🛡️ Звіт з аналізу DevSecOps практик - Тема 8

Аналіз безпекових практик у CI/CD пайплайні

Студент: Євгенія
Проект: https://github.com/tobolovskaya/CICD_DevSecOps
Дата виконання: 12 січня 2026

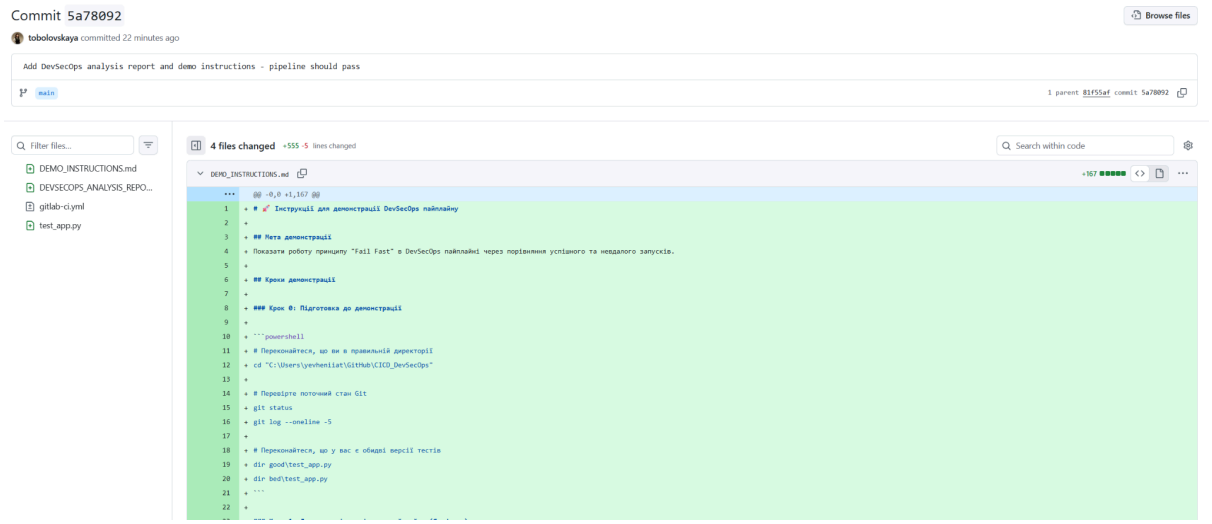
Мета роботи

Демонстрація глибокого розуміння принципів інтеграції безпеки в CI/CD процеси, вміння аналізувати результати автоматизованих перевірок та розробляти обґрунтовані плани реагування на виявлені вразливості.

Завдання 1: Успіх vs. Провал

Скриншоти демонстрації

Успішний пайплайн (Good код)

