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

До захисту допущено
Завідувач кафедри

_____ **Дмитро ЛАНДЕ**
(підпис)

«_____»_____2023 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Системи, технології та математичні методи кібербезпеки»
спеціальності 125 «Кібербезпека»

на тему: Розробка прототипу інфраструктурної конфігурації DevOps для безпечної розробки програм

Виконав (-ла): здобувач вищої освіти **IV** курсу, групи ФБ-96
(шифр групи)

Сендецький Костянтин В'ячеславович 
(прізвище, ім'я, по батькові) (підпис)

Керівник к.т.н., доцент кафедри ІБ, 

Коломицев Михайло Володимирович
(посада, науковий ступінь, вчене звання, прізвище, ім'я, по батькові) (підпис)

Рецензент старший викладач кафедри ММАД 
Тітков Дмитро Валерійович
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище, ім'я, по батькові) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Здобувач вищої освіти 

Київ – 2023 року

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

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

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ Дмитро ЛАНДЕ
(підпис)

«___»_____2023 р.

ЗАВДАННЯ
на дипломну роботу здобувачу вищої освіти

Сендецький Костянтин В'ячеславович
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка прототипу інфраструктурної конфігурації DevOps для безпечної розробки програм,
керівник роботи Коломицев Михайло Володимирович, к.т.н., доцент,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «26» травня 2023 р. № 2028-с

2. Термін подання здобувачем вищої освіти роботи «15» червня 2023 р.

3. Вихідні дані до роботи: методи та практики для забезпечення безпечного процесу розробки програмного забезпечення в DevOps

4. Зміст роботи: Поняття DevOps, роль безпеки в DevOps, визначення основних загроз, поняття DevSecOps, огляд основних методологій забезпечення безпеки, розробка методології для безпечної розробки, аналіз інструментів DevOps, побудова типового DevOps конвеєра, аналіз інструментів для DevSecOps, побудова типового DevSecOps конвеєра на основі створеної методології.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо)
Презентація

6. Дата видачі завдання 18.04.2023

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів дипломної роботи	Примітка
1	Визначення теми роботи, погодження з дипломним керівником	01.10.2022 – 22.10.2022	Виконано
2	Ознайомлення з літературними джерелами, визначення мети дипломної роботи	23.10.2022 – 10.01.2023	Виконано
3	Визначення основних принципів DevOps	11.01.2023 – 20.02.2023	Виконано
4	Визначення питання безпеки в DevOps. Написання першого розділу.	21.02.2023 – 14.04.2023	Виконано
5	Проходження Переддипломної практики. Визначення інструментів для побудови прототипу інфраструктури DevOps. Написання другого розділу	15.04.2023 – 25.05.2023	Виконано
6	Реалізація прототипу інфраструктурної конфігурації DevOps. Написання третього розділу	26.05.2023 – 14.06.2023	Виконано
7	Передзахист дипломної роботи	16.06.2023	
8	Захист дипломної роботи	23.06.2023	

Здобувач вищої освіти  Костянтин СЕНДЕЦЬКИЙ

(підпис) (Власне ім’я, ПРІЗВИЩЕ)

Керівник роботи  Михайло КОЛОМИЦЕВ

(підпис) (Власне ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ

Дипломна робота 71 сторінок, 36 малюнок, 1 таблиця, 20 джерел, 1 додаток.

Об’єкт дослідження: загрози безпеці які виникають в DevOps.

Предмет дослідження: методи та практики безпечної розробки програм в DevSecOps.

Мета роботи: розробка методики забезпечення безпеки в процесі розробки програмного забезпечення в DevOps.

Методи дослідження: аналіз літературних джерел, практик, методів інструментів. Порівнянняк пратик та інструментів. Моделювання отриманих результатів.

В результаті роботи було розроблено та реалізовано методологію для безпечної розробки програм в контексті DevOps.

Ключові слова: DevOps, безпека, CI/CD, пайплайн, SAST, DAST, Docker, Gitlab

ABSTRACT

The thesis consists of 71 pages, including 35 illustrations, 1 table, 20 references, and 1 appendix.

Research object: security threats that arise in DevOps.

Research subject: methods and practices of secure software development in DevSecOps.

The main goal this work: development of a methodology for ensuring security in the process of software development in DevOps.

Research methods: analysis of literature sources, practices, methods and tools. Comparison of practices and tools. Modeling of the obtained results.

As a result of the work, a methodology for secure software development in the context of DevOps was developed and implemented.

Keywords: DevOps, security, CI/CD, pipeline, SAST, DAST, Docker, Gitlab

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИНЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ.

DevOps – методологія розробки програмного забезпечення, яка поєднує процеси розробки та експлуатації.

ПЗ – програмне забезпечення.

Pipeline – послідовність автоматичних кроків.

CI/CD – практика безперервної інтеграції та доставки.

DevSecOps – захищена методологія DevOps.

Build – процес збору (компілювання) програми.

Репозиторій – місце зберігання програмного коду.

Push – відправка коду до репозиторію.

Pull – отримування коду із репозиторію.

Production – середовище де виконується фінальна версія застосунку.

ВСТУП

В сучасному світі розробки програмного забезпечення головним фактором успішності проектів є їх швидкість розробки та доставки до кінцевого користувача. DevOps являється ключовим поняттям, яке об’єднує процеси розробки та адміністрації для досягнення ефективності та автоматизації процесів розробки, випуску та підтримку програм.

Проте в DevOps, як і в будь якій технології, безпека є критичним аспектом, який необхідно враховувати. Для забезпечення безпеки існує методологія DevSecOps, яка використовує практики забезпечення безпеки на кожному етапі життєвого циклу розробки ПЗ. Це фактично об’єднання процесів розробки та адміністрації з безпекою.

У даній роботі ми розглянемо загрози які можуть виникнути в процесах DevOps, методи забезпечення безпеки, та створимо свою методику для безпечної розробки програм в DevOps.

Актуальність роботи зумовлена тим, що в сучасному світі розробки програмного забезпечення все більше актуальним стає питання ефективного способу розробки програм.

Мета і завдання дослідження полягають в визначенні основних вразливостей DevOps в контексті розробки програмного забезпечення, та як їх уникнути.

Методом дослідження є аналіз літературних джерел, електронних ресурсів. Визначення необхідних інструментів. Побудова методики безпечної розробки в DevOps.

Наукова новизна одержаних результатів полягає в дослідженні вразливостей та побудова покращеної безпечної методики DevOps в контексті розробки програмного забезпечення.

Практичне значення одержаних результатів — Результати дослідження можуть використовуватися для покращення безпеки в процесі розробки програмного забезпечення в DevOps системах.

Апробація результатів роботи - Робота на тему “Аналіз інструментів SAST DAST для покращення безпеки коду в DevSecOps” була представлена на Всеукраїнській науково-практичній конференції «Theoretical and Applied Cybersecurity» (TACS-2023) присвячена 100-річному ювілею академіка В.М. Глушкова.

1 ПОНЯТТЯ DEVOPS. БЕЗПЕКА В DEVOPS

1.1 Методологія DevOps

DevOps (Development & Operations) – методологія яка поєднує команди розробників та експлуатації для планування, розробки, доставки та експлуатації впродовж усього життєвого циклу розробки програм. Це забезпечує координацію між співробітниками окремих ролей для більш ефективного, швидкого та надійного процесу розробки ПЗ. Це реалізується шляхом комунікації між командами, автоматизації окремих процесів та використання найкращих практик та інструментів.

DevOps побудований на практиках неперервної інтеграції та доставки (CI/CD). Роль CI/CD є основною в методології DevOps, яка дозволяє автоматизувати та прискорювати процес розробки та доставки ПЗ.

CI (Continuous Integration) – це практика яка дозволяє регулярно інтегрувати зміни коду у спільний репозиторій. При кожній зміні коду, автоматично будуть запускатися процеси збірки та тестування програми, для виявлення потенційних помилок та конфліктів. Це дозволяє швидко виявляти проблеми, підтримувати стабільність та ефективність в процесі розробки ПЗ.

CD (Continuous Delivery) – це практика яка забезпечує автоматичну та неперервну доставку готового ПЗ до цільового середовища. Після успішного етапу Continuous Integration, програма автоматично доставляється та розгортається в середовищі. Це забезпечує швидкий та надійний процес доставки ПЗ.

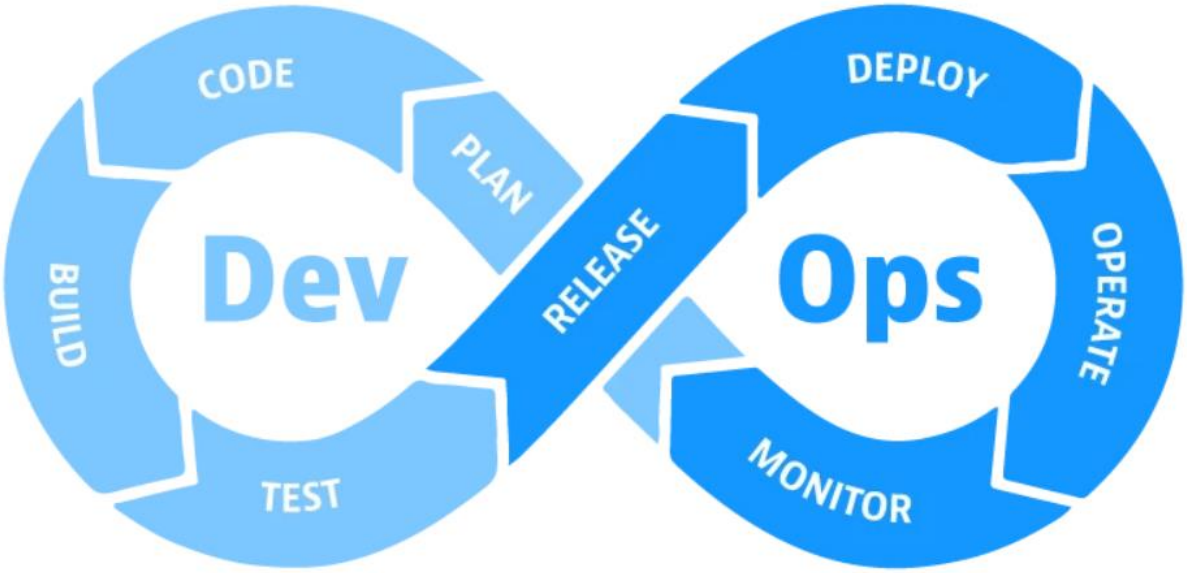


Рисунок 1.1 - Цикл DevOps

Життєвий цикл DevOps складається з таких етапів (Рисунок 1.1):

1. **Plan (Планування)** – етап визначення вимог, задач та цілей ПЗ.
2. **Code (Кодування, розробка)** – етап на якому розробники реалізують та розробляють проект з вимогами які раніше буле визначені.
3. **Build (Збірка проекту)** – на цьому етапі програма збирається та компілюється з усіма необхідними залежностями, файлами та пакетами.
4. **Test (Тестування)** – етап проведення тестування ПЗ для виявлення помилок, проблем функціональності та потенційних загроз.
5. **Release (Випуск)** – етап випуску готового ПЗ після успішного проходження тестів. Визначення версії ПЗ, створення потрібних пакетів для встановлення та підготовка до розгортання.
6. **Deploy (Розгортання)** – програма розгортається у цільовому середовищі.

7. **Operate (Експлуатація)** – програма розгорнута та виконується у середовищі. Процес обслуговування та підтримки ПЗ у цільовому середовищі.
8. **Monitor (Моніторинг)** – етап активного моніторингу роботи програми.

Для покращення швидкості та якості процесу розробки, виділяють такі основні принципи DevOps:

- **Сумісна робота.**

Практика що об’єднує спільні зусилля та знання людей різних галузей, щоб налаштувати їх на ефективне виконання конкретних цілей в загальних інтересах.

- **Автоматизація.**

Дозволяє автоматизувати процеси розробки, тестування, розгортання та моніторингу програм. Це знижує кількість ручної роботи та пришвидшує час доставки програм.

- **Контроль версій коду.**

Відстеження та керування версіями коду. Це дозволяє ефективніше працювати в команді за допомогою історії змін коду, розгалуження коду (branching) та відміною змін до попередньої версії.

- **Неперервна інтеграція (Continuous Integration, CI).**

Забезпечує регулярну та автоматизовану інтеграцію нового функціоналу в програму. Це процес автоматичної збірки та тестування програм.

- **Неперервне розгортання (Continuous Delivery, CD).**

Процес автоматичного розгортання змін в цільове середовище після успішного проходження тестів.

- **Неперервний моніторинг.**

Системи моніторингу дозволяють відстежувати роботу програм, збирати та аналізувати дані про продуктивність та проблеми. Це допомагає оперативно реагувати на несправності та покращувати якість ПЗ або цільового середовища.

- **Інфраструктура як код (IaC).**

Підхід який дозволяє керувати та налаштовувати інфраструктуру шляхом виконання програмного коду. Можна наприклад швидко встановлювати потрібні залежності**.** Через те, що інфраструктура створюється та налаштовується за допомогою програмного коду, при критичних проблемах в інфраструктурі, швидше та ефективніше її буде заново створити та налаштувати ніж проводити аналіз несправностей та намагатися її виправити.

В 2020 році, Atlassian & CITE Research провели дослідження [4], в якому 99 відсотків опитуваних заявили що впровадження DevOps позитивно вплинуло на їх компанії. Розглянемо основні переваги впровадження DevOps в процес розробки ПЗ:

- Прискорення циклу розробки програмного забезпечення завдяки технологіям CI/CD, що дозволяє швидше впроваджувати новий функціонал.
- Покращує якість та надійність програмного забезпечення через систематичне тестування та контроль версій.
- Автоматизація процесів дозволяє знизити ймовірність виникнення помилок, які можуть виникнути в процесі ручної реалізації через людський фактор.
- Покращує комунікацію між командами розробки, тестування та експлуатації, що призводить до більш ефективної командної роботи шляхом розподілення відповідальності.

1.2 Безпека в DevOps

Під час розробки ПЗ, завжди повинно бути актуальним питання безпеки. Використання методології DevOps звісно покращувала ефективність розробки ПЗ, проте цього було недостатньо. Загроз стає все більше, а питання безпеки кінцевого продукту є ключовим для компаній розробки.

У DevOps забезпечення протидії атакам на безпеку є необхідним рішенням для захисту безпеки програм.

Існують 2 важливі документи, які дозволяють ознайомитись з найбільш поширеними та небезпечними загрозами безпеці. Це ресурси OWASP (Open Web Application Security Project) TOP 10 [8] та CWE (Common Weakness Enumeration) 25 [7]. Це списки найбільш популярних небезпечних вразливостей програмного та апаратного забезпечення.

Переглянувши та проаналізувавши ці списки можна визначити найбільш небезпечні та популярні вразливості:

- Недостатня аутентифікація та авторизація.

Тип вразливості який дозволяю зловмиснику обійти процеси аутентифікації та авторизації. Зловмисники можуть проводити атаки пібору паролю, фішинг або вкрасти особисті дані. Причиною цьому можуть стати такі вразливості, як: Слабкі паролі, відсутність їх шифрування; Відсутність двофакторної або мультифакторної авторизації; Недостатньо захищене зберігання облікових даних. Коли дані зберігаються у відкритому середовищі

- Використання зовнішніх застарілих або вразливих компонентів.

Тип вразливості який дозволяє зловмиснику отримати контроль над програмою, пошкодити файли, вкрасти важливу інформацію. Зазвичай використовують вразливості сторонніх бібліотек, функцій, плагінів, фреймворків.

- Незахищена валідація вхідних даних.

Тип вразливості який дозволяє зловмиснику впроваджувати шкідливий код через компоненти зчитування даних у програмі. Приклад популярних загроз: SQL-injection (SQL-ін’єкція), XSS-атаки (Cross-Site Scripting) та інші.

- Незахищена інфраструктура.

Тип вразливості серверів, фізичних або хмарних, коли вони мають незахищену конфігурацію, відкриті порти або інші недоліки безпеки. Зловмисники можуть використовувати різні сканери для виявлення слабких місць системи, та відповідним чином провести атаку.

- Неправильно налаштовані права доступу.

Вразливість при якій зловмисник може отримати несанкціонований доступ до конфіденційної інформації, використовуючи вразливість некоректного налаштування прав доступу.

1.3 Поняття та роль DevSecOps

Для забезпечення безпечної розробки програм існує концепція DevSecOps. Головним її аспектом являється інтеграція безпеки (Security, Sec) в кожен етап процесу розробки ПЗ, починаючи з проектування та розробки програми, закінчуючи розгортанням та моніторингом. Це означає що розробники, фахівці з безпеки та адміністратори мають спільну мету в забезпеченні безпеки.

