають цю ідею далі, роблячи підписання образів більш простим, швидким і масштабованим. Водночас інструменти для оркестрації і реєстри (Kubernetes, Harbor, та ін.) отримують можливості політик, що гарантують запуск тільки перевірених контейнерів. Завдяки цьому програмні оновлення в контейнерних середовищах можуть відбуватися швидко і безпечно, значно знижуючи ризик атак на ланцюжок постачання.

1.4 Висновки до розділу 1

У першому розділі виконано огляд предметної області безпечного оновлення контейнерів із застосуванням криптографії: розглянуто сучасні системи контейнеризації (Docker, Kubernetes) та пов’язані з ними загрози, зокрема вразливості контейнерних образів і атаки на ланцюжок постачання, а також обґрунтовано необхідність механізмів довіри до оновлень. Проаналізовано криптографічні основи захищених оновлень – симетричне й асиметричне шифрування, геш-функції та цифрові підписи, що забезпечують цілісність і автентичність образів; досліджено існуючі рішення (Docker Content

Trust, Sigstore/Cosign, Notation) і показано їх роль у формуванні ланцюга довіри від розробника до середовища розгортання, що підтверджує необхідність криптографічного захисту як бази для подальшої розробки власного механізму безпечного оновлення контейнерів.

РОЗДІЛ 2 АРХІТЕКТУРА ЗАХИЩЕНОГО CI/CD ТА ЛАНЦЮЖОК ДОВІРИ

2.1 Схема захищеного процесу CI/CD

У сучасних умовах DevOps вкрай важливо забезпечити цілісність та довіру програмних оновлень на всіх етапах конвеєра доставки. Нещодавні гучні інциденти (наприклад, атаки на ланцюги поставок SolarWinds, Codecov тощо) продемонстрували, що зловмисники можуть впроваджувати шкідливий код на етапах збірки або залежностей. Для протидії таким загрозам необхідно криптографічно пов’язати кожен артефакт із його джерелом походження [11]. Іншими словами, потрібен ланцюжок довіри (Chain of Trust), який простежується від моменту створення коду розробником до розгортання оновлення на кінцевому пристрої.

Захищений процес CI/CD у даному проєкті побудований таким чином, що кожен рівень підтверджує довіру до отриманого артефакту перед тим, як передати його на наступний етап. Розробник ініціює випуск програмного оновлення через Git-тег, автоматична збірка на GitHub Actions підписує отриманий контейнер та формує атестацію. Далі приватний реєстр контейнерів зберігає образ та підписи. Кінцевий пристрій (Raspberry Pi) перед розгортанням перевіряє цифрові підписи та атестації. У результаті ні на одному етапі оновлення не приймається “на слово” – довіра гарантується лише засобами криптографії (цифровими підписами, сертифікатами, гешами тощо).

На рисунку 2.1 показано послідовність етапів, починаючи від створення розробником підписаного Git-тегу до безпосереднього розгортання оновлення на сервері.

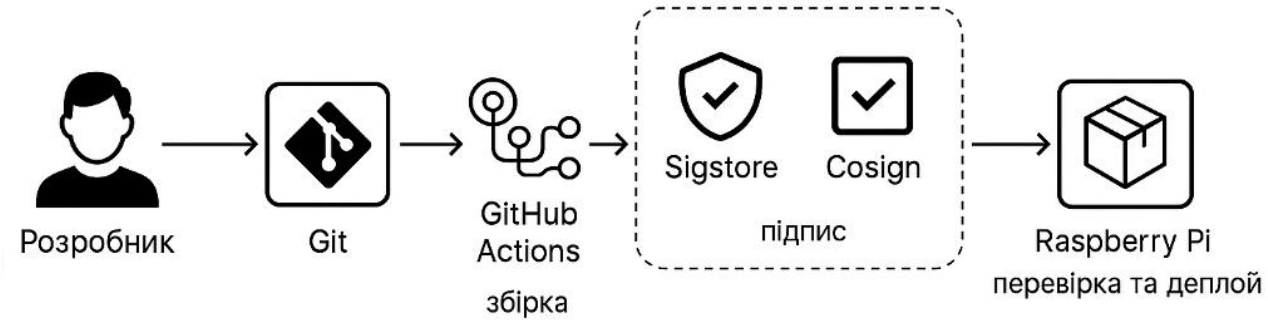

Рисунок 2.1 – Загальна архітектура захищеного CI/CD з ланцюжком довіри

Кожна стрілка відображає передачу артефактів між компонентами із перевіркою їх цілісності та автентичності. Таким чином, вибудовується безперервний ланцюжок довіри: лише ті збірки, що пройшли перевірку на кожному попередньому кроці, можуть бути автоматично прийняті на наступному етапі та врешті встановлені в систему.

У подальших підрозділах детально розглянуто кожен рівень цієї архітектури, зокрема роль розробника і Git-репозиторію, конвеєр збірки та підпису на основі GitHub Actions, приватний реєстр контейнерів, механізм безпечного оновлення на пристрої Raspberry Pi, а також те, як ці компоненти об’єднуються в єдиний ланцюжок довіри.

2.2 Рівень розробки

На рівні розробки забезпечується початковий контроль над випуском нового програмного забезпечення. Розробник працює на локальному середовищі із вихідним кодом та використовує систему керування версіями Git. В цьому проєкті створено окремий репозиторій (SysBitDev/secure_ci-cd на GitHub), де зберігається код додатка. Для відправки змін у репозиторій розробник виконує стандартні кроки – редагує код, комітить зміни та пушить їх на віддалений сервер. Однак оновлення системи на пристрої не відбувається автоматично з кожним комітом. Замість цього запроваджено спеціальний механізм контрольованого релізу за допомогою Git-тегів.

Коли накопичено достатньо змін і потрібно випустити нову версію, розробник створює нову версію Git-тег (наприклад, v1.0.1), що відповідає релізу програмного забезпечення. Такий тег містить посилання на конкретний коміт і слугує “міткою” офіційної версії. Важливо, що тег створюється у підписаному вигляді (підтримується GPG-підпис або аналогічні засоби Git для підпису тегів). Підписання тегу надає можливість криптографічно засвідчити, що саме довірений розробник або уповноважена особа позначила даний коміт як офіційний реліз, і що цей тег не був підроблений сторонньою особою. Багато відкритих проєктів використовують підписані теги для маркування релізів з підтвердженою автентичністю [12].

Створений тег пушиться до віддаленого Git-репозиторію на GitHub. Ця дія запускає процес Continuous Integration/Continuous Deployment – в налаштуваннях проєкту передбачено, що саме подія появи нового тегу (а не звичайного коміту) є тригером для конвеєра CI/CD. Роль Git-тегу у безпеці системи полягає в тому, що він виступає явним дозволом на деплой. Без наявності належним чином оформленого тегу автоматичне оновлення сервера не відбудеться. Це унеможливлює несанкціоновані або випадкові розгортання незатвердженого коду – поки відповідальна особа не позначить версію тегом, система оновлюватися не буде. Таким чином, перший рівень ланцюжка довіри запроваджує контроль з боку розробника та підтверджує аутентичність джерела змін.

2.3 Рівень CI/CD

Після того, як у репозиторії з’явився новий підписаний тег, у дію вступає рівень CI/CD – автоматизована збірка та підготовка релізу із гарантуванням цілісності. Для цього використовується сервіс GitHub Actions, який виконує скрипти збірки у відповідь на створення тегу. Архітектурно цей рівень включає хмарний runner GitHub Actions (віртуальне середовище, що виконує завдання), а також сервіси Sigstore (зокрема, відкритий постачальник OIDC-токенів від GitHub та інфраструктуру підпису Cosign).

Процес на рівні CI/CD проходить декілька етапів. По-перше, GitHub Actions завантажує вихідний код відповідної версії та виконує збірку Docker-образу застосунку. У нашому випадку Docker-образ створюється під цільову платформу (Raspberry Pi, архітектура ARM) з урахуванням усіх залежностей. Далі зібраний образ проходить тестування (за наявності автотестів) і готується до публікації.

Наступний крок – публікація контейнера в реєстр. Налаштовано використання приватного GitHub Container Registry (GHCR) в межах того ж облікового запису або організації. Pipeline автентифікується до GHCR та завантажує Docker-образ (слушно тегований, наприклад, secure_app:latest) у приватний репозиторій пакунків. Образ зберігається під криптографічним гешем (SHA-256 дайджест), що однозначно його ідентифікує.

Після успішного завантаження контейнера виконується найважливіша частина – криптографічне підписування артефактів. Для цього конвеєр GitHub Actions використовує інструмент Sigstore Cosign у режимі keyless signing, тобто без постійного зберігання власних криптографічних ключів. Замість традиційного приватного ключа, Cosign отримує короткоживучий сертифікат на основі OIDC-токена від постачальника ідентичності. У даному випадку GitHub виступає як OIDC-провайдер. Воркфлов GitHub Actions має налаштовані права id-token:write, що дозволяє йому отримати одноразовий OIDC-токен, підтверджений платформою GitHub [13]. Далі цей токен передається сервісу Sigstore Fulcio (відкритому сертифікаційному центру), який перевіряє особу запиту (зокрема, що токен виданий GitHub для конкретного репозиторія та workflow) і видає короткочасний X.509 сертифікат. Сертифікат містить публічний ключ, згенерований Cosign-ом, прив’язаний до ідентичності GitHub Actions (наприклад, включає інформацію про репозиторій та дію, що підписує образ). Після цього Cosign підписує Docker-образ отриманим приватним ключем і негайно знищує приватний ключ із пам’яті. Таким чином, досягається підпис артефакту без зберігання довготривалих ключів – замість них використовується сертифікат, дійсний лічені хвилини [14]. Підпис контейнера (цифровий відбиток, зашифрований приватним ключем) відправляється до прозорого журналу Sigstore Rekor, де зберігається як відкритий запис для аудиту. Одночасно Cosign

автоматично завантажує підпис і до нашого контейнерного реєстру GHCR, асоціюючи його з образом. Як показано на рисунку 2.2, підпис контейнерного образу здійснюється без використання постійних приватних ключів.

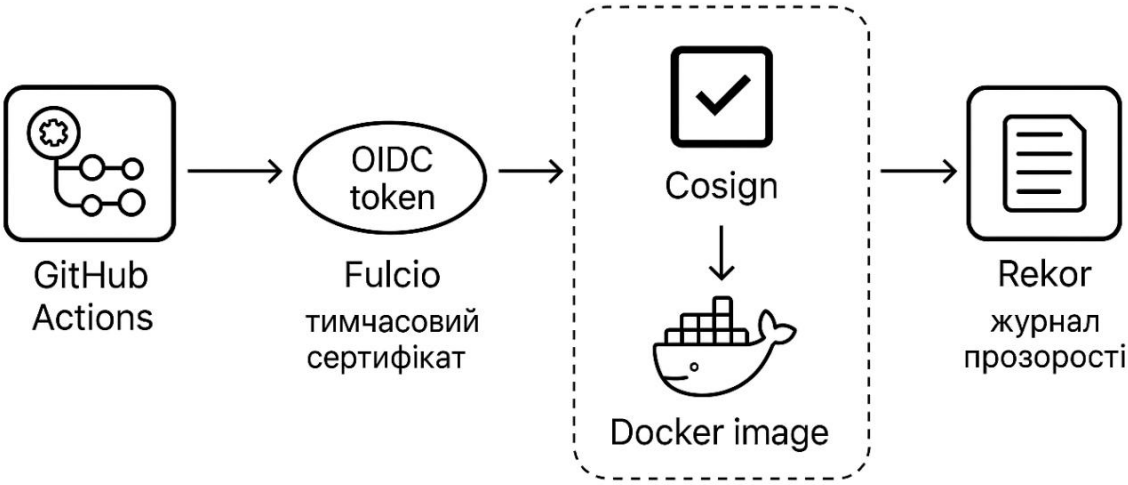

Рисунок 2.2 – Схема цифрового підпису контейнерного образу за допомогою Cosign

Замість цього застосовується механізм keyless signing, при якому GitHub Actions отримує OIDC-токен та обмінює його на короткостроковий сертифікат Sigstore. Це дозволяє підтвердити ідентичність середовища збірки та унеможливлює компрометацію секретних ключів.