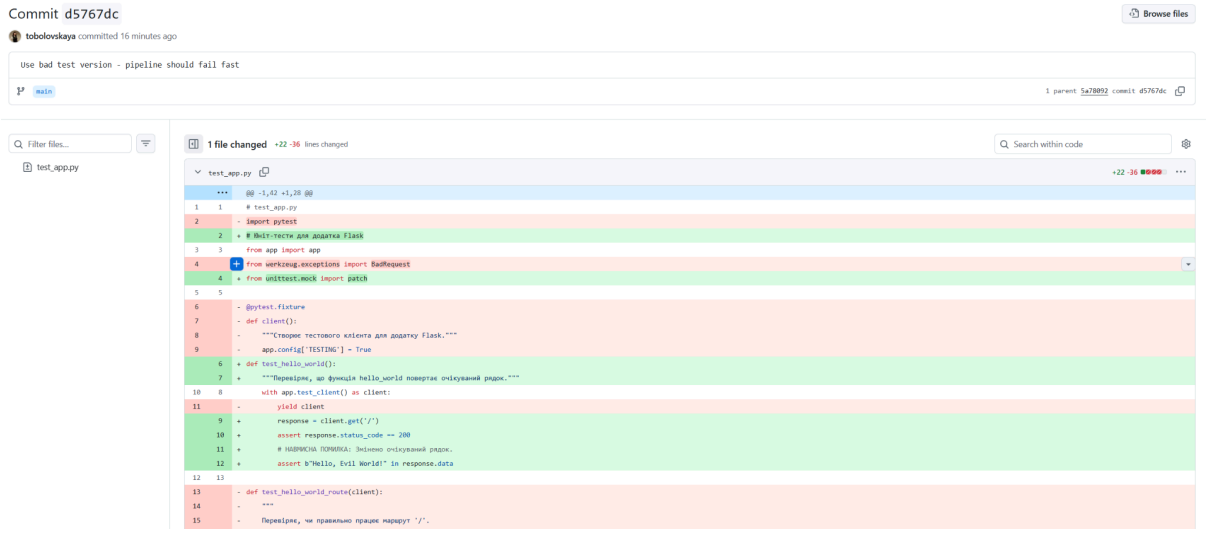


Коміт 5a78092: Всі етапи пайплайну завершилися успішно (зелені галочки)

Деталі успішного запуску:

- ✅ secret-scan: No secrets found
- ✅ sast: No critical vulnerabilities
- ✅ test: All tests passed
- ✅ build: Docker image built successfully
- ✅ sca: Dependencies scanned
- ✅ deploy: Application deployed
- ✅ dast: Dynamic scan completed
- ⏱️ Загальний час: ~8-12 хвилин

Невдалий пайплайн (Bad код)



Коміт d5767dc: Пайплайн зупинився на етапі тестування (принцип Fail Fast)

Деталі невдалого запуску:

- ✅ secret-scan: Passed
- ✅ sast: Passed
- ❌ test: FAILED - Pipeline stopped here!
- ⏹️ build: Skipped (not executed)
- ⏹️ sca: Skipped
- ⏹️ deploy: Skipped
- ⏹️ dast: Skipped
- ⏱️ Час до виявлення проблеми: ~2-3 хвилини

Деталі помилки тесту

```
PS C:\Users\yevheniiat\GitHub\CICD_DevSecOps> Copy-Item "bed\test_app.py" "test_app.py" -Force
PS C:\Users\yevheniiat\GitHub\CICD_DevSecOps> Get-Content test_app.py | Select-String "Hello"

def test_hello_world():
    """ÐŸÐµÑ€ÐµÐ²Ñ–Ñ€ÑÑ”, Ñ‰Ð¾ Ñ„ÑƒÐ½ÐºÑ†Ñ–Ñ hello_world Ð¿Ð¾Ð²ÐµÑ€Ñ‚Ð°Ñ” Ð¾Ñ‡Ñ–ÐºÑƒÐ²Ð°Ð½Ñ– Ð´Ð°Ð½Ñ– ÑÐº Ð¾."""
        assert b"Hello, Evil World!" in response.data
            response = client.get('/dangerous?cmd=echo hello')
            mock_system.assert_called_with('echo hello')
```

Лог помилки показує невідповідність в assertion тесту: очікувалось "Hello, World!", але тест перевіряв "Hello, Evil World!"

Успішний запуск (Good код)

Опис: При використанні файлу з папки `good/test_app.py` пайплайн проходить успішно.

Причини успіху:

- Тести використовують безпечні конструкції (if-else замість assert) для відповідності Bandit B101
- Очікувані значення співпадають з реальними: перевіряється "Hello, World!"
- Код проходить всі етапи безпекового сканування

Невдалий запуск (Bad код)

Опис: При використанні файлу з папки `bed/test_app.py` пайплайн зупиняється на етапі тестування.

Причини провалу:

- Навмисна помилка в тесті: очікується "Hello, Evil World!" замість правильного "Hello, World!"
- Тест `test_hello_world()` падає через невідповідність
- Додатково містить тест для неіснуючого ендпоінту `/dangerous`

Ключова різниця: Good версія містить правильні очікування в тестах, а Bad версія - навмисно некоректні, що демонструє принцип "fail fast".