Мета DevSecOps – пришвидшити випуск саме безпечних програм. Раніше під час виявлення загроз, це займало дуже великий обсяг часу на знайдення всіх причин виникнення загроз,

Існує багато практик та методологій які використовуються в DevSecOps. Основна їх ідея це інтеграція заходів безпеки на самих ранніх етапах розробки ПЗ та уникнути загрози як можна простіше та швидше.

Розглянемо деякі популярні методи забезпечення безпеки в DevSecOps:

- Agile.

Методологія зосереджена на тісному контакті всіх команд розробки та інших задіяних осіб. Як і в DevOps, agile використовується для контакту між командами розробки, а DevSecOps до них приєднується команда безпеки. Це дозволяє виявляти та виправляти помилки колективно та на ранніх етапах розробки.

- Waterfall (Водоспад).

Методологія розробки ПЗ, при якому розробка йде лінійним шляхом, причому кожен наступний крок спирається на попередній. В DevSecOps це важливо щоб вимоги безпеки були розглянуті перш ніж переходити до наступних етапів.

- Shift-left.

Методологія в якій питання безпеки, а саме тестування безпеки, виконуються на самих ранніх етапах розробки, щоб раніше виявити та вирішити проблему.

- Threat Modeling (Моделювання загроз).

Команди безпеки на ранніх етапах розробки ПЗ виявляють потенційні загрози додатку, потім на основі аналізу загроз розробляють або моделюють варіанти усунення проблеми.

- Secure Development Lifecycle (SDL).

Практика, задача якої впровадити безпеку на кожен етап розробки, від етапу планування до запуску в цільовому середовищі.

- Безпека процесу розробки.

Планування безпеки починається на самому ранньому етапі розробки. Розробники пишуть ПЗ відповідно до кращих стандартів, практик та методів побудови безпечної архітектури програми.

- Безперервний моніторинг.

Допомагає активно слідкувати за проблемами які виникають під час роботи ПЗ або цільового середовища.

1.4 Основні інструменти та методи створення процесу безпечної розробки

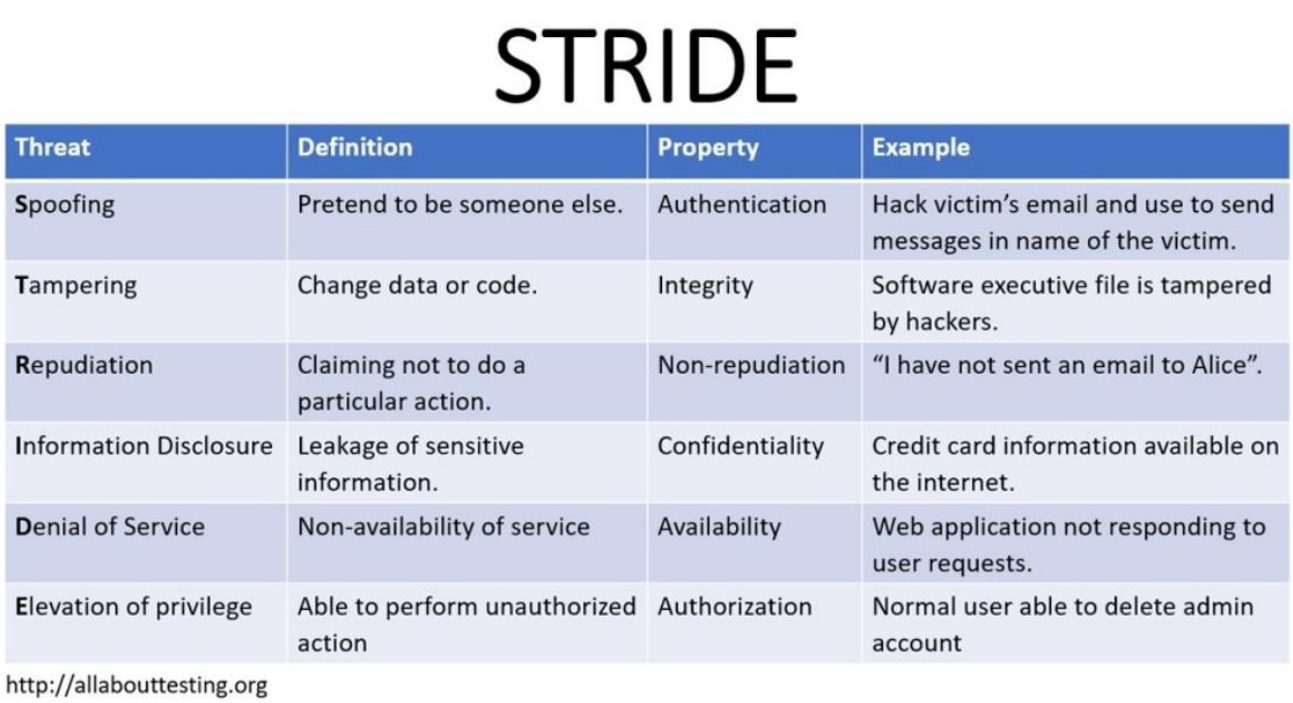
Існує дуже багато різних методів та інструментів які допомагають забезпечити безпеку в DevSecOps [16]. Далі ми їх розглянемо.

1.4.1 Аналіз загроз

Модель загроз допомагає виявити потенційні загрози системи або програми, які можуть бути виконані від лиця зловмисника. Головна мета це виявлення загроз безпеки на самому ранньому етапі розробки, а не в процесі коли застосунок знаходиться в процесі виконання.

STRIDE - один з популярних методів моделювання загроз. STRIDE розшифровується як: Spoofing (Підробка), Tampering (Втручання), Repudiation (Відмова), Information Disclosure (Розкривання інформації), Denial of Service

(Відмові в обслуговувані), Elevation of Privilege (Підвищення привілеїв). Це одні з найрозповсюджених загроз безпеці систем та застосунків. [15]



STRIDE

Threat	Definition	Property	Example
Spoofing	Pretend to be someone else.	Authentication	Hack victim's email and use to send messages in name of the victim.
Tampering	Change data or code.	Integrity	Software executive file is tampered by hackers.
Repudiation	Claiming not to do a particular action.	Non-repudiation	"I have not sent an email to Alice".
Information Disclosure	Leakage of sensitive information.	Confidentiality	Credit card information available on the internet.
Denial of Service	Non-availability of service	Availability	Web application not responding to user requests.
Elevation of privilege	Able to perform unauthorized action	Authorization	Normal user able to delete admin account

http://allabouttesting.org

Рисунок 1.2 - Модель STRIDE

Модель допомагає відповісти на такі питання [15]:

- Як зловмисник може змінити дані автентифікації?
- Який буде вплив, якщо зловмисник зможе прочитати дані користувача?
- Що станеться, якщо буде обмежено доступ до бази даних профілю користувача
- Та інші.

Реалізувати аналіз загроз можна визначивши наступні критерії:

1. Визначення та аналіз системи для якої ми будемо моделювати аналіз загроз.
2. Збір інформації архітектури нашого застосунку.
3. Ідентифікація критичних загроз які можуть вплинути на застосунок.

4. Оцінка наслідків на кожному етапі DevSecOps-конвеєра.
5. Розробка стратегій які можуть виявити або пом’якшити вплив та ймовірність виникнення загроз.

1.4.2 Інструменти статичного аналізу

Static Application Security Testing (SAST) або “White-box testing” – формат тестування програми, який сканує вихідний код та всі його компоненти для виявлення потенційних вразливостей [21].

Переваги використання SAST:

- Тестування на ранніх етапах розробки.
- Може застосовуватися до широкого спектра програм.
- Видає повну інформацію щодо вразливості, де вона виникла, з яких причин, та запропонує варіанти вирішення проблеми.
- Аналізую тільки код, без запуску програми.

Недоліки використання SAST:

- Не завжди точно визначає вразливість.
- Може видавати хибні спрацьовування.

1.4.3 Інструменти динамічного аналізу

Dynamic Application Security Testing (DAST) – або "Black-box testing" – формат тестування, який шукає вразливості шляхом впровадження типових сценаріїв атаки на додаток [21].

Переваги:

- Може виявити вразливості втрачені під час використання SAST
- Більш широкий спектр тестів у порівнянні з SAST

- Може сканувати в реальному часі

Недоліки:

- Тестування тільки після завершення розробки.
- Може видавати хибні спрацьовування.
- Займає багато часу та потребую багато ресурсів.

1.4.4 Інфраструктура як код (IaC)

Інфраструктура як код дозволяє керувати інфраструктурними ресурсами, такими як віртуальні машини, мережі сховища даних та інші з використанням коду. Замість усієї ручної роботи, є можливість просто запускати програмний код.

Найголовніша перевага це масштабування. IaC дозволяє автоматично застосовувати зміни конфігурації, створюючи або змінюючи ресурси.

1.4.5 Інструменти моніторингу

Моніторинг використовується для швидкого та ефективного виявлення загроз у додатках або інфраструктурі.

Шляхом постійного моніторингу дозволяє виявити на ранній стадії загрози та негайно їх виправити.

Чому важлива інтеграція безпеки в DevOps?

- Це дозволяє виявляти помилки та вразливості на ранніх етапах розробки, коли вони відносно прості у виправленні. Зазвичай це аналізатори відкритого коду, які аналізують сам програмний код, шукають потенційні вразливості та дають варіанти їх виправлення.

- Знижуються витрати на час виявлення та виправлення, так як виявити загрозу на ранніх етапах набагато ефективніше ніж виявлення у вже запущеному середовищі.
- Весь аналіз безпеки проходить у автоматизованому режимі, що є набагато ефективніше та швидше ніж вручну.

Висновки до розділу 1

В даному розділі ми визначили що методологія DevOps є дуже корисною для покращення ефективності в процесі розробки ПЗ. Компанії впроваджують DevOps для пришвидшення процесу випуску проектів, за допомогою його основних практик.

Було визначено що типова методологія DevOps окремо не розглядає проблеми забезпечення безпеки. Є можливість експлуатації таких типових загроз як: Недостатня аутентифікація та авторизація, Використання зовнішніх застарілих або вразливих компонентів, Незахищена валідація вхідних даних, Незахищена інфраструктура, Неправильно налаштований контроль доступу. Щоб вирішити питання безпеки було визначену методологію впровадження безпеки на кожен епат життєвого циклу розробки програмного забезпечення DevSecOps.

DevSecOps це лише поняття впровадження безпеки в процес розробки ПЗ. Для інтеграції DevSecOps існують деякі популярні методика, ми розглянули: Agile, Waterfall, Shift-left, Моделювання загроз, Secure Development Lifecycle, Безпека процесу розробки, Безперервний моніторинг. Також окремо розглянули інструменти які забезпечують виявлення загроз безпеки, такі як: інструменти

статичного аналізу, інструменти динамічного аналізу, інфраструктура як код, інструменти моніторингу.

Інтеграція безпеки в DevOps являється дуже важливою практикою, яка допомагає розробляти захищене та надійне ПЗ.

2 РОЗРОБКА МЕТОДИКИ ПРОЕКТУВАННЯ ІНФРАСТРУКТУРИ ДЛЯ БЕЗПЕЧНОЇ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Огляд методики

В цьому розділі ми будемо створювати методику для проектування інфраструктурної конфігурації DevOps для безпечної розробки програм.
В першому розділі ми розібрали основні принципи та практики побудови безпечної конфігурації DevSecOps. За допомогою нашого аналізу, ми зможемо створити методику для розробки конфігурації DevOps для безпечної розробки програм.

2.2.1 Основні компоненти та рішення реалізації

На основі отриманих знань, можемо побудувати стратегії розробки нашого прототипу інфраструктури для безпечної розробки програм

- Для початку реалізації будь якого DevOps конвеєра, треба визначити ціль з якою його треба впроваджувати. Це може бути як прискорення процесу розробки, захист даних, захист застосунку та багато чого іншого.
- Далі треба виявити основні загрози які можуть виникнути в проекті. Як вони можуть виникнути, як вони будуть впливати на безпеку.
- На основі аналізу ризиків ми можемо розробити стратегію їх протидії загрозам. Треба виявити набір практик які будуть впроваджені в процес розробки для мінімізації ризиків.
- І нарешті треба інтегрувати безпеку в процес розробки. Зробити безпеки невід’ємною частиною. Треба переконатися що безпека була впроваджена на кожен етап життєвого циклу розробки ПЗ.

В якості прикладу ми будемо розробляти веб-застосунок. Ми хочемо щоб наш застосунок був розроблений за допомогою практик DevOps. Це означає що головна наша мета це ефективна розробка ПЗ, з максимально доступними автоматизаціями головних процесів. Наша програма буде проходити всі шляхи після розробки автоматично, що буде значно пришвидшувати процес розробки ПЗ та його випуску. Також це дозволить мінімізувати шанс виникнення помилок, які могли б виникнути при ручному виконанні.

Далі нам треба визначити основні питання безпеки нашого застосунку. Будуть проаналізовані потенційні загрози, та на цій основі виберемо інструменти та практики для побудови нашого DevOps-конвеєра для безпечної розробки. Ці інструменти вбудуємо в наш CI/CD пайплайн, щоб максимально автоматизувати всі перевірки безпеки.

2.2 Огляд проекту та вибір інструментів для побудови інфраструктури типового DevOps-конвеєра

Почнемо з побудови типового DevOps-конвеєра для нашого проекту. Визначимо які саме шляхи нам потрібні:

1. Розробник пише програму або модифікацію та публікує її в репозиторії коду.
2. Виконання автоматичних модульних тестів.
3. Після успішного проходження тестів, програма автоматично збирається та упаковується в контейнер зі всіма залежностями.
4. Контейнер публікується в репозиторії.
5. Розгортання ПЗ у інфраструктурі шляхом отримання контейнеру із репозиторію.
6. Моніторинг продуктивності та доступності ПЗ в інфраструктурі.

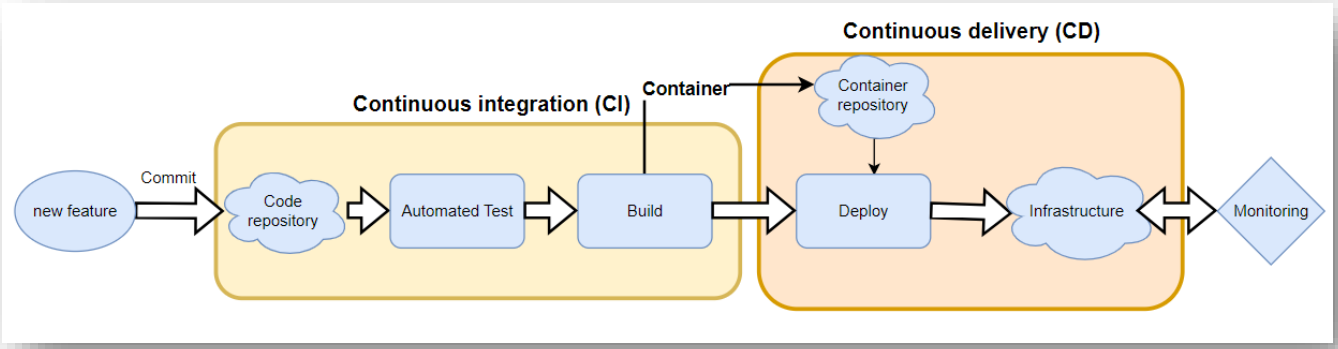


Рисунок 2.1 – Типовий DevOps-конвеєр

Далі визначимо основні інструменти для побудови DevOps-конвеєра

2.2.1 Система контролю версій

Розробник створив код на своїй локальній машині, там він його і зберігає. Проте у нього немає ніяких можливостей відслідковувати зміни які він вносив в свій проект. І тут виникає декілька питань: Що робити якщо треба протестувати новий функціонал у цьому ж проекті? Що робити якщо нові зміни у проект все зламали, та треба повернутися на попередню стабільну версію? Що робити якщо його друг-розробник захотів допомогти йому з проектом?

Щоб відповісти на ці, та подібні питання, ми ознайомимося з поняттям системи контролю версій

Система контролю версій – інструмент за допомогою якого розробники можуть відслідковувати зміни в коді або в інших файлах проекту. Це дозволяє гнучко керувати та відстежувати різні версії написаних файлів. Ця системи реагує на кожну зміну в файлі, дозволяючи повернутися до минулої версії, поєднати зміни різних розробників та керувати конфліктами при одночасному редагуванні файлів.

Головним інструментом контролю версій є Git. Код можна зберігати в спеціальних репозиторіях які розроблені на основі Git. Серед популярних репозиторіїв є GitLab, GitHub, Bitbucket. Кожен з них має деякі переваги. Наприклад GitHub має найбільшу популярність серед розробників, він має весь необхідний функціонал для спільної роботи, він широко використовується для взаємодії між програмістами над розробкою відкритого ПЗ.

