Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління
(повна назва)

Кафедра Безпеки інформаційних технологій
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Автоматизація аналізу безпеки програмного коду за допомогою платформи Kubernetes
(тема)

Виконав: студент 2 курсу, групи БДІРм-19-1
Коханевич Є.Г.
(прізвище, ініціали)
Спеціальність 125 Кібербезпека
(код і повна назва спеціальності)
Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)
Освітня програма «Безпека державних інформаційних ресурсів»
(повна назва освітньої програми)

Керівник доц. Балагура Д.С.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Халімов Г.З.
(підпис) (прізвище, ініціали)

2020 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління
(повна назва)

Кафедра Безпеки інформаційних технологій
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 125 Кібербезпека
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна, або освітньо-наукова)

Освітня програма «Безпека державних інформаційних ресурсів»
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

студентові *Коханевичу Євгенію Григоровичу*
(прізвище, ім'я, по батькові)

1. Тема роботи *Автоматизація аналізу безпеки програмного коду за допомогою платформи Kubernetes*
затверджена наказом по університету від "22" жовтня 2020 р. № 1413Ст
2. Термін подання студентом роботи (проекту) 18.12.2020
3. Вихідні дані до роботи (проекту) Теоретичні дані про тестування безпеки програмного забезпечення, VirtualBox, Vagrant, Ubuntu iso образ, Visual Studio Code
4. Перелік питань, що потрібно опрацювати в роботі (зміст пояснювальної записки)
1. Тестування, як частина процесу розробки програмного продукту
2. Тестування безпеки програмного забезпечення
3. Платформа Kubernetes
3.1. Опис платформи Kubernetes
3.2. Kubernetes Оператор, як засіб автоматизації тестування
4. Розробка Kubernetes Оператора
4.1. Розгортання платформи Kubernetes
4.2. Тестуваннння створенного Kubernetes Оператора в середовищі розробки
Висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій *Презентаційний матеріал у вигляді слайдів.*

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів магістерської атестаційної роботи	Термін виконання етапів роботи	Примітка
1	*Отримання завдання*	*02.09.20*	
2	*Пошук літератури*	*04.09.20-28.09.20*	
3	*Дослідження існуючих методик розробки програмного забезпечення та місце тестування в них*	*28.09.20-15.10.20*	
4	*Аналіз методів тестування безпеки програмного коду*	*15.10.20-11.11.20*	
5	*Аналіз платформи Kubernetes та Kubernetes операторів, як засобу автоматизації роботи додатків*	*11.11.20-25.11.20*	
6	*Розробка та тестування Kubernetes оператора для аналізу безпеки програмного коду*	*25.11.20-04.12.20*	
7	*Оформлення пояснювальної записки*	*04.12.20-16.12.20*	

Дата видачі завдання _____ _____20__ р.

Студент _____
(підпис)

Керівник роботи (проекту) _____ __доц. Балагура Д.С.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Атестаційна робота містить: 84 с., 4 табл., 18 рис., 2 додатки, 11 джерел.

KUBERNETES, ТЕСТУВАННЯ БЕЗПЕКИ, KUBERNETES ОПЕРАТОР, SAST, DAST, IAST, SDLC, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ.

Об’єкт дослідження – автоматизація тестування безпеки програмного забезпечення в процесі розробки на платформа Kubenetes.

Предмет дослідження – інструменти для автоматизації процесу тестування безпеки програмного забезпечення в процесі розробки.

Мета роботи – дослідження методів тестування безпеки програмного коду та розробка рішення автоматизації на платформі Kubernetes.

Основним завданням роботи є розробка ефективного засобу автоматизації тестування безпеки програмного коду на базі платформи Kubernetes.

В роботі запропоноване рішення автоматизації з допомогою Kubernetes оператора. Розроблено програмний засіб Kubernetes Оператор для автоматизації роботи інструмента статичного аналізу безпеки програмного коду Spotbugs. Розгорнутий кластер Kubernetes в якому створено тестове середовище розробки програмного забезпечення та проведено тестування розробленого рішення.

Область використання результатів – тестування безпеки програмного коду в процесі розробки.

ABSTRACT

Attestation work contains 84 p., 4 tables, 18 pic., 2 applications, 11 sources.

KUBERNETES, SECURITY TESTING, KUBERNETES OPERATOR, SAST, DAST, IAST, SDLC, SOFTWARE.

The property development – the automation of software security testing in the development process on the Kubenetes platform.

Purpose – to study the methods of software code security testing and to develop an automation solution on the Kubernetes platform.

The Kubernetes cluster has been deployed in which a software development test environment has been created and the developed solution has been tested.

The result – the Kubernetes Operator software has been developed to automate the operation of the Spotbugs software static security analysis tool.

Scope of usage – security testing of software code during development.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

SDLC — життєвий цикл розробки програмного забезпечення

ІБ — інформаційна безпека

ІС — інформаційна система

ІТ — інформаційні технології

ОС — операційна система

ПЗ — програмне забезпечення

API — прикладний програмний інтерфейс

RAD — швидка розробка додатків

DDS — специфікація проектної документації

SRS — специфікацію вимог до програмного забезпечення

HTTP — протокол передачі гіпертекстових документів

CI/CD — безперервна інтеграція / безперервна доставка

SAST — статичне тестування безпеки програмного додатку

DAST — динамічне тестування безпеки програмного додатку

IAST — інтерактивне тестування безпеки програмного додатку

URL — уніфікований локатор ресурсу, адреса ресурсу

IDE – інтегроване середовище розробки

DevOps — методологія: розробка та підтримка

DevSecOps — методологія: розробка, безпека та підтримка

ВСТУП

На сьогоднішній день сфера розробки програмного забезпечення зростає дуже значними темпами. Кожного дня з’являються нові розробники та компанії, які готові надавати послуги по створення програмного забезпечення. З іншої сторони, так само активно розвивається і сфера кібер-криміналітету в світі, тому вкрай важливо забезпечувати надійний рівень безпеки додатків, що розробляються.

Засоби, які використовуються для підвищення безпеки програмного забезпечення зазвичай охоплюють пошук, виправлення, попередження та запобігання вразливостей системи безпеки. Як правило, для цього використовуються засоби та методи виявлення таких вразливостей в процесі життєвого циклу програмного забезпечення, як планування, проектування, розроблення, розгортання, модифікація та технічне обслуговування. В процесі роботи над кожним з цих етапів користувач зіштовхується з деякими проблемами, які здебільшого виникають у результаті розробки програм, які розроблялися для забезпечення певних потреб користувача.

Основним методом аналізу безпеки програмного забезпечення є тестування. Методи тестування безпеки усувають уразливості у безпеці програм. Вважається, що тестування безпеки реалізується протягом усього життєвого циклу програми, щоб уразливості були знешкоджені правильно та у встановлений час.

1 ТЕСТУВАННЯ, ЯК ЧАСТИНА ПРОЦЕСУ РОЗРОБКИ ПРОГРАМНО ПРОДУКТУ

1.1 Життєвий цикл розробки програмного забезпечення (SDLC)

Успішне завершення проекту розробки програмного забезпечення залежить не тільки від команди розробників або технічних знань, воно включає групу процесів, людей, дій та відповідні необхідні навички окремої групи. Тому правильно побудований та керований процес розробки дозволяє забезпечити ефективне використання ресурсів та часу, а як результат, успішний результат проекту.

Життєвий цикл розробки програмного забезпечення (SDLC) - це процес, який виконується для програмного проекту в рамках розробки програмного забезпечення. Він складається з детального плану, що описує, як розробляти, підтримувати, замінювати та підтримувати або вдосконалювати конкретне програмне забезпечення. Життєвий цикл визначає методологію підвищення якості програмного забезпечення та загального процесу розробки [2]. Для вирішення складності програмних проектів розроблено різноманітні моделі життєвого циклу. Це постало проблемою вибору найбільш підходящого для окремого проекту. Використання відповідної моделі життєвого циклу програмного забезпечення може підвищити ефективність та результативність розробки програмного забезпечення. Вибір, як правило, залежить не тільки від типу проекту, а й від цілей, яких потрібно досягти. Тому існують механізми для визначення найбільш підходящого методу простим та ефективним способом.