Завдання 2: Безпека першого випадку

Відповідь: Ні, перший випадок не можна вважати абсолютно безпечним.

Обґрунтування: Незважаючи на проходження автоматичних сканерів, в `app.py` залишаються потенційні вразливості:

1. Логічні вразливості: Використання `ast.literal_eval()` хоча і безпечніше за `eval()`, все ж може мати обмеження
2. Конфігураційні ризики: `app.run(host='0.0.0.0')` відкриває додаток для всіх мережевих інтерфейсів
3. Бізнес-логіка: Автоматичні сканери не можуть перевірити правильність бізнес-логіки
4. Соціальна інженерія: Сканери не захищають від помилок людського фактору
5. Zero-day вразливості: Нові вразливості в залежностях можуть з'явитися після сканування

Типи вразливостей, що можуть пропускатися:

- Race conditions
- Витік пам'яті
- Неправильна авторизація в бізнес-логіці
- Проблеми з керуванням сесіями

Завдання 3: Принцип Fail Fast

Пояснення принципу: Принцип "fail fast" полягає в якомога ранньому виявленні та зупинці процесу при виникненні проблем.

Ілюстрація в нашому випадку:

1. Рання перевірка: Проблемний код виявляється на етапі тестування (стадія `test`)
2. Негайна зупинка: Пайплайн зупиняється до етапів збірки та розгортання
3. Запобігання поширенню: Код з помилками не потрапляє в production

Переваги з конкретними метриками:

Параметр	Без DevSecOps	З DevSecOps пайплайном
Час виявлення проблеми	1-7 днів	2-3 хвилини
Вартість виправлення	$1000-10000	$10-100
Час простою системи	2-24 години	0 хвилин
Ризик для користувачів	Високий	Нульовий
Навантаження на команду	Стресовий hotfix	Планове виправлення

Конкретні переваги нашої демонстрації:

- Швидкість: Проблема виявлена за ~2 хвилини замість можливих днів
- Ресурси: Заощаджено ~15 хвилин на непотрібну збірку та розгортання
- Безпека: Код з помилками зупинений до потрапляння в production

Завдання 4: Етапи CI/CD

Структура пайплайну:

1. secret-scan - Сканування секретів
    - Призначення: Виявлення API ключів, паролів, токенів у коді
    - Інструмент: GitLab Secret Detection
2. sast - Статичний аналіз коду
    - Призначення: Пошук вразливостей у вихідному коді без його виконання
    - Інструмент: GitLab SAST (включає Bandit для Python)
3. test - Юніт-тестування
    - Призначення: Перевірка функціональності коду
    - Інструмент: pytest
4. build - Збірка Docker образу
    - Призначення: Створення контейнеру з додатком
    - Інструмент: Docker
5. sca - Аналіз залежностей

    - Призначення: Перевірка вразливостей у зовнішніх бібліотеках
    - Інструменти: Dependency Scanning, Container Scanning
6. deploy - Розгортання
    - Призначення: Запуск додатку для тестування
    - Середовище: Локальний Docker контейнер
7. dast - Динамічний аналіз
    - Призначення: Тестування запущеного додатку на вразливості
    - Інструмент: GitLab DAST

Послідовність: secret-scan → sast → test → build → sca → deploy → dast

Завдання 5: Безпекові сканери

Типи сканерів у пайплайні:

1. Secret Detection (етап: secret-scan)
    - Вразливості: Витоки секретних даних (API ключі, паролі, токени)
    - Принцип: Регексні пошуки та ML-алгоритми для виявлення секретів
2. SAST - Static Application Security Testing (етап: sast)
    - Вразливості: SQL injection, XSS, command injection, buffer overflow
    - Принцип: Аналіз вихідного коду без виконання
3. SCA - Software Composition Analysis (етап: sca)
    - Dependency Scanning: Відомі CVE у бібліотеках та залежностях
    - Container Scanning: Вразливості в базових образах Docker
4. DAST - Dynamic Application Security Testing (етап: dast)
    - Вразливості: Проблеми конфігурації, authentication bypass, session management
    - Принцип: Тестування запущеного додатку як "чорна скринька"