В нашій роботі ми будемо використовувати сам GitLab, так як в нього можна без проблем інтегрувати деякі інструменти які ми в подальшому будемо використовувати. Також тут можна створити безкоштовний приватний репозиторій.

Переваги системи контролю версій:

- Дозволяє колективно працювати над проектом.
- Код зберігається в єдиному для всіх розробників репозиторії.
- Є можливість слідкувати як за своїми змінами в коді та і за змінами інших людей.
- Гілки (branches). Основна гілка на якій розробляється та зберігається фінальні зміни коду, називається main (або master). Проте кожен розробник може працювати на окремій гілці над своєю особистою задачею, потім цю гілку можна або поєднати з гілкою іншого розробника, або додати до основної.
- Є можливість відновлення до минулих версій, у разі якщо було знайдено деякі помилки.

В контексті нашого проекту це один з ключових елементів, в ньому ми будемо зберігати наш код, який в подальшому буде використовуватися для наступних кроків DevOps-конвеєру.

2.2.2 Система Ci/Cd

Як ми раніше визначили, системи CI/CD є ключовими при побудові DevOps-конвеєра. Це дозволяє автоматизувати всі процеси від розробки ПЗ до його запуску. В нашій роботі пайплайн CI/CD буде запускатися при кожній новій зміні коду, пайплайн буде містити такі кроки:

В нашій роботі ми будемо розробляти CI/CD пайплайн з наступних кроків:

1. Пайплайн запускається коли розробник написав модифікацію нашого проекту та завантажив в репозиторій.
2. Наш код компілюється, шляхом встановлення всіх необхідних зележностей та бібліотек, та створює робочу версію застосунку.
3. Після успішної збірки програми, запускаються автоматичні тести які перевіряють функціональність нашого застосунку.
4. Після успішного проходження тестів наш застосунок готовий до розгортання в цільовому середовищі. Там він і буде запущений.
5. Програма запущена в середовищі, збираються відповідні метрики щодо якості роботи.

Реалізація пайплайну дарує нам значні покращення в швидкості та ефективності розробки нашого проекту.

Існує багато інструментів CI/CD, найпопулярніші з них: Jenkins, GitLab CI/CD, Travis CI, Circle CI, TeamCity. Основна їх відмінність у інтеграції з різними платформами та репозиторіями. Найгнучкіший із них це Jenkins, для якого можна створювати та інтегрувати різні плагіни, його можна налаштовувати під кожного розробника, та під кожні вимоги.

Для нашого проекту ми будемо використовувати GitLab CI/CD, який є інструментом платформи GitLab. Ми зможемо легко налаштовувати наші пайплайни, та мати інтеграцією з репозиторієм.

2.2.3 Контейнеризація

Останнім часом у сфері розробки ПЗ стрімкої популярності набуває поняття контейнерів.

Контейнеризація – це метод забирання (упаковування) програм з усіма її залежностями в ізольованому середовище. Контейнери створюються на основі створених до цього образів. Головним інструментом є Docker.

В нашому випадку ми будемо використовувати його на етапі збору нашої програми. Для цього буде написаний спеціальний Dockerfile, в якому будуть описані правила за якими будуть встановлюватися всі залежності нашого застосунку. На основі Dockerfile буде створений образ який потім буде запускатися в якості окремого контейнеру. Тепер наш застосунок буде коректно працювати в ізольованому середовищі з усіма залежностями.

Переваги контейнеризації:

1. Відносно легкий процес збирання програми.
2. Контейнери мають ізольоване середовище, що можна налаштувати самому.
3. Контейнери можна запускати на будь-якій платформі де встановлений інструмент контейнеризації. Що звісно відрізняється від прямого запуску, наприклад, програми на другому комп’ютері на якому треба завантажити всі залежності та бібліотеки.

Наша програма буде містити декілька файлів та деякі залежності які прописані в окремому файлі. Також в майбутньому наш веб-застосунок буде завантажений на

production-сервер. Щоб все працювало правильно ми будемо використовувати контейнеризацію.

2.2.4 Інфраструктура

Після успішного проходження етапів збірки та тестування, наша програма готова до запуску в production-середовищі.

В якості інфраструктури розгортання можна обрати будь-яке середовище, наприклад апаратне (фізичні сервери), віртуальне та хмарне.

В нашому випадку будемо використовувати хмарне середовище. Хмарні послуги стрімко набирають популярність, вони надають послугу різноманітними сервісами, які самі обслуговують.

Замість того щоб фізично налаштовувати власну фізичну інфраструктуру, ми можемо її орендувати у хмарного провайдера. Один із провідних хмарних провайдерів це AWS. Тут ми будемо використовувати сервіс EC2, який надає нам доступ до окремої інфраструктури.

2.2.5 Моніторинг

Моніторинг дозволяє відстежувати та аналізувати роботу інфраструктури та програми. Це допомагає виявляти проблеми та вразливості.

2.2.6 Огляд нашого типового DevOps-конвеєра

Ми визначили основні етапи які потрібні для побудови типового DevOps-конвеєра. Це дозволяє нам швидко та ефективно розробляти ПЗ та запускати його в цільовому середовищі.

Проте питання безпеки залишається відкритим, далі ми розглянемо інструменти та практики забезпечення безпечної розробки в контексті DevOps

2.3 Інструменти та практики побудови інфраструктури DevOps безпечної розробки програм

Ми реалізували типовий DevOps-конвеєр для нашого застосунку. Проте ми взагалі ігнорували питання загроз. В цьому розділі ми розглянемо основні принципи та практики для побудови захищеного DevOps-конвеєра.

Раніше ми визначилися з основною метою впровадження DevOps методології. Один із пунктів це забезпечення безпечної розробки.

Ми визначили що треба оцінити які вразливості можуть будуть використанні в нашому застосунку та на цьому аналізі створити набір правил які ми будемо використовувати для покращення безпеки розробки ПЗ.

Головна мета це мінімізувати ризики які можуть виникнути в нашому застосунку. Щоб цього уникнути, ми будемо інтегрувати безпеку на кожен етап розробки ПЗ. Від процесу розроби до його запуску в цільовому середовищі.

2.3.1 Аналіз ризиків та їх усунення

Треба визначити область захищення. Це може бути програма або окрема система в DevSecOps-конвеєрі.

В якості прикладу будемо розглядати власний проект. В нашому випадку ми будемо розглядати веб-застосунок на JavaScript з використанням бази даних.

- Визначення архітектури, дизайн, компоненти програми та їх особливості роботи.

В нашому випадку ми розробляємо веб-застосунок на мові програмування JavaScript, з використанням фреймворку для розробки веб-застосунків Express. Застосунок містить в собі поля вводу, інформація занесена в них буде записана в базу даних, та висвітлює їх на головній сторінці. В якості БД буде використовуватися SQLite, яка зберігає дані в єдиному файлі.

- Визначення загроз які можуть спричинити проблеми безпеці програми.

Спеціалісти з інформаційної безпеки зазвичай проводять аудит безпеки на основі відомих практик, політик та інструментів. В нашому випадку, на основі попереднього аналізу архітектури застосунку, ми виявимо потенційні загрози. Далі ми зможемо використати спеціальні інструменти для виявлення вразливостей в ПЗ.

Проаналізувавши архітектуру нашого застосунку, ми виявили такі потенційно вразливі елементи:

- Програма має поля вводу. Це може бути використано для потенційних атак впровадження шкідливого коду, такі як XSS-атаки або SQL-ін’єкції, та інші.
- Використання бази даних. Може бути потенційно виконані SQL-ін’єкції, або несанкціонований доступ до даних.
- Важливі дані. Найпоширеніша вразливість це зберігання конфіденційної інформації в файлах коду. Це розглядається при побудові будь-якого застосунку. Зловмисники можуть спокійно та без проблем виявляти секрети в програмному коді, та використовувати секретні дані для власної вигоди.
- Використання сторонніх бібліотек. Будь-які фреймворки які ми використовуємо, працюють на основі інших сторонніх бібліотек, та й ті бібліотеки можуть використовувати інші модулі. Зловмисники можуть використовувати деякі вразливості інших модулів які використовуються в окремому фреймворку.

- Визначення інструментів аналізу безпеки.

Для того що переконатися в наших дослідженнях вразливостей, треба впровадити відповідні інструменти аналізу безпеки. Впроваджувати ми їх будемо в наш пайплайн для того щоб перевірка відбувалася автоматично після кожної зміни к коді. Вибір інструментів ми будемо використовувати на основі наших попередніх досліджень.

Звичайно ми будемо використовувати основні з інструментів аналізу безпеки в DevSecOps. А саме:

- Інструменти статичного аналізу. Інструменти SAST сканують наш програмний код шляхом аналізу кожної строки та співставленню її з програмною базою вразливих практик.
- Інструменти динамічного аналізу. Сканування буде проходити в режимі реального часу, будуть виконуватися типові небезпечні запити до застосунку, та аналізувати його відповідь.

Також на основі аналізу наших потенційних загроз, ми виявимо деякі потрібні сканери безпеки для нашого особистого проекту. А саме:

- Інструменти пошуку секретів. В нашій програмі потенційно можуть зберігатися секрети у відкритому вигляді. Це можуть бути різні паролі, ключі або інша конфіденційна інформація.
- Інструменти аналізу залежностей. Наш застосунок використовує деякі бібліотеки для своєї роботи. В деяких випадках нам для наших застосунків треба специфічні версії модулів, вони можуть бути на декілька версій пізніше, але на то вони й пізніші. Такі версії можуть містити різноманітні вразливості. Для цього нам і треба переконатися що наша залежність безпечна.
- Інструменти аналізу контейнерів. Так як наш застосунок буде виконуватися у ізольованому середовищі контейнера, крок його аналізу теж є важливим в

нашій системі. Можуть бути виявлені деякі вразливі залежності та конфігурації.

2.3.2 План впровадження безпеки

Далі можна визначити план за яким буде впроваджуватися безпека в наш процес розробки та які він буде реалізований

1. Планування та розробка.
    a. Аналіз потенційних загроз. Процес що допомагає виявити потенційні загрози для системи або програми на ранніх етапах розробки.
    b. Використання кращих практик розробки ПЗ, та налаштування інших залежностей.
2. Безпека розробки коду.
    a. Статичний аналіз безпеки програми (Static Application Security Testing, SAST). Автоматичне сканування вихідного коду на наявність вразливостей.
    b. Інші аналізатори, які були визначені при аналізі потенційних загроз.
3. Збирання та тестування проекту.
    a. Динамічний аналіз безпеки програми (Dynamic Application Security Testing, DAST). Сканування програми шляхом впровадження типових сценаріїв атаки на додаток. Відстежує вектори атак, поведінку вразливостей та видає детальну інформацію щодо загроз.
    b. Безпека контейнерів. Сканування контейнерів.
4. Інфраструктура.
    a. Конфігурація інфраструктури. Налаштування безпеки інфраструктури в AWS.

5. Моніторинг. Практика постійного моніторингу та аналізу програми або системи.

2.4 Принципи розробленої методики

Для розробки методики ми будемо використовувати найкращі практики в контексті розробки програмного забезпечення. Вона являє собою розширений набір рекомендацій, практик та інструментів.

- **Інтеграція безпеки від початку до кінця.** Безпека буде розглядатися як невід'ємна частина кожного етапу життєвого циклу DevOps.
- **Безпека коду.** Використання та впровадження інструментів сканування безпеки коду.
- **Автоматизація безпеки.** Інструменти та методи перевірки безпеки використовуються автоматично. Виконується це шляхом впровадження їх в CI/CD pipeline.
- **Використання методу shift-left.** Кожний наступний етап пайплайну буде розпочатий, якщо успішно завершився попередній.
- **Використання контейнеризації.** Програми будуть збиратися у вигляді контейнерів.
- **Використання хмарної інфраструктури.** Програми будуть розгортатися та запускатися в хмарному середовищі. Це гарна практика для швидкого та ефективного створення та використання хмарної інфраструктури.
- **Моніторинг інфраструктури.** Відстеження активності production-середовища.

2.5 Схема реалізації методики розробки прототипу безпечного DevOps

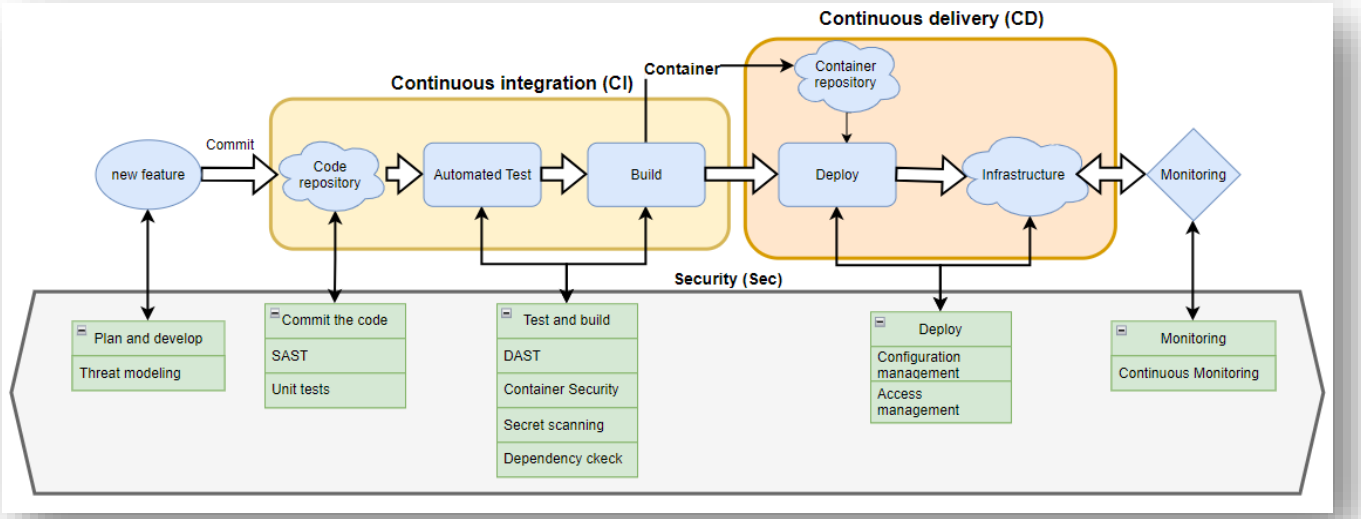


Рисунок 2.2 – Безпечний DevOps-конвеєр

Дана схема висвітлює життєвий цикл безпечної розробки програми в контексті DevOps.

Етапи життєвого циклу програми:

1. Проводиться планування та розробка архітектури програмного забезпечення. Проводиться аналіз вразливостей. Розробляються попередні плани розробки ПЗ.
2. При завантажені коду в репозиторій проводяться перевірки залежностей та статичні аналізи вразливостей та несправностей коду за допомогою інструментів SAST.
3. Після успішного проходження тестування, програма збирається в контейнер. Проводиться аналіз безпеки контейнеру за допомогою спеціальних методів та інструментів. Програма також тестується за допомогою інструментів

динамічного сканування DAST для виявлення вразливостей готового проекту.

4. Програма готова до розгортання в інфраструктуру. Інфраструктура налаштована згідно вимогам безпеки конфігурації та різними її компонентами та ресурсами. Проводиться тестування вже готової програми на її коректну роботу.
5. Програма працює в інфраструктурі. Тепер за допомогою постійного моніторингу є можливість відслідковувати несправності та оперативно на них реагувати.

Висновки до розділу 2

В даному розділі ми розглянули процес створення прототипу інфраструктурної конфігурації DevOps для безпечної розробки програм. В якості прикладу ми розробляли методику для розробки веб-застосунку. Ми визначили основні етапи які треба для побудови типового DevOps-конвеєра, головною метою якого є пришвидшення процесу розробки та випуску програм. Було побудовано модель типового DevOps-конвеєра, та вибрано основні практики та інструменти для побудови.

Було проаналізовано основні загрози які могли виникнути в нашому проекті за допомогою аналізу потенційних загроз. Програма мала потенційні вразливості, до яких ми обрали інструменти сканування безпеки, а саме інструменти статичного аналізу, інструменти динамічного аналізу, інструменти пошуку секретів, інструменти аналізу залежностей, інструменти аналізу контейнерів.

Для впровадження наших практик та інструментів ми склали план впровадження безпеки. Після цього було створено кінцеві методи та практики для розробки прототипу інфраструктурної конфігурації DevOps для безпечної розробки програм.

Він складається з таких компонентів: Інтеграція безпеки від початку до кінця, Безпека коду, Автоматизація безпеки, Використання методу Shift-left, Використання контейнеризації, Використання інфраструктури, Моніторинг інфраструктури.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОТОТИПУ ІНФРАСТРУКТУРИ DEVOPS

3.1 Огляд проекту

В якості нашого проекту для якого будемо будувати DevOps-конвеєр, як ми раніше розглянули, буде веб-застосунок [10].