Наступний крок – формування даних про походження збірки (provenance). Використовується стандарт SLSA Provenance у форматі in-toto для генерації атестації про те, як саме було побудовано образ. Зокрема, в атестації фіксуються хеші вихідних матеріалів (вихідного коду), ідентифікатор збірки (ідентичність GitHub Actions, що виконала збірку), час та інші метадані. Це дозволяє простежити програмний продукт до його джерела і зафіксувати, за яких умов він був створений [15]. Cosign (разом з інструментом in-toto) підписує проавенанс аналогічно – через Fulcio видається сертифікат і атестація підписується, після чого записується в журнал транспарентності. У приватному реєстрі поруч з образом зберігається і файл атестації (як окремий OCI-артефакт).

На рисунку 2.3 показано процес формування provenance-атестації відповідно до стандартів in-toto та SLSA.

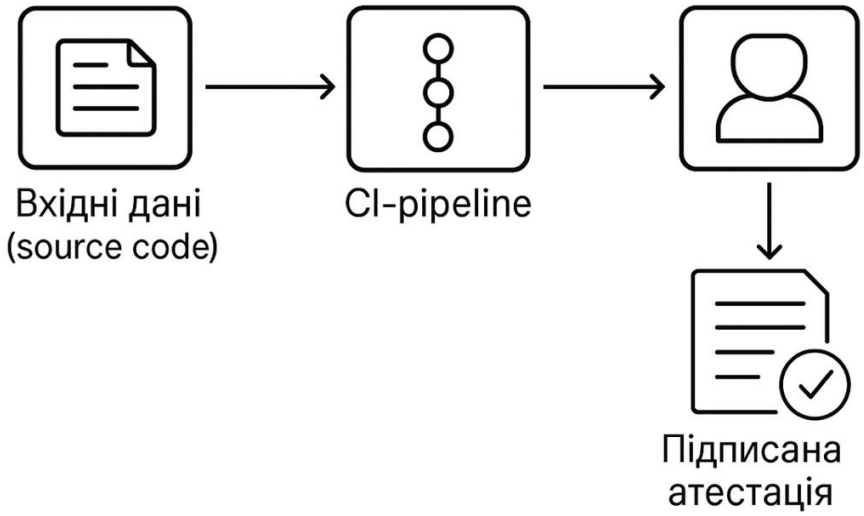

Рисунок 2.3 – Формування та підпис provenance-атестації контейнерного образу

Атестація фіксує походження контейнерного образу, використані вхідні артефакти та ідентичність середовища збірки, що забезпечує трасованість та можливість перевірки програмного продукту.

Після успішного підпису контейнера та атестації конвеєр формує спеціальний файл маніфесту релізу (release-manifest.json). У цьому маніфесті перераховані основні компоненти випуску: посилання на Docker-образ (вказується його повний image reference з включеним хешем), хеш самого образу, посилання або вбудовані дані підписів та сертифікатів, версія випуску, а також додаткова інформація (наприклад, дата, попередня версія тощо). Маніфест потрібен для того, щоб на стороні пристрою-отримувача оновлення перевірити цілісність та взаємну відповідність всіх отриманих файлів. Файл release-manifest.json теж підписується конвеєром – вже з використанням того самого механізму Sigstore (OIDC + Fulcio). Підпис маніфесту вбудовується або зберігається окремо (як файл release.sig), а сертифікат підписувача додається як release.crt.

На завершення роботи CI/CD, GitHub Actions автоматично створює GitHub Release відповідної версії. У release-додатку на GitHub публікується опис релізу,

а як вкладені файли додаються release-manifest.json, release.sig та release.crt. Таким чином, endpoint GitHub Release слугує зручним та захищеним каналом доставки метаданих оновлення для кінцевого пристрою. З нього можна завантажити маніфест і підписи.

На цьому рівні реалізовано декілька ключових принципів безпеки. По-перше, конвеєр ізольований і має мінімально необхідні привілеї. Доступ до реєстру здійснюється лише з правами write:packages для завантаження образу, а секрети (паролі, ключі) взагалі відсутні – замість них використовується OIDC-токен для отримання сертифіката. Жодні приватні ключі не зберігаються постійно. Підпис контейнера, так і підпис маніфесту виконуються за схемою із короткостроковим сертифікатом, що унеможливлює компрометацію довготривалих секретів. По-друге, кожен підпис прив’язаний до ідентичності CI-процесу – сертифікат містить інформацію про GitHub Actions workflow, репозиторій та автора, тому на стороні отримувача можна перевірити, що образ підписано саме авторизованим CI з нашого репозиторію (а не сторонньою особою). По-третє, використання відкритих сервісів Sigstore означає, що всі підписи й атестації занесені до загальнодоступного прозорого логу (Rekor), який забезпечує додатковий рівень контролю та унеможливлює непомітну підміну артефактів. Отже, рівень CI/CD не лише автоматизує випуск оновлення, але й створює криптографічні докази автентичності та цілісності отриманого Docker-образу.

2.4 Рівень реєстру

Після збірки та підпису Docker-образу, артефакти зберігаються на рівні реєстру контейнерів у приватному GitHub Container Registry (GHCR). GHCR виконує роль сховища як самих образів, так і пов’язаних з ними підписів та інших OCI-артефактів (атестацій). Регістр налаштовано як приватний, тобто доступ до нього обмежений обліковим записом та сервісами, що мають відповідний токен доступу. В контексті нашої системи, публікація образу та підписів здійснюється CI-процесом з правами write:packages, а завантаження –

лише оновлюваним пристроєм з правами read:packages. Жоден інший суб'єкт не має дозволу на зміну або читання цього реєстру, що знижує ризики витоку або несанкціонованої модифікації.

У GitHub Container Registry кожен завантажений Docker-образ зберігається із урахуванням контент-адресації. Йому призначається унікальний SHA-256 дайджест, який обчислюється з вмісту образу. Цей дайджест використовується як основа для ідентифікатора підпису. Зокрема, Cosign, підписуючи образ, автоматично додає до реєстру спеціальний тег виду sha256-<digest>.sig. Такий тег є окремим об'єктом в OCI-реєстрі, котрий містить сам цифровий підпис і сертифікат. Фактично, підпис зберігається "поруч" із відповідним образом, і зв'язок між ними встановлюється через спільний дайджест.

На рисунку 2.4 видно, що для Docker-образу присутні два теги: latest та тег починаючи з sha256-... і закінчуючи суфіксом .sig.

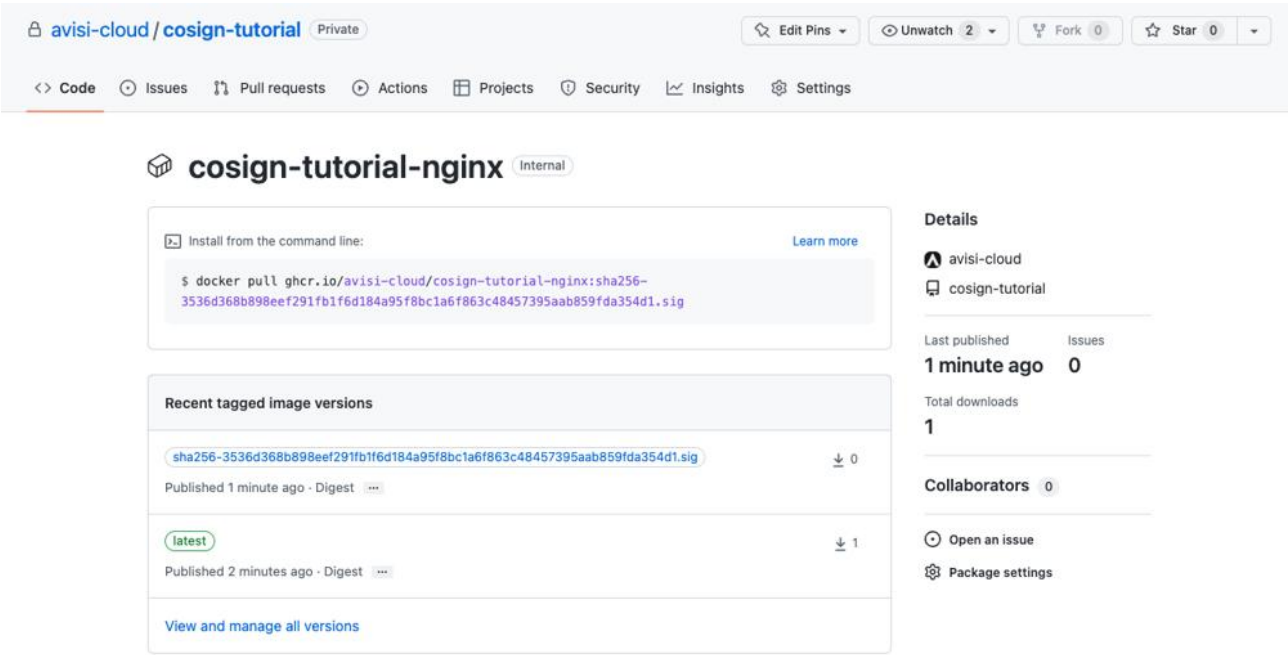

Рисунок 2.4 – Приватний репозиторій GHCR зі збереженим підписом контейнера

Останній є автоматично доданим Cosign-ом об'єктом підпису, прив'язаним до криптографічного хешу образу. Завдяки цьому, під час перевірки підпису інструмент Cosign або інша система може знайти відповідний підпис саме за дайджестом образу, гарантуючи, що перевіряється правильна комбінація "образ–

підпис". Така схема робить підпис прозорим для користувача. Він зберігається як стандартний OCI-артефакт, і Cosign може отримати його з реєстру під час верифікації без додаткових налаштувань [11].

Окрім підпису .sig, до репозиторію також завантажуються інші пов’язані об’єкти. Зокрема, атестація provenance (файл із інформацією про збірку) теж зберігається в OCI-реєстрі як артефакт (зазвичай з окремим тегом чи у спеціальному сховищі артефактів Sigstore). У нашому випадку, Cosign публікує атестацію in-toto до журналу Rekor, а за потреби може дублювати її в той же реєстр.

Важливою характеристикою контейнерного реєстру є те, що він забезпечує незмінність (immutability) зображень. Після того як образ та пов’язані з ним підписи завантажено, змінити їх або підмінити практично неможливо без втрати валідності підпису. Якщо зловмисник спробував би замінити образ у реєстрі на інший (наприклад, із шкідливим вмістом), то його криптографічний дайджест зміниться, і наявний підпис .sig більше не відповідатиме новому образу. Будь-яка спроба підміни артефактів буде виявлена на етапі перевірки підписів, адже цифровий підпис порушиться – Cosign вкаже, що підпис не відповідає даному образу або що сертифікат підпису невалідний для цього вмісту. Таким чином, GHCR виступає надійним сховищем, де кожен образ захищений від несанкціонованої модифікації за рахунок криптографічних зв’язків.

Додатковий рівень захисту забезпечується приватністю реєстру. Оскільки він недоступний публічно, зловмисник не може навіть отримати копію образу чи підпису, щоб спробувати підібрати колізію гешу або здійснити інші атаки. Доступ контролюється GitHub-токенами з вузькими правами. Разом всі ці заходи гарантують, що випущений контейнер і його метадані (підписи, атестації) збережені цілісно й доступні лише довіреним сторонам.

2.5 Рівень виконання

Рівень виконання відповідає за безпосереднє прийняття оновлення та його розгортання на цільовому пристрої – у нашому випадку це одно платний

комп’ютер Raspberry Pi, що виконує роль віддаленого вузла (Secure Node). На Raspberry Pi розгорнуто оточення для запуску контейнера (Docker Engine та Docker Compose), а також спеціальний агент оновлення, реалізований на Python (надалі Secure Updater). Цей агент періодично перевіряє наявність нових релізів програмного забезпечення в GitHub (наприклад, через запити до GitHub Releases API або шляхом завантаження останньої інформації про випуски з репозиторію). Коли з’являється новий випуск, Secure Updater ініціює процес завантаження та верифікації оновлення.

Послідовність дій на цьому рівні наступна. Перш за все, агент отримує з GitHub останній реліз (за тегом) та завантажує опубліковані в ньому файли: маніфест релізу (release-manifest.json), підпис маніфесту (release.sig) і сертифікат підписувача (release.crt). Після успішного завантаження цих трьох компонентів починається етап криптографічної перевірки ланцюжка довіри.

Secure Updater, використовуючи утиліту Cosign або вбудовані криптографічні бібліотеки, спочатку перевіряє підпис маніфесту на відповідність вкладеному сертифікату release.crt. Сертифікат випущено центром Fulcio і прив’язано до ідентичності, тому другим кроком є перевірка властивостей цього сертифіката. Агент впевнюється, що сертифікат видано саме службою Sigstore (перевіряється ланцюжок підписів до кореневого сертифіката Fulcio) і що його OIDC-claims відповідають очікуваним значенням (наприклад, поле суб’єкта містить URL нашого репозиторію GitHub та назву workflow, що мав право підпису). Таким чином підтверджується, що маніфест підписав наш довірений CI/CD-процес на GitHub, а не хтось інший [13]. Лише за умови валідності цифрового підпису маніфесту та достовірності сертифіката агент переходить до наступних перевірок.

