Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних
технологій та комп'ютерної інженерії
Кафедра захисту інформації

Комплексна бакалаврська дипломна робота на тему:
«Система тестування безпеки програмного застосунку. Частина 1. Підсистема статичного тестування»

Виконав: студент 4 курсу групи 1БС-20б
спеціальності 125 Кібербезпека
_____Олександр РОГОЗИНСЬКИЙ

Керівник: к. т. н., доцент каф. ЗІ
_____ Віталій ЛУКІЧОВ
«14» червня 2024 р.
Рецензент: к. т. н. доцент каф. ПЗ
_____ Вікторія ВОЙТКО
«14» червня 2024 р.

Допущено до захисту
Завідувач кафедри ЗІ
д. т. н., проф.
_____ Володимир ЛУЖЕЦЬКИЙ
«14» червня 2024 р.

Вінниця ВНТУ – 2024 року

Міністерство освіти і науки України
Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра захисту інформації
Рівень вищої освіти І (бакалаврський)
Галузь знань – 12 Інформаційні технології
Спеціальність – 125 Кібербезпека
Освітньо-професійна програма – Безпека інформаційних і комунікаційних систем

ЗАТВЕРДЖУЮ
Завідувач кафедри ЗІ,
д. т. н., проф.
Володимир ЛУЖЕЦЬКИЙ
«14» березня 2024 року

З А В Д А Н Н Я
НА КОМПЛЕКСНУ БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ
СТУДЕНТУ
Олександру РОГОЗИНСЬКОМУ

1. Тема роботи: «Система тестування безпеки програмного застосунку. Частина 1. Підсистема статичного тестування»

   керівник роботи: Віталій ЛУКІЧОВ, к. т. н., доцент кафедри ЗІ, затверджені наказом ректора ВНТУ від 11 березня 2024 року №80.

2. Строк подання студентом роботи 14 червня 2024 р.
3. Вихідні дані до роботи:
   - Галузь застосування: Тестування безпеки веб-додатків.
   - Метод тестування: Статичне тестування безпеки (SAST).
   - Використовувані інструменти: SonarQube, Semgrep, Horusec, AWS, OWASP Benchmark.
4. Зміст текстової частини: Вступ. 1. Аналіз функцій та інструментів статичного тестування безпеки програмних застосунків. 2. Проектування підсистеми статичного тестування. 3. Реалізація підсистеми статичного тестування. Висновки. Список використаних джерел. Додатки.
5. Перелік ілюстративного матеріалу: аналіз інструментів SAST (плакат А4), схема системи тестування безпеки (плакат А4), процеси підсистеми статичного тестування безпеки (плакат А4), схема інфраструктури (плакат

А4), схема шаблону CloudFormation (плакат А4), кількість виявлених вразливостей (плакат А4).

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
1	В. ЛУКІЧОВ, к.т.н., доц. каф.ЗІ	11.03.24	31.03.24
2	В. ЛУКІЧОВ, к.т.н., доц. каф.ЗІ	11.03.24	16.04.24
3	В. ЛУКІЧОВ, к.т.н., доц. каф.ЗІ	11.03.24	04.05.24

7. Дата видачі завдання 14 березня 2024 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської дипломної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз завдання. Вступ	12.03.24 – 14.03.24	
2	Аналіз літературних джерел за напрямком бакалаврської дипломної роботи	15.03.24 – 31.03.24	
3	Розробка рішень, моделей, алгоритмів	01.04.24 – 15.04.24	
4	Практична реалізація, моделювання, експериментування, результати	16.04.24 – 01.05.24	
5	Аналіз виконання ПЗ, висновки	02.05.24 – 10.05.24	
6	Оформлення пояснювальної записки	30.05.24 – 07.06.24	
7	Попередній захист БДР	10.06.24 – 12.06.24	
8	Виправлення зауважень, підготовка ілюстративного матеріалу	12.06.24 – 13.06.24	
9	Представлення БДР до захисту, рецензування	13.06.24 – 17.06.24	
10	Захист БДР	18.06.24 – 21.06.24	

Студент _____ Олександр РОГОЗИНСЬКИЙ
(підпис)

Керівник комплексної бакалаврської

дипломної роботи _____ Віталій ЛУКІЧОВ
(підпис)

АНОТАЦІЯ

Комплексна бакалаврська дипломна робота складається з 52 сторінок формату А4, на яких є 10 рисунків, 6 таблиць, список використаних джерел містить 16 найменувань. Комплексна бакалаврська дипломна робота присвячена дослідженню інструментів статичного тестування безпеки програмного забезпечення (SAST) та розробці підсистеми SAST. У результаті аналізу було з’ясовано, що сучасні інструменти статичного тестування можуть виявляти широкий спектр вразливостей у програмному забезпеченні, однак кожен з них має свої особливості та обмеження. Було запропоновано підхід до вибору оптимального інструменту, зокрема врахування таких факторів, як підтримувані мови програмування, інтеграція з CI/CD, точність та час роботи.

Проведено експериментальне тестування з використанням декількох популярних інструментів SAST, таких як Semgrep, SonarQube та Horusec, результати якого показали їх ефективність.

Результати експерименту показали, що Semgrep має найкращі показники виявлення вразливостей, SonarQube відзначається низьким рівнем хибних спрацьовувань, а Horusec має найнижчу загальну ефективність.

Вибрано інструмент Semgrep для реалізації підсистеми статичного тестування. Було використано підхід Infrastructure as Code (IaC) та AWS CloudFormation для автоматизації розгортання та управління інфраструктурою. Підсистему було успішно протестовано на застосунку OWASP Juice Shop, виявивши значну кількість вразливостей.

Ключові слова: кібербезпека, статичне тестування, вразливості, SAST, програмне забезпечення, інструменти тестування, інтеграція, CI/CD, підсистема, автоматизація.

ABSTRACT

The complex bachelor's work consists of 52 pages of A4 format, which includes 10 figures, 6 tables, and a list of references containing 16 titles. The comprehensive bachelor's work is devoted to the study of static software security testing (SAST) tools and the development of a SAST subsystem. As a result of the analysis, it was found that modern static testing tools can detect a wide range of vulnerabilities in software, but each of them has its own characteristics and limitations. An approach to choosing the optimal tool is proposed, including consideration of such factors as supported programming languages, integration with CI/CD, accuracy and operation time.

Experimental testing was conducted using several popular SAST tools, such as Semgrep, SonarQube, and Horusec, and the results showed their effectiveness.

The results of the experiment showed that Semgrep has the best vulnerability detection rate, SonarQube has a low false positive rate, and Horusec has the lowest overall efficiency.

The Semgrep tool was selected to implement the static testing subsystem. The Infrastructure as Code (IaC) approach and AWS CloudFormation were used to automate infrastructure deployment and management. The subsystem was successfully tested on the OWASP Juice Shop application, revealing a significant number of vulnerabilities.

Keywords: cybersecurity, static testing, vulnerabilities, SAST, software, testing tools, integration, CI/CD, subsystem, automation.

ЗМІСТ

ВСТУП

У світі швидко розвивається сфера інформаційних технологій, а разом з тим зростає і загроза кібербезпеці. З урахуванням частоти та складності кібератак, компанії та розробники програмного забезпечення все більше звертають увагу на засоби тестування безпеки застосунків. Статичне тестування є одним із важливих етапів у процесі виявлення та усунення потенційних вразливостей, що робить його темою важливою та актуальною для дослідження. Статичне тестування включає аналіз коду без виконання програми, що дозволяє виявити потенційні вразливості на ранніх стадіях розробки. Цей метод допомагає розробникам знайти і виправити помилки до того, як вони стануть загрозою для кінцевого користувача.

Крім того, із зростанням складності програмного забезпечення і впровадженням нових технологій, ускладнюється і процес забезпечення його безпеки. Стандартні методи тестування можуть бути недостатніми для виявлення всіх потенційних вразливостей, тому статичне тестування стає ключовим елементом в процесі забезпечення безпеки програмного забезпечення. Воно дозволяє виявляти вразливості, які можуть бути використані для атаки, навіть до випуску програми в продукцію.

Метою роботи є підвищення безпеки застосунків за допомогою підсистеми статичного тестування.

Об'єктом роботи є процес тестування безпеки програмних застосунків.

Предметом роботи є методи та інструменти статичного тестування безпеки програмних застосунків.Для досягнення поставленої мети необхідно розв’язати такі задачі:

- Розглянути функції статичного тестування та здійснити аналіз популярних інструментів.
- Оцінити ефективність інструментів тестування за допомогою проведення експерименту.
- Розробити та реалізувати підсистему статичного тестування виходячи з результатів експерименту.

- Отримати і проаналізувати результат роботи підсистеми статичного тестування на практиці.

Результати роботи були апробовані на конференції із публікацією тез та матеріалів доповідей, Функції та методи статичного тестування безпеки. Всеукраїнська науково-практична інтернет-конференції «Молодь в науці: дослідження, проблеми, перспективи (МН-2024)» [1].

1 АНАЛІЗ ФУНКЦІЙ ТА ІНСТРУМЕНТІВ СТАТИЧНОГО ТЕСТУВАННЯ БЕЗПЕКИ ПРОГРАМНИХ ЗАСТОСУНКІВ

Безпека програмного забезпечення є важливим аспектом розробки програмного забезпечення, і статичне тестування безпеки застосунків (SAST) є важливим інструментом для забезпечення безпеки програм.

SAST – тип тестування безпеки що передбачає аналіз вихідного коду або скомпільованих двійкових файлів для виявлення вразливостей безпеки. SAST інструменти можуть допомогти розробникам знайти недоліки безпеки на ранніх етапах життєвого циклу розробки, зменшуючи ризик інцидентів безпеки та забезпечення дотримання стандартів безпеки.

SAST виникає на початку життєвого циклу розробки програмного забезпечення, оскільки він аналізує код у невиконаному стані і не потребує робочої програми [2].

1.1 Функції статичного тестування безпеки

Статичний аналіз може виконуватись вручну або за допомогою спеціалізованих інструментів. Ручний аналіз передбачає детальне вивчення коду розробниками або аудиторами безпеки, що може бути досить трудомістким процесом. З іншого боку, автоматичний статичний аналіз виконується за допомогою програмного забезпечення, яке автоматично сканує код на наявність відомих вразливостей і помилок.

Інструменти SAST статично аналізують код без виконання, щоб визначити шаблони та розпізнати вразливості.

Ці інструменти можуть виявляти такі проблеми як переповнення буфера, SQL-ін’єкції, міжсайтовий скриптинг (XSS) і інші поширені вразливості [3].

Деякі загальні функції інструментів SAST такі:

− Аналіз коду. Інструменти SAST аналізують вихідний код і виявляють потенційні вразливості. Вони перевіряють помилки коду, небезпечні практики написання коду та інші проблеми, які можуть призвести до вразливостей безпеки.

− Сканування на основі правил. Використовуються попередньо визначені правила для сканування коду та визначення потенційних вразливостей. Ці правила базуються на найкращих практиках і відомих уразливостях, їх можна налаштувати відповідно до конкретних умов організації;

− Хибно позитивне зменшення. Використовуються такий метод як аналіз потоку даних, і контролювати аналіз потоку для зменшення помилкових спрацьовувань. Помилкові спрацьовування виникають, коли інструмент ідентифікує вразливість якої немає в коді;

− Інтеграція з інструментами розробки. Інструменти SAST можна інтегрувати в середовища розробки (IDE) та системи збірок для сканування коду під час процесу розробки. Це дозволяє розробникам виявити та усунути вразливості на ранній стадії життєвого циклу розробки програмного забезпечення (SDLC).

− Звітування. Надаються докладні звіти з переліком виявлених уразливостей і надаються вказівки для їх виправлення. Звіти можуть містити фрагменти коду або запропоновані зміни коду для усунення вразливостей.

− Перевірка відповідності. Інструменти SAST можуть перевіряти код на відповідність стандартам безпеки та правилам, таким як OWASP Top Ten, PCI DSS і HIPPA.

1.2 Вирішення проблем за допомогою SAST

Статичний аналіз використовується в досить широких сферах, оскільки існує багато типів інструментів статичного аналізу, кожен з яких використовується для різних цілей. Розглянемо проблеми які можуть бути вирішені різними інструментами статичного аналізу.

1.2.1 Перевірка типів

Найбільш поширеною формою статичного аналізу, з якою знайома більшість програмістів, є перевірка типів. Багато програмістів не приділяють цьому багато уваги. Адже правила визначаються мовою програмування та забезпечуються компілятором. Наприклад, це запобігає присвоєння цілочислених значень змінній

з плаваючою комою. Виявляючи помилки під час компіляції, перевірка типів запобігає виникнення помилок під час виконання програми.

1.2.2 Перевірка стилю

Перевірка стилів також є статичним аналізом. Зазвичай він вимагає більш поверхневого набору правил ніж перевірка типів. Перевірники стилів застосовують правила пов’язані з пробілами, іменуванням, використання застарілих функцій, коментуванням, структурою програми тощо. Через те що багато програмістів є прихильниками власного стилю, більшість таких програм мають досить гнучкий набір правил які вони забезпечують. Помилки виявлені такими інструментами часто пов’язані з читабельністю та підтримкою коду, але ніяк не вказують на помилки які можуть статись при запуску програми.

Може бути важко імплементувати засіб напівдорозі великого проекту, тому що різні програміти ймовірно мають різні уявлення про “правельність” стилю. Після того як проект розпочато, робити перегляд коду для того аби зменшити кількість помилок даним інструментом є не вигідно, адже ціною великих незручностей буде отримано лише незначну користь. Перевірки стилю простіше всього прийняти на початку проекту.

1.2.3 Розуміння програми

Інструменти для розуміння програми допомагають розібратись в великий кількості вихідного коду. Інтегровані середовища розробки завжди включають принаймні деякі функції розуміння програм. Наприклад аби знайти всі використання якогось конкретного методу чи знайти оголошення змінної. Більш просунутий аналіз може підтримувати функції автоматичного рефакторингу програми, такі як перейменування зміних або поділ однієї функції на декілька.

Інструменти розуміння програм високо рівня намагаються допомогти програмістам отримати уявлення про те, як працює програма. Деякі намагаються реконструювати інформацію про дизайн програми на основі аналізу реалізації, таким чином надаючи програмісту загальне уявлення про програму. Це особливо

корисно для програмістів, яким потрібно зрозуміти великий обсяг коду, який вони не писали [4].

1.2.4 Перевірка властивостей та верифікація

Інструмент приймає специфікацію та частину коду, а потім намагається довести, що код є точною реалізацією специфікації. Якщо специфікація є повним описом усього, що має виконувати програма, інструмент перевірки програми може виконати перевірку еквівалентності, щоб переконатися, що код і специфікація точно збігаються.