Початкові концепції SDLC були зароджені в 1960-х роках для розробки широкомасштабних функціональних бізнес-систем в епоху великих бізнес-конгломератів. У перші часи комп’ютерного програмування єдині моделі, які використовувались для розробки таких складних речей, були в будівельній та обробній галузях. Таким чином, було цілком зрозуміло, що структуровані

підходи, що використовуються в цих галузях, повинні застосовуватися і до розробки комп'ютерних систем. Наприклад, у сферах будівництва бізнес-аналітики спочатку зрозуміють вимоги клієнта. Кроки виконуються архітекторами, що розробляють рішення та інженерами для розробки та будівництва будівель, мостів чи доріг. Підводячи до останнього кроку, протестуйте, вдосконаліть і підпишіть сертифікат на продукцію. Отже, в 1970-х роках велика група бізнес-аналітиків будівельної та обробної промисловості потрапила в область обчислювальної техніки, щоб проаналізувати вимоги бізнесу до нових систем . Значна кількість інженерів також увійшла в область обчислювальної техніки як програмісти [2].

Це дуже традиційний процес розробки, який йде послідовно від початку до кінця. Деякі процедури накладання неминучі, такі як тестування та вдосконалення. Тим не менше, схрещування між основними фазами не є загальним явищем. У старі часи методи програмування були дуже складними, а мови програмування було непросто вивчити та маніпулювати ними. Таким чином, розробка комп’ютерних систем слідувала структурованому, а послідовний підхід мав багато сенсу.

За останні 50 років комп’ютерні системи відігравали важливу роль у корпораціях. Від розсилки листів з листоношами до розсилки електронної пошти через Інтернет, від заповнення паперових заявок до електронних заявок, від аудиту фінансових журналів до електронних таблиць, що зберігаються в корпоративних системах, кожен аспект тісно пов'язаний з інформаційними технологіями. Отже, численні компанії сприймають це дуже серйозно і витрачають значні гроші, ресурси та зусилля на безпеку та управління інформаційними технологіями.

Протягом 5 десятиліть розвивалися концепції розробки програмного забезпечення, і з'явилися нові уявлення та дизайни в орієнтованих на клієнта додатках та рішеннях. Кожен підхід має свої плюси і мінуси, сильні та слабкі сторони. Реальним є той факт, що одне єдине рішення вже не може помістится

в мільйонах організацій через різний досвід, структуру, відповідальність, бажання та цілі. Проте можна знайти спільні цілі на кожному етапі розробки програмного забезпечення. Не повинно бути великих різниць у тому, як роботи описуються, організовуються та управляються з різними організаційними передумовами та вимогами. Тому сучасні життєві цикли розробки програмного забезпечення є достатньо гнучкими для використання у різних типах бізнесу, продуктів та послуг.

Не обмежуючись переліченими нижче моделями, існують різні моделі, що використовуються в процесі життєвого циклу розробки програмного забезпечення.

— Водоспад (Waterflow)
— Ітеративні (Iterative)
— Гнучка (Agile)
— Швидка розробка додатків (RAD)

Життєвий цикл розробки програмного забезпечення складається з детальних етапів та заходів, які описують, як розробляти, розробляти, підтримувати, замінювати, змінювати, вдосконалювати, тестувати або навіть запускати програмне забезпечення.

Діяльність можна розбити на дуже детальний рівень, але водночас їх можна згрупувати за п’ятьма основними категоріями: планування, проектування, розробка, випробування та розгортання.

На рисунку 1.1 наведено графічне зображення, яке відображає типовий життєвий цикл розробки програмного забезпечення.

Рисунок 1.1 — Життєвий цикл розробки програмного забезпечення

Планування, як правило, відбувається після інновації чи ініціації, яку ініціює група бізнес-кінцевих споживачів або спонсора, який визначив потребу чи можливість. На етапі планування визначається обсяг або межа понять. Вивчення техніко-економічного обґрунтування продукції у фінансовій, операційній та технічній областях проводитимуть старші члени групи за участю бізнес-користувачів. План забезпечення якості та управління ризиками також готується на етапі планування для мінімізації будь-яких непередбачуваних ризиків. Документація з бізнес-кейсів повинна бути готова на цьому етапі, щоб узагальнити всі ідеї та мати цілісний погляд на повний план [2].

Розробка продукту починається з чіткого визначення вимог. Документ про специфікацію вимог до програмного забезпечення (SRS), який складається з усіх деталей вимог до продукту, повинен бути затверджений клієнтами або замовниками до початку проектування продукту.

Маючи в своєму розпорядженні SRS, буде запропоновано більше одного дизайну архітектури продукту на основі вимог SRS. Вони будуть задокументовані молодшими членами команди у специфікації проектної

документації (DDS) та передані старшим членам, зацікавленим сторонам проекту для ознайомлення. DDS буде оцінюватися на основі різних критеріїв, але не обмежуючись бюджетом, часом, зручністю користування, ризиком, інтеграцією тощо.

Після того, як було обрано найкращий або найбільш відповідний дизайн, впровадження починається негайно. Програмісти повинні розробляти програмне забезпечення відповідно до DDS і одночасно слідувати стандартам кодування, визначеним компанією. Інструменти програмування повинні бути обмежені тими, що надаються компанією, аби всі програмісти могли узгоджувати свої роботи. Функціональна специфікація (FS) повинна бути написана програмістами для запису всіх функцій, які надаються на технічному рівні.

Тестування програмного забезпечення слід проводити на всіх етапах як підетап. Тим не менше, дві основні з них повинні зробити програмісти, кінцеві користувачі та експерти із забезпечення якості. Причина полягає в тому, що програмісти найкраще знають, як працює програма, і тому вони можуть визначити найбільш вразливі ділянки програмного забезпечення. Кінцеві користувачі звертатимуть більше уваги на свої рутинні завдання, що може допомогти забезпечити відповідність програмного забезпечення вимогам. І останнє, але не менш важливе: експерти з контролю якості досліджують програмне забезпечення в цілому з різних точок зору, таких як архітектура, безпека, інтеграція з іншими системами тощо. Як результат, для трьох груп тестувальників слід підготувати кілька різних типів планів тестування. проводити на етапі тестування [1].

Перше, що потрібно зробити на етапі розгортання, - це перевірити всі запущені тестові кейси, щоб забезпечити успішне виконання програмного забезпечення, повноту та коректність.

Потрібно прийняти остаточне рішення, якщо програмне забезпечення слід розміщувати у виробничому середовищі, і тому на цьому етапі слід вимагати схвалення від керівництва. План розгортання (ПЗ) повинен бути чітко

визначений та затверджений для внесення будь-яких змін, які збираються внести. Також слід підготувати документальну документацію, таку як Посібник з встановлення, Посібник з адміністрування та Посібник користувача. Члени групи підтримки повинні бути готові відповісти на всілякі запитання щодо програмного забезпечення. Нарешті, Плани дій на випадок непередбачених ситуацій (CP) слід створювати відповідно до готового програмного забезпечення. Для нещодавно впровадженого програмного забезпечення загальним рішенням є перенесення дня запуску програмного забезпечення для переробки та повторного тестування. Для запущеного програмного забезпечення, швидше за все, можна повернутися до попередньої версії або відкласти день запуску, щоб виправити дефекти.

1.2 Тестування програмного забезпечення

Тестування відіграє значну роль у досягненні та оцінці якості програмного продукту. Оцінку якості програмного забезпечення можна розділити на дві широкі категорії, а саме статичний та динамічний аналіз. Статичний аналіз означає, що він базується на вивченні ряду документів, а саме документів вимог, програмних моделей, проектних документів, вихідного коду тощо. Основні методи, що використовуються у статичному аналізі, включають перегляд коду, перевірку, проходження, аналіз алгоритму та доказ правильності [1]. Це не передбачає фактичного виконання коду, що розробляється. Динамічний аналіз програмної системи передбачає фактичне виконання програми з метою виявлення можливих збоїв програми. Також спостерігаються поведінкові та робочі властивості програми. Програми виконуються як з типовими, так і з ретельно підібраними вхідними значеннями. Часто вхідні дані програми можуть бути непрактично великими. Основні цілі тестування включають те, що система працюватиме належним чином, система не працюватиме неналежним чином, знизити ризик поломки системи та знизити вартість тестування [1].