Далі Secure Updater аналізує вміст самого release-manifest.json. У маніфесті знаходиться посилання на Docker-образ (його URL в реєстрі GHCR, включно з хешем, наприклад ghcr.io/username/repo@sha256:abcdef...) та очікувані значення хешу образу, а також хеші для пов’язаних файлів атестації. Агент завантажує з приватного реєстру вказаний Docker-образ (для чого Raspberry Pi має токен з правами читання). Після завантаження виконується верифікація підпису

контейнера. За допомогою команди cosign verify (в режимі keyless) перевіряється, чи підписаний образ відповідає політикам Sigstore. Cosign автоматично отримує з реєстру підпис (OCI-об’єкт .sig), завантажує відповідний сертифікат підпису з журналу Rekor та перевіряє, що:

- підпис криптографічно валідний для даного образу (відповідає хешу образу);
- сертифікат підпису видано Fulcio і його OIDC-предмет співпадає з очікуваною ідентичністю (той самий GitHub Actions workflow);
- підпис присутній у прозорому журналі (Rekor) і не відкликаний.

Тільки якщо всі ці умови дотримано, контейнер вважається автентичним і не зміненим при транспортуванні.

Одночасно агент перевіряє атестацію про збірку (provenance). З журналу або реєстру отримується in-toto атестація, яка містить дані про процес збірки. Secure Updater звіряє хеші матеріалів у цій атестації з очікуваними (наприклад, хеш вихідного коду, закладений у атестацію, повинен відповідати хешу Git-коміту, на який вказує релізний тег). Також перевіряється, що атестація підписана тим самим сертифікатом, що і контейнер (або іншим довіреним сертифікатом Sigstore), і що вона не була модифікована. Аналізуючи provenance, система може впевнитися, що образ справді зібрано із задекларованого вихідного коду та із застосуванням відповідного процесу CI (довіреного builder-a) [15]. Це додає впевненості, що в процесі збірки не було додано стороннього коду.

Наступна перевірка – Anti-rollback контроль версії. Маніфест містить номер версії поточного випуску (наприклад, 1.0.1) та попередньої (наприклад, 1.0.0). Raspberry Pi зберігає у себе інформацію про останню успішно встановлену версію. Secure Updater звіряє, що нова версія дійсно новіша за встановлену. Якщо з якоїсь причини відбувається спроба встановити старішу версію (наприклад, через компрометацію каналу оновлення або помилку конфігурації), агент визначає це та блокує такий даунгрейд. Механізм anti-rollback запобігає тому, щоб пристрій було повернено на вразливу попередню версію прошивки/ПЗ, чим міг би скористатися зловмисник [16]. Отже, навіть маючи валідні підписи, але

версію нижчу за поточну, оновлення буде відхилено як потенційно небезпечне або помилкове.

Лише за умови успішного проходження всіх вищезазначених перевірок Secure Updater переходить до безпосереднього розгортання оновлення. Спочатку він оновлює файл конфігурації Docker Compose, встановлюючи новий image reference (з потрібним хешем) для сервісу додатка. Далі виконується команда docker compose up -d, що завантажує новий образ (якщо він ще не був завантажений) та перезапускає контейнер додатка із цією оновленою версією. Оскільки Docker-образ підтверджений як автентичний і безпечний, система переходить у стан роботи з новим програмним забезпеченням.

Варто підкреслити, що якщо хоч один із кроків перевірки не пройдено, процес оновлення негайно блокується. Secure Updater не виконує жодних змін у системі, доки не зможе гарантувати повну довіру до нового коду. У разі збою перевірки (невірний підпис, невідповідність хешів, недійсний сертифікат, версія старіша за поточну тощо) агент повідомить про помилку (наприклад, збереже лог або надішле оповіщення) і залишить працювати попередню версію без змін. Такий підхід відповідає принципам безпечного оновлення прошивок/ПЗ для IoT: система повинна відмовитися від оновлення, якщо його цілісність чи легітимність не можуть бути доведені поза сумнівами.

Схема на рисунку 2.5 ілюструє послідовність безпечного оновлення контейнерів на кінцевому вузлі. У разі невдалого проходження хоча б одного етапу перевірки оновлення автоматично блокується, що унеможливлює розгортання скомпрометованих або недовірених контейнерів.

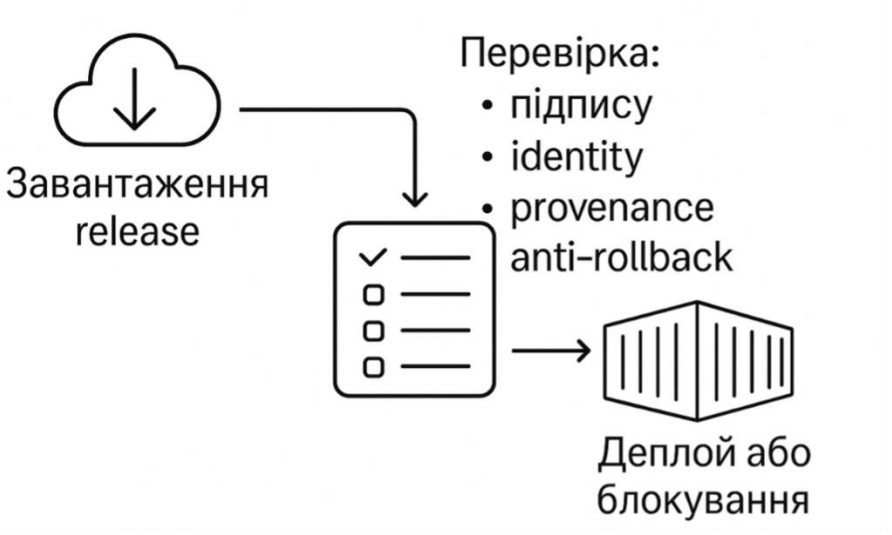

Рисунок 2.5 – Схема безпечного оновлення контейнерів із перевіркою підпису

Таким чином, Raspberry Pi виконує роль останньої ланки в ланцюжку довіри, де відбувається остаточна верифікація та застосування оновлення. Завдяки комплексній перевірці (підпис маніфесту, сертифікат CI, підпис контейнера, атестація збірки, версія) до пристрою ніколи не буде застосовано неперевірений або потенційно шкідливий код. Це особливо важливо, коли пристрій знаходиться в потенційно небезпечному середовищі (наприклад, в мережі Інтернет) і оновлюється віддалено без участі людини.

2.6 Ланцюжок довіри

Побудована система формує наскрізний ланцюжок довіри від моменту випуску коду до його запуску в продуктивному середовищі. Кожен попередній рівень передає наступному не сирі дані, а криптографічно засвідчені артефакти. Нижче підсумовано цей потік довіри. Git-тег розробника ініціює процес і слугує першою ланкою довіри. Підписаний тег гарантує, що випуск версії санкціонований автором проєкту і відповідає конкретному набору змін у коді [12]. Він дає сигнал системі, що цю версію можна збирати і поширювати. CI-процес GitHub Actions отримавши тригер у вигляді тегу, довірений середовище збірки компілює код. Завдяки OIDC-аутентифікації до Sigstore, конвеєр отримує сертифікат, який підтверджує його особу, і підписує збірку (контейнер) та маніфест релізу. Таким чином, CI виступає підписантом, а його підпис

визнається довіреним, бо сертифікат видано авторитетним центром на основі підтвердженої ідентичності (нашого репозиторію/workflow) [13]. Підписаний контейнер та атестація Docker-образу постачається в репозиторій вже із вкладеним криптографічним підписом і атестацією, що описує його походження. Підпис контейнера гарантує, що образ не було змінено після збірки, а атестація надає прозорість щодо процесу його отримання [11]. Підписаний маніфест релізу пов’язує воєдино всі частини випуску (образ, хеші, попередня версія тощо) і також підписаний CI-процесом. Підпис маніфесту гарантує, що жоден з перелічених у ньому компонентів не був змінений і що сам маніфест створено довіреною стороною. Перевірка на Raspberry Pi це – фінальна ланка, де сходяться всі дані довіри. Пристрій перевіряє цифрові підписи маніфесту та образу, сертифікати (що верифікують особу підписанта як GitHub Actions), зіставляє хеші та підтверджує відсутність даунгрейду версії. Лише коли всі зв’язки ланцюжка збігаються, оновлення завантажується в систему.

Представлений на рисунку 2.6 ланцюжок довіри демонструє, що кожен наступний етап базується виключно на криптографічно перевірених артефактах попереднього рівня. Таким чином, довіра в системі не передається декларативно, а формується на основі цифрових доказів, які можна перевірити.

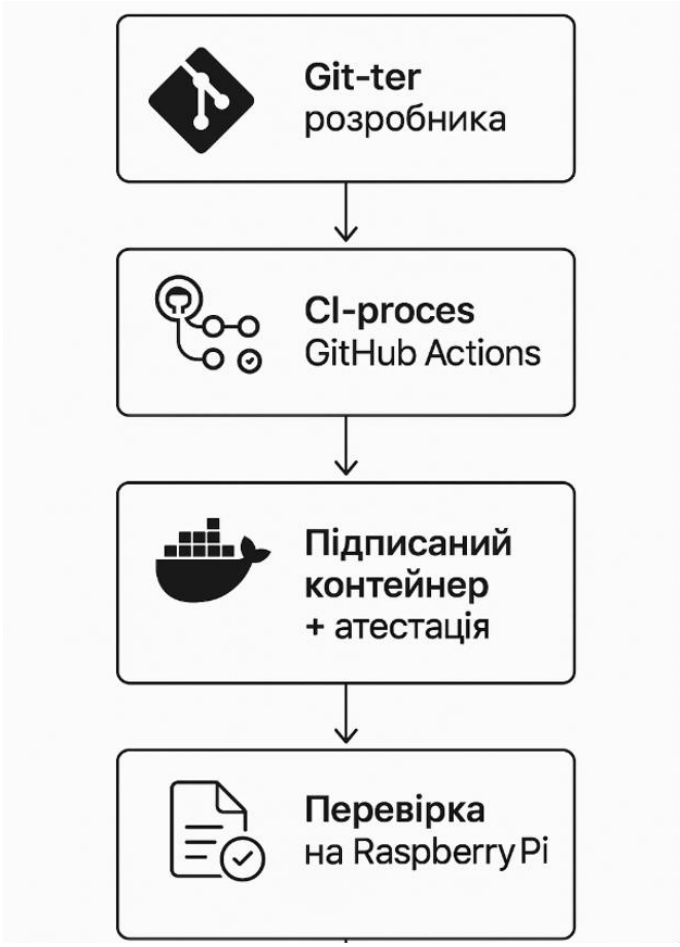

Рисунок 2.6 – Ланцюжок довіри в системі безпечного оновлення контейнерів

Отже, ланцюжок довіри ніколи не “передається на слово” на жодному з етапів. Кожна наступна дія ґрунтується виключно на криптографічно перевірених доказах від попереднього етапу. У підсумку, якщо хоча б одна ланка виявиться недостовірною, весь процес зупиниться, що забезпечує високий рівень безпеки і стійкості до атак на ланцюг поставок програмного забезпечення.

2.7 Висновки до розділу 2

У другому розділі було спроєктовано та детально розглянуто архітектуру захищеного процесу CI/CD з наскрізним ланцюжком довіри для автоматичного оновлення програмного забезпечення. Запропонована багаторівнева система включає рівень розробки (контроль релізів через підписані Git-теги), рівень збірки CI/CD (автоматизована збірка контейнера із підписуванням за допомогою Sigstore Cosign без зберігання постійних ключів), рівень реєстру (приватне

сховище GHCR, де образи й підписи захищені від підміни), а також рівень виконання на кінцевому пристрої (Raspberry Pi з агентом Secure Updater, який верифікує оновлення перед деплоєм). В розділі показано, як ці компоненти взаємодіють, утворюючи єдиний ланцюжок довіри. Кожен компонент перевіряє цілісність та автентичність даних, отриманих від попереднього.