Рідко програмісти мають специфікацію, яка є достатньо детальною, щоб її можна було використовувати для перевірки еквівалентності, і робота зі створення такої специфікації може в кінцевому підсумку обернутися більшою роботою, ніж написання коду, тому цей стиль перевірки трапляється не дуже часто.

Також цей метод відомий як формальна верифікація.

Формальна верифікація - це метод перевірки коректності програмного забезпечення шляхом використання математичних моделей і логічних доведень. Цей метод дозволяє довести, що програма відповідає специфікаціям і не містить певних типів помилок або вразливостей [4].

Основні етапи формальної верифікації:

− Моделювання - створення формальної моделі програми або її частин. Це може включати абстрактні описи алгоритмів, структур даних та взаємодії між компонентами.

− Специфікація - визначення формальних специфікацій, які описують бажану поведінку програми. Специфікації можуть включати інваріанти, передумови та постумови для функцій, а також властивості системи, які повинні бути виконані.

− Доведення - використання логічних методів і автоматизованих інструментів для доведення того, що модель програми відповідає специфікаціям. Це може включати перевірку логічної еквівалентності, символьне виконання та аналіз моделі.

Формальна верифікація може бути особливо корисною для критично важливих систем, де помилки можуть мати серйозні наслідки, таких як авіаційні системи, медичне обладнання та банківське програмне забезпечення. Проте цей метод має і свої обмеження, зокрема високу складність і вартість впровадження, а також вимоги до високої кваліфікації розробників і аналітиків.

1.2.5 Вразливості які шукає SAST

Список вразливостей які зазвичай зможуть знаходити інструменти статичного тестування безпеки:

− Ін’єкція команд - вразливість, коли зловмисники можуть впровадити та виконати власні команди або системні виклики через додаток, який використовує вхідні дані для створення системних команд.

− Небезпечні кукі-файли - інформація, збережена в куках (cookies), не зашифрована або відправляється через незахищені канали, вона стає доступною для атак з перехопленням.

− LDAP ін’єкція - атака полягає в тому, що зловмисники вводять спеціально сформовані дані, які використовуються для маніпулювання запитами до LDAP-систем, часто для викрадення даних або отримання несанкціонованого доступу.

− Обхід шляху - вразливість дозволяє зловмисникам отримувати доступ до файлів та каталогів, до яких вони зазвичай не мають доступу, шляхом маніпулювання шляхами файлів у запитах до сервера.

− SQL ін’єкції - вразливість, яка дозволяє зловмисникам впроваджувати та виконувати SQL-запити до бази даних через недостатньо фільтровані введені дані.

− Межа довіри - місце в програмі, де довіряються вхідні дані. Якщо воно порушується, можна отримати недозволений доступ до функцій чи ресурсів.

− Слабкий алгоритм шифрування - використання слабкого алгоритму шифрування, який легко піддається атакам, наприклад, атакам методом перебору.

− Слабкий алгоритм хешування - використання слабкого алгоритму хешування для зберігання паролів або інших конфіденційних даних, що може легко бути скомпрометовано.

− Слабка випадковість - використання недостатньо випадкових значень при генерації ключів, токенів або інших важливих випадкових об'єктів, що робить їх прогнозованими.

− XPath ін’єкція - вразливість дозволяє зловмисникам впроваджувати та виконувати XPath-запити з метою отримання конфіденційної інформації або виконання несанкціонованих дій.

− Міжсайтовий скриптінг (XSS) - вразливість, яка дозволяє зловмисникам впроваджувати та виконувати скрипти на веб-сторінках, що відображаються у веб-браузері користувача.

1.3 Популярні інструменти для статичного тестування

Ринок інструментів для статичного тестування пропонує широкий вибір рішень, кожне з яких має свої особливості та переваги. Деякі з найпопулярніших інструментів включають:

Veracode - пропонує комплексне рішення для статичного тестування та має велику базу даних відомих вразливостей. Veracode використовує хмарне середовище для аналізу коду, що дозволяє знизити витрати на інфраструктуру та забезпечує швидкий і зручний доступ до результатів аналізу [5].

SonarQube - відкритий інструмент для статичного аналізу коду, який підтримує численні мови програмування та інтеграції з іншими інструментами. SonarQube надає розширені можливості для аналізу коду, включаючи виявлення вразливостей, помилок стилю кодування та інших проблем. Інструмент також підтримує інтеграцію з популярними CI/CD системами, що дозволяє автоматизувати процес статичного тестування [6].

Fortify Static Code Analyzer - інструмент від Micro Focus, який спеціалізується на виявленні вразливостей у вихідному коді. Fortify використовує комплексний підхід до аналізу коду, включаючи перевірку на відповідність стандартам безпеки та виявлення широкого спектру вразливостей. Інструмент також пропонує інтеграцію з іншими інструментами розробки, що дозволяє легко впровадити його у процес розробки програмного забезпечення [7].

Для більш чіткої картини виконано порівняльний аналіз згаданих інструментів, та записано його результати в табл. 1.1.

Таблиця 1.1 – Аналіз інструментів статичного тестування

	VeraCode	SonarQube	Fortify SCA
Тип	SaaS	Локальне, SaaS	Локальне, SaaS
Підтримувані мови	Широкий спектр включаючи Java, .NET, JavaScript, Python	Широкий спектр включаючи Java, .NET, JavaScript, Python	Широкий спектр включаючи Java, .NET, JavaScript, Python
Інтеграція	CI/CD інструменти, IDE, системи відстеження помилок	CI/CD інструменти, IDE, SCM	CI/CD інструменти, IDE, системи відстеження помилок
Сканування контейнерів	Обмежений	Обмежений	Так
Хибні спрацювання	Іноді виявляються хибні спрацьовування	Низький рівень хибних спрацьовувань	Середній рівень
Ціна	Вища вартість, підписка	Низька вартість для спільнотної версії, плагіни можуть збільшити вартість	Вища вартість, особливо для великих впроваджень
Масштабованість	Висока масштабованість, хмарний сервіс	Може вимагати багато ресурсів для великих проектів	Масштабований, але може бути дорогим
Продуктивність	Швидке сканування з хорошими показниками	Ефективний, може бути повільним для великих кодових баз	Висока продуктивність, але вимогливий до ресурсів

Veracode є одним з найпопулярніших інструментів для статичного тестування завдяки своїй великій базі даних вразливостей та хмарному рішенню, що дозволяє знизити витрати на інфраструктуру. Однак, висока вартість цього інструменту може бути недоліком для малих компаній або стартапів.

SonarQube, будучи відкритим інструментом, пропонує широкий спектр можливостей для аналізу коду та підтримує численні мови програмування. Це робить його привабливим для розробників, які шукають безкоштовне або доступне рішення. Однак, можливості SonarQube можуть бути обмеженими без додаткових плагінів.

Fortify SCA від Micro Focus спеціалізується на глибокому аналізі безпеки та підтримці стандартів безпеки. Це робить його ідеальним вибором для великих компаній та організацій, що потребують високого рівня безпеки. Проте, складність налаштування та висока вартість можуть бути недоліками цього інструменту.

1.4 Висновки до розділу

Отже, SAST - це важливий процес у розробці програмного забезпечення, який допомагає виявити вразливості безпеки на ранній стадії життєвого циклу розробки. Аналізуючи вихідний код програми на предмет потенційних проблем безпеки, інструменти SAST можуть допомогти розробникам виправити проблеми до того, як вони будуть розгорнуті у виробництво, зменшуючи ризик порушень безпеки.

SAST пропонує кілька переваг, включаючи раннє виявлення вразливостей, економічно ефективне тестування безпеки та можливість виявлення проблем безпеки в сторонньому коді. Однак, він також має обмеження, включаючи можливість помилкових спрацьовувань і хибних негативних результатів, а також нездатність виявляти певні типи проблем безпеки.

Було проведено аналіз декількох популярних інструментів для статичного аналізу коду. Також SAST слід використовувати разом з іншими заходами безпеки, такими як DAST та ручне тестування на проникнення, щоб забезпечити багаторівневий підхід до захисту додатків.

2 ПРОЕКТУВАННЯ ПІДСИСТЕМИ СТАТИЧНОГО ТЕСТУВАННЯ

2.1 Огляд загальної системи тестування безпеки

ІТ-система великих підприємств складається зі значної кількості додатків, які потребують постійного тестування на наявність вразливостей безпеки. Особливо це стосується власної розробки програмного забезпечення.

У сфері тестування безпеки додатків існує багато різних типів методів тестування безпеки, таких як статичне тестування безпеки додатків (SAST), динамічне тестування безпеки додатків (DAST). Кожен з цих методів тестування зазвичай підтримується одним комерційним продуктом для захисту програмного забезпечення.

Але нажаль більшість компаній або нехтують заходами безпеки або використовують лише один з методів тестування тим самим не забезпечуючи повний спектр заходів для перевірки безпеки свого продукту.

Хоч і багато компаній зосереджуються на статичному аналізі, динамічний аналіз може виявити типи вад, які може бути важко знайти при статичному аналізі. І хоча додавання динамічного тестування безпеки додатків призведе до виявлення більшої кількості вразливостей, компанії які поєднують динамічне сканування зі статичним, в результаті закривають більше вразливостей швидше.

Це ж стосується і відновлення роботи після того, як якась вразливість була використана зловмисником. Компанії, які комбінують SAST (Статичний аналіз безпеки застосунків) та DAST (Динамічний аналіз безпеки застосунків) у своїх процесах розробки, у середньому відновлюють роботу свого продукту після інциденту на 24,5 дні швидше, ніж інші компанії, які не використовують такий комплексний підхід [8].

Тому для забезпечення максимальної безпеки застосунку та прискорення реагування на можливі інциденти пропонується використовувати систему, яка комбінує SAST та DAST, як зображено на рис. 2.1. Такий підхід дозволяє виявляти вразливості на ранніх стадіях розробки тестуючи застосунок у реальних умовах та скоротити час реагування на інциденти.

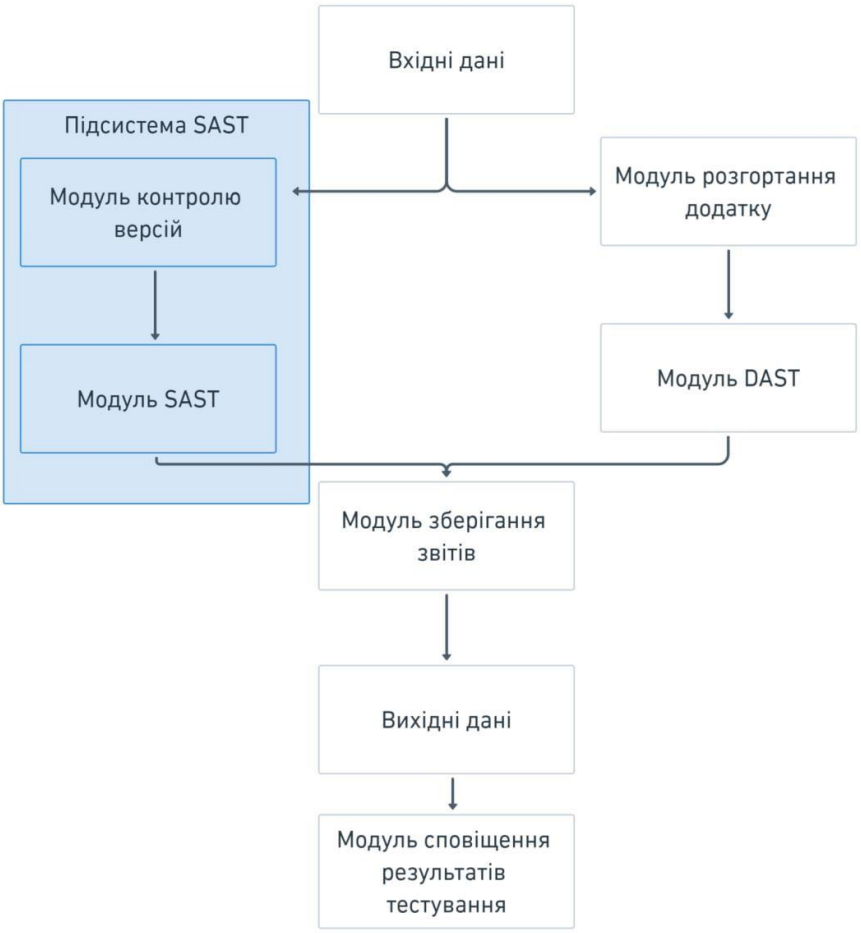


Рисунок 2.1 – Загальна схема системи тестування безпеки застосунку

2.2 Підсистема статичного тестування

Підсистема статичного тестування (SAST) — це сукупність заходів і робочих процесів, що використовуються для виконання статичного аналізу безпеки коду. Цей процес включає аналіз вихідного коду або скомпільованих бінарних файлів для виявлення вразливостей безпеки, таких як помилки кодування, недотримання стандартів безпеки та інші потенційні проблеми, які можуть призвести до вразливостей у застосунку.

Нижче наведено огляд типового процесу та робочого циклу SAST:

− Вибір інструменту. Першим кроком є вибір відповідного інструменту SAST, який відповідає вимогам додатку, що тестується. При цьому враховуються

такі фактори, як мова програмування, складність програми та тип вразливостей, які необхідно виявити.

− Конфігурація. Інструмент SAST потрібно налаштувати для аналізу вихідного коду програми. Це включає визначення обсягу аналізу, наприклад, які файли та каталоги слід проаналізувати, а також вказівку будь-яких користувацьких правил або налаштувань.

− Сканування коду. Інструмент SAST сканує вихідний код програми для виявлення потенційних вразливостей безпеки. Цей процес виконується автоматично і зазвичай займає кілька годин, залежно від розміру та складності програми.

− Виявлення вразливостей. Інструмент SAST виявляє потенційні вразливості у вихідному коді програми на основі попередньо визначених правил та алгоритмів. Інструмент генерує звіт, який містить список вразливостей, а також інформацію про місце розташування вразливості в коді, серйозність проблеми та рекомендації щодо її усунення.

− Перевірка та аналіз. Звіт розглядається та аналізується експертами з безпеки, щоб підтвердити достовірність виявлених вразливостей та визначити їх пріоритетність на основі серйозності та потенційного впливу. Цей етап може також включати подальше дослідження виявлених вразливостей, наприклад, тестування, щоб визначити, чи можуть вони бути використані або чи існують якісь пом'якшувальні фактори.