Застосунок складається з таких основних файлів: index.html, style.css, script.js, server.js, tests.test.js.

Програма написана на мові JavaScript, це сайт який приймає від користувача вхідні дані “name” та “email”, потім виводить ці дані на екран. Реалізація була за допомогою модулів express, який реалізує логіку всього веб додатка, та sqlite3, для взаємодії з базою даних, яка зберігає дані в файлі. Програма прослуховується на порту 5000.

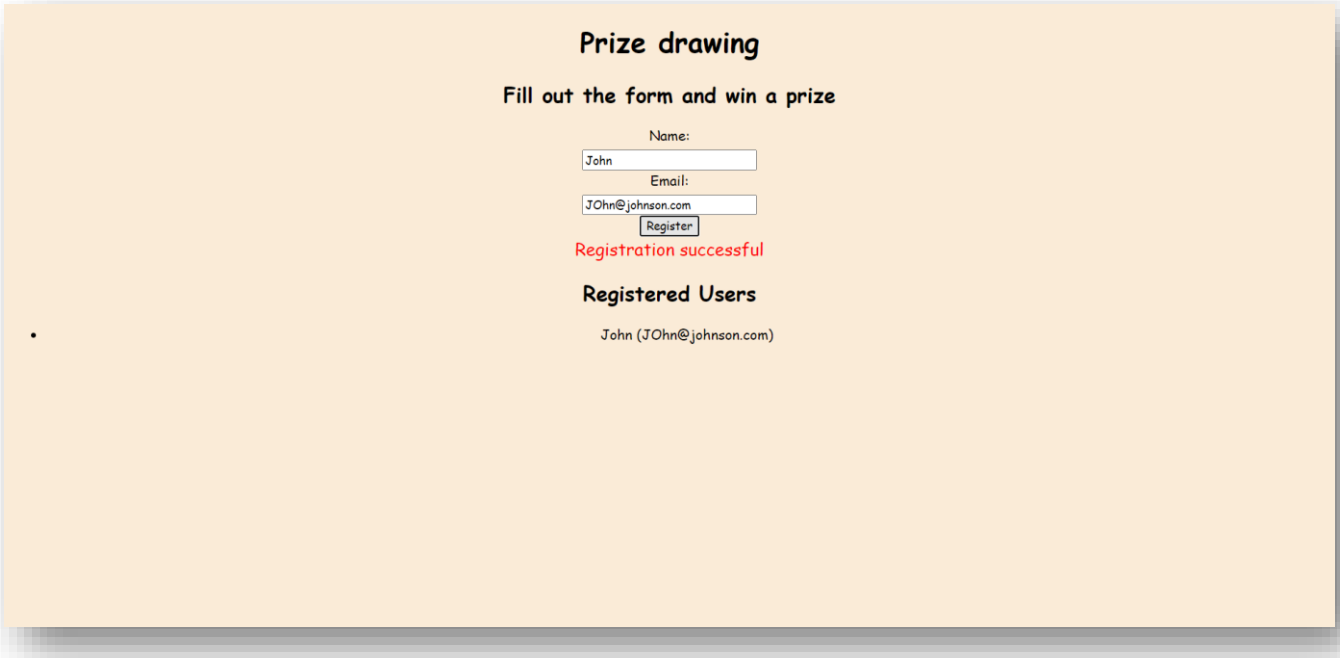


Рисунок 3.1 - Огляд цільового додатку

3.2 Розробка та налаштування прототипу інфраструктури DevOps

3.2.1 Процес розробки та зберігання коду в репозиторії Gitlab

Першим етапом побудови інфраструктури DevOps для безпечної розробки програм буде вибір системи контролю версій. В нашому випадку це буде Git, а в якості репозиторію буде GitLab.

Створимо репозиторій. Додаємо ключ ssh для нашого з'єднання з репозиторієм. Спочатку згенеруємо його на нашій машині за допомогою ssh-keygen, та запишемо його в налаштування GitLab.

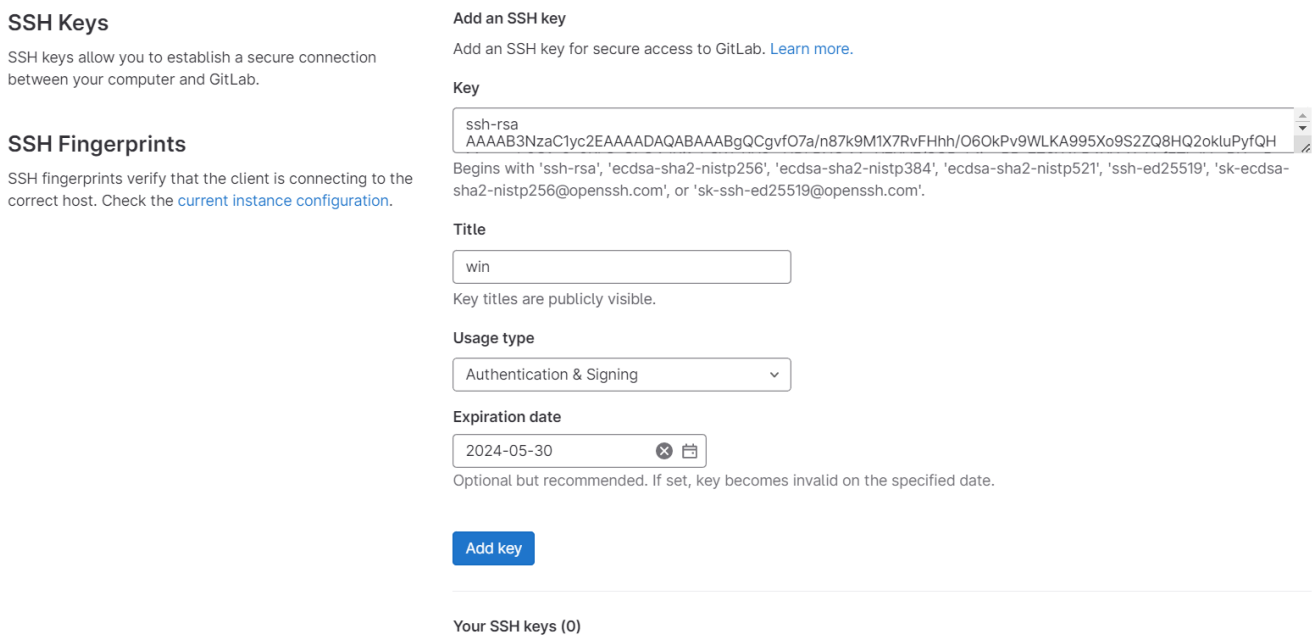


Рисунок 3.2 - З'єднання з GitLab

Завантажимо (git pull) репозиторій на власну машину, та запушимо (git push) всі файли нашого проект в репозиторій.

```
Konstantin@DESKTOP-F60K1GH MINGW64 ~/Desktop/proj/web-site (main)
$ git commit -m "New Version"
[main f7cd90e] New Version
 10 files changed, 5176 insertions(+), 5 deletions(-)
 create mode 100644 .gitignore
 create mode 100644 Dockerfile
 create mode 100644 app/index.html
 create mode 100644 app/script.js
 create mode 100644 app/style.css
 create mode 100644 package-lock.json
 create mode 100644 package.json
 create mode 100644 server.js
 create mode 100644 tests.test.js

Konstantin@DESKTOP-F60K1GH MINGW64 ~/Desktop/proj/web-site (main)
$ git push
Enumerating objects: 15, done.
Counting objects: 100% (15/15), done.
Delta compression using up to 8 threads
Compressing objects: 100% (12/12), done.
Writing objects: 100% (13/13), 51.45 KiB | 7.35 MiB/s, done.
Total 13 (delta 0), reused 0 (delta 0), pack-reused 0
To https://gitlab.com/Kostiantyn__/web-site.git
   79afca8..f7cd90e  main -> main
```

Рисунок 3.3 - Завантаження проекту в репозиторій

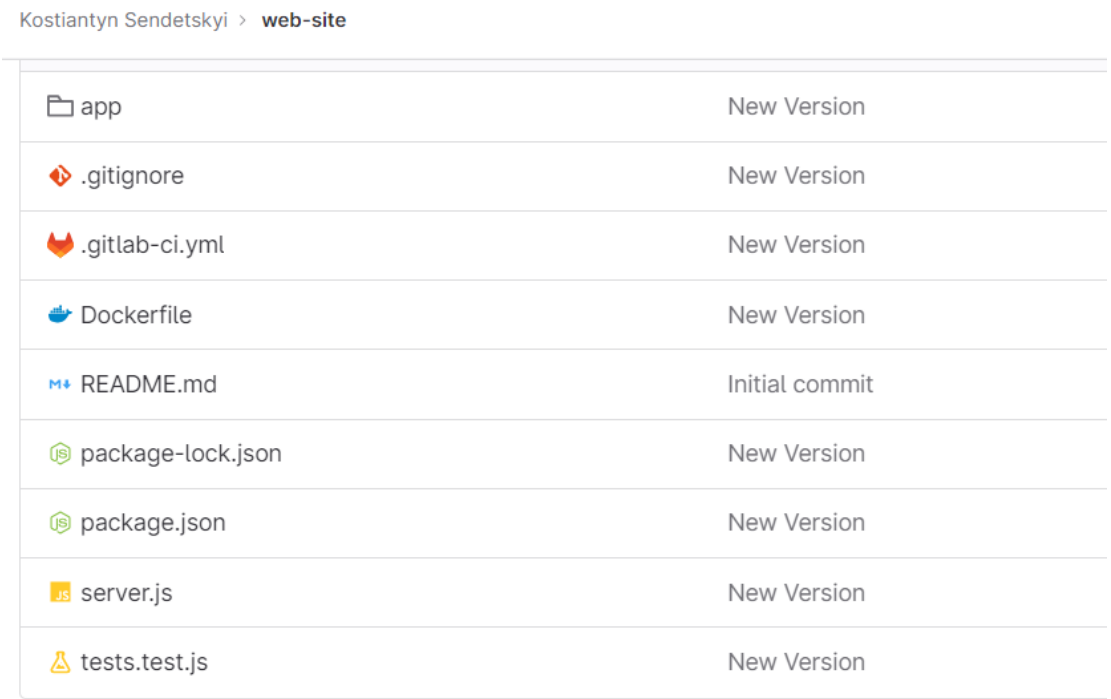


Рисунок 3.4 - Репозиторій GitLab

3.2.2 Docker

Як ми бачимо (Рисунок 3.4), в репозиторії існують такі важливі файли як: package-lock.json, package.json – які містять всі залежності нашого додатку та Dockerfile – файл збірки нашого проекту в контейнер.

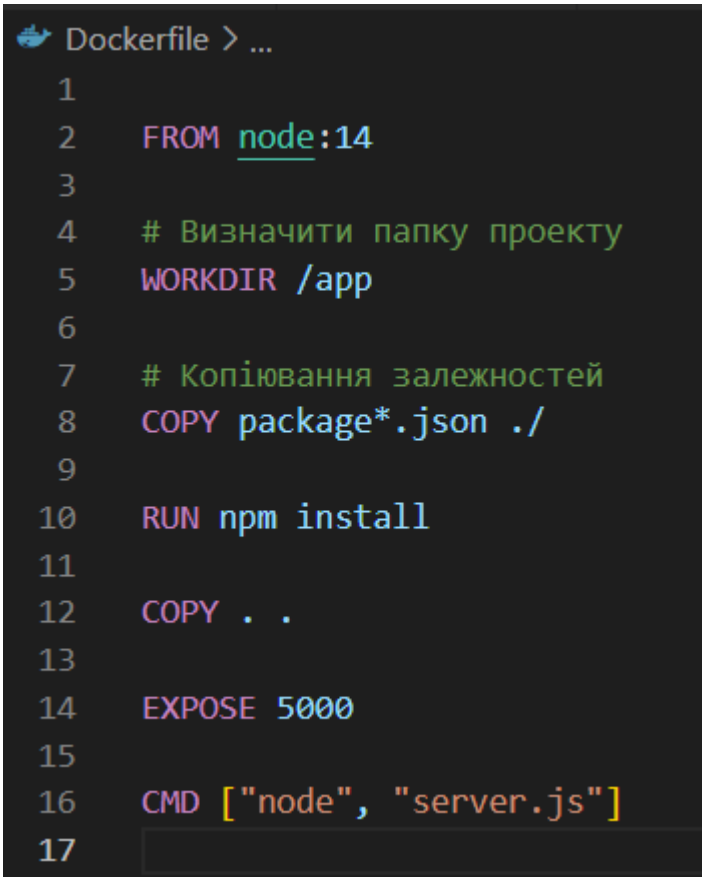


Рисунок 3.5 - Dockerfile

За допомогою Dockerfile (Рисунок 3.5) ми можемо зібрати наш додаток в єдиний контейнер. Образ збирається на основі готового образу node:14, далі копіюються файли залежностей, за допомогою яких додаток запускається, та прослуховується на порту 5000.

Наш зібраний образ повинен десь зберігатися, для цього ми будемо використовувати репозиторій образів Docker Hub.

3.2.3 Credentials

Важливою складовою налаштування інфраструктури є налаштування облікових даних. Як ми раніше зазначили зберігання даних у відкритому вигляді в коді є дуже серйозною залежністю. Тому для подальшого налаштування деяких інструментів між собою, ми будемо використовувати Змінні оточення. Зараз розглянемо на прикладі репозиторію Docker Hub, який нам знадобиться для розміщення образів нашої програми.

Треба зробити щоб облікові дані репозиторію були доступні в GitLab, щоб в подальшому наш пайплайн, який ми будемо розглядати далі, міг без проблем залогінитися в репозиторій та запушити докер образ. Ці данні це логін та пароль. Для цього ми створимо Secret Type Of Variables, які будуть доступні в нашому пайплайні.

Settings -> Ci/Cd -> variables.

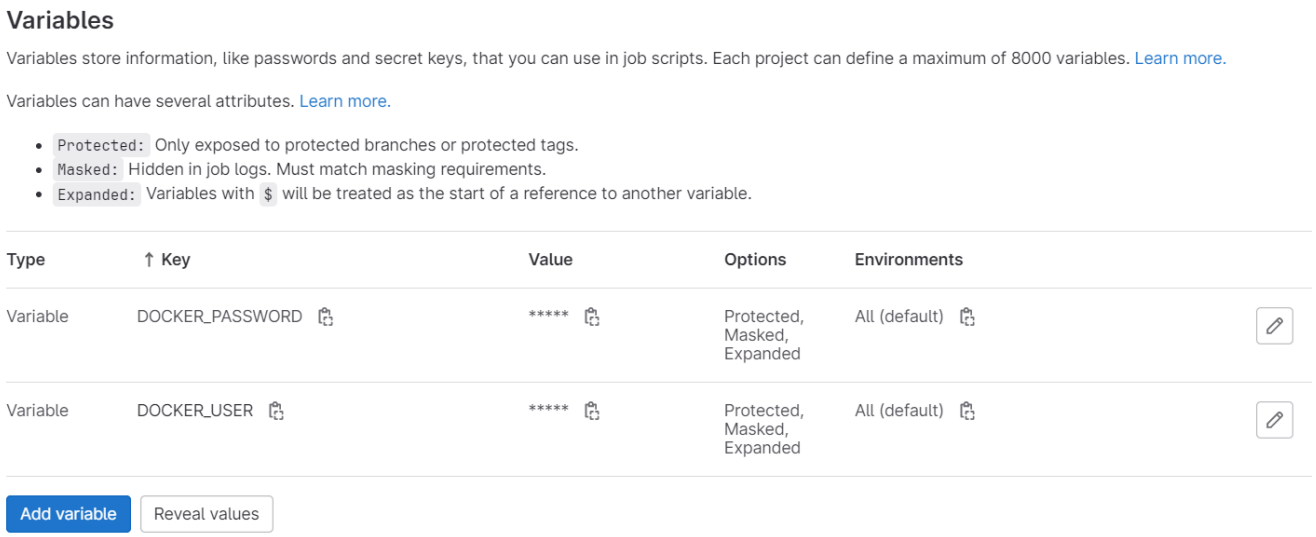


Рисунок 3.6 – Налаштовані Credentials

3.2.4 Цільове середовище

Далі ми створимо та налаштуємо сервер на якому ми будемо розгортати наш додаток. Це буде звичайний EC2 інстанс від AWS.

Для початку додамо SSH ключ для підключення gitlab до нашого серверу. Також додамо необхідні змінні для подальшого підключення.