Такий підхід значно підвищує безпеку процесу розгортання оновлень. Завдяки використанню сучасних механізмів криптографічного підпису та атестації (Fulcio OIDC сертифікатів, прозорого логування Rekor, in-toto атестацій) досягається захист від цілого класу атак, пов’язаних із компрометацією CI/CD або реєстрів контейнерів. Жоден неперевірений код не може потрапити на сервер, оскільки на кожному кроці є перевірка підписів і відповідності очікуванням. Також було враховано захист від атак типу rollback – пристрій відмовиться встановлювати старіші, потенційно вразливі версії прошивки.

РОЗДІЛ 3 РЕАЛІЗАЦІЯ ЗАХИЩЕНОГО CI/CD-КОНВЕЄРА ДЛЯ КОНТЕЙНЕРНИХ СИСТЕМ

3.1 Архітектура системи та її компоненти

Для забезпечення безпечного оновлення контейнерів з криптографічною верифікацією розгорнуто комплексну систему, що складається з кількох вузлів із чітко визначеними ролями. Архітектура включає чотири основні вузли. Вузол A (ПК розробника) – локальне середовище розробника. Тут відбувається розробка коду застосунку, коміт змін до репозиторію Git, а також створення тегів (релізів) у форматі `vX.Y.Z`. Тегування коду ініціює процес CI/CD. Вузол B (GitHub – репозиторій та CI) – віддалений GitHub-репозиторій, де зберігається вихідний код проєкту, а також платформа GitHub Actions для автоматичного виконання роботи конвеєра CI/CD. При появі нового Git-тегу репозиторій запускає workflow, що збирає Docker-образ, публікує його у реєстрі та підписує за допомогою Sigstore Cosign. Додатково генерується файл доказів походження (provenance attestation) для підтвердження автентичності збірки. Після успішного проходження CI/CD створюється реліз на GitHub із вкладеними артефактами (маніфестом оновлення та цифровими підписами). Вузол C (GitHub Container Registry) – приватний реєстр контейнерів GHCR (GitHub Container Registry), де зберігається зібраний Docker-образ. Образ є недоступним публічно і вимагає аутентифікації для завантаження. Саме в цей реєстр надсилається нова версія контейнера після успішної збірки та підпису. Вузол D (Raspberry Pi – сервер деплою) – віддалений сервер (на базі Raspberry Pi), на якому працює сам контейнерний застосунок. Цей вузол виконує роль середовища продакшн. На ньому встановлено Docker Engine з Docker Compose для керування контейнерами [20,21]. Найважливішим компонентом є скрипт Secure Updater, який періодично перевіряє наявність нових релізів у репозиторії (через GitHub Releases), завантажує маніфест оновлення та підписані артефакти, криптографічно перевіряє їхню автентичність (через Cosign), і лише після успішної верифікації зупиняє старий контейнер та запускає оновлений. Цей механізм також

захищений від відкату версій (rollback) – тобто, від випадків, коли зловмисник може спробувати нав’язати попередню уразливу версію замість актуальної.

Таким чином, архітектура забезпечує кінцевий ланцюжок довіри. Від розробника, який маркує реліз і тригерить CI, до збірки та підпису контейнера на стороні GitHub, і далі до сервера, який оновлює контейнер лише якщо довіряє отриманим артефактам. Нижче наводиться покрокова реалізація цієї системи.

3.2 Налаштування репозиторію та CI/CD-конвеєра

На платформі GitHub було створено новий приватний репозиторій з вихідним кодом проєкту. Структура репозиторію містить мінімально необхідні файли для побудови контейнера та налаштування CI/CD-конвеєра (див. рисунок 3.1).

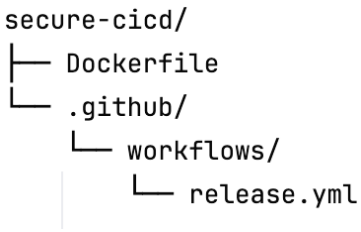

Рисунок 3.1 – Структура файлів у репозиторії проекту на GitHub

На рисунку 3.1 показано вміст кореневого каталогу репозиторію з Dockerfile та конфігурацією GitHub Actions. Файл Dockerfile описує, як побудувати образ Docker, а каталог `.github/workflows` містить YAML-файл з описом CI/CD-процесу, що запускається на кожен релізний тег.

Файл `Dockerfile` (див. рисунок 3.2) визначає образ контейнера із застосунком. У нашому випадку застосунок демонстраційний, тому Dockerfile досить простий: базується на офіційному легковагому образі Alpine Linux, створює некореневого користувача для запуску процесу та запускає нескінченний цикл (в реальному застосунку тут міг би бути старт сервера чи інша логіки).

```
FROM alpine:3.20

RUN adduser -D appuser
USER appuser

CMD ["sh", "-c", "echo secure_ci-cd running; sleep infinity"]
```

Рисунок 3.2 – Dockerfile з визначенням образу контейнерного застосунку

Використання некореневого користувача всередині контейнера підвищує безпеку контейнерного середовища.

Як видно, для базового образу використано `alpine:3.20`. Команда `adduser -D appuser` створює системного користувача appuser без пароля, а директива `USER appuser` переключає виконання контейнера на цього користувача (щоб процес не працював від імені root). Завершальна команда CMD просто виводить повідомлення і засинає, імітуючи постійну роботу сервісу. Така конфігурація достатня для перевірки роботи механізму оновлення.

Файл робочого процесу CI/CD `.github/workflows/release.yml` реалізує повний автоматизований цикл збірки, криптографічного підпису та публікації контейнерного образу. Workflow налаштовано на запуск у момент створення Git-тегу, що відповідає шаблону семантичного створення версій `v*.*.*`, що забезпечує контрольоване формування релізів. На початковому етапі у YAML-конфігурації визначається подія запуску та необхідні дозволи GitHub Actions (див. рисунок 3.3).

```
name: secure-release

on:
  push:
    tags:
      - "v*.*.*"

permissions:
  contents: write
  packages: write
  id-token: write
```

Рисунок 3.3 – Тригер запуску workflow та перелік дозволів

Даний фрагмент визначає, що workflow активується виключно при пуші Git-тегів, які відповідають семантичному створення версій. Це виключає випадкові збірки та гарантує, що кожен реліз є усвідомленим. Дозвіл `packages: write` необхідний для публікації образів у GitHub Container Registry, `contents: write` – для створення GitHub Release, а `id-token: write` – для отримання OIDC-токена, який використовується механізмом keyless-підпису Sigstore.

Далі у workflow задаються глобальні змінні середовища, які використовуються впродовж усього процесу (див. рисунок 3.4).

```yaml
env:
  IMAGE: ghcr.io/xxxxxx/secure_ci-cd
  PROVENANCE_TYPE: https://xxx.xxx/secure-ci-cd/provenance/v1
```

Рисунок 3.4 – Оголошення глобальних змінних середовища

Змінна `IMAGE` містить повну адресу контейнерного образу в реєстрі GHCR. Змінна `PROVENANCE_TYPE` визначає унікальний ідентифікатор типу доказу походження (provenance attestation), який дозволяє відрізняти власні attestations від стандартних або сторонніх форматів. Основний job виконується на віртуальному середовищі `ubuntu-latest`. Початкові кроки відповідають за отримання коду та підготовку інструментів збірки (див. рисунок 3.5).

```yaml
jobs:
  build-sign:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Docker Buildx
        uses: docker/setup-buildx-action@v3
```

Рисунок 3.5 – Ініціалізація job та налаштування Docker Buildx

На цьому етапі виконується клонування репозиторію та ініціалізація Docker Buildx – розширеного механізму збірки, який підтримує багатоплатформені образи та дозволяє одразу публікувати результати в реєстр.

Наступний фрагмент коду відповідає за автентифікацію у приватному реєстрі GHCR (див. рисунок 3.6).

```yaml
- name: Login to GHCR
  uses: docker/login-action@v3
  with:
    registry: ghcr.io
    username: ${{ github.actor }}
    password: ${{ secrets.GITHUB_TOKEN }}
```

Рисунок 3.6 – Авторизація в GitHub Container Registry

Вхід до реєстру здійснюється за допомогою стандартного `GITHUB_TOKEN`, який автоматично генерується GitHub і має необхідні права завдяки налаштованим дозволам workflow. Це усуває потребу в ручному управлінні обліковими даними.

Після підготовки середовища виконується збірка Docker-образу та його публікація з тегом, що відповідає версії релізу (див. рисунок 3.7).

```yaml
- name: Build & Push image
  run: |
    docker buildx build \
      --push \
      --tag $IMAGE:${{ github.ref_name }} \
      .
```

Рисунок 3.7 – Збірка та публікація контейнерного образу

Команда `--push` забезпечує негайне завантаження образу в реєстр після успішної збірки, а використання `${{ github.ref_name }}` гарантує синхронізацію версії контейнера з Git-тегом.

Оскільки подальші операції підпису виконуються не над тегом, а над криптографічним діджестом, workflow отримує SHA-256-ідентифікатор образу (див. рисунок 3.8).

```yaml
- name: Resolve image digest
  run: |
    DIGEST=$(docker buildx imagetools inspect $IMAGE:${{ github.ref_name }} \
      --format '{{json .Manifest.Digest}}' | tr -d '"')
    echo "IMAGE_REF=$IMAGE@$DIGEST" >> $GITHUB_ENV
```

Рисунок 3.8 – Визначення діджеста контейнерного образу

Збереження повної адреси образу у форматі `<image>@<digest>` забезпечує незмінність посилання та захищає від атак, пов’язаних із перевизначенням тегів.

Після цього встановлюється утиліта cosign і виконується keyless-підпис контейнера (див. рисунок 3.9).

```yaml
- name: Install cosign
  uses: sigstore/cosign-installer@v3

- name: Sign container image
  run: |
    cosign sign --yes $IMAGE_REF
```

Рисунок 3.9 – Keyless-підпис контейнерного образу

Cosign використовує OIDC-токен GitHub Actions для формування підпису без зберігання приватних ключів. Інформація про підпис автоматично публікується в Sigstore Transparency Log.

Далі створюється JSON-файл з інформацією про походження збірки та формується attestation (див. рисунок 3.10).

```yaml
- name: Create provenance predicate
  run: |
    cat > prov.json <<EOF
    {
      "repo": "SysBitDev/secure_ci-cd",
      "workflow": "secure-release",
      "ref": "${{ github.ref }}",
      "commit": "${{ github.sha }}",
      "runner": "${{ runner.name }}",
      "created_at": "${{ github.run_started_at }}"
    }
    EOF

- name: Attest provenance
  run: |
    cosign attest --yes \
      --type $PROVENANCE_TYPE \
      --predicate prov.json \
      $IMAGE_REF
```

Рисунок 3.10 – Формування та публікація provenance attestation

Цей етап створює криптографічно захищений доказ походження образу, який підтверджує, що він був зібраний саме цим workflow з конкретного репозиторію.

На завершальному етапі формується маніфест релізу та виконується його окремий підпис (див. рисунок 3.11).

```
- name: Create release manifest
  run: |
    cat > release-manifest.json <<EOF
    {
      "version": "${{ github.ref_name }}",
      "image": "$IMAGE_REF",
      "provenance_type": "$PROVENANCE_TYPE",
      "issuer": "https://token.actions.githubusercontent.com",
      "workflow": "secure-release"
    }
    EOF

- name: Sign release manifest
  run: |
    cosign sign-blob --yes \
      --output-signature release.sig \
      --output-certificate release.crt \
      release-manifest.json
```

Рисунок 3.11 – Створення та підпис маніфесту релізу

Маніфест об’єднує всі критичні метадані, необхідні серверній частині для верифікації оновлення.

Фінальним кроком є створення GitHub Release з прикріпленими криптографічними артефактами (див. рисунок 3.12).

```
- name: Publish GitHub Release
  uses: softprops/action-gh-release@v2
  with:
    files: |
      release-manifest.json
      release.sig
      release.crt
```

Рисунок 3.12 – Публікація релізу з підписаними артефактами

Таким чином, workflow забезпечує повний ланцюг довіри – від збірки контейнера до перевірюваного релізу з криптографічно підтвердженим походженням.

Для успішного виконання описаного workflow необхідно було виконати кілька підготовчих кроків у налаштуваннях GitHub-репозиторію. Дозволи для GitHub Actions. У налаштуваннях репозиторію (Settings -> Actions -> General) ввімкнено розширені права для GitHub Actions. Вибрано опцію Read and write permissions замість дефолтних Read-only. Це потрібно, щоб workflow міг завантажувати пакети в GHCR та створювати релізи. Токен для доступу до GHCR. Контейнерний реєстр GHCR за замовчуванням приватний для образів у приватних репозиторіях. Хоча GitHub Actions має права записати образ, для завантаження образу на сервері (вузол D) потрібен окремий токен. Створено Personal Access Token (classic) з мінімально необхідними правами: repo (для читання релізів) та read:packages (для скачування образів з GHCR). Цей PAT-токен буде зберігатися на сервері і використовуватися скриптом оновлення для доступу до артефактів релізу та образу. Обмеження доступу до пакета. Після першого успішного запуску CI і завантаження образу в GHCR було перевірено налаштування пакету (контейнерного образу) у розділі Packages. За необхідності, видимість пакету встановлюється в Private, щоб гарантувати, що жоден сторонній не отримає доступ до образу або його підписів. У нашому випадку, образ і так приватний, оскільки репозиторій приватний, але цей крок додає впевненості.