− Звітність. Після завершення процесів SAST формується детальний звіт, який містить вичерпну інформацію про виявлені вразливості, їх класифікацію за ступенем серйозності, а також рекомендації щодо їх усунення. Цей звіт є цінним інструментом для команди розробників, дозволяючи їм ефективно планувати та впроваджувати заходи з підвищення безпеки застосунку.

Процеси підсистеми можна представити у вигляді схеми зображеної на рис. 2.2.

Ця схема ілюструє взаємозв'язок між різними компонентами підсистеми та послідовність їх виконання.

Рисунок 2.2 – Процеси підсистеми статичного тестування

Важливо вимірювати ефективність та результативність проведеного аналізу, використовуючи спеціальні показники:

− Кількість знайдених вразливостей. Автоматизована технологія тестування часто генерує значний відсоток хибних спрацьовувань. Тому загальна кількість знайдених вразливостей може вводити в оману. Важливіше визначити, чи є виявлені ризики та вразливості насправді атакованими - тобто, чи може зловмисник використати їх для експлуатації та компрометації додатку, а отже, чи становлять вони реальну загрозу [9].

− Рівень виправлення - відсоток вразливостей, які можна використати для атаки і які дійсно виправляються, повинен бути дуже близьким до 100%, якщо вразливості підтверджені як потенційні загрози. Зазвичай, менше 10% загроз, виявлених інструментами, можуть бути використані для атак. Однак важливо переконатися щодо серйозності та терміновості кожної вразливості [9].

− Щільність вразливостей - показник кількості вразливостей на рядок коду. Ця метрика може допомогти виявити ділянки коду, які можуть потребувати додаткового тестування та виправлення [9].

Підсистема статичного тестування (SAST) забезпечує високий рівень безпеки програмного забезпечення, дозволяючи виявляти вразливості на ранніх етапах розробки.

2.3 Головні фактори вибору інструменту SAST

Задля визначення найкращого інструменту для використання у підсистемі статичного тестування безпеки необхідно виконати експеримент з використанням інструментів SAST. Метою даного експерименту є проведення детального та всебічного порівняльного аналізу ефективності різних засобів статичного тестування безпеки для виявлення вразливостей у програмному коді. Було визначено головні фактори які будуть впливати на вибір інструментів.

Виявлення хибних спрацьовувань. Важливим аспектом ефективності інструментів SAST є їх здатність точно ідентифікувати реальні вразливості у коді, мінімізуючи при цьому кількість хибних спрацьовувань. Для цього необхідно провести оцінку відсотка хибних позитивів (false positives) та хибних негативів (false negatives) для кожного інструменту. Після виконання сканування кожним інструментом, зібрані виявлені вразливості фіксуються в окремому звіті, де міститься інформація про знайдені вразливості, їх опис, місце розташування в коді та рівень серйозності.

Оцінка зручності. Зручність використання інструментів SAST є важливим аспектом, оскільки зручний інструмент сприяє підвищенню продуктивності та ефективності команди розробників і тестувальників. Для цього проводиться

детальний аналіз інтерфейсу користувача, швидкості сканування, якості документації та легкості налаштування.

Перш за все, розглядається інтерфейс користувача кожного інструменту. Інструмент повинен бути інтуїтивно зрозумілим, з чіткою та логічною структурою меню і функцій. Це включає в себе оцінку зручності навігації, зрозумілості відображення результатів сканування та легкості доступу до необхідних функцій. Інструмент з добре продуманим інтерфейсом значно зменшує час, необхідний для навчання та адаптації користувачів.

Підсумковий аналіз ефективності. Підсумковий аналіз ефективності інструментів SAST є завершальним етапом експерименту, на якому проводиться комплексна оцінка всіх зібраних даних для визначення оптимального інструменту. Цей етап включає інтеграцію результатів усіх попередніх оцінок та порівняльний аналіз за кількома ключовими параметрами.

Підсумковий аналіз включає оцінку точності виявлення вразливостей. Збираються дані про кількість виявлених реальних вразливостей, а також про кількість хибних позитивів та хибних негативів для кожного інструменту. Інструмент, який виявив найбільше реальних вразливостей з найменшою кількістю хибних спрацьовувань, вважається найбільш точним та надійним.

Далі проводиться аналіз зручності використання, що охоплює інтерфейс користувача, швидкість сканування, якість документації та легкість налаштування. Інструменти з найвищими оцінками за цими критеріями сприяють підвищенню продуктивності команди та ефективності робочих процесів.

2.4 Обґрунтування об'єкту тестування

Проект OWASP Benchmark - це набір тестів на Java, призначений для оцінки точності, покриття та швидкості роботи автоматизованих інструментів виявлення вразливостей програмного забезпечення. Без можливості виміряти ці інструменти важко зрозуміти їхні сильні та слабкі сторони, а також порівняти їх між собою [10].

OWASP Benchmark складається з повноцінного веб-додатку з відкритим вихідним кодом. У цьому веб-додатку ми можемо знайти тисячі придатних для

використання тестових кейсів, кожен з яких пов'язаний зі слабкими місцями в загальному переліку вразливостей (CWE), які можна проаналізувати за допомогою будь-якого інструменту тестування безпеки додатків, включаючи інструменти SAST і DAST. Всі ці вразливості навмисно включені у веб-додаток, можуть бути використані і можуть бути виявлені інструментами. Бенчмарк також включає десятки генераторів оцінок для численних відкритих і комерційних інструментів для тестування безпеки застосунків, і набір підтримуваних інструментів постійно зростає.

Вибір OWASP Benchmark для проведення експерименту з дослідження інструментів статичного тестування безпеки можна обгрунтувати кількома чинниками:

− Визнаність та широка підтримка. OWASP (Open Web Application Security Project) є відомою та авторитетною організацією, яка зосереджена на забезпеченні безпеки веб-додатків. OWASP Benchmark - це проект, який широко визнаний в галузі тестування безпеки, і має значний рівень довіри в спільноті інформаційної безпеки.

− Актуальність та реалістичність сценаріїв атак. OWASP Benchmark надає набір реалістичних сценаріїв атак, які базуються на реальних вразливостях, знайдених в веб-додатках. Це дозволяє проводити тестування на реальних викликах, з якими можуть стикатися розробники.

− Велика кількість тестових випадків. OWASP Benchmark містить широкий набір тестових випадків, що охоплюють різні типи вразливостей. Це дозволяє проводити глибокий і всебічний аналіз різних аспектів безпеки.

− Активне співтовариство та оновлення. OWASP постійно оновлює та вдосконалює Benchmark з урахуванням нових загроз і вразливостей, що виникають у сфері безпеки веб-додатків. Це дозволяє експерименту бути актуальним та відповідати сучасним викликам безпеки.

− Відкритість та доступність. OWASP Benchmark є відкритим проектом, що забезпечує доступ до його матеріалів для широкої спільноти. Це робить його

привабливим для використання в дослідженнях та експериментах, оскільки дозволяє виконувати повторні перевірки та порівняння результатів.

Хоча WAVSEP (Web Application Vulnerability Scanner Evaluation Project) та SAMATE (Software Assurance Metrics And Tool Evaluation) також є важливими джерелами для оцінки інструментів безпеки, використання OWASP Benchmark може бути виправданим через його широку визнаність, актуальність та доступність в розробці та дослідженнях веб-додатків.

2.5 Обґрунтування вибору інструментів тестування

Здійснено вибір інструментів SAST, ефективність яких буде досліджено. Ці інструменти сканують вихідний код додатків та знаходять в ньому вразливості безпеки. Існує безліч таких інструментів, як комерційних, так і з відкритим вихідним кодом. Саме другий тип, з відкритим вихідним кодом, буде використовуватись, оскільки вони не тільки дозволяють проводити дослідження безкоштовно, але й надають багато переваг та функціональних можливостей.

Першим інструментом було обрано Semgrep. Semgrep є легким і швидким інструментом статичного аналізу коду, який підтримує широкий спектр мов програмування. Однією з головних переваг Semgrep є його простота використання та налаштування: він дозволяє розробникам швидко писати власні правила для виявлення вразливостей, завдяки зрозумілій синтаксису, що базується на знайомих конструкціях мов програмування. Semgrep також відзначається високою точністю виявлення вразливостей та мінімальною кількістю хибних спрацьовувань, що робить його надійним інструментом для забезпечення безпеки коду.

Крім того, Semgrep легко інтегрується у процеси CI/CD, забезпечуючи безперервний моніторинг безпеки під час розробки програмного забезпечення [11].

Його активна спільнота та регулярні оновлення гарантують, що інструмент завжди відповідає сучасним стандартам безпеки.

Другим було обрано Horusec. Його вибір для проведення експерименту з дослідження інструментів статичного тестування безпеки обґрунтований його сучасними можливостями та спеціалізованою фокусованістю на безпеці коду.

Horusec - це інструмент з відкритим вихідним кодом, який організовує роботу інших інструментів безпеки та виявляє недоліки або вразливості в проектах, а всі результати поміщає в базу даних для аналізу та генерації метрик [12].

Однією з ключових переваг Horusec є його здатність інтегруватися з DevSecOps процесами, що дозволяє автоматизувати перевірки безпеки на всіх етапах розробки, від початкового написання коду до деплою. Крім того, Horusec надає інтуїтивно зрозумілі звіти та детальні описи вразливостей, включаючи рекомендації щодо їх усунення, що значно полегшує роботу розробникам і фахівцям з безпеки. Інструмент також підтримує інтеграцію з популярними CI/CD системами та репозиторіями коду, що сприяє більш зручному і ефективному впровадженню у вже існуючі робочі процеси.

Третім інструментом обрано SonarQube. SonarQube є одним з найбільш популярних і всебічних інструментів для статичного аналізу коду, що підтримує понад 25 мов програмування. Крім того, SonarQube має зручний інтерфейс користувача та надає докладні звіти з аналізу, які допомагають швидко виявляти та виправляти проблеми в коді [6].

Його активна спільнота та регулярні оновлення забезпечують актуальність і надійність інструменту. Останнє, що слід зазначити про SonarQube - це можливість розширювати його за допомогою плагінів.

Ці інструменти були обрані для експериментального дослідження завдяки їх функціональним можливостям та популярності серед розробників, а також через їх безкоштовність та відкритість. Вони дозволять провести всебічний аналіз безпеки коду та порівняти їх ефективність у реальних умовах.

2.6 Підготовка до експерименту

Для забезпечення об'єктивного та всебічного аналізу ефективності інструментів статичного тестування безпеки, було проведено експеримент. Основна мета цього експерименту - оцінити здатність інструментів Semgrep, SonarQube та Horusec виявляти вразливості у веб-додатках та порівняти їх

результати для визначення найбільш ефективного рішення для подальшого використання в загальній системі тестування безпеки програмного застосунку.

2.6.1 Налаштування середовища для експерименту

Для проведення експерименту необхідно налаштувати середовище тестування на операційній системі Kali Linux. Це середовище забезпечить стабільність та необхідні інструменти для аналізу безпеки. Дистрибутив Kali Linux на базі Debian Linux, який має відкритий вихідний код, містить багато вбудованого функціоналу та орієнтований на використання для тестування на проникнення, що робить його найбільш підходящим для даної роботи.

Перш за все потрібно встановити операційну систему Kali Linux, встановлюємо її на віртуальну машину VirtualBox. Для цього потрібно завантажити саму програму VirtualBox та встановити на неї операційну систему, рекомендується завантажувати відразу готові віртуальні машини які можна найти на офіційному сайті Kali. Це робиться для того аби забезпечити максимально швидке розгортання віртуальної машини і зменшити час для налаштування, адже на таких машинах вже заздалегіть встановлені Python, Java та інші необхідні засоби.

Після встановлення рекомендується оновити список доступних пакунків та їхніх версій, аби в подальшому при завантажені інструментів не було помилок, це можна зробити наступними командами:

```
sudo apt update
sudo apt upgrade
```

Для того аби завантажити OWASP Benchmark нам знадобиться docker, інструмент для роботи з контейнерами. Аби встановити його виконаємо наступні команди:

```
sudo apt install -y docker.io
sudo systemctl enable docker –now
echo "deb [arch=amd64 signed-by=/etc/apt/keyrings/docker.gpg]
https://download.docker.com/linux/debian bookworm stable" | \
sudo tee /etc/apt/sources.list.d/docker.list
kali@kali:~$                    curl                    -fsSL
https://download.docker.com/linux/debian/gpg |
sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io
```

Аби перевірити чи docker було встановлено коректно можна використати команду:

```
docker --version
```

Вона відображає поточну версію docker як зображено на рис. 2.3.

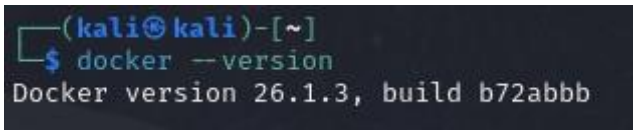


Рисунок 2.3 – Поточна версія Docker

Наступним буде завантаження OWASP Benchmark з Github. Для цього використаємо команду:

```
git clone https://github.com/OWASP-Benchmark/BenchmarkJava
```

Далі переходимо в папку VMs аби зібрати і запустити docker, що робиться шляхом послідовного запуску файлів BuildDockerImage.sh і runDockerImage.sh командами:

```
bash buildDockerImage.sh
bash runDockerImage.sh
```

Після виконання всіх команд OWASP Benchmark аби перевірити чи програму було встановлено коректно можна отримати до неї доступ за допомогою веб-браузеру за адресою localhost на порту 8443 як зображено на рис. 2.4.