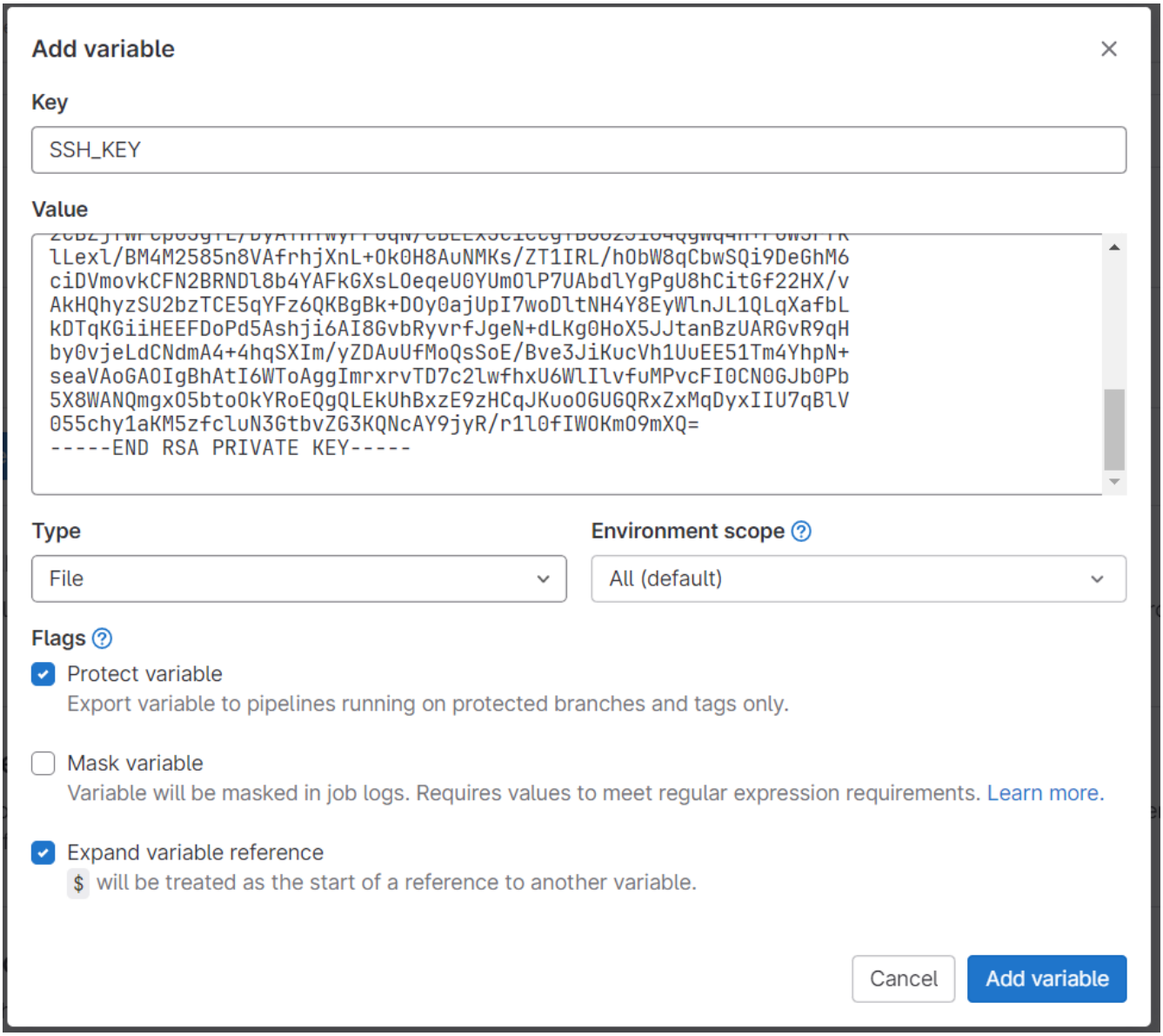


Рисунок 3.7 – Додавання ssh ключа

Type	↑ Key	Value	Options	Environments	
Variable	DOCKER_PASSWORD	*****	Protected, Masked, Expanded	All (default)	
Variable	DOCKER_USER	*****	Protected, Masked, Expanded	All (default)	
Variable	SERVER_IP	*****	Protected, Masked, Expanded	All (default)	
File	SSH_KEY	*****	Protected, Expanded	All (default)	

Рисунок 3.8 – Дані для підключення

Також попередньо налаштуємо сервер, а саме встановимо Docker.

```
ubuntu@prod:~$ docker -v
Docker version 20.10.21, build 20.10.21-0ubuntu1~22.04.3
```

Рисунок 3.9 – Встановленний Docker на сервері

Також відкриємо порт 5000 для доступу до докер контейнеру.

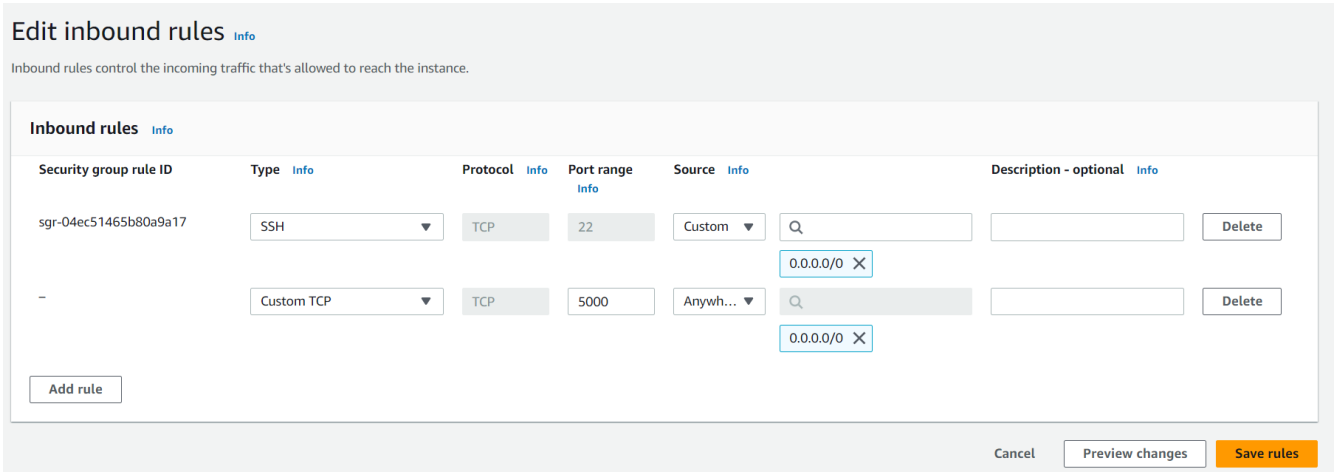


Рисунок 3.10 – Налаштування портів інстантса EC2

3.2.5 GitLab Ci

Треба налаштувати наш головний процес, який буде реалізовувати всі наші процеси збірки, тестування та деплою. Будемо використовувати інструмент GitLab Ci/Cd, а саме налаштовувати наш пайплайн в файлі .gitlab-ci.yml.

Для початку треба встановити GitLab Runner на локальну машину Ubuntu 22.04.

Використовуючи інструкцію з офіційного сайту, успішно встановили та запустили ранер.

```
ubuntu@gitlab-runner:~$ sudo gitlab-runner status
Runtime platform                                    arch=amd64 os=linux pid=1012 revision=79704081 version=16.0.1
gitlab-runner: Service is running
```

Рисунок 3.11 – Налаштування портів інстантса EC2

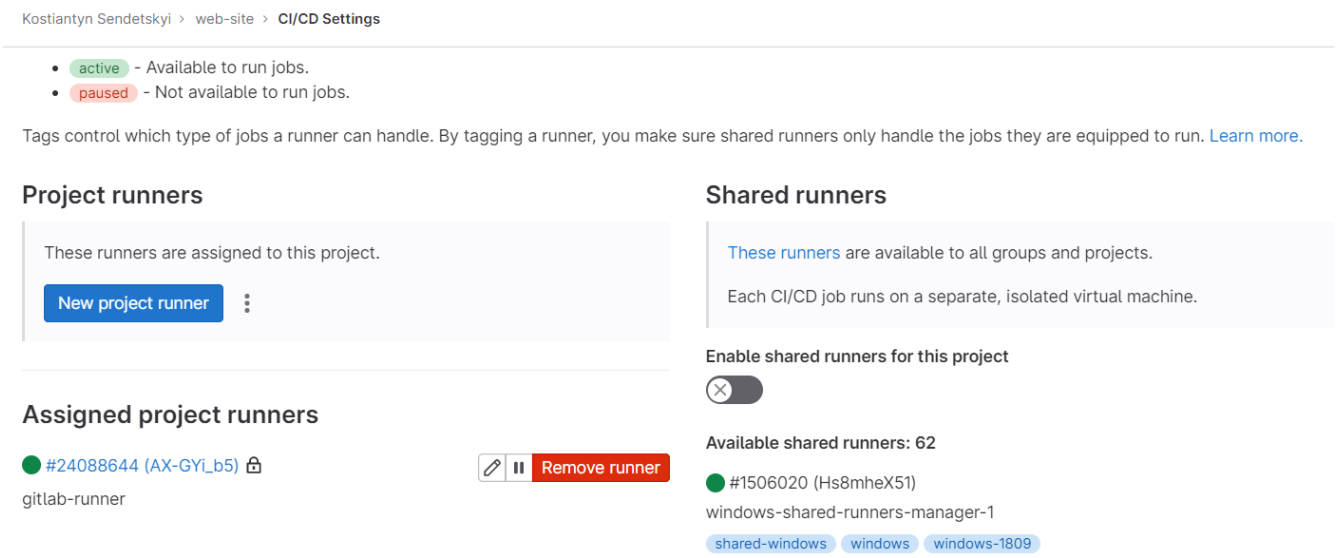


Рисунок 3.12 – Запущений GitLab runner

Тепер ми можемо створити GitLab Ci pipeline. В нашому репозиторії є файл .gitlab-ci.yml, який буде містити код нашого пайплайну.

```yaml
stages:
  - test
  - build
  - deploy

variables:
  DOCKER_IMAGE_NAME: konstanteenn/web-site
  DOCKER_IMAGE_TAG: nodejs-app-1.1

test:
  image: node:latest
  script:
    - npm install
    - npm run test

build:
  stage: build
  image: docker:latest
  services:
    - docker:dind
  variables:
    DOCKER_TLS_CERTDIR: "/certs"
  before_script:
    - docker login -u $DOCKER_USER -p $DOCKER_PASSWORD
  script:
    - docker build -t $DOCKER_IMAGE_NAME:$DOCKER_IMAGE_TAG .
    - docker push $DOCKER_IMAGE_NAME:$DOCKER_IMAGE_TAG

deploy:
  stage: deploy
  before_script:
    - chmod 400 $SSH_KEY
  script:
    - ssh -o StrictHostKeyChecking=no -i $SSH_KEY ubuntu@$SERVER_IP "
      sudo docker login -u $DOCKER_USER -p $DOCKER_PASSWORD &&
      sudo docker stop \$(sudo docker ps -aq) || true &&
      sudo docker rm \$(sudo docker ps -aq) || true &&
      sudo docker run -d -p 5000:5000 $DOCKER_IMAGE_NAME:$DOCKER_IMAGE_TAG"
```

Рисунок 3.13 – Gitlab CI/CD пайплайн

Наш пайплайн поділяється на 3 етапи:

- Test. Етап на якому наш додаток збирається в спеціальному контейнері з залежностями node за допомогою npm install та запускається тестування (у нас це файл tests.test.js, який за допомогою модуля jest та supertest проводить простий тест роботи програми. Це тести, які повертають code respond 200, якщо були успішно відправлені GET та POST запити).
- Build. Етап на якому збирається наш Docker образ на основі Dockerfile та завантажується в репозиторій Docker Hub. Етап виконується на спеціальному образі Docker з сервісом Docker:dind, який дозволяє запускати Docker-контейнер в середині іншого Docker-контейнера.
- Deploy. Етап на якому наш GitLab Runner підключається до віддаленого серверу за допомогою ssh, логіниться в DockerHub, копіює образ з нашого DockerHub та запускає Docker контейнер на нашому сервері.

Тепер при зміні будь якої ділянки коду, наш пайплайн буде автоматично запускатися. Для прикладу змінимо фон нашого веб застосунку в index.html та перевіримо роботу пайплайну. Зробимо наступні кроки на нашій локальній машині в папці проекту:

git add . > git commit -m “New version” > git push.

Запустився наш пайплайн з трьома етапами. Проходять етапи тестування, збірки та розгортання.

Рисунок 3.14 – Процес виконання пайлайну

Та через деякий час успішно завершився

Рисунок 3.15 – Виконаний пайлайн

Ми зробили деяку модифікацію додатку, він успішно пройшов всі етапи та завантажився на наш сервер, тепер ми можемо зайти на нього на порту 5000.

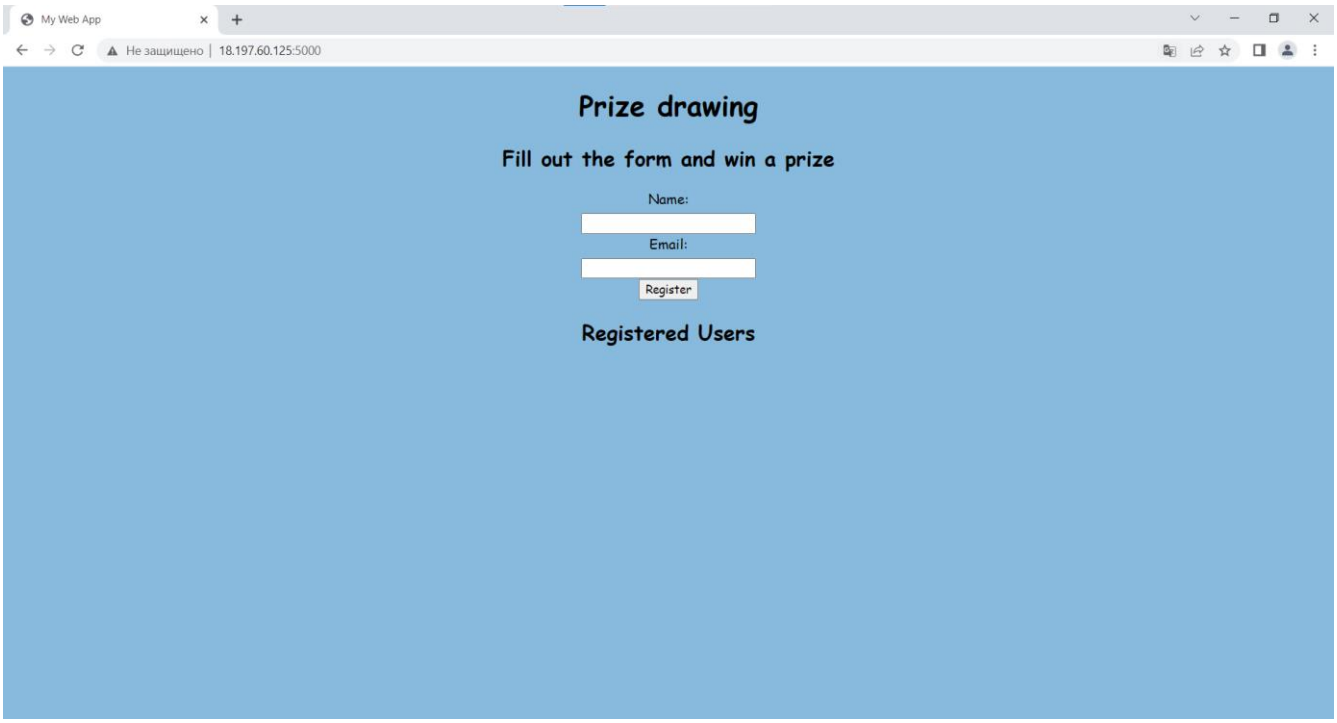


Рисунок 3.16 – Застосунок після проходження пайплайну

3.3 Інтеграція безпеки в pipeline

В розділі 2.3 ми проаналізували основні вразливості які можуть виникнути в нашому застосунку, та визначили інструменти які допоможуть нам визначити проблеми безпеки. В цьому розділі ми їх інтегруємо в наш пайплайн.

3.3.1 Статичний аналіз

Ми дослідили що наш застосунок має потенційні вразливості в коді. Щоб це перевірити, ми будемо використовувати інструменти статичного аналізу коду.

Існує велика кількість різних інструментів статичного аналізу коду. Відрізняються вони всі своїм функціоналом. Вибирати їх слід під конкретні вимоги проекту. Слід враховувати системні вимоги машини яка буде проводити тестування, на це може впливати кількість аналізаторів, розмір проекту, складність правил а також інші процеси які виконуються на сервері.