3.3 Процес випуску нової версії

Після налаштування репозиторію та workflow, випуск нової версії застосунку виконується розробником у декілька кроків. Спочатку розробник комітить всі необхідні зміни в гілку `main` (або іншу основну гілку) та пушить їх на GitHub. Далі, коли код готовий до випуску, створюється Git-тег з номером версії.

На рисунку 3.12 показано послідовність команд для випуску версії 1.0.0. Після виконання цих дій GitHub автоматично розпочне workflow secure-release на основі тегу `v1.0.0`. В ході workflow зберуться всі артефакти, як описано раніше.

```
git add .
git commit -m "secure pipeline"
git push origin main

git tag v1.0.4
git push origin v1.0.4
```

Рисунок 3.12 – Створення нового релізу з коміт змін та пуш Git-тегу версії v1.0.0 в репозиторій

Результатом успішного виконання CI/CD є поява нового релізу на GitHub. На сторінці Releases репозиторію з'явиться запис про версію v1.0.0, який містить прикріплені файли: `release-manifest.json`, `release.sig` та `release.crt`.

На рисунку 3.13 представлено інтерфейс GitHub із прикладом створеного релізу та переліком артефактів.

Рисунок 3.13 – Сторінка GitHub Release для версії v1.0.4 з прикріпленими артефактами маніфесту та цифрового підпису

Файл маніфесту містить інформацію про образ та версію, а файли підпису і сертифіката підтверджують, що цей маніфест підписано нашим CI-процесом.

3.4 Налаштування середовища розгортання на Raspberry Pi

Після того, як перший реліз сформовано, можна готувати сервер (вузол D) до прийому оновлень. Роль сервера виконує Raspberry Pi з ОС Linux (наприклад, Raspberry Pi OS 64-bit). Основні компоненти, які треба встановити і налаштувати на цьому вузлі: Docker (для запуску контейнера), Docker Compose, утиліта Cosign (для перевірки підписів) та власне скрипт оновлення з політикою безпеки. Усі дії на Raspberry Pi виконуються під користувачем, що має права `sudo`.

Спершу оновлюємо систему і встановлюємо Docker. Для цього використано офіційний інсталяційний скрипт Docker (команда `curl | sh`). Після інсталяції користувача додають до групи `docker` для можливості запускати команди Docker без `sudo`, і активують нові групові права командою `newgrp docker`. Також встановлюється Docker Compose плагін. Послідовність команд показано на рисунку 3.14.

```
sudo apt update && sudo apt upgrade -y
curl -fsSL https://get.docker.com | sh
sudo usermod -aG docker $USER
newgrp docker
sudo apt install docker-compose-plugin -y
```

Рисунок 3.14 – Встановлення Docker Engine та плагіну Docker Compose на Raspberry Pi

Після цього на сервері доступна команда `docker` і `docker compose`. Можна перевірити версії, виконавши `docker --version` та `docker compose version`, щоб впевнитись у коректній інсталяції.

Оскільки наш образ зберігається у приватному GitHub Container Registry, серверу потрібен доступ для його завантаження. Для цього використаємо раніше створений PAT-токен. Щоб не зберігати токен у відкритому вигляді в історії

команд, використовуємо команду `docker login` з передачею пароля через stdin. На рисунку 3.15 показано, як виконати логін. Потрібно вставити значення PAT замість `PASTE_PAT_HERE`. Після успішного логіну Docker демон повідомить Login Succeeded..

```
echo "PASTE_PAT_HERE" | docker login ghcr.io -u <ім'я користувача GitHub> --password-stdin
```

Рисунок 3.15 – Вхід на сервері в приватний GitHub Container Registry за допомогою PAT-токена

Для перевірки підписів на сервері встановлюється утиліта cosign (розробка Sigstore). Проект Cosign публікує готові бінарні файли для різних архітектур. Raspberry Pi 4 має процесор ARM64, тому завантажуємо відповідний реліз. На рисунку 3.16 – команди завантаження останньої версії cosign, надання прав виконання та переміщення виконуваного файлу в системний каталог `/usr/local/bin`. Після цього команда `cosign version` повинна вивести поточну версію та інформацію, що cosign готовий до роботи.

```
curl -Lo cosign https://github.com/sigstore/cosign/releases/latest/download/cosign-linux-arm64
chmod +x cosign
sudo mv cosign /usr/local/bin/
```

Рисунок 3.16 – Інсталяція утиліти Cosign на Raspberry Pi (ARM64)

Для зручності, на сервері створюється окремий каталог, де будуть зберігатися файли, пов'язані з деплоєм та оновленням. Наприклад, використовуємо шлях `/opt/secure-ci-cd/`. Командою `mkdir -p` створено структуру директорій, включно з підкаталогом `updater` для скрипта оновлення та його даних. Далі права власності на цей каталог передано поточному користувачу, щоб не виконувати додаткові дій від root при записі файлів (див. рисунок 3.17).

```
sudo mkdir -p /opt/secure-ci-cd/updater
sudo chown -R $USER:$USER /opt/secure-ci-cd
```

Рисунок 3.17 – Створення каталогу /opt/secure-ci-cd на сервері для файлів проекту та надання прав доступу користувачу

Для управління запуском контейнера використовується Docker Compose, що дозволяє визначити сервіси у YAML-форматі. Створимо файл `/opt/secure-ci-cd/docker-compose.yml` (див. рисунок 3.18), в якому опишемо сервіс `app`. У полі `image` використано змінну `${IMAGE_REF}` – вона міститиме повну адресу образу з діджестом, який потрібно запускати. Змінну буде задано у файлі оточення окремо. Параметр `container_name` задає ім'я контейнера (для зручності підтримки), а `restart: always` вказує Docker автоматично перезапускати контейнер у разі його аварійного завершення або при перезапуску системи.

```
services:
  app:
    image: ${IMAGE_REF}
    container_name: secure_ci_cd_app
    restart: always
```

Рисунок 3.18 – Конфігураційний файл Docker Compose для сервісу застосунку

Щоб Docker Compose міг підставити значення `${IMAGE_REF}`, створимо файл середовища `/opt/secure-ci-cd/.env` (див. рисунок 3.19). У ньому буде дві змінні: `IMAGE_REF` – посилання на поточний образ (за замовчуванням, поки що placeholder `sha256:PLACEHOLDER`, який далі заміниться реальним), та `GITHUB_TOKEN` – наш PAT-токен для доступу до GitHub. Надалі скрипт оновлення самостійно оновлюватиме значення `IMAGE_REF` у цьому файлі та збереже токен. Наявність токена у файлі середовища дозволить скрипту отримувати доступ до GitHub API і приватного реєстру при кожному запуску.

```
IMAGE_REF=ghcr.io/company/secure-cicd@sha256:PLACEHOLDER
GITHUB_TOKEN=PASTE_PAT_HERE
```

Рисунок 3.19 – Файл середовища .env із початковими значеннями для образу та токену доступу

Одним з елементів Zero Trust підходу є перевірка provenance attestation – доказу того, що образ зібрано з очікуваного вихідного коду та CI-процесу. Раніше на етапі CI ми вбудували дані про репозиторій, workflow та хеш коміту у JSON-файл `prov.json` і завантажили його як attestation для образу. Тепер, на стороні сервера, визначимо політику, яка перевірятиме ці дані. Для цього використовується мова політик Open Policy Agent (OPA) Rego. Створимо файл політики `/opt/secure-ci-cd/updater/policy.rego` (див. рисунок 3.20).

```
package signature

default allow = false

allow {
  input.predicateType == "https://xxx.xxx/secure-ci-cd/provenance/v1"
  lower(input.subject[0].name) == "ghcr.io/xxx/secure_ci-cd"
  input.subject[0].digest.sha256 != ""
}
```

Рисунок 3.20 – Політика OPA Rego для перевірки походження контейнерного образу

Політика описана у модулі `package signature`. За замовчуванням правило `allow` має значення `false` (оновлення заборонено). У блоці `allow { ... }` перелічено умови, за яких політика дозволить оновлення контейнера.

Параметр `input.predicateType == "https://xxx.xxx/secure-ci-cd/provenance/v1"` – тип предикату (доказу) повинен відповідати нашому очікуваному значенню. Це гарантує, що ми перевіряємо саме наш файл provenance, а не який-небудь інший. Параметр `startswith(lower(input.subject[0].name), "ghcr.io/company/secure-`

`cicd")` – поле `subject[0].name` у attestation має починатися з нашого шляху образу в реєстрі. Тобто, attestation стосується образу з GHCR нашого проєкту. `lower()` використано для уніфікації регістру символів. Параметр `input.subject[0].digest.sha256 != ""` – поле digest (геш SHA-256 образу) повинно бути непорожнім. Це опосередковано підтверджує, що attestation прив'язано до конкретного діджеста образу.

Якщо всі ці умови істинні, політика поверне `allow = true`, і оновлення буде дозволено. У протилежному випадку скрипт оновлення отримає відмову при спробі криптографічної перевірки attestation.

Головним компонентом механізму безпечного оновлення на стороні сервера є Python-скрипт `update.py`, розташований у каталозі `/opt/secure-ci-cd/updater/update.py`. Даний скрипт виконується періодично за розкладом і відповідає за виявлення нових релізів, криптографічну перевірку їх автентичності та автоматичне оновлення контейнерного застосунку.

На початку скрипта відбувається імпорт стандартних бібліотек Python та ініціалізація ключових констант, які використовуються протягом усього процесу оновлення (див. рисунок 3.20).

```python
import json
import os
import subprocess
import sys
import urllib.request

OWNER = "Company"
REPO = "secure_ci-cd"

BASE_DIR = "/opt/secure_ci-cd"
UPDATER_DIR = f"{BASE_DIR}/updater"
ENV_FILE = f"{BASE_DIR}/.env"
STATE_FILE = f"{UPDATER_DIR}/state.json"
```

Рисунок 3.20 – Початкові імпорти та ідентифікатори репозиторію

Використані бібліотеки забезпечують роботу з JSON-даними, доступ до змінних середовища, виконання зовнішніх команд, завершення роботи скрипта у разі помилок та взаємодію з GitHub API через HTTP-запити.

Файл `state.json` використовується для збереження інформації про останню успішно розгорнуту версію контейнера. Це дозволяє реалізувати захист від відкату версій та гарантує, що сервер не перейде на старішу або потенційно вразливу версію.

Наступним кроком задаються параметри, які визначають допустимий контекст підпису (див. рисунок 3.21).

```
ISSUER = "https://token.actions.githubusercontent.com"
IDENTITY_REGEX = r"https://github.com/xxxxx/secure_ci-cd/.github/workflows/release.yml@refs/tags/v.*"
PROVENANCE_TYPE = "https://xxx.xxx/secure-ci-cd/provenance/v1"
```

Рисунок 3.21 – Параметри OIDC-верифікації та тип provenance

Змінна `ISSUER` містить очікуваного видавця OIDC-токенів – GitHub Actions. Регулярний вираз `IDENTITY_REGEX` визначає допустимий ідентифікатор сертифіката, який однозначно вказує, що підпис було створено workflow `release.yml` у заданому репозиторії та саме на події створення тегу. Значення `PROVENANCE_TYPE` ідентифікує тип attestation, який скрипт очікує знайти та перевірити.

Для спрощення логіки реалізовано низку допоміжних функцій (див. рисунок 3.22).