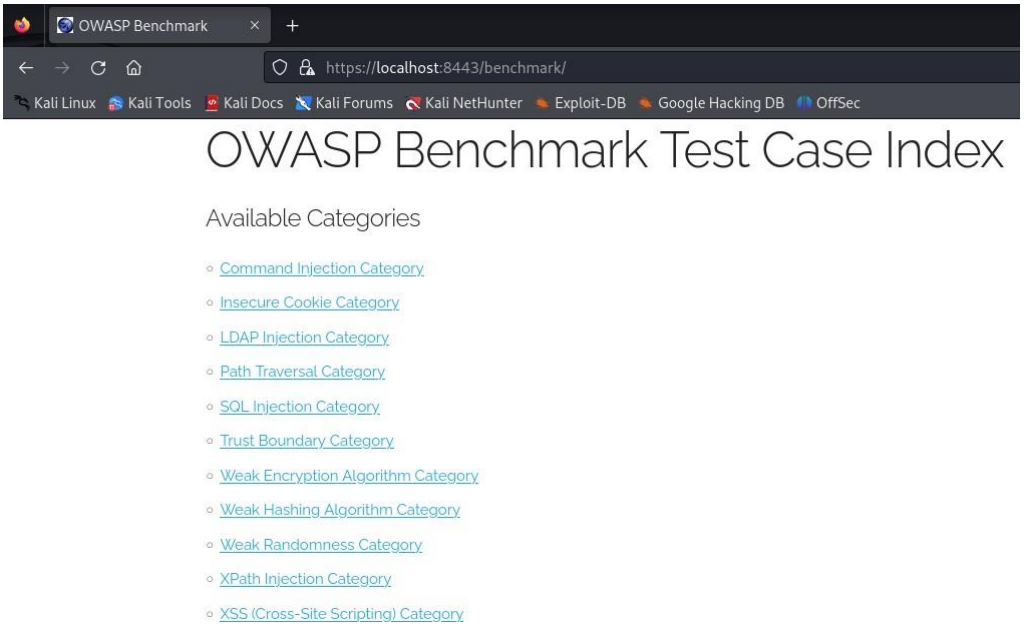


Рисунок 2.4 – OWASP Benchmark у веб-браузері

Далі почнемо встановлення інструментів для тестування. Першим буде Semgrep:

```
docker pull semgrep/semgrep
```

А для перевірки чи правильно було встановлено інструмент використаємо наступну команду:

```
docker run --rm semgrep/semgrep semgrep –version
```

Вона виведе нам поточну версію застосунку як зображено на рис. 2.5.

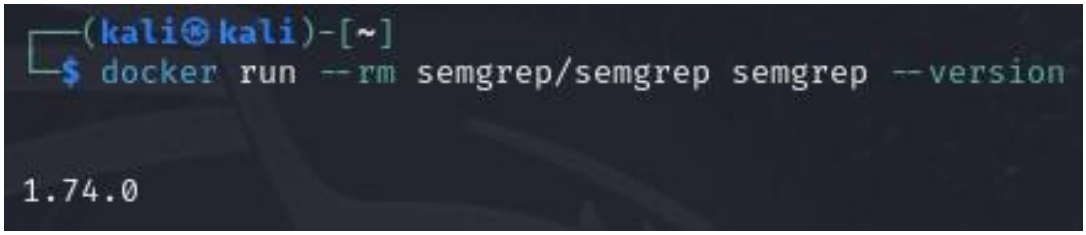


Рисунок 2.5 – Поточна версія Semgrep

Наступним кроком встановлюємо Horusec, для цього виконаємо команду:

```
curl -fsSL
https://raw.githubusercontent.com/ZupIT/horusec/main/deployments/scr
ipts/install.sh | bash -s latest
```

Для перевірки коректного встановлення інструменту виконаємо команду horusec version як зображено на рис. 2.6.

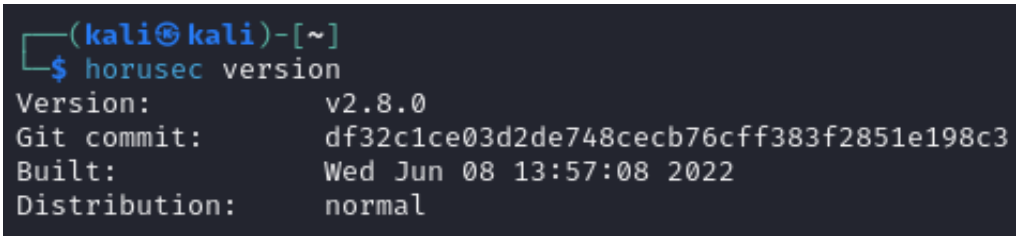


Рисунок 2.6 – Поточна версія Horusec

Останнім інструментом для встановлення буде SonarQube.

В OWASP Benchmark є спеціально створений скриптовий файл для встановлення та запуску SonarQube, аби запустити його перейдемо в папку бенчмарку та виконаємо команди:

```
chmod 755 ./scripts/runSonarQube.sh
bash ./scripts/runSonarQube.sh
```

2.6.2 Визначення метриків

Визначимо метрики за якими буде відбуватись порівняння інструментів SAST. Вони базуються на обраному OWASP Benchmark, оскільки він має

функціональні можливості для визначення, розрахунку та представлення надзвичайно важливих показників та метрик.

Додаток BenchmarkScore для генерації оцінок входить до складу бенчмарку. Він аналізує вихідні файли, згенеровані будь-яким з підтримуваних інструментів безпеки, запущених на бенчмарку, і порівнює їх з очікуваними результатами, а також створює набір веб-сторінок, на яких детально описується точність і швидкість роботи відповідних інструментів.

Завдяки їм отримується повна картина ефективності інструментів тестування, а шляхом їх порівняння можна провести детальний, коректний і надійний порівняльний аналіз між використовуваними статичними інструментами тестування безпеки додатків.

У поточній роботі використовуються індикатори та метрики для оцінки сканерів на основі вже обраного бенчмарку. Ці метрики автоматично розраховуються і візуалізуються за допомогою функції оцінки результатів OWASP Benchmark.

Метрики та індикатори представлені в таблиці 2.1.

Таблиця 2.1 – Узагальнення використаних метрик

Назва	Формула	Опис	Опис в OWASP Benchmark
Правдиві Позитиви (TP)		Реальна вразливість, яку потрібно виправити	Тести з реальними вразливостями, які були правильно повідомлені інструментом як уразливі
Хибні Негативи (FN)		Реальна вразливість не виявлена інструментом що є серйозною проблемою	Тести з реальними вразливостями, які не були правильно повідомлені інструментом як уразливі

Продовження таблиці 2.1

Правдиві Негативи (TN)		Код без вразливостей який не потребує дій	Тести з фальшивими вразливостями, які були правильно не повідомлені інструментом як уразливі
Хибні Позитиви (FP)		Код без вразливостей повідомлений як помилковий, це попередження слід ігнорувати	Тести з фальшивими вразливостями, які були неправильно повідомлені інструментом як уразливі
Показник Правдивих Позитивів (TPR)	$TPR = \frac{TP}{TP + FN}$	Пропорція позитивних випадків, які правильно класифіковані як позитивні. Також називається Recall або чутливість	Відсоток вразливостей які інструмент правильно повідомляє
Показник Хибних Позитивів (FPR)	$FPR = \frac{FP}{FP + TN}$	Представляє співвідношення негативів, які неправильно класифіковані як позитивні. Також називається fall-out	Відсоток фальшивих фразливостей які інструмент повідомляє як реальні

До основних метрик також додамо ще деякі параметри по яким будем порівнювати кожен з інструментів:

− Тип інтерфейсу. Чи має інструмент графічний інтерфейс користувача (GUI) або інтерфейс командного рядка (CLI). GUI зручніший для новачків завдяки простоті використання, тоді як CLI пропонує більше можливостей для автоматизації та гнучкість для досвідчених користувачів.

− Підтримка мов програмування. Які мови програмування може аналізувати інструмент. Чим більше мов підтримується, тим універсальніший інструмент у різних середовищах розробки. Зазвичай підтримувані мови включають Java, JavaScript, Python, Ruby, C# тощо.

− Час, необхідний для сканування. Тривалість сканування інструментом. Швидкість сканування може варіюватися в залежності від складності та розміру тестованої програми. Коротший час сканування зазвичай кращий, але не повинен знижувати якість аналізу.

− Кількість виявлених попереджень про безпеку. Кількість виявлених під час сканування попереджень або вразливостей. Більша кількість виявлених проблем свідчить про ретельність інструменту, але важливо також враховувати точність, щоб уникнути помилкових спрацьовувань.

2.7 Аналіз результатів використання інструментів

Після запуску кожного з інструментів отримуємо файли формату xml. Аби виконати аналіз результатів використаємо інструмент для створення карток з результатами BenchmarkUtils. На цьому кроці з консолі запускається скрипт, який генерує картки на основі вже розміщених файлів з результатами, отриманими під час сканування інструментами SAST. Для цього запускається файл createScorecards.sh з каталогу OWASP Benchmark.

Після його виконання генеруються картки у форматі html, які містять детальну інформацію про отримані результати, а також ілюструють їх за допомогою графіки. Крім цього файлу, скрипт генерує ще два файли: перший - це png-зображення вищезгаданого графіка, а другий - csv-файл, що містить обсяг даних з тестів та аналізів. Таким чином, на основі згенерованих оціночних карток проводиться порівняльний аналіз між інструментами статичного тестування.

Отримавши результати запишемо їх в таблиці. Першим для аналізу буде інструмент Semgrep. Як видно з таблиці 2.2 Semgrep має досить високий показник TPR, це означає що інструмент правильно виявляє 96.84% справжніх вразливостей в програмному коді.

Це досить високий рівень ефективності, що свідчить про те, що Semgrep надійно виявляє справжні вразливості. Водночас він також має досить високий показник FPR (False Positive Rate), це означає, що приблизно 57.74% із тих випадків, які Semgrep відмічає як потенційно небезпечні або вразливі, насправді не

є вразливостями. Хоча це може створювати додаткову роботу для розробників, які мають перевіряти ці помилкові спрацювання, важливо пам'ятати, що Semgrep все ж таки допомагає виявляти значну кількість реальних проблем безпеки, які могли б залишитися непоміченими.

Таблиця 2.2 - Показники роботи Semgrep

Параметр	Результати
Тип інтерфейсу	GUI та CLI
Час сканування (хв)	3
Загальна кількість тестових випадків	2740
Кількість виявлених попереджень про безпеку	2028
Істинно-позитивні (TP), хибно-позитивні (FP), істинно-негативні (TN), хибно-негативні (FN)	TP = 1325, FP = 703, TN = 637, FN = 75
Істинно-позитивний (TPR) та хибно-позитивний показники (FPT) (%)	TPR = 96.84%, FPR = 57.74%
Оцінка (%)	Оцінка = TPR - FPR = 39.1%

Наступним інструментом, який було проаналізовано, є SonarQube. Як вказано в таблиці 2.3, його показники виявлення вразливостей є нижчими, ніж у Semgrep. Однак, завдяки низькому відсотку хибних спрацювань, його загальна оцінка лише на декілька відсотків нижча, ніж у Semgrep. Це свідчить про те, що SonarQube є більш точним інструментом, хоча й може пропускати деякі вразливості.

На відміну від Semgrep, SonarQube має декілька версій: безкоштовну Community та декілька платних версій. Варто зазначити, що безкоштовна версія SonarQube не підтримує такі мови програмування, як C, C++, Objective-C, Swift та COBOL. Тому, якщо проект використовує ці мови, доведеться обрати одну з платних версій SonarQube або розглянути інші інструменти.

Загалом, вибір між Semgrep та SonarQube залежить від конкретних потреб та вимог. Semgrep може бути кращим вибором для проектів, де пріоритетом є

швидкість та ефективність виявлення вразливостей, тоді як SonarQube може бути більш підходящим для проектів, які потребують більш широкого спектру функцій та детального аналізу коду.

Таблиця 2.3 - Показники роботи SonarQube

Параметр	Результати
Тип інтерфейсу	GUI та CLI
Час сканування (хв)	5
Загальна кількість тестових випадків	2740
Кількість виявлених попереджень про безпеку	748
Істинно-позитивні (TP), хибно-позитивні (FP), істинно-негативні (TN), хибно-негативні (FN)	TP = 607, FP = 141, TN = 1184, FN = 60
Істинно-позитивний (TPR) та хибно-позитивний показники (FPT) (%)	TPR = 50.36%, FPR = 17.02%
Оцінка (%)	Оцінка = TPR- FPR = 33.34 %

І останнім на черзі є Horusec. З поміж трьох він показав себе найгірше, як видно з таблиці 2.4. Показник виявлення вразливостей, хоч і є нормальним, але через досить високий показник виявлення хибних вразливостей, загальний результат інструменту є досить низьким. Це означає, що Horusec може бути менш ефективним у порівнянні з іншими інструментами, оскільки його використання може призвести до додаткових витрат часу та ресурсів на аналіз хибних спрацювань.

Однак, варто зазначити, що Horusec є повністю безкоштовним інструментом з відкритим вихідним кодом, що може бути привабливим для невеликих проектів або команд з обмеженим бюджетом.

Крім того, Horusec активно розвивається та вдосконалюється, тому його ефективність може зрости в майбутньому.

Але на даний момент Horusec найгірше підходить для використання в підсистемі статичного тестування через свою низьку ефективність.

Таблиця 2.4 - Показники роботи Horusec

Параметр	Результати
Тип інтерфейсу	CLI
Час сканування (хв)	4
Загальна кількість тестових випадків	2740
Кількість виявлених попереджень про безпеку	184
Істинно-позитивні (TP), хибно-позитивні (FP), істинно-негативні (TN), хибно-негативні (FN)	TP = 716, FP = 426, TN = 899, FN = 699
Істинно-позитивний (TPR) та хибно-позитивний показники (FPT) (%)	TPR = 47.64%, FPR = 35.99%
Оцінка (%)	Оцінка = TPR- FPR = 11.65 %

2.8 Висновки до розділу

Виконано ряд дій для підготовки та налаштування середовища для проведення експерименту з використанням різних інструментів. Було встановлено програму бенчмарк яка буде тестуватись та всі інструменти для тестування і необхідні для їхньої роботи додаткові застосунки.

Було сформовано основні метрики та вимоги до інструментів. Ці метрики використані для виявлення найкращого інструменту.

Першим для аналізу було розглянуто інструмент Semgrep. Semgrep має досить високий показник TPR , що свідчить про його здатність правильно виявляти 96.84% справжніх вразливостей у програмному коді. Це високий рівень ефективності, що вказує на надійність Semgrep у виявленні реальних загроз. Проте інструмент також має високий показник FPR , приблизно 57.74%, що означає, що понад половину позначених ним як потенційно небезпечних випадків насправді не є вразливостями.

Наступним було розглянуто інструмент SonarQube. Хоча його показники є нижчими за Semgrep, SonarQube має низький відсоток хибних спрацьовувань, що забезпечує майже таку ж загальну оцінку, як і в Semgrep. Важливо зазначити, що SonarQube має декілька версій: безкоштовну Community та кілька платних версій. Безкоштовна версія не підтримує такі мови, як C, C++, Objective-C, Swift, COBOL, на відміну від платних версій, що може бути важливим фактором при виборі інструменту.

Останнім розглянуто інструмент Horusec. Він показав найгірші результати серед трьох інструментів. Хоча його показник виявлення вразливостей є прийнятним, високий показник хибних спрацьовувань значно знижує його загальну ефективність. Це свідчить про те, що Horusec може створювати значну кількість помилкових попереджень, що може ускладнювати роботу розробників та відволікати їх від реальних проблем.