Перша мета охоплює тестові випадки, щоб перевірити, чи система буде працювати належним чином з усіма зазначеними операціями, цей тест має на меті перевірити основний потік моделі використання, і вона охоплює позитивний сценарій тестування. Друга мета охоплює тестові випадки, щоб система вийшла з ладу , тому перевірте всі альтернативні потоки, зазначені під час використання моделі, і вона охоплює негативний сценарій тестування. Третя ціль охоплює тестові випадки для відмови або явного збою системи, щоб знати межі та межі системи, так само як і максимальна система ємність з точки зору швидкості обробки, розподілу пам'яті, обробки переривань тощо, може бути відома. Четверта мета зосереджена на розробці мінімальних тестових випадків, щоб охопити максимальне тестування системи, майже неможливо протестувати всю комбінацію тестових випадків, отже, завдання тестування - охопити всю систему тестування з мінімальною кількістю тестових випадків та оптимальне використання ресурсів із зменшенням пов’язаних із цим витрат [3]. Генерація тестових кейсів є одним з найважливіших завдань при тестуванні. Тестовий випадок - це проста пара <введення, очікуваний результат>. У системах без стану: вихід повністю залежить від вводу, приклад включає в себе обчислення куба чисел, користувачеві просто потрібно надати вхід в систему. У державно-орієнтованих системах: вихід однієї послідовності виступає як вхід для іншої послідовності тощо, приклад включає телефонну комутаційну систему, банкомат тощо.

Тестові кейси, як зазначино в [3], можуть формуватися з різних джерел інформації, оскільки процес розробки програмного забезпечення велика кількість такі артефакти, як специфікація вимог, проектний документ та вихідний код. Для того, щоб створити ефективні тести за нижчою ціною, розробники тестів вивчають та аналізують різні джерела інформації: вимоги та функціональні характеристики, вихідний код, вхідні та вихідні домени, експлуатаційний профіль та модель несправності, яка передбачає некоректність системи . За цією моделлю дизайнер може передбачити наслідки певних

несправностей. Для того, щоб протестувати програму, інженер-тестувальник повинен виконати послідовність тестових дій, які включають, визначити мету, яку потрібно перевірити, вибрати вхідні дані, обчислити очікуваний результат, налаштувати середовище виконання програми, виконати програму, проаналізувати результати тесту [1].

Двома широкими поняттями в тестуванні, для проектування тестів, є тестування білої та чорної скриньок. Методи випробовування в білій скриньці також називають методами структурних випробувань. Як випливає з назви, ця методологія тестування забезпечує розуміння програми та створення тестових кейсів. Тестер має доступну структуру вихідного коду та інші деталі програми, тому на його основі можна підготувати тестові кейси. Це дозволяє побачити, що відбувається всередині програми [1]. Наприклад, розглянемо банкомат банківської системи, тут кодом, що стоїть за системою, буде надання операцій з перевірки PIN-кодів, переліку пунктів меню, перевірки балансу, зняття готівки, прив'язки картки adhar тощо. зосередьтеся на правильній роботі кожного модуля, перш ніж він тестується за допомогою користувальницького інтерфейсу. Тут основним джерелом вхідних даних для тестових випадків є основна структура вихідного коду. Цей метод тестування може застосовуватися на рівні модулів та на рівні інтеграції, коли код розробляється та тестується паралельно.

Методики тестування в чорній скриньці називають методами функціонального тестування. Як випливає з назви, тестувальник не має уявлення про тестується програму. Тестер розуміє вимоги та перевіряє їх за допомогою користувальницького інтерфейсу системи. Тестер не знає коду, що стоїть під тестуваним інтерфейсом. Ця методологія тестування розглядає, які доступні вхідні дані для програми та які очікувані результати повинні бути результатом кожного введення [1]. Наприклад, розглянемо систему банкоматів банківських систем, тестер розуміє можливі операції, що підтримуються банкоматом, готує набір входів і очікуваних результатів для перевірки кожної операції, на момент тестування тестер має доступ лише до інтерфейсу користувача, джерело код не

надається тестувальнику. Тут основним джерелом вхідних даних для тестових випадків є кількість входів та очікувані результати для тестування інтерфейсу користувача. Цей метод тестування може бути застосований при тестуванні на рівні системи, де всі інтерфейси підготовлені та перевірені для кожної функціональності. Основною відмінністю між тестуванням чорної скриньки та білої скриньки є сфери, на яких вони вирішили зосередитись. Найпростішими словами, тестування чорної скриньки орієнтоване на результати. Якщо дія вживається, і вона дає бажаний результат, то процес, який фактично був використаний для досягнення цього результату, не має значення. З іншого боку, тестування білої скриньки стосується деталей. Він фокусується на внутрішній роботі системи, і лише тоді, коли всі шляхи були перевірені, то тестування можна вважати завершеним [1].

1.3 Безпека в процесі розробки програмного продукту

Хоча компанії часто хочуть випустити новий код якомога швидше, щоб максимізувати можливості на ринку, ця стратегія іноді не може належним чином врахувати проблеми безпеки. Деякі компанії можуть виявити ненавмисні уразливості, які можуть серйозно скомпрометувати власні корпоративні дані, а також дані своїх клієнтів. Деякі з найбільш серйозних порушень, які з'явилися в заголовках газет за останні роки, сталися через те, що задіяні компанії не належно розставили пріоритети щодо питань безпеки досить рано в SDLC [4].

Оскільки усвідомлення важливості безпеки додатків зросло за останні роки, все більше компаній почали враховувати проблеми безпеки раніше в SDLC. Роблячи це, вони можуть краще пом'якшити потенційні ризики, виявити помилки раніше, раніше виявити проблеми з користувацьким досвідом та зменшити витрати, пов'язані з усуненням усіх цих проблем пізніше в процесі розробки програмного забезпечення. DevSecOps, орієнтована на безпеку, еволюція популярної концепції DevOps проектування та розгортання

програмного забезпечення, прагне чітко вбудувати найкращі практики безпеки програм раніше в SDLC [5].

Кіберзлочинці все частіше орієнтуються на веб-додатки, тому компанії повинні надавати пріоритети проблемам безпеки раніше в SDLC. Це особливо вірно, якщо програмне забезпечення, про яке йдеться, є критично важливим. Використання переваг сканера безпеки веб-додатків та проведення інших форм перевірки безпеки веб-додатків на початку процесу допомагає вашому бізнесу зменшити ризик, вирішити виникаючі проблеми, перш ніж вони стануть головними головними болями, і зменшити витрати [4].

SDLC є ефективною методологією для проектування та створення програмного забезпечення, але особливо яскраво виглядає, коли всі зацікавлені сторони ставлять пріоритети в питаннях безпеки та продумано включають тестування безпеки на початку процесу. Застосовуючи безпечний підхід до SDLC та заохочуючи ефективну співпрацю між командами, бізнес може вивести на ринок високоякісне програмне забезпечення за менший час та з меншою кількістю головних болей.

1.4 Визначення задач дослідження

1. Провести аналіз питання безпеки під час розробки програмного забезпечення, шляхом визначення основних етапів в життєвому циклі розробки програмного забезпечення та місця безпеки на кожному етапі. Провести аналіз проблеми автоматизації тестування безпеки в процесі розробки програмних додатків.
2. Провести огляд платформи Kubernetes, як найбільш популярного середовища для життєвого циклу розробки програмного забезпечення. Детально розглянути Kubernetes оператори та їх можливе використання для автоматизації тестування безпеки.
3. Розробити Kubernetes оператор для автоматизації роботи інструмента тестування безпеки на прикладі сканера статичного тестування безпеки програмного забезпечення Spotbugs.
4. Провести розгортання кластера Kubernetes для подальшого тестування. Розгорнути створений оператор в кластері Kubernetes. Створити середовище розробки програмного забезпечення на базі інструмента непреривної інтеграції Jenkins. Розробити Jenkins пайплайн для автоматизації життєвого циклу програмного забезпечення та інтегрувати в нього тестування безпеки програмного коду з допомогою сканера безпеки, який був розгорнутий з допомогою створеного Kubernetes оператора. Провести декілька тестових запусків пайплайна для перевірки надійності роботи оператора.

2 ТЕСТУВАННЯ БЕЗПЕКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Безпечний цикл розробки програмного забезпечення (Secure SDLC)

Що стосується створення, випуску та обслуговування програмного забезпечення, більшість організацій мають добре організований процес. Однак, коли справа стосується безпеки цього програмного забезпечення, не так вже й багато. Багато команд розробників досі сприймають безпеку як втручання - щось, що створює перешкоди та змушує їх переробляти, не даючи їм виходити на ринок нових цікавих функцій. Але небезпечне програмне забезпечення піддає бізнес зростаючому ризику. Нові цікаві функції не захистять вас або ваших клієнтів, якщо ваш продукт пропонує вразливі місця для хакерів. Натомість вашій команді потрібно інтегрувати безпеку у весь життєвий цикл розробки програмного забезпечення (SDLC), щоб вона забезпечувала, а не гальмувала, доставку на ринок високоякісних високозахищених продуктів [4].