Функція `run()` використовується для запуску зовнішніх утиліт, таких як `docker` та `cosign`. У разі помилки виконання команда негайно завершує роботу скрипта, що запобігає продовженню процесу оновлення в некоректному стані.

```
def run(cmd: list[str]) -> str:  6 usages (1 dynamic)
    p = subprocess.run(
        cmd,
        stdout=subprocess.PIPE,
        stderr=subprocess.STDOUT,
        text=True
    )
    if p.returncode != 0:
        print(p.stdout)
        raise SystemExit(p.returncode)
    return p.stdout
```

Рисунок 3.22 – Функція виконання системних команд

Робота зі станом версій реалізована через окремі функції (див. рисунок 3.23). Це дозволяє коректно порівнювати версії виду `v1.2.3` і визначати, чи є оновлення новішим.

```
def load_state() -> dict:  1 usage  SysBitDev
    if not os.path.exists(STATE_FILE):
        return {"version": None}
    with open(STATE_FILE, "r", encoding="utf-8") as f:
        return json.load(f)

def save_state(state: dict):  1 usage  SysBitDev
    with open(STATE_FILE, "w", encoding="utf-8") as f:
        json.dump(state, f, indent=2)

def semver_tuple(v: str):  2 usages  SysBitDev
    v = v.lstrip("v")
    a, b, c = v.split(".")
    return (int(a), int(b), int(c))
```

Рисунок 3.23 – Робота зі станом версій та функцію порівняння семантичних версій

Після ініціалізації скрипт виводить службове повідомлення та завантажує змінні середовища з файлу `.env` (див. рисунок 3.24).

```
print("=== secure updater ===")

for line in open(ENV_FILE):
    if "=" in line:
        k, v = line.strip().split("=", 1)
        os.environ[k] = v
```

Рисунок 3.24 – Завантаження змінних середовища

Це дозволяє отримати актуальний `GITHUB_TOKEN`, який використовується для доступу до GitHub API та реєстру контейнерів.

Далі формується HTTPS-запит до GitHub API для отримання інформації про останній реліз (див. рисунок 3.25).

```
api = f"https://api.github.com/repos/{OWNER}/{REPO}/releases/latest"

req = urllib.request.Request(
    api,
    headers={
        "Authorization": f"Bearer {token}",
        "Accept": "application/vnd.github+json"
    }
)
```

Рисунок 3.25 – Отримання інформації про останній реліз з GitHub

Отриманий JSON містить метадані релізу та перелік прикріплених артефактів.

Після ідентифікації потрібних asset-ів скрипт завантажує файли `release-manifest.json`, `release.sig` та `release.crt` (див. рисунок 3.26), використовуючи спеціальний заголовок `Accept: application/octet-stream`.

```
print(run([
    "cosign", "verify-blob",
    "--certificate", crt_path,
    "--signature", sig_path,
    "--certificate-oidc-issuer", ISSUER,
    "--certificate-identity-regexp", IDENTITY_REGEX,
    manifest_path
]))
```

Рисунок 3.26 – Перевірка підпису маніфесту релізу

Ця команда гарантує, що маніфест було підписано саме довіреним workflow у потрібному репозиторії. У разі невідповідності хоча б одного параметра виконання скрипта припиняється.

Після успішної верифікації маніфесту виконується перевірка версій (див. рисунок 3.27).

```
if state["version"] is not None:
    if semver_tuple(version) < semver_tuple(state["version"]):
        print(f"rollback blocked: {version} < {state['version']}")
        sys.exit(0)
```

Рисунок 3.27 – Механізм захисту від rollback-атак

Цей механізм унеможливлює повернення до старішої версії навіть у випадку появи некоректного або скомпрометованого релізу.

Після завантаження образу за діджестом виконується перевірка його підпису та перевірка attestation (див. рисунок 3.28).

```
print("verify image signature...")
print(run([
    "cosign", "verify",
    "--certificate-oidc-issuer", ISSUER,
    "--certificate-identity-regexp", IDENTITY_REGEX,
    image_ref
]))

print("verify provenance attestation...")
print(run([
    "cosign", "verify-attestation",
    "--type", PROVENANCE_TYPE,
    "--policy", f"{UPDATER_DIR}/policy.rego",
    image_ref
]))
```

Рисунок 3.28 – Верифікація підпису контейнера та provenance attestation

Таким чином підтверджується, що образ зібрано у довіреному CI-середовищі та не було змінено після збірки.

У разі успішного проходження всіх перевірок скрипт оновлює файл `.env`, після чого виконує `docker compose up -d`. Docker автоматично перезапускає контейнер з новим образом, а поточна версія зберігається у `state.json`.

Скрипт завершує роботу повідомленням про успішне оновлення, що підтверджує коректність виконання всього ланцюга безпечного оновлення.

Щоб скрипт оновлення запускався регулярно і без втручання користувача, на Raspberry Pi налаштовано пару unit-файлів systemd: служба та таймер. Служба (`secure-update.service`) визначає, як запускати скрипт, а таймер (`secure-update.timer`) – як часто.

У файлі сервісу (див. рисунок 3.29) в секції `[Unit]` зазначено, що він має виконуватись після запуску мережі та Docker (`After=network-online.target docker.service`), щоб гарантовано мати доступ до інтернету і працюючий Docker. Секція `[Service]` визначає тип виконання `Type=oneshot` (одноразовий запуск) і команду `ExecStart`, яка виконує наш скрипт за допомогою Python 3.

```
[Unit]
Description=Secure container updater
After=network-online.target docker.service
Wants=network-online.target

[Service]
Type=oneshot
ExecStart=/usr/bin/python3 /opt/secure-ci-cd/updater/update.py
```

Рисунок 3.29 – Юніт-файл systemd secure-update.service для одноразового виконання скрипта оновлення

Таймер (див. рисунок 3.30) налаштовано таким чином, щоб запускати службу кожні 10 хвилин. В секції `[Timer]` вказано `OnBootSec=3min` – перший запуск через 3 хвилини після старту системи, `OnUnitActiveSec=10min` – далі кожні 10 хвилин після останнього виконання. Опція `Persistent=true` гарантує, що у випадку, якщо система була вимкнена довше цього інтервалу, при ввімкненні таймер одразу запускає пропущене оновлення (тобто, не чекає наступного інтервалу).

```
[Unit]
Description=Run secure updater every 10 minutes

[Timer]
OnBootSec=3min
OnUnitActiveSec=10min
Persistent=true

[Install]
WantedBy=timers.target
```

Рисунок 3.30 – Юніт-файл systemd secure-update.timer для періодичного (кожні 10 хв) запуску служби оновлення

Після створення цих файлів, виконуються команди `sudo systemctl daemon-reload` (перезавантаження конфігурації systemd) та `sudo systemctl enable --now secure-update.timer` (увімкнення таймера при старті системи

та негайний його запуск). Відтепер кожні 10 хвилин буде автоматично викликатися наш скрипт, який перевірятиме наявність нових релізів.

3.5Перевірка роботи системи оновлення

Після виконання наведених налаштувань система готова до безпечного отримання оновлень. Розглянемо, як відбувається повний цикл оновлення на практиці.

Розробник підвищує версію застосунку і створює новий тег, наприклад `v1.0.4`, який пушить до репозиторію. GitHub Actions автоматично запускає workflow secure-release для цього тегу. В ході роботи CI генерується новий Docker-образ, підписаний через cosign, публікується attestation, формується новий `release-manifest.json` і підписується. У репозиторії з'являється реліз v1.0.1 з необхідними файлами.

На Raspberry Pi таймер через короткий проміжок (не більше 10 хвилин) запускає скрипт `update.py`. Скрипт звертається до GitHub і бачить, що останній реліз – v1.0.4 (відрізняється від збереженого стану v1.0.0). Він завантажує новий маніфест і підписи. Далі виконується послідовно: перевірка підпису маніфесту (щоб упевнитись, що файли дійсно від нашого CI), порівняння версій (щоб виключити відкат), завантаження образу, перевірка його підпису, перевірка attestation з політикою.

Якщо усі перевірки успішні, скрипт оновлює Docker Compose до нового образу. Старий контейнер з версією v1.0.3 буде зупинено і замінено новим екземпляром контейнера версії v1.0.4. Наприкінці скрипт залишає запис у `state.json` про встановлену версію. На рисунку 3.31 наведено фрагмент консольного виводу скрипта оновлення на сервері при успішному застосуванні оновлення. Тут видно результати команд cosign verify-blob, cosign verify, cosign verify-attestation та повідомлення `OK updated to v1.0.4`, що підтверджує успішний перехід на нову версію.

```
deploy compose...
 Container secure_ci_cd_app Running

OK updated to v1.0.4
```

Рисунок 3.31 – Консольний вивід роботи скрипта update.py на сервері Raspberry Pi під час оновлення до версії v1.0.1

Варто зазначити, що при спробі потенційної атаки (наприклад, якщо буде завантажено некоректний підпис чи невірний образ), оновлення не відбудеться. Сценарії, які захищені системою. Якщо зловмисник якимось чином отримає доступ до реєстру або спробує видати себе за сервер GitHub, підписи не співпадуть. Cosign перевіряє і маніфест, і сам образ на коректність підпису та відповідність нашому workflow. Без наявності приватного ключа (якого у нашому випадку навіть не існує, бо використовується OIDC) або доступу до GitHub Actions від нашого репозиторію зловмисник не зможе згенерувати валідний підпис. Навіть якщо атакувальник спробує опублікувати образ від іншого імені чи без workflow, наш сервер це виявить на етапі перевірки attestation. Політика OPA перевіряє саме наші атрибути збірки (repo, digest тощо). Будь-яке відхилення (інший репозиторій, відсутність attestation) призведе до відхилення оновлення. Якщо нападник чи помилково розробник спробує позначити стару вразливу версію як "останню", сервер не встановить її, бо номер версії менший за вже наявний. Це важливо, оскільки відкат на стару версію, це типовий прийом, щоб експлуатувати відомі вразливості.

Таким чином, механізм оновлення побудовано за принципом повної довіри до коду (Pipeline-to-Prod Trust). Сервер виконує оновлення лише тоді, коли всі перевірки пройдені успішно, і відхиляє оновлення при найменшому невідповідному параметрі.

3.6 Висновки до розділу 3

У третьому розділі реалізовано захищений CI/CD-конвеєр для контейнеризованого застосунку та продемонстровано його роботу. Був

створений приватний GitHub-репозиторій з усіма необхідними конфігураціями для автоматизованої збірки і підпису контейнерів. Налаштовано GitHub Actions workflow, що здійснює збірку Docker-образу при створенні нового тегу версії, завантажує цей образ у приватний реєстр GHCR та накладає на нього криптографічний підпис (за допомогою Sigstore Cosign) разом з файлом доказу походження збірки (provenance).

На стороні сервера розгорнуто систему безпечного оновлення. Встановлено Docker і Docker Compose для керування контейнером, налаштовано регулярне виконання скрипта оновлення через systemd timer. Скрипт оновлення інтегрується з GitHub API та реєстром контейнерів, завантажує інформацію про останні релізи і нові образи. Запроваджено багаторівневу перевірку безпеки перед оновленням: верифікація підпису маніфесту релізу, перевірка підпису самого контейнерного образу, а також перевірка його походження через attestation і OPA-політику. Лише у випадку успішного проходження всіх етапів перевірки відбувається автоматичне оновлення запущеного контейнера до нової версії.

Продемонстровано, що система стійка до типових загроз, пов’язаних з ланцюгом постачання: підміни образів, неавторизованих змін у конвеєрі CI/CD, та спроб відкату на уразливі версії. Впровадження keyless-підписів через OIDC забезпечує зручне керування ключами (по суті, відсутність потреби у власних ключах) і водночас довіру до середовища збірки. Поєднання GitHub Releases як каналу доставки метаданих та Docker Registry як сховища образів дозволило досягти повністю автоматизованого, але безпечного процесу оновлення. Цей розділ фактично описує прототип production-grade рішення, яке може бути застосоване для безпечного розгортання контейнерних застосунків зі збереженням цілісності та автентичності на кожному етапі.

РОЗДІЛ 4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1 Охорона праці

Метою кваліфікаційної роботи є розробка та практична реалізація механізму безпечного оновлення контейнеризованих застосунків на основі захищеного CI/CD-конвеєра з використанням криптографічних методів перевірки автентичності та цілісності програмного забезпечення. Оскільки виконання такого типу робіт передбачає застосування комп'ютерної техніки, зокрема ПК і периферійних пристроїв, обов'язковим є дотримання вимог з охорони праці та техніки безпеки.