Таким чином, вибір між цими інструментами залежить від специфічних потреб і умов розробки. Semgrep є найкращим варіантом для швидкого та надійного виявлення вразливостей, незважаючи на високу частку хибних спрацьовувань. SonarQube пропонує збалансований підхід з низьким відсотком хибних спрацьовувань, але обмеженою підтримкою мов у безкоштовній версії. Через свою ефективність в підсистемі статичного тестування буде використовуватись саме Semgrep.

Для забезпечення всебічного захисту додатків, SAST слід використовувати в поєднанні з іншими заходами безпеки, такими як динамічний аналіз безпеки додатків (DAST), ручне тестування на проникнення та регулярний аудит безпеки. Такий комплексний підхід дозволяє створити багаторівневу систему захисту, яка мінімізує ризики та забезпечує високий рівень безпеки програмного забезпечення.

3. РЕАЛІЗАЦІЯ ПІДСИСТЕМИ СТАТИЧНОГО ТЕСТУВАННЯ

3.1 Обгрунтування вибору середовища для реалізації

Для втілення системи та її підсистеми необхідно вирішити, як і де їх реалізувати. Для нашої системи потрібна хмарна платформа, ринок пропонує досить шикорий вибір. Розглянемо деякі з них.

Microsoft Azure являє собою хмарну платформу, яка надає широкий спектр хмарних послуг обчислення, зберігання даних, аналітики, штучного інтелекту та інших хмарних послуг. Microsoft Azure дозволяє підприємствам будувати, розгортати та керувати застосунками та послугами на інфраструктурі Microsoft. Має широкий спектр послуг: віртуальні машини, контейнери, бази даних, аналітику даних, інтелектуальну аналітику, штучний інтелект, Інтернет речей та саме. Крім того, вона пропонує гнучкі моделі ціноутворення і інтегрується з різноманіттю інструментів для розробників, зокрема Visual Studio.

Серед інших продуктів, платформа пропонує компаніям Azure App Service, який допомагає створювати та розміщувати вебпрограми, мобільні серверні системи та RESTful API будь-якою мовою програмування у хмарі, не керуючи інфраструктурою [13].

Google Cloud Platform (GCP) - це хмарна платформа від компанії Google, яка надає широкий спектр хмарних послуг, включаючи обчислення, зберігання даних, штучний інтелект, аналіз даних, розробку програмного забезпечення та інші послуги.

За допомогою GCP клієнти можуть отримати доступ до комп’ютерних ресурсів, розміщених у центрах обробки даних Google по всьому світу, як безкоштовно, так і на платній основі по мірі використання сервісів [14].

Amazon Web Services (AWS) - це провідна хмарна платформа, яка пропонує широкий спектр послуг обчислення, зберігання даних, баз даних, аналітики, штучного інтелекту, Інтернету речей (IoT), розробки програмного забезпечення та інших хмарних рішень. AWS є частиною Amazon.com Inc., та вважається одним з найбільших та найпопулярніших постачальників хмарних послуг у світі.

Розглянемо декілька аргументів, чому використання Amazon Web Services (AWS) може бути кращим вибором порівняно з Google Cloud Platform (GCP) або Microsoft Azure:

− Більший досвід та стабільність. AWS є одним з найдосвідченіших та найдавніших постачальників хмарних послуг. Їхній рівень досвіду і стабільності може бути важливим фактором для бізнесів, особливо для тих, які потребують надійних і випробованих рішень.

− Широкий спектр послуг. AWS пропонує один з найширших асортиментів хмарних послуг, включаючи обчислення, зберігання даних, бази даних, аналітику, штучний інтелект, IoT та багато іншого. Це дозволяє підприємствам забезпечити всі свої потреби за допомогою одного постачальника, що спрощує управління та інтеграцію.

− Гнучкість і масштабованість. AWS надає широкий спектр конфігурацій і можливостей масштабування, що дозволяє бізнесам підлаштувати свої обчислювальні потреби до різних сценаріїв. Це включає в себе можливості автоматичного масштабування, що дозволяє динамічно змінювати обсяги ресурсів залежно від попиту.

− Глобальна присутність. AWS має широку мережу центрів обробки даних по всьому світу, що дозволяє бізнесам розгортати свої додатки та послуги в найближчих до їхніх користувачів регіонах. Це може поліпшити продуктивність та ефективність послуг.

Важливою вимогою до системи це її можливість бути описаною за допомогою IaC.

Інфраструктура як код (IaC) - це підхід до автоматизації управління інфраструктурою комп'ютерних систем через опис програмного коду, замість ручного налаштування або використання інтерактивних консолей.

Основні переваги інфраструктури як код включають:

− Автоматизація. Інфраструктура, описана у вигляді коду, може бути автоматично розгорнута та налаштована. Це зменшує ризики людських помилок та забезпечує відповідність налаштувань.

− Швидкість розгортання. За допомогою інфраструктури як коду, нові середовища можуть бути розгорнуті за допомогою автоматичного процесу. Це дозволяє швидко реагувати на зміни вимог і вимоги бізнесу.

− Масштабованість. Збільшення або зменшення ресурсів може бути автоматично здійснене за допомогою зміни коду, що дозволяє швидко адаптуватися до змін в навантаженні або потребах.

− Версіонування та контроль змін. Код інфраструктури може бути збережений у системі контролю версій, що дозволяє відстежувати зміни, відновлювати попередні версії та спільно працювати над інфраструктурою.

− Реюзабельність. Опис інфраструктури у вигляді коду дозволяє повторно використовувати компоненти, шаблони та бібліотеки для створення різних середовищ інфраструктури.

Для опису інфраструктури як коду буде використовуватись серів AWS CloudFormation.

AWS CloudFormation - це сервіс, який допомагає моделювати та налаштовувати ресурси AWS, аби витрачати менше часу на управління цими ресурсами і більше зосереджуватися на додатках, які працюють в AWS. Ми створюємо шаблон, який описує всі потрібні ресурси AWS, а CloudFormation піклується про забезпечення та налаштування цих ресурсів. Не потрібно окремо створювати і налаштовувати ресурси AWS і з'ясовувати, що від чого залежить [15].

За допомогою CloudFormation можна описувати вашу інфраструктуру у вигляді шаблонів, які містять опис різних ресурсів AWS, таких як віртуальні машини, бази даних, мережі, політики безпеки тощо.

Основні переваги AWS CloudFormation:

− Автоматизація розгортання. CloudFormation дозволяє автоматизувати процес розгортання інфраструктури.

− Відстеження змін. CloudFormation забезпечує відстеження змін до інфраструктури, що дозволяє легко відновлювати попередні версії, а також відстежувати та аудитувати всі зміни.

− Шаблони зі спільнотою. AWS CloudFormation має широкий вибір готових шаблонів, які можна використовувати для розгортання різноманітних ресурсів. Крім того, існують спільноти користувачів, які діляться своїми шаблонами та найкращими практиками.

− Інтеграція з іншими сервісами. AWS CloudFormation інтегрується з іншими сервісами AWS, що дозволяє легко управляти всіма аспектами інфраструктури, включаючи ресурси, що залежать від інших сервісів.

Для реалізації системи було створено схему інфраструктури з використанням різних інструментів і сервісів AWS, що включає головним чином наступні компоненти:

− Розробники починають процес, вносячи зміни до коду.

− AWS CodeCommit: зміни в коді зберігаються в репозиторії коду AWS CodeCommit.

− AWS CodePipeline: AWS CodePipeline автоматично розпочинає процес інтеграції та доставки (CI/CD) після внесення змін до репозиторію коду.

− Збірка:

а) SAST Check (Semgrep): На етапі збірки виконується статичний аналіз безпеки коду (SAST) за допомогою інструменту Semgrep. Semgrep перевіряє код на наявність вразливостей без його виконання.

− Розгортування:

а) Prepare build infra. Підготовка інфраструктури для збірки.

б) Execute build and run. Виконання збірки і запуск коду.

в) DAST Check (OWASP ZAP): виконується динамічний аналіз безпеки коду (DAST) за допомогою OWASP ZAP. OWASP ZAP тестує запущений код на наявність вразливостей у виконуваному середовищі.

− Amazon S3. Результати перевірки безпеки та інші артефакти зберігаються в Amazon S3.

− Amazon EventBridge. EventBridge використовує події, згенеровані під час процесу, для запуску подальших дій.

– Amazon SNS. За допомогою Amazon Simple Notification Service (SNS) розробники отримують повідомлення про результати сканування безпеки та інші важливі події по електронній пошті.

На рис. 3.1 зображено схему яка ілюструє процес інтеграції та розгортання коду для забезпечення безпеки та автоматизації в розробці програмного забезпечення.

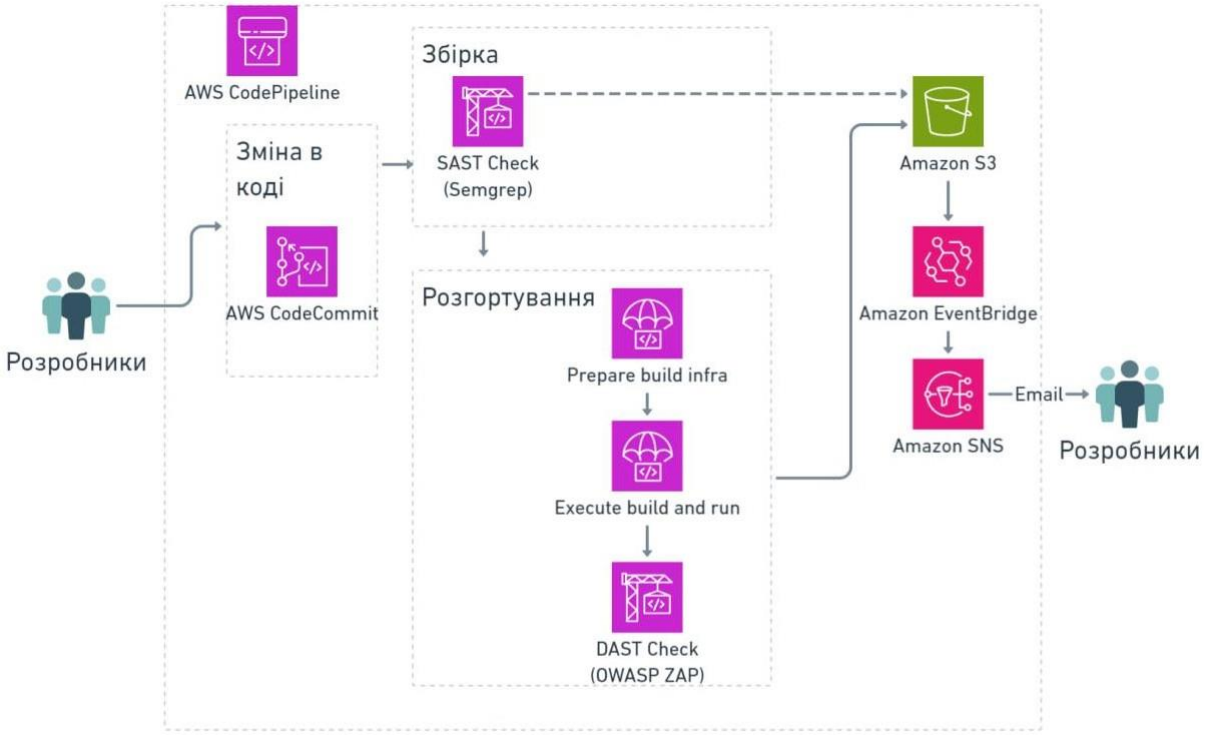


Рисунок 3.1 – Схема інфраструктури

На рис. 3.2 можна побачити шаблон взаємодії між різними сервісами AWS, який в даному випадку зосереджений на процесах статичного (SAST) та динамічного (DAST) аналізу безпеки застосунків. Як видно зі схеми, CodeBuild відіграє центральну роль у виконанні сканування, а S3 використовується для зберігання результатів. Події, пов'язані з цими процесами, відстежуються та обробляються за допомогою EventBridge, а сповіщення про стан сканування та його результати надсилаються через SNS.

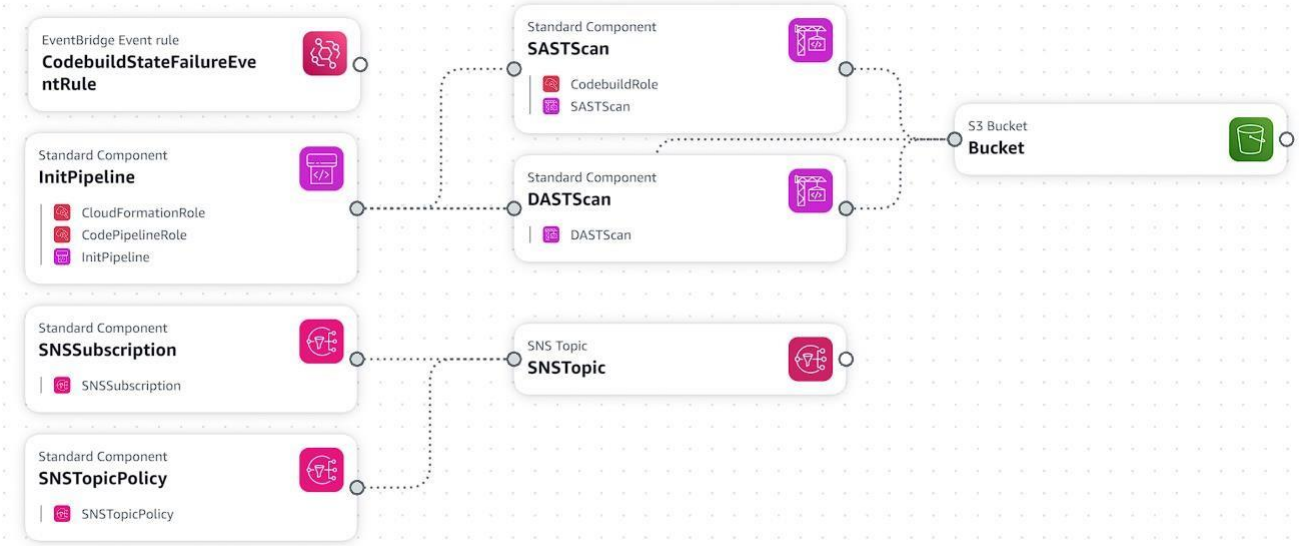


Рисунок 3.2 – Шаблон CloudFormation

3.2 Реалізація підсистеми статичного тестування за допомогою підходу IaC

Для налаштування автоматизованого статичного аналізу безпеки коду (SAST) у пайплайні CI/CD за допомогою AWS CodeBuild та CloudFormation, використовується елемент шаблону CloudFormation, який описує конфігурацію SAST сканування. Нижче описано конфігурацію елемента шаблону:

```
SASTScan:
    Type: AWS::CodeBuild::Project
    Properties:
      Artifacts:
        Type: S3
        Location: !Ref Bucket
        Name: Semgrep
        Packaging: ZIP
      Cache:
        Location: !Sub ${Bucket}/Semgrep.zip
        Type: S3
      BadgeEnabled: false
      Name: SASTCheck
      Environment:
        ComputeType: BUILD_GENERAL1_SMALL
```

```
        Image: aws/codebuild/amazonlinux2-x86_64-standard:2.0
        Type: LINUX_CONTAINER
        EnvironmentVariables:
          - Name: TechnologyStack
            Value: !Ref TechnologyStack
          - Name: Semgrep_CLI
            Value: !Ref Semgrep_CLI
      ServiceRole: !Ref CodebuildRole
      Source:
        Type: CODECOMMIT
        Location: !GetAtt CodeCommitRepository.CloneUrlHttp
        BuildSpec: SASTBuildconf.yaml
```

Цей код створює проект AWS CodeBuild для виконання SAST перевірок коду за допомогою Semgrep. Проект отримує код з репозиторію CodeCommit, використовує Docker-образ amazonlinux2-x86_64-standard:2.0 для середовища побудови, і виконує конфігурацію побудови, визначену у файлі SASTBuildconf.yaml. Результати побудови зберігаються у S3, і кеш також зберігається у S3 для оптимізації майбутніх побудов. Проект використовує роль IAM для надання необхідних дозволів.