Розглянемо основні інструменти:

Таблиця 3.1 - Популярні інструменти SAST

Інструмент	Опис	Підтримувані мови
1	2	3
Checkmarx	Проводить сканування вихідного коду та виводить звіт реальному часі	Java, .NET, PHP, Python, Ruby, Swift, C/C++, Objective-C, Scala, Kotlin, JavaScript
SonarQube	Інструмент який забезпечує безперервну перевірку коду та звітування про проблеми якості коду	Більше 25 популярних мов програмування

Кінець таблиці 3.1

1	2	3
Fortify	Інструмент що аналізує вразливості та видає рекомендації щодо покращення безпеки	Java, .NET, C/C++, Python, JavaScript
Veracode	Аналізує код на відповідність відомим стандартам, генерує звіт та дає рекомендації покращення.	Більше 25 популярних мов програмування
Semgerp	Інструмент що використовує звірення за допомогою паттернів. Легко налаштовується	Більше 25 популярних мов програмування
CodeQL	Проводить сканування за допомогою спеціальної унікальної бази даних. Що дозволяє виконувати комплексний аналіз коду.	Більше 25 популярних мов програмування

Так як у нас маленький проект, ми використаємо один аналізатор зі звичайним набором правил.

Будемо використовувати semgrep, так як він підходить мові нашого проекту, його легко налаштувати, та він стабільно та якісно виявляє популярні вразливості.

Для налаштування треба створити аккаунт та новий проект [17].

Далі додано наступний фрагмент до нашого .gitlab-ci.yml.

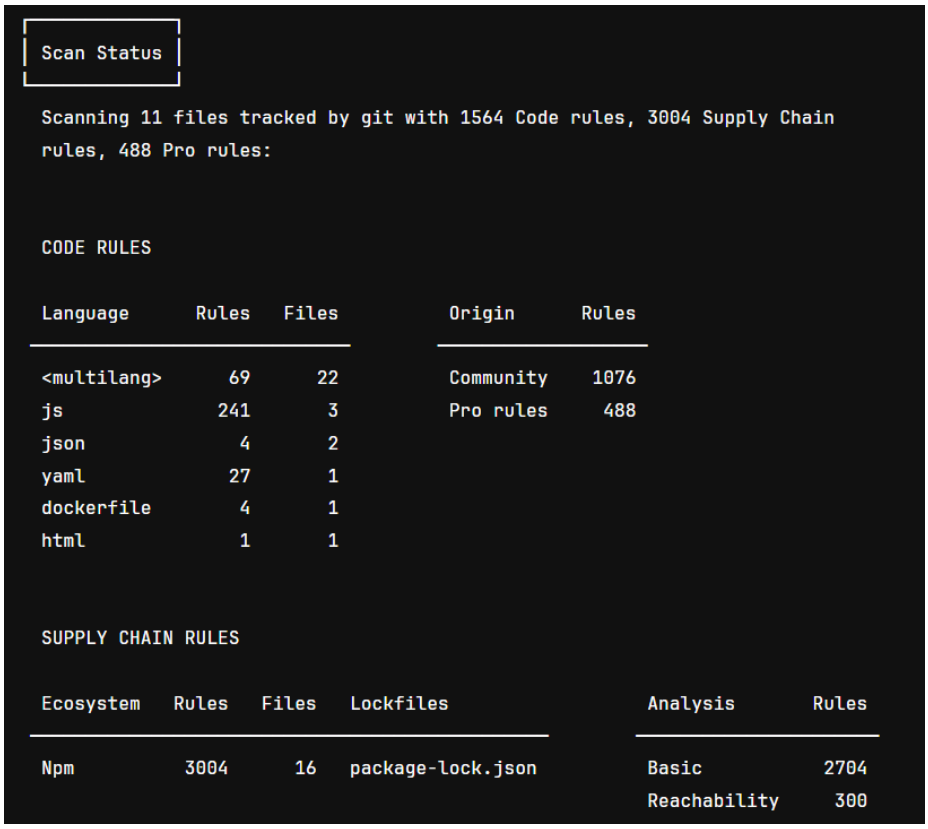
```
sast_semgrep:
  stage: security_test
  image: returntocorp/semgrep
  script:
    - semgrep --version
    - semgrep ci
  variables:
    SEMGREP_APP_TOKEN: $SEMGREP_APP_TOKEN
```

Рисунок 3.17 – Фрагмент статичного аналізу

Даний крок працює на спеціальному образі semgrep, який містить всі потрібні залежності для запуску сканера. Сканування запускається командою semgrep ci. Також використано деякі правила запуску. Сканування запускається якщо запуск був через web.

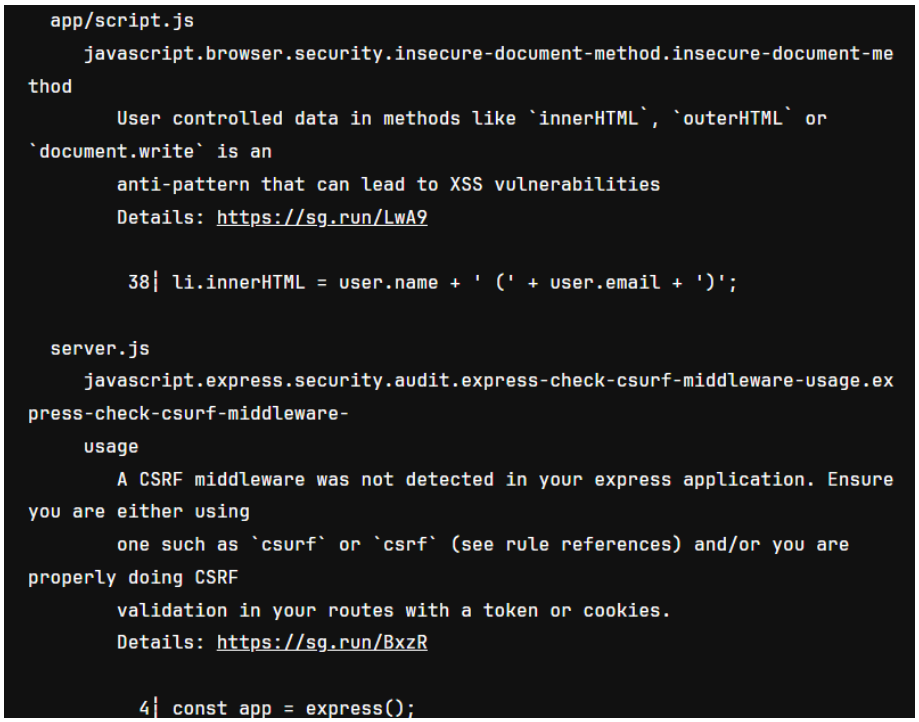
Після виконання даного сканування, ми можемо переглянути як логи в самому GitLab CI, так і на сайті semgrep.

Було виявлено файли проекту, та проаналізовано на основі правил в базі даних. Інструменти знайшов 2 потенційні вразливості.



```
┌─────────────┐
│ Scan Status │
└─────────────┘

 Scanning 11 files tracked by git with 1564 Code rules, 3004 Supply Chain
 rules, 488 Pro rules:


 CODE RULES

 Language      Rules   Files          Origin      Rules
 ──────────────────────────────       ──────────────────
 <multilang>      69      22          Community    1076
 js              241       3          Pro rules     488
 json              4       2
 yaml             27       1
 dockerfile        4       1
 html              1       1


 SUPPLY CHAIN RULES

 Ecosystem   Rules   Files   Lockfiles                Analysis       Rules
 ──────────────────────────────────────────           ──────────────────────
 Npm          3004      16   package-lock.json        Basic           2704
                                                     Reachability     300
```

Рисунок 3.18 – Статус виконання статичного аналізу



```
  app/script.js
     javascript.browser.security.insecure-document-method.insecure-document-me
thod
        User controlled data in methods like `innerHTML`, `outerHTML` or
`document.write` is an
        anti-pattern that can lead to XSS vulnerabilities
        Details: https://sg.run/LwA9

         38┆ li.innerHTML = user.name + ' (' + user.email + ')';

  server.js
     javascript.express.security.audit.express-check-csurf-middleware-usage.ex
press-check-csurf-middleware-
     usage
        A CSRF middleware was not detected in your express application. Ensure
you are either using
        one such as `csurf` or `csrf` (see rule references) and/or you are
properly doing CSRF
        validation in your routes with a token or cookies.
        Details: https://sg.run/BxzR

          4┆ const app = express();
```

Рисунок 3.19 – Виявлені загрози статичним аналізом

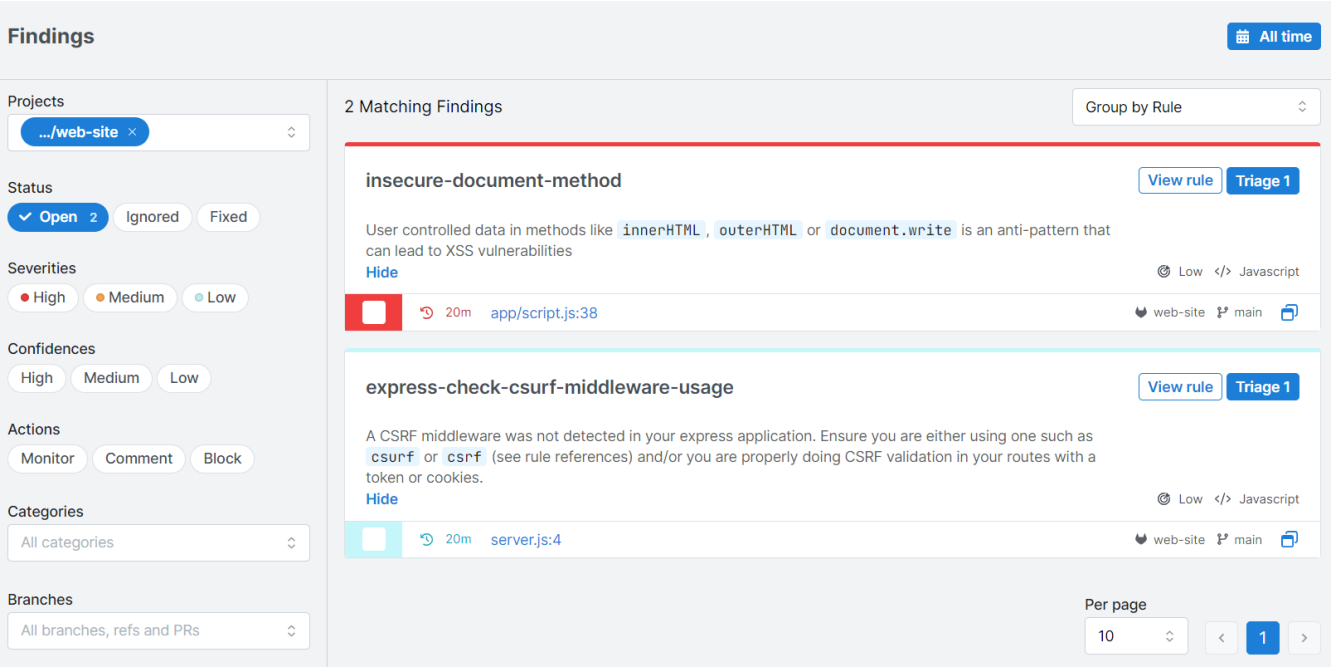


Рисунок 3.20 – Виявлені загрози в веб версії

Вразливості поділяються на деякі категорії. А саме на High, Medium, Low. В нашому випадку було знайдено одну критичну вразливість та одну потенційно небезпечну вразливість. Сканер знайшов незахищені методи які можуть спричинити до атаки XSS.

3.3.2 Динамічний аналіз

Вразливості програмного коду ми знайшли, проте цього не достатньо. Інстурменти DAST сканують застосунок шляхом взаємодії з ним, так само як користувач. Надсилаючи запити та аналізуючи отримані відповіді. Одні з основних інструментів це OWASP ZAP [18] та Burp Suite. Кожен охоплює велику кількість перевірок. В нашому випадку будемо використовувати OWASP ZAP так як його легше інтегрувати в наш пайплайн.

Для реалізації динамічного аналізу в наш пайлайн треба додати відповідний фрагмент:

```
dast_owasp_zap:
  stage: dast_and_container
  image: owasp/zap2docker-stable:latest
  allow_failure: true
  script:
    - mkdir /zap/wrk
    - /zap/zap-baseline.py -t http://$SERVER_IP:5000/ -g gen.conf -r report.html || true
  after_script:
    - cp /zap/wrk/report.html .
  artifacts:
    when: always
    expire_in: 30 days
    paths:
      - report.html
  needs: [build]
```

Рисунок 3.21 – Фрагмент динамічного аналізу

Сканування виконується на етапі dast_scan. В якості контейнеру буде використовуватися спеціальний образ від OWASP який містить сам сканер [18].

Сканування виконується за допомогою скрипта zap-baseline.py. Він сканує наш додаток який працює на порту 5000 нашого серверу, за допомогою спеціального файлу конфігурації та генерує звіт який потім можна завантажити у вигляді артефакту з GitLab Job artifacts.

Після успішного сканування, буде згенерований .html файл звіту. Проаналізувавши файл можна визначити потенційні вразливості в нашому застосунку. Про кожну вразливість виводиться додаткова інформація, щодо причини виникнення, рекомендації щодо усунення вразливості та інші.

Site: http://18.197.60.125:5000

Generated on Sun, 11 Jun 2023 12:18:57

ZAP Version: 2.12.0

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	4
Low	3
Informational	5
False Positives:	0

Alerts

Name	Risk Level	Number of Instances
Absence of Anti-CSRF Tokens	Medium	2
CSP: Wildcard Directive	Medium	2
Content Security Policy (CSP) Header Not Set	Medium	2
Missing Anti-clickjacking Header	Medium	2
Permissions Policy Header Not Set	Low	5
Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)	Low	6
X-Content-Type-Options Header Missing	Low	4
Information Disclosure - Sensitive Information in URL	Informational	1
Information Disclosure - Suspicious Comments	Informational	1
Modern Web Application	Informational	2
Storable and Cacheable Content	Informational	2
Storable but Non-Cacheable Content	Informational	4

Рисунок 3.22 – Виявлені загрози динамічним аналізом

Medium	CSP: Wildcard Directive
Description	Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks. Including (but not limited to) Cross Site Scripting (XSS), and data injection attacks. These attacks are used for everything from data theft to site defacement or distribution of malware. CSP provides a set of standard HTTP headers that allow website owners to declare approved sources of content that browsers should be allowed to load on that page — covered types are JavaScript, CSS, HTML frames, fonts, images and embeddable objects such as Java applets, ActiveX, audio and video files.
URL	http://18.197.60.125:5000/robots.txt
Method	GET
Parameter	Content-Security-Policy
Attack	
Evidence	default-src 'none'
URL	http://18.197.60.125:5000/sitemap.xml
Method	GET
Parameter	Content-Security-Policy
Attack	
Evidence	default-src 'none'
Instances	2
Solution	Ensure that your web server, application server, load balancer, etc. is properly configured to set the Content-Security-Policy header.
Reference	http://www.w3.org/TR/CSP2/ http://www.w3.org/TR/CSP/ http://caniuse.com/#search=content+security+policy http://content-security-policy.com/ https://github.com/shapesecurity/salvation https://developers.google.com/web/fundamentals/security/csp#policy_applies_to_a_wide_variety_of_resources
CWE Id	693
WASC Id	15
Plugin Id	10055

Рисунок 3.23 – Опис знайденої вразливості

Загрози поділені на різні рівні ризику. В нашому випадку було виявлено 4 помилки середнього рівня, які можуть спричинити виникненню серйозних проблем.

3.3.3 Сканування залежностей

Так як ми використовуємо сторонні залежності, нам необхідно переконатися що вони безпечні для нашого проекту.

Сканування залежностей може бути присутнім в деяких інструментах статичного аналізу. Проте в нашому випадку ми будемо використовувати спеціальний інструмент для цього сканування, який називається OWASP Dependency Check. Він сканує проект та всі його залежності на наявність вразливостей. Він використовую базу даних вразливостей NVD (National Vulnerability Database), та генерує звіт знайдених вразливостей.