Для результативної й безпечної роботи колективу працівників, залучених до розробки СВВ, необхідно забезпечити належні та безпечні умови праці. При цьому керівник організації несе пряму відповідальність за порушення нормативно-правових актів з охорони праці [22]. Крім того, на робочих місцях працівників слід гарантувати виконання вимог, затверджених Наказом Мінсоцполітики від 14.02.2018 № 207 «Про затвердження Вимог щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями». Відповідно до Вимог приміщення, у яких розміщені робочі місця операторів (за винятком приміщень, де розташовані робочі місця операторів великих ЕОМ загального призначення – сервер), мають бути обладнані системою автоматичної пожежної сигналізації згідно з такими нормативами:

– Переліком однотипних за призначенням об'єктів, що підлягають обладнанню автоматичними установками пожежогасіння та пожежної сигналізації, затвердженим наказом Міністерства України з питань надзвичайних ситуацій та у справах захисту населення від наслідків Чорнобильської катастрофи від 22.08.2005 № 161, зареєстрованим у Міністерстві юстиції України 05.09.2005 за № 990/11270 (НАПБ Б.06.004-2005);

– Державними будівельними нормами «Інженерне обладнання будинків і споруд. Пожежна автоматика будинків і споруд», затвердженими наказом

Держбуду України від 28.10.98 № 247 (далі – ДБН В.2.5-56:2014), із застосуванням димових пожежних сповіщувачів та переносних вуглекислотних вогнегасників.

В інших приміщеннях дозволяється встановлення теплових пожежних сповіщувачів. Приміщення з робочими місцями операторів повинні бути забезпечені вогнегасниками, кількість яких визначається відповідно до вимог ДСТУ 4297:2004 «Пожежна техніка. Технічне обслуговування вогнегасників». Загальні технічні вимоги, а також з урахуванням граничнодопустимих концентрацій вогнегасної рідини – відповідно до вимог НАПБ А.01.001-2014. Приміщення, у яких розміщені робочі місця операторів сервера загального призначення, оснащуються системою автоматичної пожежної сигналізації та засобами пожежогасіння згідно з вимогами ДБН В.2.5-56:2014, ДБН В.2.5-56:2010, НАПБ А.01.001-2014 та вимогами нормативно-технічної і експлуатаційної документації виробника. Проходи до засобів пожежогасіння повинні залишатися вільними.

Лінія електромережі для живлення комп’ютера та периферійних пристроїв має виконуватися як окрема групова трипровідна мережа з прокладанням фазового, нульового робочого і нульового захисного провідників. Нульовий захисний провідник застосовується для заземлення (занулення) електроприймачів. Забороняється використовувати нульовий робочий провідник як нульовий захисний. Нульовий захисний провідник прокладається від стійки групового розподільного щита або розподільного пункту до розеток електроживлення. Не допускається під’єднання на щиті до одного контактного затискача нульового робочого й нульового захисного провідників. Площа перерізу нульового робочого та нульового захисного провідників у груповій трипровідній мережі повинна бути не меншою за площу перерізу фазового провідника. Усі провідники мають відповідати номінальним параметрам мережі та навантаження, умовам навколишнього середовища, умовам прокладання провідників, температурному режиму та типам апаратури захисту, а також вимогам НПАОП 40.1-1.01-97.

У приміщенні, де одночасно експлуатується понад п’ять комп’ютерів, у помітному та доступному місці встановлюється аварійний резервний вимикач, який може повністю вимкнути електроживлення приміщення, окрім освітлення. Підключення комп’ютерів до електромережі повинно здійснюватися лише через справні штепсельні з’єднання та електророзетки заводського виготовлення.

У штепсельних з’єднаннях і електророзетках, крім контактів фазового та нульового робочого провідників, мають бути передбачені спеціальні контакти для підключення нульового захисного провідника. Конструкція повинна забезпечувати, щоб приєднання нульового захисного провідника відбувалося раніше, ніж приєднання фазового та нульового робочого провідників; під час відключення порядок роз’єднання має бути зворотним. Забороняється підключати комп’ютери до звичайної двопровідної електромережі, зокрема із застосуванням перехідних пристроїв. Електромережі штепсельних з’єднань і електророзеток для живлення комп’ютерної техніки повинні виконуватися за магістральною схемою – по 3–6 з’єднань або електророзеток в одному колі. Штепсельні з’єднання та електророзетки для напруги 12 В і 42 В за конструкцією мають відрізнятися від штепсельних з’єднань для напруги 127 В і 220 В. Також штепсельні з’єднання та електророзетки, розраховані на 12 В і 42 В, повинні візуально (за кольором) відрізнятися від штепсельних з’єднань, розрахованих на 127 В і 220 В.

Під час підвищення ефективності контролю доступу до приміщення, де для забезпечення безпеки мешканців, співробітників і збереження майна використовуються ДС, важливим з точки зору охорони праці є забезпечення достатнього рівня природного та штучного освітлення, визначеного у НПАОП 0.00-7.15-18.

Організація робочого місця фахівця з дослідження методів і програмно-апаратних засобів оптимізаційних процесів на основі ГА повинна гарантувати відповідність усіх елементів робочого місця та їхнього розташування ергономічним вимогам ДСТУ 8604:2015 «Дизайн і ергономіка. Робоче місце для виконання робіт у положенні сидячи. Загальні ергономічні вимоги». Відстань від

екрана до очей фахівців, що працюють за комп’ютером, визначається відповідно до вимог ДСанПіН 3.3.2.007-98.

Розміщення принтера або іншого пристрою введення-виведення інформації на робочому місці повинно забезпечувати хорошу видимість екрана комп’ютера та зручність ручного керування пристроєм введення-виведення в межах досяжності моторного поля згідно з вимогами ДСанПіН 3.3.2.007-98.

Отже, в результаті аналізу вимог щодо охорони праці користувачів комп’ютерів визначено особливості організації робочих місць, вимоги з електробезпеки, а також параметри природного й штучного освітлення, необхідні для ефективної та безпечної роботи фахівців із розробки СВВ, в основі якої використовуються методи детектування аномалій трафіку мережі.

4.2 Вплив електромагнітного імпульсу ядерного вибуху на елементи виробництва та заходи захисту

У воєнний період, у разі застосування ядерної зброї проти України, на електронно-обчислювальне обладнання насамперед діятиме електромагнітний імпульс (ЕМІ) ядерного вибуху – у формі короткочасного імпульсу, що уражає переважно електричну та електронну апаратуру. ЕМІ виникає головним чином унаслідок взаємодії гамма-випромінювання з атомами навколишнього середовища. На формування ЕМІ витрачається незначна частка ядерної енергії, однак він здатний індукувати великі імпульси струмів і напруг у кабелях повітряних та підземних ліній зв’язку, сигналізації, керування, електропередачі, а також в антенах радіостанцій. Дія ЕМІ може призводити до виходу з ладу чутливих електронних і електричних елементів, з’єднаних із великими антенами чи відкритими проводами, а також до збоїв у роботі обчислювальних пристроїв. Вплив ЕМІ необхідно враховувати для всіх електричних та електронних систем [23].

Характерною ознакою ЕМІ як уражаючого чинника є його здатність поширюватися в навколишньому середовищі на десятки й сотні кілометрів. Через це ЕМІ може діяти на об’єкти в тих зонах, де ударна хвиля, світлове

випромінювання та проникаюча радіація вже втрачають значення як уражаючі фактори. За наземних і низьких повітряних вибухів у лініях зв'язку та електроживлення виникають наведені напруги, що можуть спричиняти пробій ізоляції провідників і кабелів відносно землі, а також пробій ізоляції елементів приладів, підключених до повітряних і підземних ліній.

Найуразливішими до впливу ЕМІ є системи зв'язку, сигналізації та керування. Кабелі й апаратура, що використовуються в таких системах, мають обмежену електричну міцність – не більше 10 кВ імпульсної напруги, тоді як наведені ЕМІ імпульси напруги можуть перевищувати ці значення. Особливо чутливою до ЕМІ є апаратура на напівпровідникових елементах та інтегральних схемах, які працюють на малих струмах і напругах, а отже, є сприйнятливими до зовнішніх електричних і магнітних полів, зокрема й елементи програмного засобу керування процесом міграції віртуальних машин в обчислювальній хмарі. ЕМІ здатний пробивати ізоляцію, пошкоджувати (спалювати) елементи електричних схем радіоапаратури, викликати короткі замикання в радіопристроях, іонізацію діелектриків, змінювати або повністю стирати магнітний запис. ЕМІ також негативно впливає на резистори, спричиняє іскріння в їхніх міжконтактних з'єднаннях і на окремих ділянках провідної поверхні. Найбільшу небезпеку ЕМІ становить для апаратури, встановленої в особливо міцних спорудах, здатних витримувати високі тиски ударної хвилі: там обладнання зазвичай не руйнується механічно, проте ЕМІ може вивести з ладу всю незахищену апаратуру систем зв'язку, сигналізації та керування. Максимальні значення часто досягають напруги, що наводяться між кабелем і землею.

Розглянемо можливі підходи до розв'язання задачі захисту від ЕМІ. Найефективнішим (ідеальним) варіантом було б повне екранування приміщення, у якому розміщена радіоелектронна апаратура, суцільним металевим екраном [23]. Водночас очевидно, що в багатьох випадках реалізувати такий захист практично неможливо, оскільки для функціонування апаратури часто потрібен її електричний зв'язок із зовнішніми пристроями. Тому застосовують менш надійні засоби: струмопровідні сітки або плівкові покриття на вікнах, щільникові

металеві конструкції для повітрозабірників і вентиляційних отворів, а також контактні пружинні прокладки, які встановлюють по периметру дверей і люків.

Складнішою з технічної точки зору вважається проблема захисту від проникнення ЕМІ в апаратуру через різноманітні кабельні вводи. Радикальним шляхом могло б бути заміщення електричних мереж зв'язку волоконно-оптичними, які практично не чутливі до ЕМІ. Однак заміна напівпровідникових приладів на електронно-оптичні пристрої в усьому спектрі їх функцій можлива лише у віддаленій перспективі. Тому нині для захисту кабельних вводів найширше застосовують фільтри (зокрема волоконні), а також іскрові розрядники, металлоокисні варистори й високошвидкісні зенеровські діоди [23].

Зазначені засоби мають як сильні сторони, так і обмеження. Так, ємнісно-індуктивні фільтри є достатньо результативними для захисту від ЕМІ невеликої інтенсивності, а волоконні фільтри забезпечують захист у відносно вузькому діапазоні надвисоких частот. Іскрові розрядники характеризуються значною інерційністю й переважно придатні для захисту від перевантажень, що виникають під дією напруг і струмів, індукованих в обшивці літака, кожусі апаратури та в оплетенні кабелю. Металоокисні варистори – це напівпровідникові прилади, провідність яких різко зростає за високої напруги. Водночас, застосовуючи їх як засіб захисту від ЕМІ, слід враховувати недостатню швидкодію та погіршення характеристик за багаторазових впливів навантажень. Цих недоліків не мають високошвидкісні зенеровські діоди, дія яких ґрунтується на різкій лавиноподібній зміні опору – від великого значення практично до нуля – за перевищення прикладеної напруги граничного рівня. Крім того, на відміну від варисторів, параметри зенеровських діодів після багаторазових впливів високих напруг і перемикань режимів не погіршуються.

Найбільш раціональним підходом до проєктування засобів захисту кабельних входів від ЕМІ є створення таких роз'ємів, у конструкції яких передбачено спеціальні заходи для формування елементів фільтрів і встановлення вбудованих зенеровських діодів. Таке рішення дозволяє отримувати дуже малі значення ємності та індуктивності, що є необхідним для захисту від імпульсів малої тривалості й, відповідно, з потужною

високочастотною складовою. Використання роз’ємів подібної конструкції дає змогу розв’язати проблему обмеження малогабаритних характеристик пристрою захисту. Складність завдання захисту від EMI та висока вартість розроблених для цього засобів і методів змушують застосовувати їх вибірково – передусім у особливо важливих системах озброєння та військової техніки. Аналогічний підхід використовується і для захисту систем значної протяжності, керування та зв’язку. Водночас основним способом розв’язання проблеми фахівці вважають створення так званих розподілених мереж зв’язку [23].

ВИСНОВКИ

Під час виконання кваліфікаційної роботи було розроблено та реалізовано механізм безпечного оновлення контейнеризованих застосунків із використанням криптографічних методів перевірки автентичності та цілісності програмного забезпечення. Запропоноване рішення ґрунтується на побудові захищеного CI/CD-конвеєра з наскрізним ланцюжком довіри, який охоплює всі етапи життєвого циклу програмного продукту – від моменту випуску версії розробником до її автоматичного розгортання на кінцевому пристрої.

У першому розділі проведено огляд предметної області безпечного оновлення контейнерів. Розглянуто сучасні системи контейнеризації та оркестрації (Docker, Kubernetes), проаналізовано актуальні загрози безпеці контейнерних середовищ, зокрема вразливості образів і атаки на ланцюжок постачання програмного забезпечення. Досліджено криптографічні основи захищених оновлень, включно з асиметричним шифруванням, геш-функціями та цифровими підписами. Також виконано аналіз існуючих рішень для захищеного оновлення контейнерів (Docker Content Trust, Notation), що дозволило обґрунтувати доцільність використання сучасних механізмів підпису та атестації як бази для побудови власного рішення.