Життєвий цикл розробки програмного забезпечення (SDLC) – це основа для процесу побудови програми від початку до виведення з експлуатації. Протягом багатьох років з’явилося безліч моделей SDLC – від водоспаду та ітерації до нещодавно гнучких та CI / CD, які збільшують швидкість та частоту розгортання.

Загалом життєвий цикл розробки програмного забезпечення включає такі етапи:

— Планування та вимоги.
— Архітектура та дизайн.
— Планування випробувань.
— Кодування.
— Тестування та результати.
— Випуск та обслуговування.

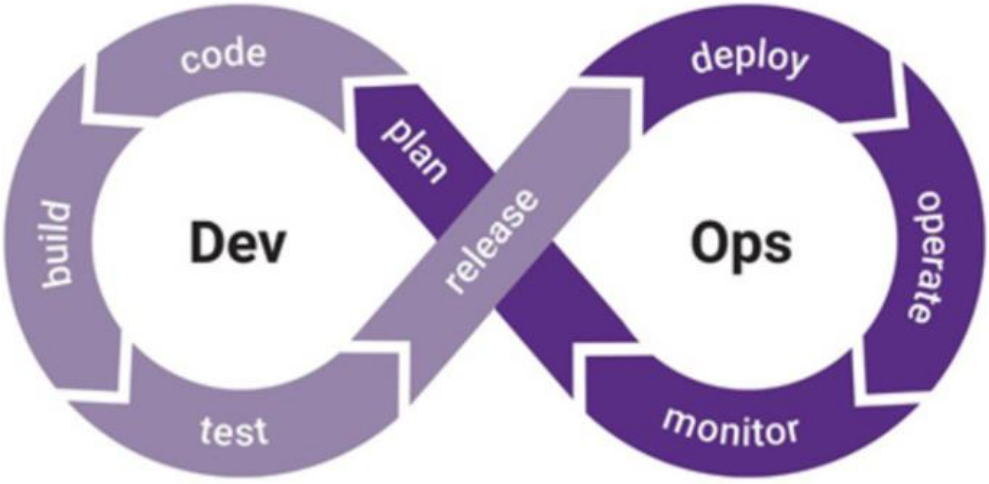

Рисунок 2.1 - Життєвий цикл розробки програмного забезпечення

Раніше організації, як правило, виконували діяльність, пов’язану з безпекою, лише в рамках тестування - наприкінці життєвого циклу. В результаті цієї пізньої техніки, вони не знайдуть помилок, недоліків та інших вразливих місць, поки ті не стануть набагато дорожчими та заберуть багато часу. Гірше того, що вони взагалі не знайдуть будь-яких вразливих місць безпеки.

Тож набагато краще, не кажучи вже про швидше та дешевше, інтегрувати тестування безпеки в SDLC, а не лише в кінці, щоб допомогти виявити та зменшити вразливі місця на ранній стадії, ефективно будуючи безпеку. Діяльність із забезпечення безпеки включає аналіз архітектури під час проектування, огляд коду під час кодування та побудови, а також тестування на проникнення перед випуском [5]. Ось основні переваги безпечного підходу SDLC:

— ваше програмне забезпечення є більш безпечним, оскільки про безпеку постійно турбуються.

— усі зацікавлені сторони знають про міркування безпеки.

— ви виявляєте недоліки дизайну рано, ще до їх кодування.

— ви зменшуєте свої витрати завдяки ранньому виявленню та усуненню дефектів.

— ви зменшуєте загальні внутрішні бізнес-ризики для вашої організації.

Важливо розуміти процеси, які організація використовує для побудови безпечного програмного забезпечення, оскільки якщо процес не зрозумілий, важко визначити його слабкі та сильні сторони. Також корисно використовувати загальні рамки для керівництва вдосконаленням процесів та оцінювати процеси на основі загальної моделі для визначення сфер вдосконалення. Моделі процесів сприяють загальним заходам організаційних процесів протягом усього життєвого циклу розробки програмного забезпечення (SDLC). Ці моделі визначають багато технічних та управлінських практик [4]. Хоча дуже мало з цих моделей були розроблені з нуля для вирішення питань безпеки, є вагомі докази того, що ці моделі стосуються належних практик програмної інженерії для управління та побудови програмного забезпечення.

Навіть коли організації відповідають певній моделі процесу, немає гарантії того, що створене ними програмне забезпечення не містить ненавмисних уразливих місць безпеки чи навмисних шкідливих кодів. Однак, ймовірно, існує більша ймовірність побудови безпечного програмного забезпечення, коли організація дотримується надійних практик програмного забезпечення з акцентом на хороший дизайн, якісні практики, такі як перевірки та огляди, використання ретельних методів тестування, належне використання інструментів, управління ризиками, проект управління та управління людьми [5].

Взагалі кажучи, захищений SDLC передбачає інтеграцію тестування безпеки та інших видів діяльності в існуючий процес розробки. Приклади включають написання вимог безпеки поряд з функціональними вимогами та проведення аналізу архітектурного ризику на етапі проектування SDLC.

Безпечний SDLC важливий, оскільки важлива безпека додатків. Часи випуску продукту в дику природу та усунення помилок у наступних виправленнях минули. Тепер розробники повинні усвідомлювати потенційні проблеми безпеки на кожному кроці процесу. Це вимагає інтеграції безпеки у

ваш SDLC способами, які раніше не потрібні. Оскільки кожен може потенційно отримати доступ до вашого вихідного коду, вам потрібно переконатись, що ви кодуєте з урахуванням потенційних вразливостей. Таким чином, наявність надійного та безпечного процесу SDLC є критично важливим для забезпечення того, щоб ваш додаток не піддавався атакам хакерів та інших недоброзичливих користувачів.

Впровадження безпеки SDLC впливає на кожну фазу процесу розробки програмного забезпечення. Для цього потрібен спосіб мислення, орієнтований на безпечну доставку, підняття питань на вимогах та етапах розвитку, коли вони виявляються. Це набагато ефективніше - і набагато дешевше, ніж чекати, поки ці проблеми безпеки виявляться в розгорнутому додатку. Безпечні процеси життєвого циклу розробки програмного забезпечення включають безпеку як складову кожного етапу SDLC.

Незважаючи на те, що вбудовування безпеки в кожну фазу SDLC - це, перш за все, спосіб мислення, який потрібно викласти кожному, міркування щодо безпеки та відповідні завдання насправді будуть істотно відрізнятися залежно від фази SDLC.

Використовується багато захищених моделей SDLC, але однією з найвідоміших є життєвий цикл розробки системи безпеки Microsoft (MS SDL) , де описано 12 практик, які організації можуть застосувати для підвищення безпеки свого програмного забезпечення. Також організація NIST опублікувала остаточну версію своєї програми розробки безпечного програмного забезпечення, яка фокусується на процесах, пов’язаних із безпекою, які організації можуть інтегрувати в свої існуючі SDLC.

Життєвий цикл розвитку надійної обчислювальної безпеки (SDL) - це процес, який Microsoft прийняла для розробки програмного забезпечення, яке має протистояти атакам безпеки. Процес додає ряд заходів, спрямованих на безпеку, і результати на кожному етапі процесу розробки програмного забезпечення Microsoft. Ці заходи безпеки та результати включають визначення

вимог до функцій безпеки та заходів забезпечення на етапі вимог, моделювання загроз для ідентифікації ризиків безпеки на етапі проектування програмного забезпечення, використання засобів сканування коду статичного аналізу та оглядів коду під час впровадження, а також тестування, зосереджене на безпеці, включаючи тестування Fuzz, на етапі тестування [4]. Додатковий захист включає остаточний огляд нового, а також застарілого коду на етапі перевірки. Нарешті, на етапі випуску команда центральної безпеки Microsoft проводить остаточний огляд безпеки.

Корпорація Майкрософт розширила SDL завдяки обов'язковому навчанню з питань безпеки для персоналу, що розробляє програмне забезпечення, з показниками безпеки та наявним досвідом з питань безпеки за допомогою команди центральної безпеки Microsoft. Корпорація Майкрософт повідомляє про обнадійливі результати продуктів, розроблених із використанням SDL, що вимірюється кількістю критичних та важливих бюлетенів безпеки, випущених корпорацією Майкрософт для продукту після його випуску.