Наступним є конфігураційний файл buildspec.yaml, що використовується в AWS CodeBuild для автоматизації процесу сканування безпеки коду за допомогою інструменту Semgrep. Він описує етапи побудови та команди, які повинні бути виконані на кожному з цих етапів.

```
phases:
  pre_build:
    commands:
      - echo Pre-build phase - ensuring environment setup pwd
      - apt-get update && apt-get install -y jq
      - echo Environment setup complete date
  install:
    commands:
      - echo Installing Semgrep - pwd
      - pip install semgrep
```

```
            - echo Semgrep installation complete date
        build:
          commands:
            - echo Starting Semgrep scanning date in pwd
            - semgrep --config auto --output report-semgrep.json --
json
            - echo Semgrep scan complete date
      artifacts:
        files:
          - report-semgrep.json
```

Файл складається з кількох основних розділів, які визначають дії на різних етапах процесу побудови. Перший розділ описує фази, pre_build фаза виконується перед початком основної збірки. Вона включає команди для налаштування середовища, оновлення списку пакетів та встановлення jq, інструменту для обробки JSON. Install фаза виконує встановлення необхідних інструментів або залежностей, а саме встановлення Semgrep за допомогою pip. Build фаза є основною, саме в ній виконується сканування коду, запускається Semgrep для перевірки коду з автоматичним конфігуруванням та збереженням результатів у файл report-semgrep.json.

Наступним розділом є розділ artifacts. Він визначає файли, які будуть збережені як артефакти після завершення збірки. У цьому випадку це файл report-semgrep.json, який містить результати сканування Semgrep.

Ця конфігурація забезпечує автоматизований процес збирання та аналізу програмного коду за допомогою інструменту Semgrep у середовищі CI/CD. Вона визначає три основні фази: підготовка до збірки, установка необхідних залежностей та сам процес збірки.

3.3 Обгрунтування об’єкту тестування

Для перевірки роботи розробленої підсистеми необхідно обрати застосунок який буде тестуватися. Для цих цілей ідеально підходить OWASP Juice Shop.

OWASP Juice Shop - це, мабуть, найсучасніший і найскладніший незахищений веб-додаток! Його можна використовувати для тренінгів з безпеки, ознайомчих демонстрацій, CTF і в якості піддослідного кролика для тестування інструментів безпеки [16].

Ось кілька причин, чому саме цей інструмент є кращим вибором для перевірки тестування:

− Відкритий код. Juice Shop є проектом з відкритим кодом, що дозволяє будь-кому вивчати його кодову базу, пропонувати покращення та використовувати його безкоштовно. Це також сприяє прозорості та надійності коду.

− Повнофункціональний веб-додаток. Juice Shop імітує реальний інтернет-магазин, включаючи різні функціональні модулі, такі як автентифікація користувачів, кошик покупок, історія замовлень тощо. Це дозволяє тестувальникам практикувати свої навички у реалістичному середовищі.

− Широкий спектр вразливостей. Juice Shop включає в себе широкий спектр поширених вразливостей, таких як SQL-ін'єкції, XSS, CSRF, недоліки автентифікації та авторизації, і багато інших. Це дозволяє тестувальникам навчатися виявляти та експлуатувати різні типи вразливостей.

− Документація та підказки. Проект добре документований, з детальними описами кожної вразливості та методами їх виявлення і експлуатації. Існують також підказки для початківців, які допомагають у вивченні основних концепцій безпеки.

− Підтримка різних платформ. Juice Shop може бути розгорнутий на різних платформах, включаючи Docker, Heroku, і локальні середовища розробки. Це робить його доступним для широкого кола користувачів з різними технічними налаштуваннями.

− Інтерактивне навчання. Інструмент підтримує інтерактивне навчання з вбудованими підказками та завданнями, які допомагають користувачам крок за кроком освоювати різні аспекти безпеки веб-додатків.

− Спільнота та підтримка. OWASP Juice Shop має активну спільноту розробників і користувачів, які готові допомогти з вирішенням проблем та

обговоренням нових ідей. Це створює додаткову цінність для тих, хто навчається, оскільки вони можуть отримувати підтримку від більш досвідчених колег.

− Безпека навчання. Використання Juice Shop дозволяє тестувальникам тренувати свої навички без ризику завдати шкоди реальним веб-додаткам або даним користувачів.

OWASP Juice Shop є повнофункціональним веб-додатком, який імітує реальний інтернет-магазин. Він надає користувачам можливість переглядати каталог товарів, де вони можуть знайти різноманітні продукти, включаючи електроніку, одяг та аксесуари. Користувачі можуть додавати ці товари до кошика, змінювати їх кількість або видаляти. Крім того, додаток підтримує реєстрацію та автентифікацію користувачів, що дозволяє створювати персональні облікові записи для збереження історії покупок та особистих даних.

Така комплексність функціоналу робить OWASP Juice Shop реалістичним інструментом для навчання та перевірки навичок тестування безпеки веб-додатків, адже він охоплює всі ключові аспекти, притаманні сучасним веб-додаткам.

3.4 Показ роботи підсистеми статичного тестування

Процес статичного тестування починається з того що розробник вносить зміни в код. Зміни коду зберігаються в репозиторій AWS CodeCommit. AWS CodePipeline автоматично розпочинає процес інтеграції та доставки (CI/CD) після внесення змін до репозиторію коду. На етапі збірки виконується статичний аналіз безпеки коду (SAST) за допомогою інструменту Semgrep. Це середовище використовується для розміщення веб-додатка OWASP Juice Shop, який вже знаходиться у репозиторії AWS CodeCommit. Розгортання додатка здійснюється за допомогою інструменту CloudFormation.

Етап SAST включає виконання сканування безпеки додатку. Це сканування виявляє потенційні вразливості та генерує звіт про результати. Якщо вразливості будуть знайдені, вони будуть внесені в звіт для подальшого аналізу та усуненню.

Якщо етап SAST завершується успішно, процес переходить до етапу розгортування, на якому буде виконуватись вже DAST, а результати роботи SAST

будуть збережені в Amazon S3. Вигляд успішно виконаного тестування зображено на рис. 3.3.

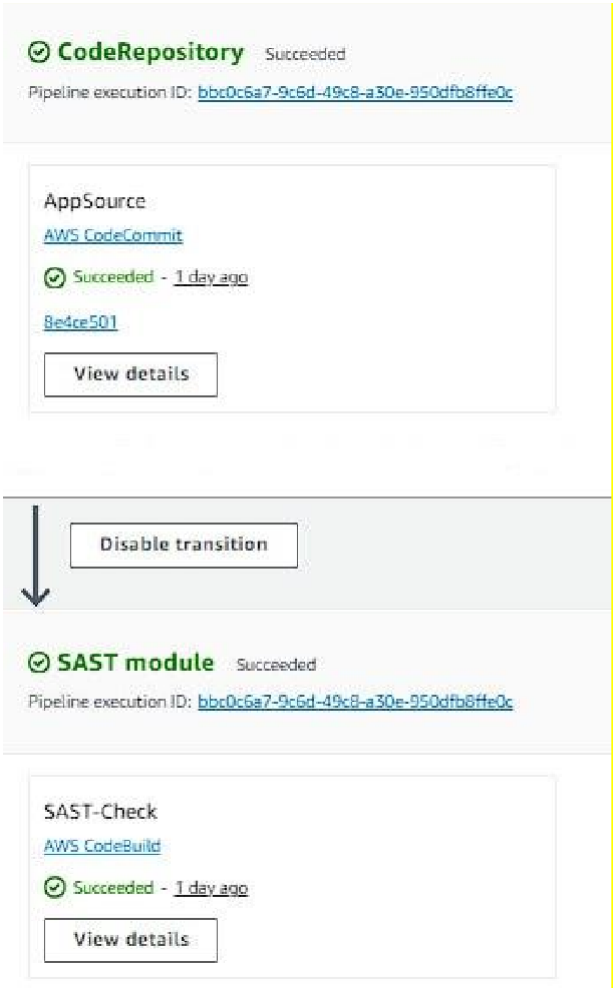


Рисунок 3.3 - Виконане тестування в пайплайні.

В результаті виконання отримаємо звіт який зберігається у сховищі S3. В звіті надаються наступні параметри:

− Check ID (Ідентифікатор перевірки) - унікальний ідентифікатор для конкретної перевірки або правила, яке було застосовано до коду. Це допомагає ідентифікувати, яке саме правило виявило проблему.

− Path (Шлях) - шлях до файлу в проекті, де була знайдена проблема. Це допомагає розробникам знайти конкретний файл, який потребує уваги.

− Start Line (Початкова лінія) - номер рядка у файлі, з якого починається проблема. Це дозволяє розробникам швидко перейти до початкової точки проблеми.

− End Line (Кінцева лінія) - номер рядка у файлі, на якому закінчується проблема. Це допомагає зрозуміти обсяг проблеми в коді.

− Message (Повідомлення) - детальний опис проблеми або знайденого питання. Це повідомлення зазвичай надає уявлення про те, що є проблемою і іноді підказує, як її вирішити.

− Severity (Серйозність) - вказує на рівень впливу, який проблема має на додаток. Звичайні рівні серйозності включають Низький, Середній, Високий і Критичний. Це допомагає пріоритизувати виправлення проблем на основі їхнього потенційного впливу.

− Category (Категорія) - тип виявленої проблеми, наприклад, безпека, продуктивність, найкращі практики тощо. Це категоризація допомагає групувати і розуміти природу проблем.

− Confidence (Впевненість) - міра впевненості, що виявлена проблема дійсно є проблемою. Рівні можуть включати Низький, Середній і Високий, що вказує на те, наскільки впевнено інструмент в тому, що це істинно позитивний результат.

− Impact (Вплив) - описує потенційні наслідки або шкоду, яка може виникнути, якщо проблему не виправити. Це часто корелює із серйозністю, але дає більше контексту про ефекти.

− Likelihood (Ймовірність) - вказує на ймовірність того, що проблема може бути експлуатована або спричинити проблеми. Рівні можуть включати Низький, Середній і Високий.

− Shortlink (Коротке посилання) - посилання на більш детальну документацію або ресурси, пов'язані з конкретною проблемою або правилом. Це може надати додатковий контекст або керівництво щодо усунення проблеми.

− Code (Код) - реальні рядки коду, які були позначені інструментом. Це надає розробникам прямий огляд проблемного коду, щоб полегшити його налагодження та виправлення.

Ці поля часто є частиною результатів статичного аналізу коду, який використовується для автоматичного сканування вихідного коду на потенційні вразливості, недоліки коду або порушення стандартів кодування.

Звід надається у вигляді json файлу, і через це досить проблематичний є прочитання інформації з нього. Для більш зручної роботи було написано скрипт який виводить основні поля у вигляді таблиць в зручному форматі html.

Розглянемо один фрагмент результатів тестування який зображено на рис. 3.4.

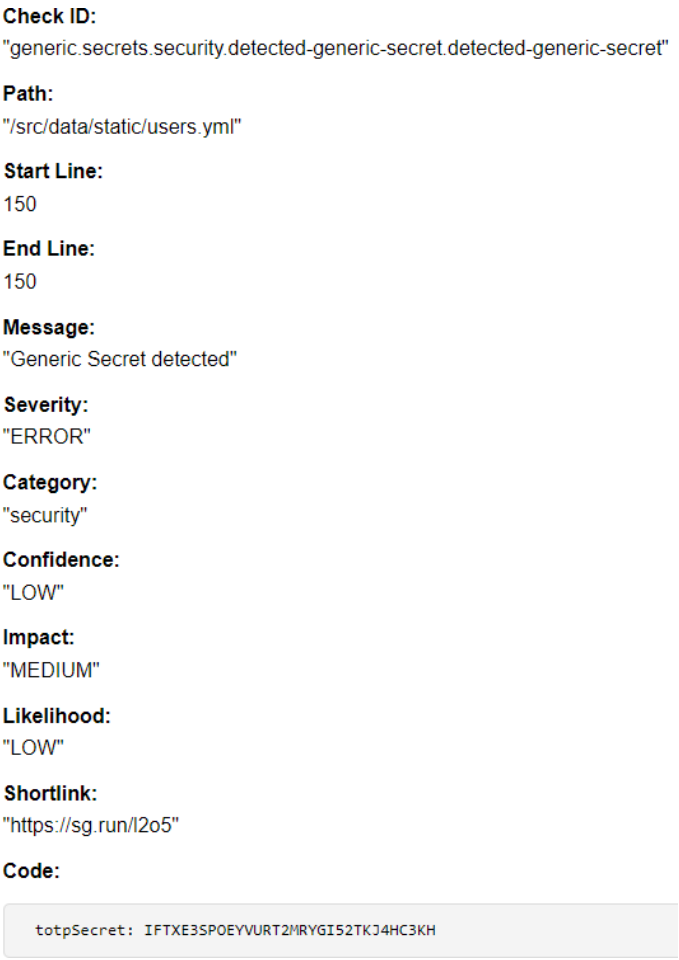


Рисунок 3.4 - Фрагмент результатів тестування

Давайте розглянемо цей фрагмент результатів та пояснимо його поля:

– Check ID (Ідентифікатор перевірки) - generic.secrets.security.detected-generic-secret.detected-generic-secret, це унікальний ідентифікатор правила, яке виявило проблему. Це правило стосується виявлення загальних секретів у коді, таких як ключі API, паролі тощо.