У другому розділі спроєктовано архітектуру захищеного CI/CD-процесу з наскрізним ланцюжком довіри. Детально описано ролі основних компонентів системи: рівень розробки з контрольованими релізами через підписані Git-теги, рівень CI/CD на основі GitHub Actions із використанням keyless-підписів Sigstore Cosign, рівень приватного контейнерного реєстру GitHub Container Registry, а також рівень виконання на кінцевому пристрої Raspberry Pi. Показано, як за допомогою цифрових підписів, сертифікатів OIDC, прозорого журналу Rekor та атестацій походження SLSA формується безперервний ланцюжок довіри, який унеможливлює потрапляння неперевіреного або скомпрометованого коду в середовище виконання.

У третьому розділі реалізовано практичну частину роботи – повноцінний прототип захищеного CI/CD-конвеєра для контейнерного застосунку.

Налаштовано приватний репозиторій GitHub та workflow GitHub Actions для автоматизованої збірки Docker-образу, його публікації в приватному реєстрі та криптографічного підпису без зберігання довготривалих ключів. На стороні сервера Raspberry Pi розгорнуто механізм безпечного оновлення, який автоматично перевіряє підпис маніфесту релізу, підпис контейнерного образу, атестацію походження збірки та коректність версії перед розгортанням. Продемонстровано працездатність системи.

Результати виконаної роботи підтверджують, що розроблений механізм забезпечує високий рівень довіри до процесу оновлення контейнерних застосунків. Запропоноване рішення дозволяє:

- гарантувати автентичність та цілісність контейнерних образів перед їх розгортанням;
- захистити процес оновлення від атак на ланцюжок постачання програмного забезпечення;
- автоматизувати оновлення без зниження рівня безпеки;
- унеможливити встановлення несанкціонованих або застарілих версій програмного забезпечення.

Практичне значення одержаних результатів полягає в можливості застосування розробленої системи в реальних DevOps- та IoT-середовищах, де критично важливими є безпечні автоматичні оновлення. Запропонований підхід є масштабованим, не прив’язаний до конкретного постачальника та може бути адаптований для різних контейнерних платформ, CI/CD-систем і типів кінцевих пристроїв.

Поставлені в роботі мета та завдання були повністю досягнуті. Розроблений захищений CI/CD-конвеєр з наскрізним ланцюжком довіри демонструє ефективність у забезпеченні безпечного оновлення контейнерних систем і може слугувати основою для подальших досліджень та впровадження сучасних підходів до захисту програмного забезпечення в умовах зростання загроз supply chain атак.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. EPAM Ukraine. *Docker security approach*. EPAM Ukraine Blog. https://careers.epam.ua/blog/docker-security-approach
2. HackYourMom. (n.d.). *Безпека Docker: безпека контейнерів (частина 5)*. HackYourMom. https://hackyourmom.com/en/pryvatnist/bezpeka-docker-bezpeka-kontejneriv-chastyna-5/
3. Sysdig. *Secure Kubernetes deployment: Signature verification*. Sysdig Blog. https://www.sysdig.com/blog/secure-kubernetes-deployment-signature-verification
4. Goyal, A. *Securing your Kubernetes deployments: Docker image signing and verification with Cosign and Kyverno*. Medium. https://medium.com/@anil.goyal0057/securing-your-kubernetes-deployments-docker-image-signing-and-verification-with-cosign-and-kyverno-e9bed3ae3efd
5. HackYourMom. *Захоплюючий світ шифрування: огляд типів алгоритмів та їх застосувань*. HackYourMom. https://hackyourmom.com/pryvatnist/zahoplyuyuchyj-svit-shyfruvannya-rozglyad-typiv-algorytmiv-ta-yih-zastosuvan/
6. Wiz. *Container image signing*. Wiz Academy. https://www.wiz.io/academy/container-security/container-image-signing
7. Amazon Web Services. *Cryptographic signing for containers*. AWS Containers Blog. https://aws.amazon.com/blogs/containers/cryptographic-signing-for-containers/
8. Collabnix. *Docker Content Trust*. Collabnix Documentation. https://collabnix.com/docs/docker-for-experts/docker-content-trust-18486/
9. Docker, Inc. *Retiring Docker Content Trust*. Docker Blog. https://www.docker.com/blog/retiring-docker-content-trust/

10. Harbor. *Sign images*. Harbor Documentation. https://goharbor.io/docs/2.6.0/working-with-projects/working-with-images/sign-images/
11. Hovhannisyan, G. *Securing Docker images with Sigstore Cosign*. Medium. https://grigorkh.medium.com/securing-docker-images-with-sigstore-cosign-208a19801b72
12. Stack Overflow. *What is the point in signing tags in Git?* https://stackoverflow.com/questions/5663733/what-is-the-point-in-signing-tags-in-git
13. Sysdig. *How to use GitHub and Sigstore*. Sysdig Documentation. https://docs.sysdig.com/en/docs/sysdig-secure/policies/supply_chain_policies/how-to-github-sigstore/
14. Sigstore. *Cosign signing overview*. Sigstore Documentation. https://docs.sigstore.dev/cosign/signing/overview/
15. SLSA. *Provenance*. Supply-chain Levels for Software Artifacts. https://slsa.dev/spec/v0.1/provenance
16. PSA Certified. *Anti-rollback explained*. PSA Certified Blog. https://www.psacertified.org/blog/anti-rollback-explained/
17. Tymoshchuk, D., Yasniy, O., Mytnyk, M., Zagorodna, N., Tymoshchuk, V., (2024). Detection and classification of DDoS flooding attacks by machine learning method. CEUR Workshop Proceedings, 3842, pp. 184 - 195.
18. Petliak, N., Klots, Y., Karpinski, M., Titova, V., Tymoshchuk, D. Hybrid system for detecting abnormal traffic in IoT. CEUR Workshop Proceedings, (2025), 4057, pp. 21 – 36
19. Klots, Y., Titova, V., Petliak, N., Tymoshchuk, D., Zagorodna, N. Intelligent data monitoring anomaly detection system based on statistical and machine learning approaches. CEUR Workshop Proceedings, (2025), 4042, pp. 80 – 89
20. ТИМОЩУК, Д., & ЯЦКІВ, В. (2024). USING HYPERVISORS TO CREATE A CYBER POLYGON. MEASURING AND COMPUTING DEVICES IN TECHNOLOGICAL PROCESSES, (3), 52-56. https://doi.org/10.31891/2219-9365-2024-79-7

21. ТИМОЩУК, Д., ЯЦКІВ, В., ТИМОЩУК, В., & ЯЦКІВ, Н. (2024). INTERACTIVE CYBERSECURITY TRAINING SYSTEM BASED ON SIMULATION ENVIRONMENTS. MEASURING AND COMPUTING DEVICES IN TECHNOLOGICAL PROCESSES, (4), 215-220. https://doi.org/10.31891/2219-9365-2024-80-26
22. T. Lechachenko, R. Kozak, Y. Skorenkyy, O. Kramar, O. Karelina. Cybersecurity Aspects of Smart Manufacturing Transition to Industry 5.0 Model. CEUR Workshop Proceedings, 2023, 3628, pp. 325–329.
23. Y. Skorenkyy, R. Zolotyy, S. Fedak, O. Kramar, R. Kozak. Digital Twin Implementation in Transition of Smart Manufacturing to Industry 5.0 Practices. CEUR Workshop Proceedings, 2023, 3468, pp. 12–23.
24. Derkach, M., Matiuk, D., Skarga-Bandurova, I., & Zagorodna, N. (2025). CrypticWave: A zero-persistence ephemeral messaging system with client-side encryption.
25. Sachenko A.O., Kochan V.V., Bykovyy P.Ye., Zahorodnia D.I., Osolinskyy O.R., Skarga-Bandurova I.S., Derkach M.V., Orekhov O.O., Stadnik A.O., Kharchenko V.S., Fesenko H.V. Internet of Things for intelligent transport systems: Practicum / A.O. Sachenko (Eds.) – Ministry of Education and Science of Ukraine, Ternopil National Economic University, Volodymyr Dahl East Ukrainian National University, National Aerospace University “Kharkiv Aviation Institute”, 2019. – 135 p. ISBN 978-617-7361-92-2. https://aliot.eu.org/wp-content/uploads/2019/10/ALIOT_ITM3_IoT-for-Int-TransSys_web.pdf
26. Stanko, A., Wieczorek, W., Mykytyshyn, A., Holotenko, O., & Lechachenko, T. (2024). Realtime air quality management: Integrating IoT and Fog computing for effective urban monitoring. CITI, 2024, 2nd.
27. Заікіна Д., Глива В. Основи охорони праці та безпека життєдіяльності. 2019. URL: https://doi.org/10.31435/rsglobal/001
28. Кучма М.М. Цивільна оборона (цивільний захист). Навчальний посібник. Львів : Магнолія 2006 . 2024. 296 с.

Додаток А Публікація

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ІМЕНІ ІВАНА ПУЛЮЯ

М А Т Е Р І А Л И

XIII НАУКОВО-ТЕХНІЧНОЇ КОНФЕРЕНЦІЇ

«ІНФОРМАЦІЙНІ МОДЕЛІ, СИСТЕМИ ТА ТЕХНОЛОГІЇ»

17-18 грудня 2025 року

ТЕРНОПІЛЬ
2025

УДК 004.056.5
Н. Вівчарівський; Т. Лечаченко, доктор філософії
(Тернопільський національний технічний університет імені Івана Пулюя, Україна)

МЕХАНІЗМ БЕЗПЕЧНОГО ОНОВЛЕННЯ КОНТЕЙНЕРІВ ІЗ ЗАСТОСУВАННЯМ КРИПТОГРАФІЇ

UDC 004.056.5
N. Vivcharivskyi; T. Lechachenko, Ph.D.

SECURE CONTAINER UPDATE MECHANISM USING CRYPTOGRAPHY

Сучасні CI/CD-конвеєри пришвидшують розробку програмного забезпечення, проте стають ціллю атак на ланцюги постачання. Компрометація етапів збірки чи деплойменту контейнерів може призвести до впровадження шкідливого коду або витоку секретів. Показовим прикладом є атака GhostAction (вересень 2025 року), коли зловмисники вбудували шкідливі CI-воркфлови у GitHub-репозиторії та викрали понад 3 тисячі секретів із більш ніж 800 проєктів [1]. Це актуалізує потребу у механізмах довіри до контейнерів і конфігурацій на всіх етапах їх життєвого циклу.

У роботі запропоновано механізм безпечного оновлення контейнеризованих застосунків, що поєднує два рівні захисту: криптографічний підпис контейнерних образів і шифрування конфігураційних файлів. Для підпису використано інструмент Cosign з екосистеми Sigstore, який зберігає сигнатури в OCI-сумісному реєстрі та дає змогу виявляти несанкціоновані зміни образу під час розгортання [2]. Конфіденційність секретів (паролі, токени, змінні оточення) забезпечує утиліта Mozilla SOPS, що шифрує YAML/JSON-файли із застосуванням сучасних криптоалгоритмів, тоді як ключі зберігаються у захищених сховищах на кшталт GitHub Secrets або хмарних KMS [3].

Рішення реалізовано як автоматизований CI/CD-процес на базі GitHub Actions: після оновлення коду конвеєр збирає та підписує Docker-образ, завантажує його до реєстру разом із сигнатурою, а конфігурації шифруються SOPS і розшифровуються лише в довіреному середовищі під час деплойменту. На цільових серверах та edge-пристроях перед запуском контейнера виконується перевірка підпису; у разі невдалої валідації оновлення блокується, що є контейнерним аналогом механізму Secure Boot [2]. Експериментальні випробування показали, що такий підхід майже не збільшує час розгортання, проте суттєво зменшує ризики атак на ланцюг постачання, гарантуючи виконання лише довіреного коду та захищеність секретних налаштувань на всьому шляху доставки.

Література

1. Guo, T. (2023, February 22). *Supply Chain Security: Sigstore and Cosign (Part II)*. GitGuardian Blog. https://blog.gitguardian.com/supply-chain-security-sigstore-and-cosign-part-ii/.
2. Wiz. (2023). *What is container image signing?* Wiz Academy. https://www.wiz.io/academy/container-security/container-image-signing.
3. getsops. (30.11.2025). *GitHub – getsops/sops: Simple and flexible tool for managing secrets* (Repository). GitHub. https://github.com/getsops/sops.