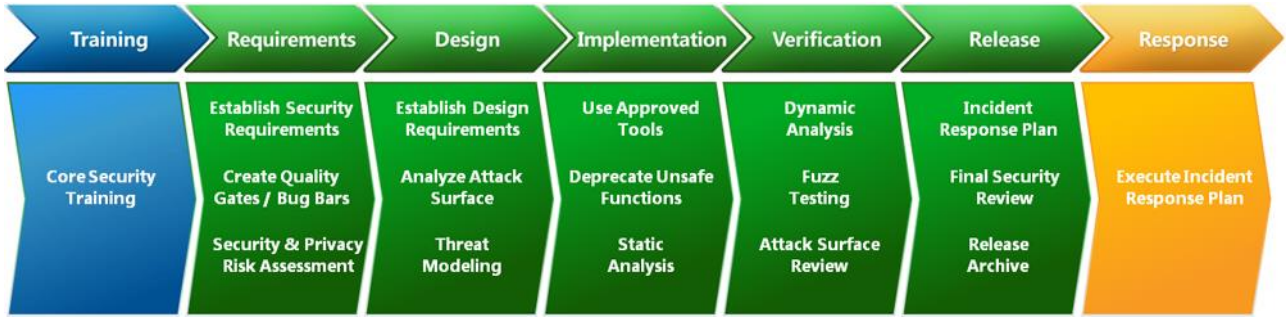

Рисунок 2.2 – Безпечний цикл розробки програмного забезпечення

2.2 Тестування безпеки в процесі розробки програмного продукту

2.2.1 Тестування, як частина безпечної розробки

Загальна схема створення інформаційних систем складається, як правило, з одних і тих самих модулів і процесів [4]:

— управління проектом у вигляді координації зусиль проектної команди, спрямованих на досягнення цілей проекту оптимальним шляхом;

— постановка задачі у вигляді визначення вимог і наступних робіт з ними;

— управління змінами в проекті: зміна може стосуватися як безпосередньо самих вимог до системи, так і торкатися організаційну схему процесу, і можуть породжуватися або самим замовником (бізнес-аналітиком), або бути наслідком виявлених в інформаційній системі дефектів;

— розробка ПЗ, як безпосереднє кодування програмної реалізації функціональних вимог і проектування схем збереження і руху інформації в інформаційній системі;

— тестування ПЗ - процес, вирішальний завдання верифікації відповідності вимог висунутих до інформаційної системи та їх програмної реалізації;

— експлуатація ПЗ як сума завдань, спрямована на забезпечення технічної і технологічної підтримки процесу роботи інформаційної системи, включаючи підтримку і необхідне системне адміністрування.

Як бачимо, процес розробки інформаційної системи складається з декількох взаємозв'язаних модулів, якими вже в свою чергу і оперують автори методологій і підходів, зміщуючи пріоритети між напрямками або змішуючи завдання декількох напрямів (пропонуючи, наприклад, здійснення завдань тестування в рамках діяльності щодо безпосередньої розробки програмної реалізації і т.д.).

Якість програмного продукту характеризується набором властивостей, що визначають, наскільки продукт «хороший» з точки зору зацікавлених сторін, таких як замовник продукту, спонсор, кінцевий користувач, розробники і тестувальники продукту, інженери підтримки, співробітники відділів маркетингу, навчання і продажів [5]. Кожен з учасників може мати різне уявлення про продукт і те, наскільки він хороший чи поганий, тобто про те,

наскільки висока якість продукту. Таким чином, постановка задачі забезпечення якості продукту виливається у завдання визначення зацікавлених осіб, їх критеріїв якості і потім знаходження оптимального рішення, що задовольняє цим критеріям. Тестування є одним з найбільш усталених способів забезпечення якості розробки програмного забезпечення і входить в набір ефективних засобів сучасної системи забезпечення якості програмного продукту.

З технічної точки зору тестування полягає у виконанні програми на деякій множині вихідних даних і звірці одержуваних результатів із заздалегідь відомими (еталонними) з метою встановити відповідність різних властивостей і характеристик програми замовленим властивостями.

Тестування програмного забезпечення – це перевірка відповідності між реальною та очікуваною поведінкою програми, що здійснюється на кінечному наборі тестів, вибраних певним чином [4]. Тестування є одним з етапів розробки програмного забезпечення.

Тестувати програмні додатки стає все важче, оскільки продовжує зростати їх технічна і функціональна складність. На жаль, технологія більшості процесів тестування не встигає за новими типами додатків. Виникає невідповідність піддає ризику якість програм та бюджет проекту - процес тестування потребує перегляду.

Для більшості проектів цей процес включає тестування коду програми з очікуваними результатами. Потім розробники «фіксують» додаток до тих пір, поки він не забезпечить потрібний результат. Або відбувається коригування очікуваного результату, якщо виникла помилка. Такий підхід фокусується на коді програми та його поведінці на безлічі тестових умов. Виявляється, що ретельне тестування додатка вимагає великої кількості тестових умов. При такому підході до тестування передбачається, що якість програми є функцією від кількості тестів - чим більше тестів, тим краще якість.

Сучасні програми мають більше можливостей завдяки взаємодії багатьох модулів. Число тестових умов, необхідних для звернення до додатків цього типу,

може перевищити бюджет і вимоги плану. Це відбувається через неефективність процесу тестування, зосередженого на дослідженні коду закінченого додатка. Тільки організації, здатні витримати такі витрати і гнучкий часовий графік, можуть і далі збільшувати тривалість тестування і відношення часу тестування до часу розробки [4].

Як вже зазначалося, традиційні технології тестування орієнтовані на код закінченого додатка (тобто додаток може тестуватися лише після того, як зібраний). Цей підхід виявляється неефективним з точки зору як якості виконуваної роботи, так і бюджету. Усунення помилок при завершенні процесу розробки обходиться дорожче і вимагає більше часу, ніж їх виправлення на більш ранніх стадіях (аналіз, проектування і т.д.). Ризик збою програмного забезпечення в результаті цих змін також збільшується на завершальних стадіях розробки програми. Одночасно зі збільшенням ціни і ризику зменшуються можливості розробника щодо внесення змін [5]. Очевидно, що помилки слід знаходити і виправляти якомога раніше.

Ідея пошуку помилок у момент, коли закінчено кодування програми, припускає розміщення моменту виявлення помилки безпосередньо в процес розробки, де, знову-таки, ціна і ризик високі, а вносити зміни вже складно. Невідповідність між стадією виявлення, часом і ціною виправлення цих помилок робить існуючі процеси тестування все менш ефективними для сучасних додатків.

Уразливості залишають програми відкритими до будь-якого використання. Методи тестування безпеки усувають уразливості у безпеці програм. Вважається, що тестування безпеки реалізується протягом усього життєвого циклу програми, щоб уразливості були знешкоджені правильно та у встановлений час. Є випадки, у яких тестування часто виконується як наслідок, у кінці життєвого циклу [5].

Сканери уразливостей та зокрема сканер веб-застосунків, інакше відомих як інструменти тестування, історично використовувались організаціями та

консультантами з безпеки для автоматичного тестування безпеки HTTP запитів-відповідей. Але це не замінює потребу в фактичному огляді вихідного коду. Початковий код програм може бути перевірений вручну або автоматично. Враховуючи індивідуальний звичайний розмір програм людський мозок не в змозі провести аналіз потоку даних з необхідним аналізом для повного контролю програми, щоб знайти програмні вразливості. Людський мозок спрямований на фільтрацію, переривання та звіт про вихід автоматизованих інструментів аналізу вихідного коду, доступних у комерційному відношенні, а також про можливі шляхи через скомпільовану кодову базу для виявлення уразливостей на рівні виникнення головної причини [5].

Існує багато видів автоматизованих інструментів для ідентифікації програмних уразливостей. Деякі з них вимагають великого досвіду проведення експертизи безпеки, а інші використовуються для повного автоматичного застосування. Результати залежать від типу інформаційних елементів (вихідної, бінарної, HTTP — трафіку, конфігурації, бібліотек, з'єднань), що надають інструменту якість аналізу і виправлення уразливостей. Звичайні технології для ідентифікації програмної вразливості включають:

— Статичне тестування безпеки (SAST, англ. Static Application Security Testing) — це технологія, яка здебільшого використовує інструмент аналізу початкового коду (англ. Source Code Analysis Tool). Метод аналізує безпеку коду програми перед її запуском і використовується для посилення коду. Цей метод дає меншу кількість неточностей, але вимагає доступу до початкового коду програми.