Використаємо ресурс [12] [13]. Для початку треба додати наступний скрипт в файл пакетів package.json нашого проекту:

```
"scripts": {
   "owasp": "owasp-dependency-check --project \"WEB-SITE\" --scan \"package-lock.json\" --exclude \"dependency-check-bin\" --out \"owasp\" --format HTML"
  }
```

І додати відповідний фрагмент до нашого пайплайну. Який просто запускає тестування за допомогою npm, та виводить знайдені загрози.

```
owaps_dependency_check:
  stage: security_test
  extends: .parent-job
  script:
    - npm install -D owasp-dependency-check
    - npm audit || true
```

Рисунок 3.24 – Фрагмент сканування залежностей

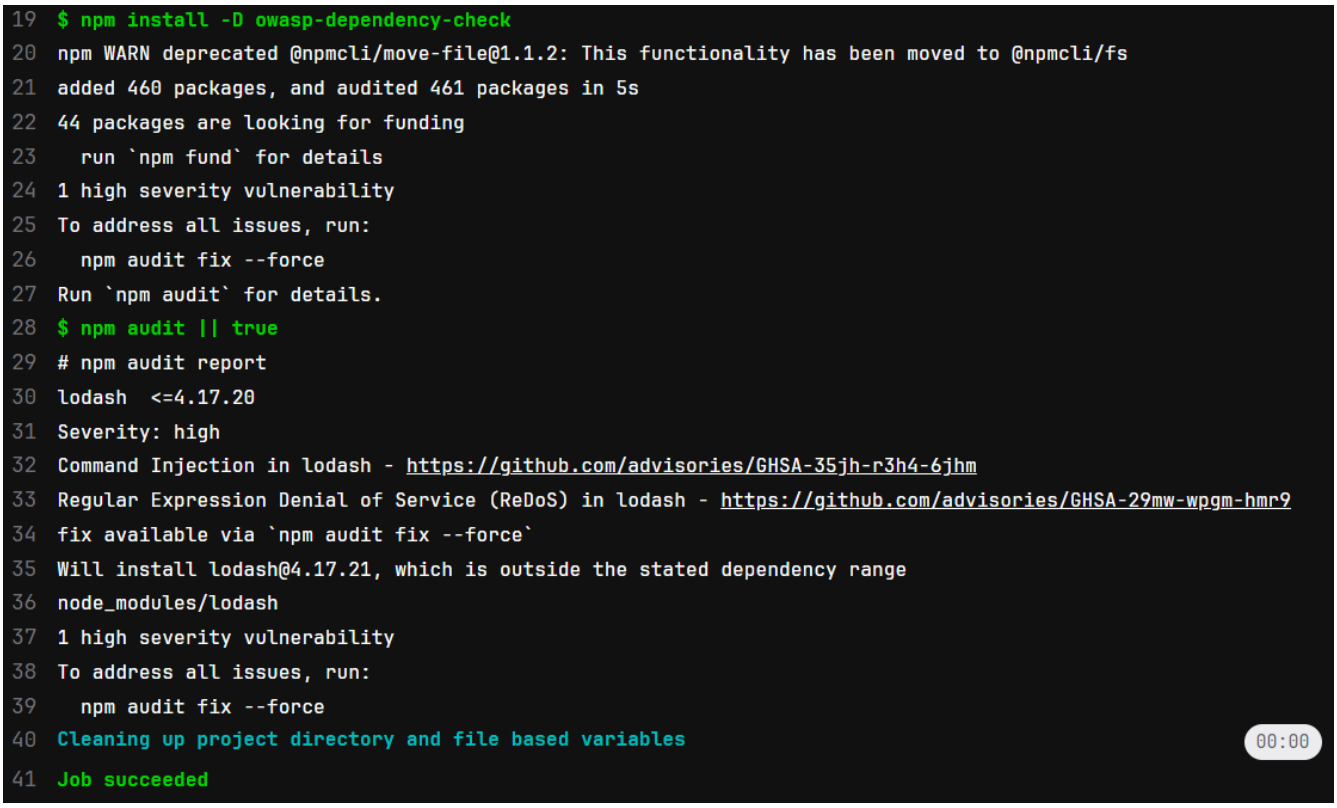


Рисунок 3.25 – Результати аналізу залежностей

Було перевірено всі доступні пакети. Серед них було виявлено одну дуже серйозну вразливість. Також було порекомендовано застосувати команду **‘npm audit fix –force’**, яка автоматично усунить вразливості в залежностях. Проте це не дуже надійно так як вона ігнорує потенційні конфлікти між іншими залежностями, це призводить до деяких проблем несумісності в проекті. Тому треба ретельно перевірити та проаналізувати результати виводу, або усунути проблему вручну.

3.3.4 Сканування контейнерів

Так як ми використовуємо технології контейнеризації, ми маємо схожу проблему як і в безпеці залежностей. В контейнерах залежності описуються в образах. Наш образ проекту також застосовує інші образи, на яких наш додаток працює. Для аналізу образів використаємо один із популярних інструментів Clair.

Clair сканує образи на наявність потенційних загроз та генерує звіт знайдених вразливостей.

```
container_scan_clair:
  stage: dast_and_container
  image: docker:latest
  services:
    - docker:dind
  before_script:
    - docker login -u $DOCKER_USER -p $DOCKER_PASSWORD
  script:
    - wget https://github.com/arminc/clair-scanner/releases/download/v12/clair-scanner_linux_amd64
    - chmod +x clair-scanner_linux_amd64
    - docker run -p 5432:5432 -d --name db arminc/clair-db:latest
    - docker run -p 6060:6060 --link db:postgres -d --name clair arminc/clair-local-scan:latest
    - docker pull $DOCKER_IMAGE_NAME:$DOCKER_IMAGE_TAG
    - docker images
    - ./clair-scanner_linux_amd64 -c http://172.17.0.3:6060 --ip $(hostname -i) $DOCKER_IMAGE_NAME:$DOCKER_IMAGE_TAG >
clair_scan_result.txt || true
  artifacts:
    when: always
    expire_in: 30 days
    paths:
      - clair_scan_result.txt
```

Рисунок 3.26 – Фрагмент сканування контейнерві

Етап сканування контейнерів виконнується на офіційному образі Docker, з сервісом dind, щоб запустити Clair сканування через контейнер. В скрипті ми завантажуємо сам сканер, та запускає потрібні Docker контейнери середовища виконання сканера та його бази даних. Завантажуємо образ нашого проекту, запускаємо скрипт сканера та зберігаємо звіт в файл.

```
180 $ ./clair-scanner_linux_amd64 -c http://172.17.0.3:6060 --ip $(hostname -i) $DOCKER_IMAGE_NAME:$DOCKER_IMAGE
    _TAG > clair_scan_result.txt || true
181 2023/06/12 16:26:33 [INFO] ▶ Start clair-scanner
182 2023/06/12 16:26:51 [INFO] ▶ Server listening on port 9279
183 2023/06/12 16:26:51 [INFO] ▶ Analyzing 4df45bd16271698ce5a31ebff6db0111358b41e56603026ffdc2273c3a3d12a1
184 2023/06/12 16:26:56 [INFO] ▶ Analyzing bb3338878a4971e9980dd90f549ed3c7e86d1d1bffa68b88504c4a6defca3863
185 2023/06/12 16:26:56 [INFO] ▶ Analyzing f0051b821797a3b7ad79e6ad524184f8486593197780a2e9c62df095a59b517f
186 2023/06/12 16:26:56 [INFO] ▶ Analyzing e960dffc7e3e6ed5e1c22090cc670580a8b0b3e0663aeb10315c955371a2142c
187 2023/06/12 16:26:57 [INFO] ▶ Analyzing dafb46cf9f8fa38444f88244ee2957e7b1016c33e5df95aeb117ed0e8759617f
188 2023/06/12 16:26:58 [INFO] ▶ Analyzing cc4fadc57f33b63ae21632b765e3e0d3a1c55f0f0d9257b64aaacc2d84b1c381
189 2023/06/12 16:26:58 [INFO] ▶ Analyzing 58c43d7af051ec4d748abe680d1b9c8b3f572123ca2bfda6817e5869b3f2ebce
190 2023/06/12 16:26:58 [INFO] ▶ Analyzing 0c01a300e8556293cd1d0c79bbcc63ca3b61dd20a9c3ee8fe7a6b76cc4422aa1
191 2023/06/12 16:26:58 [INFO] ▶ Analyzing 4cf73163b67ee1c0d1781a85fc041262fed0f4c86ed84b7bf78caa1910003ab6
192 2023/06/12 16:26:58 [INFO] ▶ Analyzing ebcbe0271c985bb03a3971a173bb11ccc74c23bd3ef6c8d28f867877955cce07
193 2023/06/12 16:26:58 [INFO] ▶ Analyzing d69164ba6ce293cc53b378d18c518f516f4c6613488ff7aedecc387cf40cb31f
194 2023/06/12 16:26:58 [INFO] ▶ Analyzing 62776fcf89b22e5d4c5150bbf2a06d7bf60b8896a6702bd9b40b6c845280e803
195 2023/06/12 16:26:59 [INFO] ▶ Analyzing 6c3130d84d449dfc6e099283ad3e16e90f93a6bf9ae10f515d175bade5341f68
196 2023/06/12 16:26:59 [WARN] ▶ Image [[MASKED]/web-site:nodejs-app-1.2] contains 565 total vulnerabilities
197 2023/06/12 16:26:59 [ERRO] ▶ Image [[MASKED]/web-site:nodejs-app-1.2] contains 565 unapproved vulnerabilitie
    s
198 Uploading artifacts for successful job
199 Uploading artifacts...
```

Рисунок 3.27– Процес скануавння контейнерві

clair_scan_result.txt - Notepad

File Edit Format View Help

STATUS	CVE SEVERITY	PACKAGE NAME	PACKAGE VERSION	CVE DESCRIPTION
[1;31mUnapproved[0m	Critical CVE-2019-19814	linux	4.19.269-1	In the Linux kernel 5.0.21, mounting a crafted f2fs filesystem image can cause __remove_dirty_segment slab-out-of-bounds write access because an array is bounded by the number of dirty types (8) but the array index can exceed this. https://security-tracker.debian.org/tracker/CVE-2019-19814
[1;31mUnapproved[0m	High CVE-2021-20246	imagemagick	8:6.9.10.23+dfsg-2.1+deb10u4	A flaw was found in ImageMagick in MagickCore/resample.c. An attacker who submits a crafted file that is processed by ImageMagick could trigger undefined behavior in the form of math division by zero. The highest threat from this vulnerability is to system availability. https://security-tracker.debian.org/tracker/CVE-2021-20246
[1;31mUnapproved[0m	High CVE-2020-26541	linux	4.19.269-1	The Linux kernel through 5.8.13 does not properly enforce the Secure Boot Forbidden Signature Database (aka dbx) protection mechanism. This affects certs/blacklist.c and certs/system_keyring.c. https://security-tracker.debian.org/tracker/CVE-2020-26541
[1;31mUnapproved[0m	High CVE-2021-3737	python3.7	3.7.3-2+deb10u4	A flaw was found in python. An improperly handled HTTP response in the HTTP client code of python may allow a remote attacker, who controls the HTTP server, to make the client script enter an infinite loop, consuming CPU time. The highest threat from this vulnerability is to system availability. https://security-tracker.debian.org/tracker/CVE-2021-3737
[1;31mUnapproved[0m	High CVE-2019-15794	linux	4.19.269-1	Overlayfs in the Linux kernel and shiftfs, a non-upstream patch to the Linux kernel included in the Ubuntu 5.0 and 5.3 kernel series, both replace vma->vm file in

Ln 18, Col 92 | 100% | Unix (LF) | UTF-8

Рисунок 3.28 – Результат сканування контейнерів

Після виконання скрипту, ми можемо побачити що було знайдено 565 потенційних вразливостей. Загрози поділяються на такі рівні: Критичні, Високі, Середні, Низькі, Незначні. В нашому випадку це одна критична, 10 високих, 45 середніх та переважна більшість незначних. Слід розуміти що знайдені загрози мають статус Unapproved (непідтверджені), це означає що вони не мають офіційних підтверджень від організацій, такі як MITRE [19] або NVD [20], які відповідають за виявлення та звітування загроз. Непідтверджені загрози можуть бути причиною загрози безпеці, тому треба визначити додаткову інформацію та перевірити в офіційних джерелах інформацію про безпеку загроз.

3.3.5 Сканування секретів

Для налаштування більшості додатків треба використовувати зміні які містять інформацію паролів, ключів він інших залежностей та інші секретні інформації.

Для аналізу нашого проекту будемо використовувати інструмент Gitleaks який виявляє конфіденційну інформацію та секрети, які випадковим чином були додані до репозиторію Git. Він проводить аналіз на основі файлу з регулярними виразами, які описує по якім правилам шукати конфіденційну інформацію.

Сканування буде працювати на етапі secret-detection. Ми будемо використовувати офіційний Docker образ сканеру. Та за допомогою виконання скрипту, який буде сканувати поточний шлях проекту на основі файлу з базовими шаблонами.

```yaml
gitleaks:
  stage: security_test
  variables:
    GIT_DEPTH: 100
  image:
    name: "zricethezav/gitleaks"
    entrypoint: [""]
  script:
    gitleaks detect -v --source=$PWD
```

Рисунок 3.29– Фрагмент сканування секретів

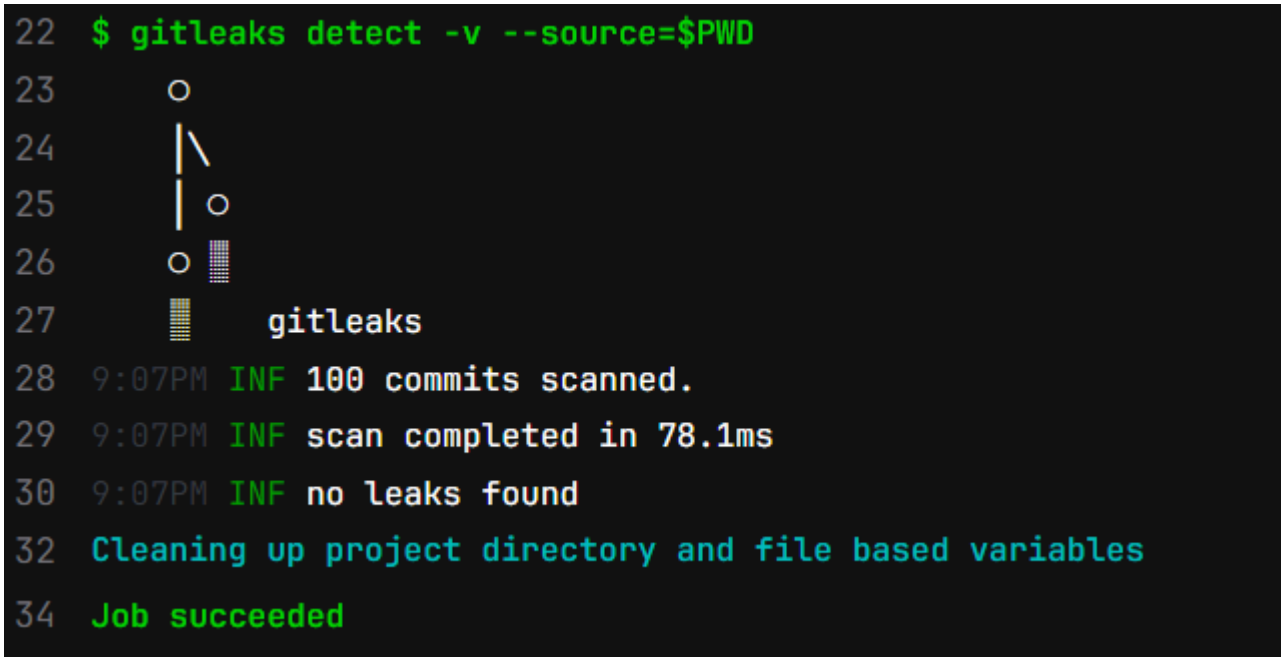


Рисунок 3.30 – Результат сканування секретів

Після вдалого виконання скрипту, було проаналізовано 20 останніх комітів, в яких не було виявлено жодних секретів, які прописані в базовомі файлі шаблонів. Всі наші секрети ми зберігли в змінних оточення в GitLab. Звісно можна самому налаштувати свій файл шаблонів та на його основі використовувати сканування.

3.4 Огляд реалізації безпечного процесу розробки в DevOps

Наш DevOps-конвеєр реалізує основі принципи DevOps для швидкої та ефективної розробки ПЗ. Розробник пише зміни в коді, далі всі шляхи від тестування до розгортання автоматизовані. Також ми впровадили автоматизовані інструменти аналізу безпеки, що дозволяє швидко аналізувати та виправляти основні проблеми нашого проекту. Детальна інформація щодо інструментів розглянута в Додатку Б.