− Path (Шлях) - "/src/data/static/users.yml" це шлях до файлу в проекті, де була знайдена проблема. В даному випадку проблема знаходиться у файлі users.yml, який знаходиться в директорії src/data/static.

− Start Line (Початкова лінія) - номер рядка у файлі, з якого починається проблема. Проблема починається з рядка 150.

− End Line (Кінцева лінія) - номер рядка у файлі, на якому закінчується проблема. Проблема знаходиться в одному рядку, тобто закінчується на рядку 150.

− Message (Повідомлення) - Generic Secret detected ,у цьому випадку було виявлено загальний секрет, що може свідчити про те, що в коді присутній конфіденційний або приватний ключ.

− Severity (Серйозність) - вказує на рівень впливу, який проблема має на додаток. ERROR означає, що це серйозна проблема, яка потребує негайного вирішення.

− Category (Категорія) - тип виявленої проблеми. У цьому випадку це проблема, пов'язана з безпекою.

− Confidence (Впевненість) - LOW означає, що інструмент не повністю впевнений, що це істинно позитивний результат, і може знадобитися додаткова перевірка.

− Impact (Вплив) - MEDIUM означає, що це може мати середній вплив на додаток.

− Likelihood (Ймовірність) - LOW означає, що ймовірність експлуатації цієї проблеми є низькою.

− Shortlink (Коротке посилання) - https://sg.run/l2o5 посилання на правило.

− Code (Код) - totpSecret: IFTXE3SPOEYVURT2MRYGI52TKJ4HC3KH, у цьому випадку виявлено значення поля totpSecret, яке, ймовірно, є конфіденційним або приватним ключем, що не повинно бути збережене у відкритому коді.

Розробникам рекомендується перевірити і видалити або захистити цей секрет у коді.

Звіти в Semgrep відіграють ключову роль у забезпеченні безпеки та якості коду. Вони надають цінну інформацію про потенційні вразливості, помилки та

недоліки в коді, допомагаючи розробникам виявити та виправити проблеми на ранніх етапах розробки.

3.5 Розгляд проведенного тестування

В результаті проведення тестування було виявлено 65 вразливостей. З них 14 є вразливостями високого ступеня серйозності, 47 середнього ступеня та 4 низького ступеня серйозності. Серйозність (severity) визначає наскільки критичними є проблеми.

Кількість вразливостей по різним категоріям зображено в таблиці 3.1.

Таблиця 3.1. - Кількість виявлених вразливостей

Вразливість	Кількість випадків
Injection	15
XSS	7
Broken Authentication	4
Broken Access Control	20
Security Misconfiguration	8
Software and Data Integrity Failures	4
Vulnerable and Outdated Components	1
Insecure Design	4
XML External Entities (XXE)	1
Server-Side Request Forgery (SSRF)	1

Як відомо OWASP Juice Shop має 107 вразливостей, тобто підсистема статичного тестування виявила близько 60% всіх вразливостей що є досить високим показником.

3.6 Висновки до розділу

У цьому розділі було розглянуто реалізацію підсистеми статичного тестування в рамках системи забезпечення безпеки програмного забезпечення. Було описано процес вибору середовища для реалізації, обґрунтовано використання AWS як хмарної платформи, та детально розглянуто реалізацію

підсистеми статичного тестування з використанням підходу IaC (Інфраструктура як код).

В результаті проведеного аналізу було виявлено значну кількість вразливостей в застосунку OWASP Juice Shop, що підтверджує ефективність підходу статичного аналізу в виявленні потенційних проблем безпеки на ранніх етапах розробки.

Використання хмарних технологій та IaC дозволяє забезпечити автоматизацію та масштабованість процесу статичного тестування, що є важливим фактором для забезпечення безпеки сучасних програмних систем.

ВИСНОВКИ

У цій роботі було розглянуто важливість та актуальність проблеми забезпечення безпеки програмного забезпечення, особливо в контексті зростаючої кількості кібератак та ускладнення програмних систем. Було проведено аналіз методів та інструментів статичного тестування безпеки застосунків (SAST), що дозволило виявити їх переваги та недоліки.

Було досліджено основні функції SAST, такі як аналіз коду, сканування на основі правил, зменшення хибних спрацьовувань, інтеграція з інструментами розробки, звітування та перевірка відповідності. Проведено порівняльний аналіз популярних інструментів SAST, таких як Veracode, SonarQube та Fortify SCA, враховуючи їх тип, підтримувані мови, інтеграцію, сканування контейнерів, хибні спрацьовування, ціну, масштабованість та продуктивність.

Також було проведено експеримент з використанням інструментів Semgrep, SonarQube та Horusec на основі OWASP Benchmark. Визначено ключові метрики для оцінки ефективності, такі як показник істинно позитивних та хибно позитивних результатів, час сканування, кількість виявлених попереджень та загальна оцінка. Результати експерименту показали, що Semgrep має найкращі показники виявлення вразливостей, SonarQube відзначається низьким рівнем хибних спрацьовувань, а Horusec має найнижчу загальну ефективність.

Вибрано інструмент Semgrep для реалізації підсистеми статичного тестування. Було використано підхід Infrastructure as Code (IaC) та AWS CloudFormation для автоматизації розгортання та управління інфраструктурою. Підсистему було успішно протестовано на застосунку OWASP Juice Shop, виявивши значну кількість вразливостей.

Таким чином, у цій роботі було успішно досягнуто поставленої мети - підвищення безпеки застосунків за допомогою підсистеми статичного тестування. Розроблена підсистема дозволяє автоматизувати процес виявлення вразливостей у коді, що сприяє підвищенню ефективності та якості процесу розробки програмного забезпечення.

Semgrep, обраний для реалізації підсистеми статичного тестування, продемонстрував високу ефективність у виявленні вразливостей, що підтверджує його потенціал для використання у реальних проектах. Проте, слід враховувати, що Semgrep може генерувати значну кількість хибних спрацьовувань, що вимагає додаткової уваги та додаткової перевірки результатів ручними тестувальниками.

Використання підходу Infrastructure as Code (IaC) та AWS CloudFormation дозволило автоматизувати процес розгортання та налаштування інфраструктури для статичного тестування, що значно спрощує його впровадження та підтримку. Це також сприяє підвищенню надійності та відтворюваності результатів тестування.

Проте, слід зазначити, що статичний аналіз коду не є універсальним рішенням для забезпечення безпеки програмного забезпечення. Він має свої обмеження та не може виявити всі типи вразливостей, особливо ті, що пов'язані з динамічною поведінкою програми. Тому, для досягнення максимального рівня безпеки, статичний аналіз слід використовувати в поєднанні з іншими методами, такими як динамічний аналіз та ручне тестування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Рогозинський О.Б., Лукічов В. В. Функції та методи статичного тестування безпеки. Матеріали Всеукраїнської науково-практичної інтернет-конференції «Молодь в науці: дослідження, проблеми, перспективи (МН-2024)», Вінниця, 11-20 травня 2024 р. 2024. URL: https://conferences.vntu.edu.ua/index.php/mn/mn2024/paper/view/21607 (дата звернення: 21.05.2024).
2. Verma V. S. Implementing DevSecOps Practices: Supercharge your software security with DevSecOps excellence. Birmingham-Mumbai : Packt, 2023. 234 p..
3. OWASP Top Ten. URL: https://owasp.org/www-project-top-ten/ (accessed: 10.05.2024).
4. West J., Chess B. Secure Programming with Static Analysis. Boston : Addison-Wesley, 2007. 559 p..
5. Static Code Analysis. URL: https://www.veracode.com/security/static-code-analysis (accessed: 22.05.2024).
6. SonarQube 10.5 Documentation. URL: https://docs.sonarsource.com/sonarqube/latest/ (accessed: 22.05.2024).
7. Fortify Static Code Analyzer (SCA) Static Application Security Testing. URL: https://www.microfocus.com/media/data-sheet/fortify_static_code_analyzer_static_application_security_testing_ds.pdf (accessed: 22.05.2024).
8. Veracode State of Software Security v11. 2020. URL: https://info.veracode.com/report-state-of-software-security-volume-11.html (accessed: 22.05.2024).
9. The ShiftLeft Team 8 AppSec Metrics You Should Be Monitoring. 2022. URL: https://securityboulevard.com/2022/01/8-appsec-metrics-you-should-be-monitoring/ (accessed: 21.05.2024).
10. OWASP Benchmark. URL: https://owasp.org/www-project-benchmark/ (accessed: 22.05.2024).

11. Add Semgrep to CI. URL: https://semgrep.dev/docs/deployment/add-semgrep-to-ci (accessed: 22.05.2024).
12. Horusec overview. URL: https://docs.horusec.io/docs/overview/ (accessed: 22.05.2024).
13. Київстар Бізнес Що таке Microsoft Azure. Найважливіше про глобальну хмарну платформу. URL: https://hub.kyivstar.ua/articles/shho-take-microsoft-azure-najvazhlyvishe-pro-globalnu-hmarnu-platformu (дата звернення: 22.05.2024).
14. Що таке Google Cloud? URL: https://wiseit.com.ua/shho-take-google-cloud-platform/ (дата звернення: 22.05.2024).
15. What is AWS CloudFormation? URL: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html (accessed: 23.05.2024).
16. OWASP Juice Shop. URL: https://owasp.org/www-project-juice-shop/ (accessed: 23.05.2024).

ДОДАТКИ

Додаток А

Текст коду конфігурації. Ініціалізація пайплайну.

```
InitPipeline:
    Type: AWS::CodePipeline::Pipeline
    Properties:
      Name: CICDASTSAST
      RoleArn: !GetAtt CodePipelineRole.Arn
      Stages:
        - Name: CodeRepository
          Actions:
            - Name: CloudSecurityStaticCodeAnalyser
              ActionTypeId:
                Category: Source
                Owner: AWS
                Version: 1
                Provider: CodeCommit
              OutputArtifacts:
                - Name: CloudSecurityStaticCodeAnalyserOutput
              Configuration:
                BranchName: main
                RepositoryName: !GetAtt CodeCommitRepository.Name
              RunOrder: 1
        - Name: SecurityValidation
          Actions:
            - Name: SAST-Check
              InputArtifacts:
                - Name: CloudSecurityStaticCodeAnalyserOutput
              ActionTypeId:
                Category: Build
                Owner: AWS
                Version: 1
                Provider: CodeBuild
              OutputArtifacts:
                - Name: SASTCheckArtifact
              Configuration:
                ProjectName: !Ref SASTScan
              RunOrder: 1
        - Name: DAST-sub-system
          Actions:
            - Name: Preper infra
              InputArtifacts:
                - Name: CloudSecurityStaticCodeAnalyserOutput
              ActionTypeId:
                Category: Deploy
                Owner: AWS
                Version: 1
                Provider: CloudFormation
              OutputArtifacts:
                - Name: Preper-infra-ChangeSet
              Configuration:
                ActionMode: CHANGE_SET_REPLACE
                ChangeSetName: infraCBChangeSet
```

```
                RoleArn: !GetAtt CloudFormationRole.Arn
                Capabilities: CAPABILITY_NAMED_IAM
                StackName: !Sub infra-${AWS::StackName}
                TemplateConfiguration:
CloudSecurityStaticCodeAnalyserOutput::CFNTemplate/config-infra.json
                TemplatePath:
CloudSecurityStaticCodeAnalyserOutput::CFNTemplate/Main-Stack.yml
              RunOrder: 1
            - Name: ExecuteinfraChangeSet
              InputArtifacts:
                - Name: CreatedinfraChangeSet
              ActionTypeId:
                Category: Deploy
                Owner: AWS
                Version: 1
                Provider: CloudFormation
              OutputArtifacts:
                - Name: Run-and-Deploy
              Configuration:
                ActionMode: CHANGE_SET_EXECUTE
                ChangeSetName: Run-and-DeployhangeSet
                StackName: !Sub Run-and-Deploy-${AWS::StackName}
              Namespace: CloudFromationDeployNamespace
              RunOrder: 2
            - Name: DAST-Check
              ActionTypeId:
                Category: Build
                Owner: AWS
                Version: 1
                Provider: CodeBuild
              OutputArtifacts:
                - Name: DASTScanArtifact
              Configuration:
                ProjectName: !Ref DASTScan
                EnvironmentVariables:
'[{"name":"APP_URL","value":"#{CloudFromationDeployNamespace.URL}","typ
e":"PLAINTEXT"}]'
              RunOrder: 3
      ArtifactStore:
        Type:  S3
Location: !Ref Bucket    }
```

Додаток Б

Текст коду конфігурації. Ініціалізація SAST сканування.

```
SASTScan:
    Type: AWS::CodeBuild::Project
    Properties:
      Artifacts:
        Type: S3
        Location: !Ref Bucket
        Name: Semgrep
        Packaging: ZIP
      Cache:
        Location: !Sub ${Bucket}/Semgrep.zip
        Type: S3
      BadgeEnabled: false
      Name: SASTCheck
      Environment:
        ComputeType: BUILD_GENERAL1_SMALL
        Image: aws/codebuild/amazonlinux2-x86_64-standard:2.0
        Type: LINUX_CONTAINER
        EnvironmentVariables:
          - Name: TechnologyStack
            Value: !Ref TechnologyStack
          - Name: Semgrep_CLI
            Value: !Ref Semgrep_CLI
      ServiceRole: !Ref CodebuildRole
      Source:
        Type: CODECOMMIT
        Location: !GetAtt CodeCommitRepository.CloneUrlHttp
        BuildSpec: SASTBuildconf.yaml
```

Додаток В

Протокол перевірки
бакалаврської дипломної роботи
на наявність текстових запозичень

Назва роботи: Система тестування безпеки програмного застосунку. Частина 1. Підсистема статичного тестування
Автор роботи: Рогозинський Олександр Борисович
Тип роботи: бакалаврська дипломна робота.
Підрозділ кафедра захисту інформації ФІТКІ.
(кафедра, факультет)

Показники звіту подібності Unicheck

Оригінальність – 96,1%. Схожість – 3,9%.
Аналіз звіту подібності (відмітити потрібне):

√ 1. Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.

□ 2. Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її виконання автором. Роботу направити на розгляд експертної комісії кафедри.

□ 3. Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку _____ Валентина КАПЛУН
(підпис) (прізвище, ініціали)

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck щодо роботи.