— Динамічне тестування безпеки (DAST, англ. Dynamic Application Security Testing) — це технологія, яка може знайти видиму вразливість пропускаючи URL ресурсу через автоматичний сканер. Цей метод дуже масштабний, добре інтегрований та швидкий.[джерело?] Недоліки DAST полягають у складності конфігурації і високої імовірності помилкового спрацювання.

— Інтерактивне тестування безпеки додатка (IAST, англ. Interactive Application Security Testing) — це таке тестування, яке оцінює додатки з використанням ПЗ. Ця методика дозволяє IAST комбінувати сили обох методів SAST і DAST, та забезпечувати доступ до коду, HTTP — трафіку, бібліотек, зворотніх зв'язків та інформації про конфігурацію. Деякі продукти IAST вимагають реалізації атаки, у той час як інші можуть використовуватися під час нормального тестування [4].

2.2.2 Статитичний аналіз безпеки програмного забезпечення (SAST)

Статичне тестування безпеки додатків (SAST) або статичний аналіз - це методологія тестування, яка аналізує вихідний код для виявлення вразливостей системи безпеки, які роблять програми сприйнятливими до атак. SAST сканує програму перед компіляцією коду. Це також відоме як тестування білих ящиків.

SAST відбувається дуже рано в життєвому циклі розробки програмного забезпечення (SDLC), оскільки він не вимагає робочої програми і може мати місце без виконання коду. Це допомагає розробникам виявляти вразливості на початкових етапах розробки та швидко вирішувати проблеми, не порушуючи збірки та не передаючи вразливості до остаточного випуску програми.

Існує три основних типи тестування SAST: аналіз вихідного коду, аналіз байтового коду та необроблений аналіз двійкового коду. Рішення безпеки SAST можна інтегрувати безпосередньо в середовище розробки, дозволяючи розробникам постійно контролювати свій код і швидко пом'якшувати вразливі місця при виявленні. Оскільки засоби безпеки SAST дають розробникам зворотний зв'язок у режимі реального часу під час кодування, вони можуть виправляти проблеми до того, як вони переходять у наступну фазу SDLC, виявляючи та виправляючи проблеми набагато швидше, ніж пізніше в SDLC [4].

Інструменти SAST дають розробникам зворотний зв'язок у реальному часі під час кодування, допомагаючи їм вирішити проблеми, перш ніж передати код на наступну фазу SDLC. Це запобігає тому, щоб проблеми, пов’язані з безпекою,

розглядалися як додаткові задуми. Інструменти SAST також надають графічні зображення знайдених проблем, від джерела до потоку. Вони допоможуть вам простіше орієнтуватися в коді. Деякі інструменти вказують на точне розташування вразливостей та виділяють ризикований код. Інструменти також можуть надати глибокі вказівки щодо того, як виправити проблеми та найкраще місце в коді для їх усунення, не вимагаючи глибокої експертизи доменів безпеки.

Розробники також можуть створювати персоналізовані звіти, які їм потрібні, за допомогою інструментів SAST; ці звіти можна експортувати в автономному режимі та відстежувати за допомогою інформаційних панелей. Організоване відстеження всіх проблем безпеки, про які повідомляє інструмент, може допомогти розробникам швидко усунути ці проблеми та випустити програми з мінімальними проблемами. Цей процес сприяє створенню безпечного SDLC [5].

Важливо зазначити, що інструменти SAST повинні запускатися в додатку регулярно, наприклад, під час щоденних / щомісячних збірок, кожного разу, коли код реєструється, або під час випуску коду.

Розробники значно перевищують кількість працівників служби безпеки. Для організації може бути складним завданням знайти ресурси для перевірки коду навіть на частці своїх програм. Ключовою силою інструментів SAST є можливість аналізу 100% кодової бази. Крім того, вони набагато швидші, ніж перевірки захищеного коду вручну, що виконуються людьми. Ці інструменти можуть сканувати мільйони рядків коду за лічені хвилини [5]. Засоби SAST автоматично з високою довірою визначають критичні вразливості, такі як переповнення буфера, ін'єкція SQL, сценарії між сайтами та інші. Таким чином, інтеграція статичного аналізу в SDLC може дати значні результати в загальній якості розробленого коду.

SAST також відомий як "тестування білої скриньки", тобто він перевіряє внутрішні структури або роботу програми, на відміну від її функціональності. Він працює на тому ж рівні, що і вихідний код, для виявлення вразливостей.

Оскільки аналіз SAST проводиться перед компіляцією коду і без його виконання, цей інструмент може бути застосований на ранніх етапах життєвого циклу розробки програмного забезпечення (SDLC). Більшість інструментів SAST підтримують основні веб-мови: PHP, Java та .Net та деякі форми C, C ++ або C #.

До переваг SAST належать:

— Інструменти SAST виявляють дуже складні вразливості на перших етапах розробки, які можна швидко усунути;

— Оскільки воно встановлює особливості проблеми, включаючи рядок коду, це полегшує виправлення;

— Його можна інтегрувати в існуюче середовище в різні моменти циклу розробки програмного забезпечення;

— Для вивчення коду потрібно небагато, і це порівняно з ручним аудитом;

Недоліками SAST є:

— Не всі компанії чи приватні особи готові надати дані для аналізу двійкового або байт-коду та вихідного коду;

— Розгортання технології в масштабі може бути складним завданням;

— Він схильний моделювати поведінку коду неточно. Тому розробникам доводиться мати справу з багатьма помилковими спрацюваннями та помилковими негативними даними;

— Динамічно набрані мови створюють проблеми; Інструменти SAST повинні семантично розуміти багато рухомих фрагментів коду, які можуть бути написані різними мовами програмування.

— Він не може перевірити програму в реальному середовищі, тому вразливості в логіці програми або незахищені конфігурації не виявляються.

Інструменти SAST є дуже цінною технологією, але не замінюють інші методи. Розробники застосовували комбінацію методів протягом усього процесу, щоб проводити оцінки та ловити недоліки перед початком виробництва [4].

Засоби аналізу вихідного коду , які також називаються Інструментами статичного тестування безпеки програм (SAST), призначені для аналізу вихідного коду або компільованих версій коду, щоб допомогти знайти недоліки безпеки.

Деякі інструменти починають переходити в IDE. Що стосується типів проблем, які можна виявити під час самої фази розробки програмного забезпечення, це потужний етап у життєвому циклі розробки для використання таких інструментів, оскільки він забезпечує негайний зворотний зв'язок з розробником щодо проблем, які вони можуть вводити в код під час коду сам розвиток. Цей негайний відгук дуже корисний, особливо якщо порівнювати з виявленням вразливостей набагато пізніше в циклі розробки.

Таблиця 2.1 – Популярні інструменти статичного сканування безпеки коду

Інструмент	Мови програмування	Доступність
ABASH	Bash	Безкоштовний
Astree	C	Комерційний
C/C++test	C, C++	Безкоштовний
JTest	Java	Безкоштовний
CxSAST	Java, JavaScript, PHP, C#, VB.NET, VB6, ASP.NET, C/C++, Apex, Ruby, Perl, Objective-C.	Комерційний
CodeCenter	C	Комерційний
CodeSonar	C, C++	Комерційний
FindBugs	Java, Groovy, Scala	Безкоштовний
Gosec	Go	Безкоштовний
Jlint	Java	Безкоштовний
PHP-Sat	PHP	Безкоштовний
pylint	Python	Безкоштовний
SonarQube	Java, C#, PHP, Python, JavaScript, TypeScript, Kotlin, T-SQL, PL/SQL, Apex, COBOL та інші.	Безкоштовний, Комерційний
SpotBugs	Java	Безкоштовний
UNO	C	Безкоштовний
Vet	Go	Безкоштовний
Yasko	Java, C/C++, JavaScript	Безкоштовний

Ринок інструментів тестування наповнений пропозиціями SAST, часто в комплекті з додатковими рішеннями, що робить складним завдання знайти відповідний варіант для вашої організації.

Список критеріїв OWASP для вибору правильних інструментів SAST може допомогти компаніям звузити варіанти та вибрати рішення, яке найкраще допомагає їм вдосконалити свої стратегії безпеки додатків:

Мовна підтримка: головним фактором є те, якими мовами користується ваша організація. Переконайтеся, що інструмент SAST, який ви використовуєте, пропонує повне охоплення цих мов.