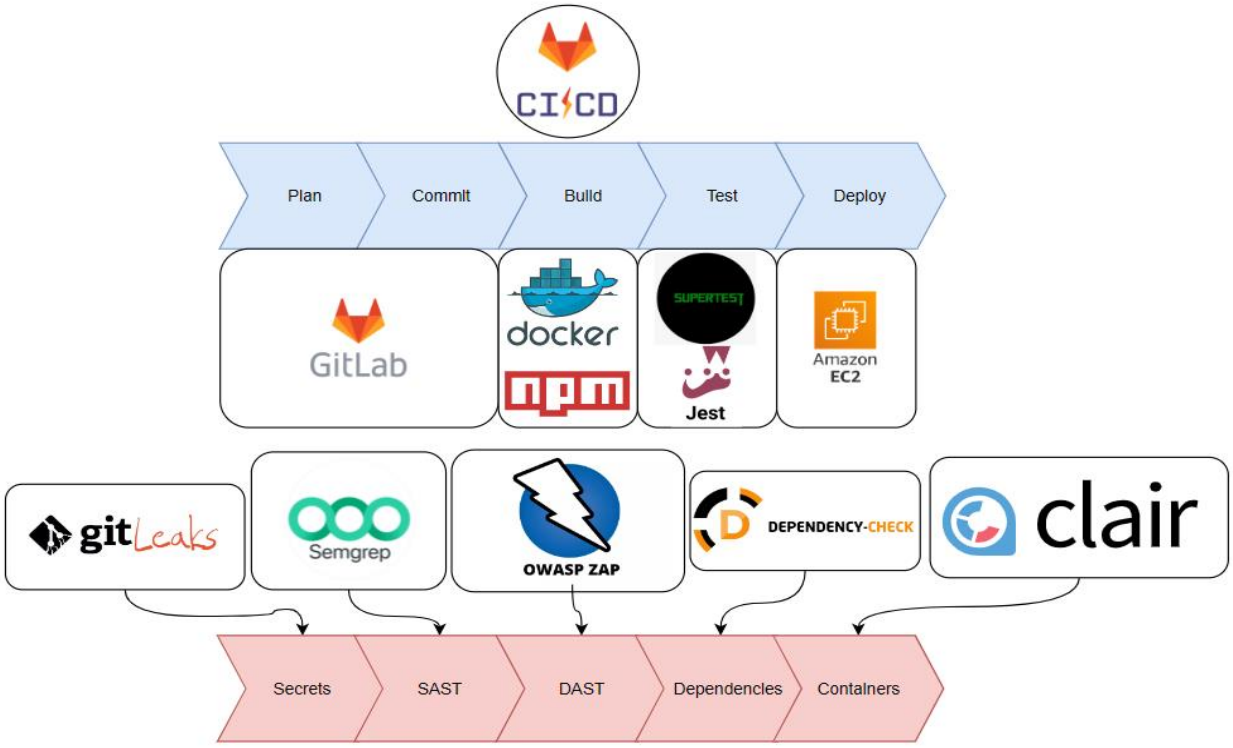


Рисунок 3.31 – Схема інструментів для побудови пайплайну

Наш реалізований пайплайн в GitLab CI виглядає наступним чином:

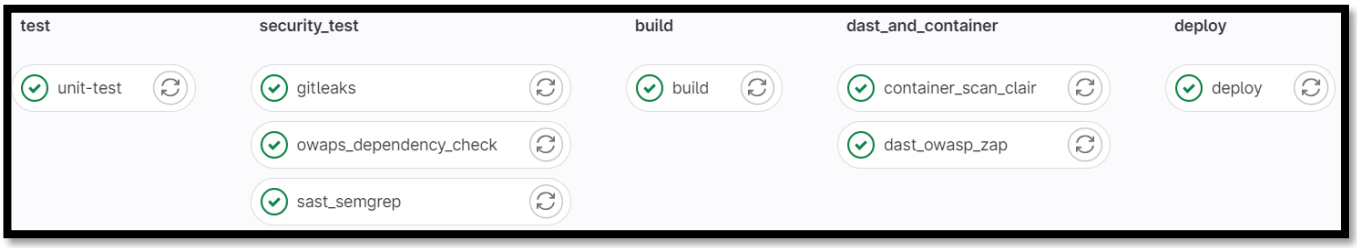


Рисунок 3.32 – Реалізований пайплайн

Ми впровадили основні практики нашої розробленої методики, а саме:

- Впровадження безпеки було реалізовано за допомогою пайплайну CI/CD.
- Всі інструменти аналізу було автоматизовано.
- Впровадили метод Shift-left, пайплайн виконується після успішного виконання попереднього етапу.
- Було задіяно практику безпечного зберігання секретів.
- Наш застосунок збирається в єдиний ізольований контейнер з усіма необхідними залежностями.
- Хмарна інфраструктура налаштована для безпечного розгортання нашого застосунку в контейнері.

Реалізація нашої методики цілком вирішує питання розробки прототипу інфраструктурної конфігурації DevOps для безпечної розробки програм. Ключовим в нашій методиці було питання безпеки саме нашого застосунку, та найважливішим завданням було автоматичне та швидке виявлення потенційних загроз для подальшої мінімізації ризиків проекту. Наш пайплайн був виконаний за 4 хвилини 37 секунд, що є чудовим результатом для швидкої розробки програм.

Також головним аспектом DevOps та DevSecOps являється постійне покращення процесів та практик для розробки ПЗ в DevOps. Технології ніколи не стоять на місці, та постійно з’являються нові реалізації, ідеї, інструменти, практики я потрібно активно впроваджувати в свої системи

Так, для покращення практик нашої методики, можна виділити деякі рекомендації:

- Використання методу Інфраструктури як код. Іноді для нашого проекту можуть знадобитися декілька інфраструктур для розгортання. Або потрібно розробити гнучке управління інфраструктурою. Для цього можна використовувати такі інструменти: Ansible, Terraform

- Використання оркестрації контейнерів. Основним інструментом є Kubernetes. Це система яка допомагає ефективно розгортати та масштабувати контейнери, що дозволяє автоматично керувати контейнерами. Це особливо корисно коли ви маєте велику кількість контейнерів.
- Впровадження інструментів моніторингу. Вони допомагають швидко реагувати на загрози в програмі та інфраструктурі. Як приклад це інструменти Grafana, Prometheus, Zabbix.
- Впровадження інструментів аналізу даних. Це інструменти які оброблюють, аналізують, візуалізують дані (журнали, logs та інші дані). Основний інструмент це ELK Stack, який включає інструменти Elasticsearch, Logstash, Kibana.

Висновки до розділу 3

В даному розділі ми практично реалізували впровадження нашої методики в процес DevOps для безпечної розробки програм. Ми визначили основні інструменти які можуть використовуватися для забезпечення безпеки в процесі розробки ПЗ. Сканування безпеки було інтегровано в пайплайн CI/CD, де аналіз проходить в автоматичному режимі. Що дозволяє ефективніше розробляти безпечне ПЗ.

Було продемонстровано шляхи впровадження інструментів та показано як вони можуть виявляти потенційні загрози. Також продемонстровані результати аналізу, які можна використовувати для покращення безпеки нашого проекту

ВИСНОВКИ

В даній дипломній роботі було проведено аналіз основних практик та методологій в DevOps. Було розглянуто основні ризики з якими можна зіткнутися під час використання даної методології для розробки ПЗ.

Аналіз основних загроз проведено за допомогою ресурсів OWASP TOP 10 та CWE 25. Із основних загроз можна виділити недостатню авторизації та аутентифікацію, використання сторонніх вразливих залежностей та проблема безпечного зберігання секретів.

Для покращення безпеки в DevOps було використано методологію DevSecOps, яка має на увазі впровадження заходів безпеки на кожен етап життєвого циклу ПЗ. На основі існуючих кращих практик ми визначили власну методику для безпечної розробки програм в DevOps, вона містить в собі такі практики:

- Інтеграція безпеки від початку до кінця.
- Безпека коду.
- Автоматизація безпеки.
- Використання методу shift-left.
- Використання контейнеризації.
- Використання хмарної інфраструктури.
- Моніторинг інфраструктури.

Для реалізації нашої методики на практиці ми використовували наступні інструменти забезпечення безпеки:

- Статичний аналіз
- Динамічний аналіз
- Сканування залежностей
- Сканування контейнерів

- Сканування секретів

В результаті практичної реалізації нашої методики, ми отримали інфраструктуру DevSecOps яка забезпечую основні вимоги швидкої та ефективної розробки, а також покращення безпеки в процесі розробки програм, в нашому випадку веб-застосунку. Ця реалізація мінімізує ризики виникнення загроз у нашому проекті.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Gene Kim. The DevOps handbook. – Boston. Massachusetts – 2018.
2. Julien Vehent. Securing DevOps: Security in the Cloud. – 2018.
3. Jeroen Mulder. Enterprise DevOps for Architects. – 2021.
4. DevOps Trends by Atlassian. [Електронний ресурс] – Режим доступу: https://www.atlassian.com/whitepapers/devops-survey-2020
5. DevSecOps Guides. [Електронний ресурс] – Режим доступу: https://devsecopsguides.com/
6. DevSecOps Controls. [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/secure/devsecops-controls
7. CWE Top 25. [Електронний ресурс] – Режим доступу: https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html
8. OWASP Top Ten. [Електронний ресурс] – Режим доступу: https://owasp.org/www-project-top-ten/
9. Microsoft SDL practices. [Електронний ресурс] – Режим доступу: https://www.microsoft.com/en-us/securityengineering/sdl/practices
10. Web application for testing. [Електронний ресурс] – Режим доступу: https://github.com/Konstanteene/web-app-sec-test
11. About DevOps. [Електронний ресурс] – Режим доступу: https://about.gitlab.com/topics/devops/
12. OWASP Dependency check in Docker. [Електронний ресурс] – Режим доступу: https://gitlab.com/gitlab-ci-utils/docker-dependency-check
13. OWASP Dependency check. [Електронний ресурс] – Режим доступу: https://www.npmjs.com/package/owasp-dependency-check

14. Clair-scanner. [Електронний ресурс] – Режим доступу: https://github.com/arminc/clair-scanner
15. Модель STRIDE. [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats
16. DevSecOps Playbook. [Електронний ресурс] – Режим доступу: https://github.com/6mile/DevSecOps-Playbook
17. Semgrep [Електронний ресурс] – Режим доступу: https://semgrep.dev/
18. OWASP ZAP [Електронний ресурс] – Режим доступу: https://hub.docker.com/r/owasp/zap2docker-stable/
19. MITRE [Електронний ресурс] – Режим доступу: https://attack.mitre.org/
20. NVD [Електронний ресурс] – Режим доступу: https://nvd.nist.gov/
21. Сендецький К.В. Аналіз інструментів DAST та SAST для покращення безпеки коду в DevSecOps. – Київ, Україна – 2023.

ДОДАТОК А КОД ПАЙПЛАЙНУ CI/CD

.gitlab-ci.yml

```
stages:
  - test
  - security_test
  - build
  - dast_and_container
  - deploy

variables:
  DOCKER_IMAGE_NAME: konstanteenn/web-site
  DOCKER_IMAGE_TAG: nodejs-app-1.2

sast_semgrep:
  stage: security_test
  image: returntocorp/semgrep
  script:
    - semgrep --version
    - semgrep ci
  variables:
    SEMGREP_APP_TOKEN: $SEMGREP_APP_TOKEN


gitleaks:
  stage: security_test
  variables:
    GIT_DEPTH: 100
  image:
    name: "zricethezav/gitleaks"
    entrypoint: [""]
  script:
    gitleaks detect -v --source=$PWD

build:
  stage: build
  image: docker:latest
  services:
    - docker:dind
  variables:
    DOCKER_TLS_CERTDIR: "/certs"
  before_script:
    - docker login -u $DOCKER_USER -p $DOCKER_PASSWORD
  script:
    - docker build -t $DOCKER_IMAGE_NAME:$DOCKER_IMAGE_TAG .
```

```
    - docker push $DOCKER_IMAGE_NAME:$DOCKER_IMAGE_TAG

container_scan_clair:
  stage: dast_and_container
  image: docker:latest
  services:
    - docker:dind
  before_script:
    - docker login -u $DOCKER_USER -p $DOCKER_PASSWORD
  script:
    - wget https://github.com/arminc/clair-scanner/releases/download/v12/clair-scanner_linux_amd64
    - chmod +x clair-scanner_linux_amd64
    - docker run -p 5432:5432 -d --name db arminc/clair-db:latest
    - docker run -p 6060:6060 --link db:postgres -d --name clair arminc/clair-local-scan:latest
    - docker pull $DOCKER_IMAGE_NAME:$DOCKER_IMAGE_TAG
    - docker images
    - ./clair-scanner_linux_amd64 -c http://172.17.0.3:6060 --ip $(hostname -i)
$DOCKER_IMAGE_NAME:$DOCKER_IMAGE_TAG > clair_scan_result.txt || true
  artifacts:
    when: always
    expire_in: 30 days
    paths:
      - clair_scan_result.txt

dast_owasp_zap:
  stage: dast_and_container
  image: owasp/zap2docker-stable:latest
  allow_failure: true
  script:
    - mkdir /zap/wrk
    - /zap/zap-baseline.py -t http://$SERVER_IP:5000/ -g gen.conf -r report.html || true
  after_script:
    - cp /zap/wrk/report.html .
  artifacts:
    when: always
    expire_in: 30 days
    paths:
      - report.html
  needs: [build]

.parent-job:
  image: node:alpine

unit-test:
  stage: test
  extends: .parent-job
```

```
  script:
    - npm install
    - npm run test

owaps_dependency_check:
  stage: security_test
  extends: .parent-job
  script:
    - npm install -D owasp-dependency-check
    - npm audit || true

deploy:
  stage: deploy
  rules:
    - if: '$CI_COMMIT_REF_NAME == "main"'
  before_script:
    - chmod 400 $SSH_KEY
  script:
    - ssh -o StrictHostKeyChecking=no -i $SSH_KEY ubuntu@$SERVER_IP "
      sudo docker login -u $DOCKER_USER -p $DOCKER_PASSWORD &&
      sudo docker stop \$(sudo docker ps -aq) || true &&
      sudo docker rm \$(sudo docker ps -aq) || true &&
      sudo docker run -d -p 5000:5000 $DOCKER_IMAGE_NAME:$DOCKER_IMAGE_TAG"
```

ДОДАТОК Б ТАБЛИЦІ РЕЗУЛЬТАТІВ ІНСТРУМЕНТІВ БЕЗПЕКИ

Тип сканування	Опис інструменту	Інстурмент сканування	Переваги
Статичний аналіз	Сканування вихідного коду на потенційні вразливості без його запуску	Semgrep	1. Широкий набір визначених правил та шаблонів для виявлення вразливостей 2. Більше 25 підтримуваних мов програмування 3. Генерація звіту вразливостей 4. Інтеграція з CI/CD системами
Динамічний аналіз	Сканування застосунку шляхом взаємодії з ним, тобто з прямим його запуском	OWASP ZAP	1. Автоматизоване сканування з можливістю налаштування та тестування веб-застосунків у реальному часі 2. Генерація звіту вразливостей 3. Інтеграція з CI/CD системами
Сканування залежностей	Сканування залежностей між компонентами ПЗ на наявність загроз	OWASP Dependency Check	1. Аналіз інформації про безпеку компонентів з публічних баз даних вразливостей (National Vulnerability Database) 2. Генерація звіту вразливостей 3. Інтеграція з CI/CD системами
Сканування контейнерів	Сканування безпеки та цілісності контейнеризованих застосунків	Clair	1. Використання баз даних вразливостей CVE та NVD. 2. Генерація звіту вразливостей 3. Інтеграція з CI/CD системами
Сканування секретів	Сканування шляхом аналізу вихідного коду та конфігураційних файлів на наявність секретів (паролі, ключі та інші)	GitLeaks	1. Підтримка різних репозиторіїв 2. Генерація звіту вразливостей 3. Інтеграція з CI/CD системами

Рисунок Б.1 – Інструменти безпеки

Тип сканування	Інстурмент сканування	Запуск програми	Формат звіту	Кількість виявлених загроз нашого застосунку	Рівень загроз	Час виконання
Статичний аналіз	Semgrep	Запуск скрипта в офіційному Docker контейнері	1. Виведення результатів в консоль 2. Посилання на сайт	2	1. High (1) 2. Medium (0) 3. Low (1)	42 секунди
Динамічний аналіз	OWASP ZAP	Запуск скрипта в офіційному Docker контейнері	Артефакт з .html файлом	12	1. High (0) 2. Medium (4) 3. Low (3) 4. Informational (5) 5. False Positive (0)	1 хвилина 53 секунди
Сканування залежностей	OWASP Dependency Check	Додати скрипт сканування в config файл застосунка та запустити його окремою командою	Виведення результатів в консоль	1	1. High (1) 2. Medium (0) 3. Low (0) 4. Informational (0)	19 секунд
Сканування контейнерів	Clair	Shell скрипт який працює з Docker контейнером Clair та його бази даних	Виведення результатів в консоль, які можна завантажити як артефакт	565	1. Unapproved (565) a) Critical (1) b) High (15) c) Medium (45) d) Negligible (504)	1 хвилина 37 секунд
Сканування секретів	GitLeaks	Запуск скрипта в офіційному Docker контейнері	Виведення результатів в консоль	0	Немає рівня загроз. Видає місцезнаходження виникненої проблеми.	9 секунд

Рисунок Б.2 – Результати використання