Автор роботи _____ Олександр РОГОЗИНСЬКИЙ
(підпис) (прізвище, ініціали)

Керівник роботи _____ Віталій ЛУКІЧОВ
(підпис) (прізвище, ініціали)

Додаток Г

ІЛЮСТРАТИВНА ЧАСТИНА

СИСТЕМА ТЕСТУВАННЯ БЕЗПЕКИ ПРОГРАМНОГО ЗАСТОСУНКУ. ЧАСТИНА 1. ПІДСИСТЕМА СТАТИЧНОГО ТЕСТУВАННЯ

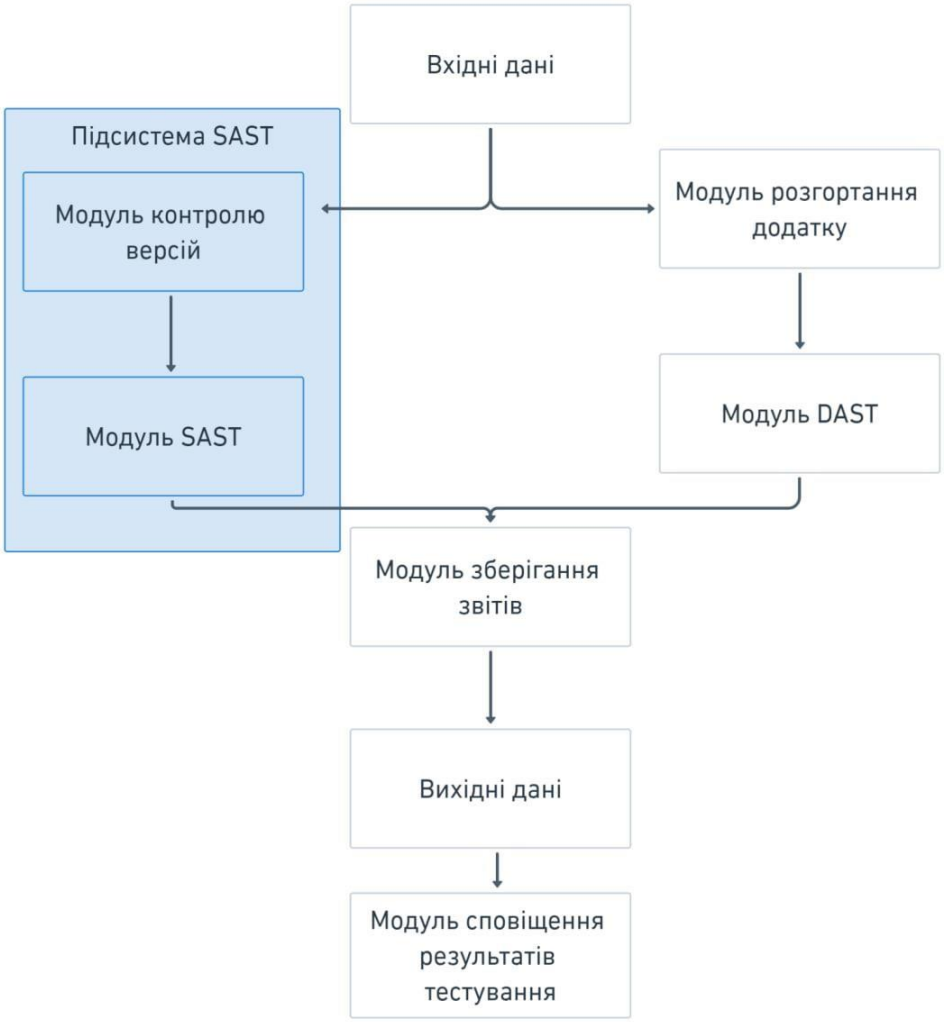
Виконав: студент 4 курсу групи 1БС-20б
спеціальності 125 Кібербезпека
_____ Олександр РОГОЗИНСЬКИЙ

Керівник: к. т. н., доцент, доцент каф. ЗІ
_____ Віталій ЛУКІЧОВ
«14» червня 2024 р.

АНАЛІЗ ІНСТРУМЕНТІВ SAST

	VeraCode	SonarQube	Fortify SCA
Тип	SaaS	Локальне, SaaS	Локальне, SaaS
Підтримувані мови	Широкий спектр включаючи Java, .NET, JavaScript, Python	Широкий спектр включаючи Java, .NET, JavaScript, Python	Широкий спектр включаючи Java, .NET, JavaScript, Python
Інтеграція	CI/CD інструменти, IDE, системи відстеження помилок	CI/CD інструменти, IDE, SCM	CI/CD інструменти, IDE, системи відстеження помилок
Сканування контейнерів	Обмежений	Обмежений	Так
Хибні спрацювання	Іноді виявляються хибні спрацьовування	Низький рівень хибних спрацьовувань	Середній рівень
Ціна	Вища вартість, підписка	Низька вартість для спільнотної версії, плагіни можуть збільшити вартість	Вища вартість, особливо для великих впроваджень
Масштабованість	Висока масштабованість, хмарний сервіс	Може вимагати багато ресурсів для великих проектів	Масштабований, але може бути дорогим
Продуктивність	Швидке сканування з хорошими показниками	Ефективний, може бути повільним для великих кодових баз	Висока продуктивність, але вимогливий до ресурсів

СХЕМА СИСТЕМИ ТЕСТУВАННЯ БЕЗПЕКИ



ПРОЦЕСИ ПІДСИСТЕМИ СТАТИЧНОГО ТЕСТУВАННЯ БЕЗПЕКИ

СХЕМА ІНФРАСТРУКТУРИ

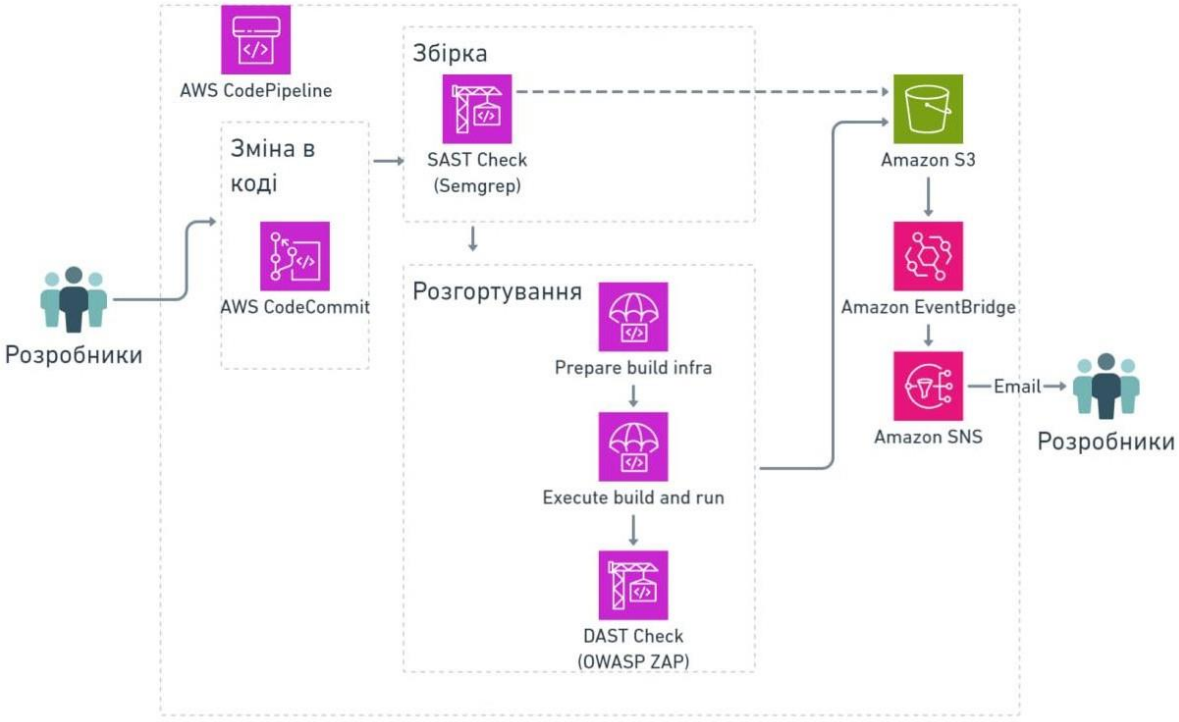
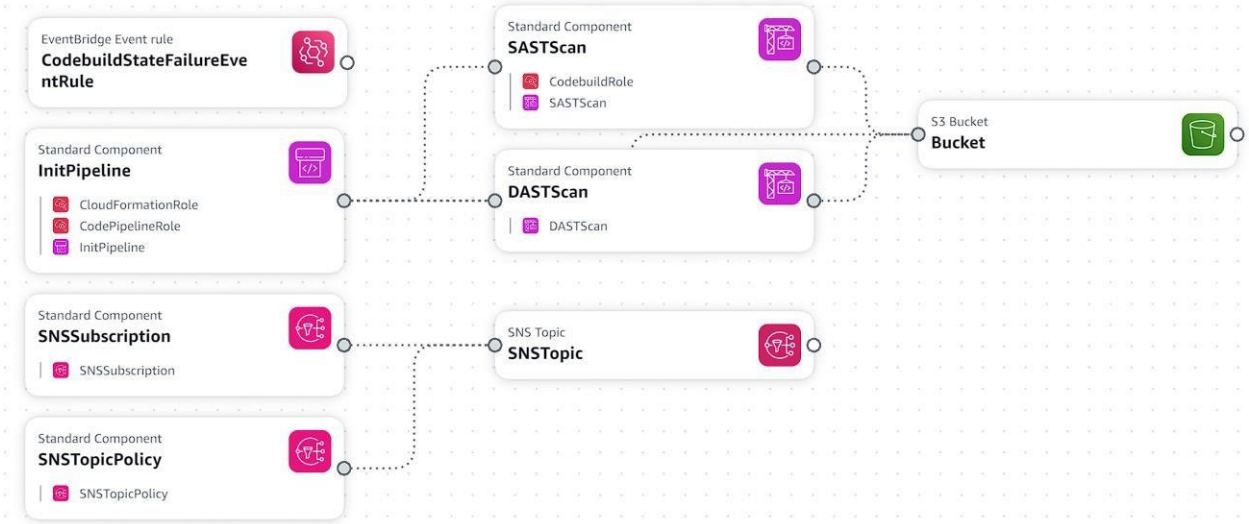


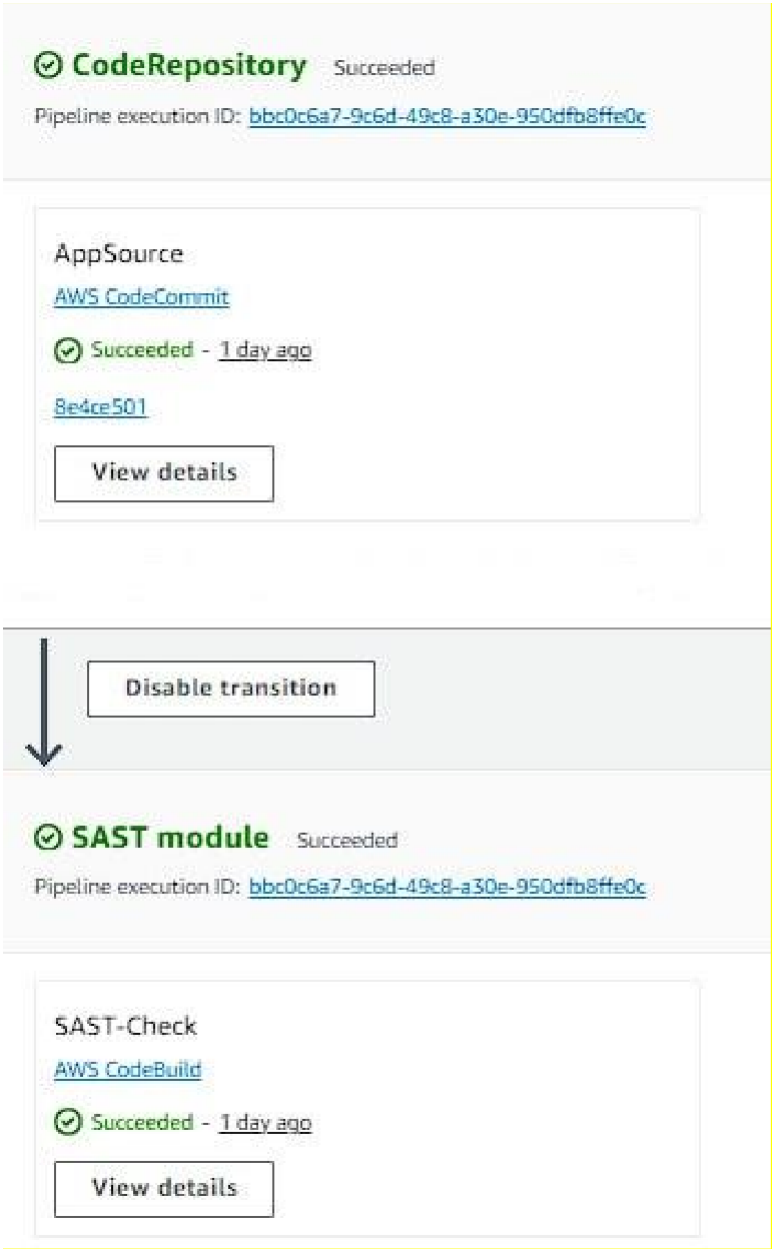
СХЕМА ШАБЛОНУ CLOUDFORMATION



КІЛЬКІСТЬ ВИЯВЛЕНИХ ВРАЗЛИВОСТЕЙ

Вразливість	Кількість випадків
Injection	15
XSS	7
Broken Authentication	4
Broken Access Control	20
Security Misconfiguration	8
Software and Data Integrity Failures	4
Vulnerable and Outdated Components	1
Insecure Design	4
XML External Entities (XXE)	1
Server-Side Request Forgery (SSRF)	1

ІЛЮСТРАЦІЇ ПРОЦЕСУ СТАТИЧНОГО ТЕСТУВАННЯ



Успішно виконані етапи підсистеми

ЗВІТ ВИКОНАНОГО СТАТИЧНОГО ТЕСТУВАННЯ

Check ID:
"generic.secrets.security.detected-generic-secret.detected-generic-secret"

Path:
"/src/data/static/users.yml"

Start Line:
150

End Line:
150

Message:
"Generic Secret detected"

Severity:
"ERROR"

Category:
"security"

Confidence:
"LOW"

Impact:
"MEDIUM"

Likelihood:
"LOW"

Shortlink:
"https://sg.run/l2o5"

Code:

```
totpSecret: IFTXE3SPOEYVURT2MRYGI52TKJ4HC3KH
```

Приклад вразливості зі звіту