Покриття вразливостей: Переконайтеся, що ваш інструмент SAST охоплює принаймні всі десятки вразливостей безпеки веб-додатків OWASP.

Точність: Хоча точність є слабким місцем для SAST, і завжди будуть помилкові спрацьовування та помилкові негативи, важливо перевірити точність інструментів SAST, які розглядає ваша організація.

Сумісність: Як і будь-який автоматизований інструмент, важливо, щоб використовуваний вами інструмент SAST підтримувався фреймворками, які ви вже використовуєте, щоб він легко інтегрувався у ваш SDLC.

Інтеграція IDE: інструмент SAST, який можна інтегрувати у вашу IDE, заощадить вам цінні ресурси відновлення.

Проста інтеграція: знайдіть інструмент SAST, який легко налаштувати, і він максимально легко інтегрується з рештою інструментів у вашому конвеєрі DevOps .

Масштабованість: переконайтесь, що інструмент SAST, який ви інтегруєте сьогодні, може бути масштабований для підтримки більшої кількості розробників та проектів завтра. Здається, інструмент SAST швидко сканує невеликий зразок проекту; переконайтеся, що це дає подібні результати для великих проектів [4].

Зростання масштабу також може вплинути на вартість рішення. У списку OWASP вказується, що важливо врахувати, чи змінюються витрати на користувача, організацію, програму чи рядок аналізованого коду.

2.2.3 Динамічний аналіз безпеки програмного забезпечення (DAST)

Динамічне тестування безпеки додатків (DAST) - це процес аналізу веб-програми через інтерфейс для виявлення вразливостей за допомогою модельованих атак. Цей тип підходу оцінює додаток ззовні, атакуючи додаток, як шкідливий користувач. Після того, як сканер DAST виконує ці атаки, він шукає результати, які не є частиною очікуваного набору результатів, та визначає вразливі місця безпеки. Принцип обертається навколо введення несправностей у тестування шляхів коду в додатку. Наприклад, він може використовувати канали даних загроз для виявлення зловмисних дій. DAST не вимагає вихідного коду або двійкових файлів, оскільки він аналізує, виконуючи додаток [4].

DAST, який іноді називають сканером вразливості веб-додатків , є різновидом тесту на безпеку. Він шукає уразливості безпеки, імітуючи зовнішні атаки на програму під час роботи програми. Він намагається проникнути в додаток іззовні, перевіряючи відкриті інтерфейси на наявність уразливостей та недоліків.

Динамічна частина назви DAST походить від тесту, що виконується в динамічному середовищі. На відміну від SAST, який сканує код програми рядок за рядком, коли програма перебуває в стані спокою, тестування DAST виконується під час запуску програми. Це не означає, що тестування проводиться під час запуску програми. Хоча DAST можна використовувати у виробництві, тестування зазвичай проводиться в середовищі контролю якості [5].

DAST надзвичайно добре знаходить видимі ззовні проблеми та уразливості. Сюди входить низка ризиків для безпеки, що входять у першу десятку OWASP, таких як міжсайтовий сценарій , помилки введення, такі як

введення SQL або введення команд , обхід шляху та небезпечна конфігурація сервера .

Однією з переваг DAST є його здатність виявляти проблеми під час виконання, що SAST не може робити у своєму статичному стані. DAST чудово знаходить проблеми з конфігурацією сервера та автентифікацією, а також недоліки, які видно лише тоді, коли відомий користувач входить у систему.

DAST працює, впроваджуючи автоматичне сканування, яке імітує зловмисні зовнішні атаки на додаток для виявлення результатів, які не є частиною очікуваного набору результатів [5]. Одним із прикладів цього є ін’єкція шкідливих даних для виявлення загальних вад ін’єкцій. DAST перевіряє всі точки доступу HTTP та HTML, а також імітує випадкові дії та поведінку користувачів для пошуку вразливостей.

Оскільки DAST не має доступу до вихідного коду програми, він виявляє вразливі місця безпеки, атакуючи програму зовні. DAST не розглядає код, тому не може спрямовувати тестерів на конкретні рядки коду, коли виявляються вразливості.

При впровадженні рішень DAST дуже покладаються на експертів з безпеки. Щоб DAST був корисним, експертам з безпеки часто потрібно писати тести або допрацьовувати інструмент. Це вимагає чіткого розуміння того, як працює програма, яку вони тестують, а також як вона використовується. Експерти з безпеки також повинні добре знати веб-сервери, сервери додатків, бази даних, списки контролю доступу, потік трафіку додатків та багато іншого для ефективного адміністрування DAST [5].

Хоча вони можуть здаватися схожими, DAST відрізняється від тестування на проникнення декількома важливими способами. DAST пропонує систематичне тестування, орієнтоване на застосування в робочому стані. З іншого боку, тестування пера використовує загальноприйняті методи злому з дозволу власника та намагається використати вразливості не лише за допомогою програми, включаючи брандмауери, порти, маршрутизатори та сервери.

DAST важливий, оскільки розробникам не потрібно покладатися виключно на власні знання при створенні додатків. Проводячи DAST під час SDLC, ви можете виявити вразливості програми, перш ніж вона буде розгорнута для загального користування. Якщо ці вразливості не перевірити, а програму розгорнути як таку, це може призвести до порушення даних, що призведе до значних фінансових втрат та шкоди репутації вашого бренду. Людська помилка неминуче зіграє роль у якийсь момент життєвого циклу розробки програмного забезпечення (SDLC), і чим швидше виявиться уразливість під час SDLC, тим дешевше її буде виправити [5].

DAST - це цінний інструмент тестування, який може виявити вразливі місця безпеки інших інструментів. Хоча DAST перевершує деякі галузі, він має свої обмеження. Давайте розглянемо основні плюси і мінуси цієї технології.

Плюси:

— Оскільки DAST не розглядає вихідний код, він не залежить від мови чи платформи. Не обмежуючись певними мовами чи технологіями, ви можете запускати один інструмент DAST у всіх своїх додатках.

— Заснований на Benchmark Project OWASP, DAST має нижчий коефіцієнт помилково позитивних результатів, ніж інші засоби тестування безпеки додатків. Тестери можуть обнуляти реальні вразливості, одночасно виправляючи шум.

— DAST перевершує пошук вразливих місць безпеки, які виникають лише тоді, коли програма працює. Крім того, DAST атакує програму ззовні всередині, ставлячи її в ідеальне положення для пошуку помилок конфігурації, пропущених іншими інструментами AST.

Мінуси:

— Одним з головних недоліків DAST є його велика залежність від експертів з питань безпеки для написання ефективних тестів, що ускладнює масштабування.

— DAST не має видимості в базі коду програми. Це означає, що DAST не може направити розробників на проблемний код для виправлення або самостійно забезпечити всебічне покриття безпеки.

— DAST не славиться своєю швидкістю, і багато користувачів повідомляють, що сканування триває занадто довго. За оцінками Форестера, сканування DAST може тривати 5-7 днів. Крім того, сканування DAST зазвичай виявляють уразливості пізніше у життєвому циклі розробки програмного забезпечення (SDLC) , коли вони є більш дорогими та трудомісткими для виправлення [4].

Сканер DAST шукає вразливості в запущеному додатку, а потім надсилає автоматичні сповіщення, якщо виявляє недоліки, які дозволяють атаки, такі як ін'єкції SQL , міжсайтові сценарії (XSS) тощо. Оскільки інструменти DAST оснащені для роботи в динамічному середовищі, вони можуть виявляти недоліки виконання, які інструменти SAST не можуть виявити.

Динамічний аналіз безпеки здатний знаходити проблеми та атакувати інші методології тестування, пропущені. Це особливо вірно, якщо ваші програми покладаються на архітектуру мікропослуг. Хоча інструменти SAST чудово піддаються скануванню коду, вони не можуть визначити, як взаємодіє кожна мікрослужба. Інструменти SAST також обмежені середовищами, які вони можуть сканувати через свою мовну залежність, і в них є помилкові спрацьовування. Рішення DAST долають ці обмеження, тестуючи додатки ззовні, у виробничому середовищі, "бачачи", як взаємодіє кожна мікрослужба. Це можна зробити від рівня запиту на витягування аж до проміжного середовища.