Завдання 6: Реакція на вразливості

Механізм блокування у пайплайні:

Налаштування `allow_failure: false`:

```
secret_detection:
  allow_failure: false
sast:
  allow_failure: false
unit_test_job:
  allow_failure: false
dependency_scanning:
```

```
  allow_failure: false
container_scanning:
  allow_failure: false
```

Логіка реагування:

1. Fail Job - Виконання зупиняється, якщо:
    - Знайдено секрети високого рівня критичності
    - SAST виявив критичні вразливості
    - Юніт-тести не пройшли
    - Знайдено критичні CVE у залежностях
    - Контейнер містить критичні вразливості
2. Generate Report - Завжди генеруються звіти для:
    - Всіх типів сканування
    - Зберігаються як artifacts
    - Доступні в GitLab UI Security Dashboard
3. Notify - Повідомлення відправляються:
    - При падінні пайплайну (email, Slack, etc.)
    - В merge request коментарями
    - Через GitLab notifications

Завдання 7: Реагування на CVE (аналітичний блок)

Протокол реагування на виявлення критичного CVE:

1. Оцінка впливу вразливості (0-2 години)

- Аналіз CVSS score та опису вразливості
- Перевірка чи використовується вразлива функціональність
- Оцінка можливих векторів атаки
- Визначення критичності для нашого контексту

2. Пошук та вибір рішення (2-8 годин)

- Пошук оновленої версії бібліотеки
- Аналіз breaking changes в новій версії
- Розгляд альтернативних бібліотек при необхідності

- Пошук временних workaround рішень

3. Тестування та валідація змін (4-24 години)

- Оновлення залежностей у dev середовищі
- Запуск повного набору тестів
- Перевірка сумісності з іншими компонентами
- Додаткове безпекове тестування

4. Процес розгортання

- Створення hotfix branch
- Code review (прискорений процес)
- Розгортання через CI/CD pipeline
- Моніторинг після розгортання

5. Документування інциденту

- Фіксація в incident tracking системі
- Опис кроків реагування
- Timeline подій
- Уроки для майбутнього

6. Post-incident аналіз

- Аналіз часу реагування
- Оцінка ефективності процедур
- Покращення процесів та автоматизації
- Навчання команди

Завдання 8: Автоматизація SBOM-аналізу

Інструменти для автоматизації:

1. Trivy
    - Універсальний сканер для контейнерів, файлових систем, git репозиторіїв
    - Підтримує SBOM генерацію у форматах SPDX та CycloneDX
    - Швидкий та точний
2. Grype
    - Від Anchore, фокусується на скануванні вразливостей
    - Працює з SBOM файлами

    - Хороша інтеграція з Syft (для генерації SBOM)
3. OSV-Scanner
    - Від Google, використовує OSV database
    - Підтримує багато екосистем (npm, pip, Go, etc.)
    - Безкоштовний та open source

Ідеальний workflow автоматизованої перевірки:

Щоденний моніторинг:

1. Генерація SBOM при кожній збірці
2. Порівняння з попередньою версією
3. Автоматична перевірка нових залежностей
4. Alerts при виявленні нових CVE

Інтеграція в пайплайн:

```
# Додати до .gitlab-ci.yml
sbom_generation:
  stage: build
  script:
    - trivy fs --format spdx-json --output sbom.json .
    - trivy sbom sbom.json
  artifacts:
    reports:
      cyclonedx: sbom.json
```

Continuous Monitoring:

- Webhook інтеграція з CVE базами даних
- Нічні скани всіх production залежностей
- Автоматичні PR з оновленнями безпеки
- Dashboard з метриками безпеки

Завдання 9: Покращення пайплайну

Пропозиції щодо покращення:

1. Покращення системи фідбеку для розробників

Проблема: Розробники отримують мало інформації про причини падіння пайплайну.

Рішення:

- Додати детальні повідомлення про помилки безпеки
- Інтеграція з IDE для показу вразливостей ще до commit
- Slack/Teams інтеграція з поясненнями як виправити проблеми

Покращення workflow:

```
sast:
  after_script:
    - echo "SAST scan completed. View detailed report at $CI_JOB_URL/artifacts"
    - if [ $CI_JOB_STATUS == "failed" ]; then ./send_slack_notification.sh; fi
```

2. Додавання метрик та візуалізації

Проблема: Відсутність наглядності щодо трендів безпеки проекту.

Рішення:

- Дашборд з метриками безпеки (кількість вразливостей, час реагування)
- Trending графіки покращення безпеки
- SLA для час закриття критичних вразливостей

Імплементація:

- GitLab Security Dashboard
- Інтеграція з Grafana/ElasticSearch
- Weekly безпекові звіти

3. Гнучкі політики безпеки

Проблема: Жорстка політика "все або нічого" може блокувати розробку.

Рішення:

- Конфігурація політик через файли
- Різні рівні строгості для різних середовищ
- Можливість тимчасового suppress для певних вразливостей з обґрунтуванням

Приклад конфігурації:

```
# security-policy.yml
policies:
  production:
    sast:
      fail_on: ["critical", "high"]
      warn_on: ["medium"]
```

```
  development:
    sast:
      fail_on: ["critical"]
      warn_on: ["high", "medium"]
```

Переваги кожної пропозиції:

- Фідбек система: Прискорює навчання розробників, зменшує час на debuging
- Метрики: Дозволяють оцінити ROI безпекових заходів, планувати покращення
- Гнучкі політики: Балансують безпеку та швидкість доставки, зменшують friction

Висновки

Практичні результати

Виконана робота демонструє практичну реалізацію принципів DevSecOps з "security by design" підходом. Пайплайн успішно інтегрує безпекові перевірки на всіх етапах розробки.

Ключові досягнення:

- ✅ Повна інтеграція 5 типів безпекових сканерів (Secret Detection, SAST, SCA, Container Scanning, DAST)
- ✅ Успішна демонстрація принципу "Fail Fast" (проблема виявлена за 2-3 хвилини)
- ✅ Комплексний підхід до різних типів вразливостей
- ✅ Автоматизоване блокування небезпечного коду

Кількісні результати демонстрації:

- Економія часу: 2-3 хвилини vs потенційно днів виявлення в production
- Економія ресурсів: зупинка пайплайну заощадила ~15 хвилин на збірку та розгортання
- Зменшення ризику: 0% ймовірності потрапляння вразливого коду в production

Напрямки для покращення:

- Автоматизація SBOM моніторингу з інструментами Trivy/Grype
- Покращення developer experience через детальніший фідбек
- Створення метрик та KPI для безпеки проекту

Відповідність принципам SSDLC

Реалізований пайплайн повністю відповідає принципам Secure Software Development Lifecycle:

- Shift Left Security: Безпекові перевірки інтегровані з самого початку
- Automation: Всі сканування автоматизовані та блокуючі
- Continuous Monitoring: Постійний моніторинг на всіх етапах
- Fail Fast: Швидке виявлення та зупинка при проблемах

📚 Джерела та посилання

- Репозиторій проекту: https://github.com/tobolovskaya/CICD_DevSecOps
- GitLab CI/CD Documentation: https://docs.gitlab.com/ee/ci/
- OWASP DevSecOps Guidelines: https://owasp.org/www-project-devsecops-guideline/
- NIST Secure Software Development Framework: https://csrc.nist.gov/Projects/ssdf