На прикладі будівлі сканер DAST можна вважати охоронцем. Однак замість того, щоб просто переконатися, що двері та вікна заблоковані, цей охоронець робить крок далі, намагаючись фізично увірватися у будівлю. Охоронець може спробувати підібрати замки на дверях або розбити вікна. Закінчивши це обстеження, охоронець міг доповісти керівнику будівлі та

пояснити, як він зміг проникнути у будівлю. Сканер DAST можна уявити таким же чином - він активно намагається знайти вразливості в робочому середовищі, щоб команда DevOps знала, де і як їх виправити.

Використовуючи DAST для виявлення вразливостей раніше в SDLC, організації можуть зменшити ризик, заощаджуючи ресурси. Найбільше занепокоєння людей з питань безпеки має якомога менше хибнопозитивних результатів, щоб зосередитись на реальних загрозах. Щоб усунути вразливість, потрібно в середньому 38 днів. Якщо вразливість не була реальною загрозою, це 38 дорогих днів непотрібної затримки [5].

Організації можуть використовувати DAST для допомоги у дотриманні PCI та інших нормативних звітів.

На додаток до впорядкування відповідності, рішення DAST може допомогти розробникам виявити помилки або проблеми у конфігурації, як зазначено в [5], а також висвітлити конкретні проблеми взаємодії з користувачем.

Таблиця 2.2 – Популярні інструменти динамічного сканування безпеки коду

Інструмент	Власник	Доступність
Abbey Scan	MisterScanner	Безкоштовний
Acunetix	Acunetix	Комерційний
App Scanner	Trustwave	Комерційний
AppSpider	Rapid7	Комерційний
Arachni	Arachni	Безкоштовний
Burp Suite	PortSwiger	Комерційний
GoLismero	GoLismero Team	Безкоштовний
Nessus	Tenable	Комерційний
QualysGuard	Qualys	Комерційний
Sentinel	WhiteHat Security	Комерційний
WebInspect	Micro Focus	Комерційний
Websecurify Suite	Websecurify	Комерційний
Zed Attack Proxy	OWASP	Безкоштовний
w3af	w3af.org	Безкоштовний

2.2.4 Інтерактивний аналіз безпеки програмного забезпечення (IAST)

IAST (інтерактивне тестування безпеки додатків) аналізує код на наявність уразливих місць безпеки, поки додаток запускається автоматизованим тестом, тестувальником або будь-яким іншим процесом, який «взаємодіє» з функціональністю програми. Ця технологія повідомляє про вразливості в режимі реального часу, а це означає, що вона не додає зайвого часу до вашого конвейєру CI / CD [4].

IAST працює всередині програми, що робить її відмінною як від статичного аналізу (SAST), так і від динамічного аналізу (DAST). Цей тип тестування також не перевіряє всю програму чи кодову базу, а лише те, що здійснюється функціональним тестом. Більше відмінностей можна побачити в таблиці 2.3.

Таблиця 2.3 – Порівняння методів тестування програмного забезпечення

	Покриття коду	Рівень негативних спрацювань	Експлуатаційність	Доступність до коду	Інтеграція в SDLC
SAST	Повне	Високий	Відсутня	Присутня	Легка
DAST	Часткове	Низький	Присутня	Відсутня	Складна
IAST	Часткове	Низький	Присутня	Присутня	Легка

Інтерактивні рішення для тестування безпеки додатків допомагають організаціям виявляти та управляти ризиками безпеки, пов’язаними з уразливими місцями, виявленими під час запуску веб-додатків, за допомогою методів динамічного тестування (що часто називають тестуванням під час виконання). IAST працює за допомогою програмних приладів або використання інструментів для моніторингу програми під час її роботи та збору інформації про те, що вона робить і як вона працює. IAST вирішує інструментальні програми шляхом розгортання агентів і датчиків у запущених додатках та постійного аналізу всіх взаємодій додатків, ініційованих ручними тестами, автоматизованими тестами або їх комбінацією для виявлення вразливостей у

реальному часі. Крім того, деякі рішення інтегрують інструменти аналізу складової програмного забезпечення для усунення відомих вразливостей компонентів та фреймворків з відкритим кодом.

IAST, як правило, реалізується шляхом розгортання агентів і датчиків у програмі після збірки [5]. Агент спостерігає за роботою програми та аналізує потік трафіку для виявлення вразливостей безпеки. Це робиться шляхом зіставлення зовнішніх підписів або шаблонів із вихідним кодом, що дозволяє виявляти більш складні вразливості.

IAST - це технологія, орієнтована на розробників, яка допомагає організаціям рухатися вліво при зверненні до тестування безпеки. Незважаючи на те, що IAST має багато переваг, він не позбавлений недоліків. Давайте розглянемо плюси та мінуси IAST.

Плюси:

— IAST має надзвичайно низький показник хибнопозитивних результатів, на відміну від SAST, який має загальновідомо високий коефіцієнт хибнопозитивних результатів. Значна кількість організацій щодня стикаються з тисячами сповіщень про безпеку. Завдяки такому обсягу, точність тестування є критично важливою для зменшення шуму та зменшення втоми.

— Щоб не відставати від темпів розвитку в наші дні, розробники вимагають швидких рішень для тестування без затримки. IAST забезпечує швидкість, надаючи результати тестування безпосередньо розробникам у режимі реального часу. IAST також добре інтегрується з інструментами CI / CD. Усунення вразливостей та перевірка чистого коду на початку життєвого циклу розробки програмного забезпечення (SDLC) допомагає організаціям економити час і гроші.

— IAST є дуже масштабованим і легко розгортається для кожного розробника в організації. Результати тесту направляють розробників до конкретних рядків проблемного коду для негайного виправлення, не вимагаючи втручання фахівця з безпеки.

Мінуси:

— Оскільки IAST вбудований у додаток, який тестується, він залежить від мови та має архітектуру на стороні сервера. IAST не охоплює певні мови та підтримує лише сучасні технологічні рамки. Сам по собі IAST не забезпечує достатнього покриття, і найкраще працює у поєднанні з іншими рішеннями AST.

— Щоб отримати максимальну користь від IAST, організаціям потрібне зріле та чітко визначене тестове середовище. IAST вимагає сучасного середовища для розробки програмного забезпечення та архітектури.

— Незважаючи на те, що IAST існує вже кілька років, він все ще не знайшов опори на ринку. Це пов’язано з тим, що він не забезпечує достатньо охоплення самостійно, немає вимірюваної рентабельності інвестицій, або він не знайшов правильних випадків використання, який ще не визначено.

2.3 Автоматизація тестування безпеки програмних продуктів

Автоматизація в будь-якій галузі приносить переваги підвищення продуктивності та зменшення витрат. При гнучкій розробці програмного забезпечення автоматизація стала настільки невід’ємною частиною гнучкого тестування, що важко уявити одне без іншого. Давайте побачимо основні причини, чому автоматизація вважається вирішальною для ефективного тестування [5].

а) Поступовий розвиток: Першим і найголовнішим фактором, який вимагає автоматизації в гнучкому тестуванні, є короткий цикл розвитку. Швидкі команди мають лише кілька тижнів, щоб зрозуміти вимогу, внести зміни в код і протестувати їх. Якби всі випробування проводились вручну, необхідний час перевищив би фактичний час розробки. В якості альтернативи тестування потрібно було б поспішити, тим самим погіршивши якість.

б) Часті зміни: гнучкі проекти не працюють із повним набором вимог. Вимоги розвиваються з часом і часто змінюються залежно від пріоритетів

клієнтів, ринкових тенденцій та потреб кінцевих споживачів. Хоча найбільш позитивною рисою спритного методу є його швидка пристосованість до змін, це також означає, що тестування повинно бути достатньо спритним, щоб відповідати змінам. Автоматизація забезпечує необхідну спритність тестування та допомагає швидше та ефективніше реагувати на зміни.

в) Постійне тестування: Спритність вимагає раннього і постійного тестування. Покриття тесту поширюється не тільки на нещодавно доданий код, але і на код із попередніх ітерацій. Це робиться для того, щоб попередня функціональність не була порушена через нещодавно додану функціональність. Це чинить великий тиск на тестувальників і може серйозно вплинути на якість продукту. Автоматизація деяких тестів означає, що тестувальники мають більше часу на пошукові тестування.

г) Отримайте швидкий огляд якості коду: автоматичне тестування дозволяє швидко протестувати код за допомогою стандартного набору тестових скриптів. Це дає тестувальнику та розробнику швидкий зазик до якості коду, і вони мають більше часу для реакції, якщо код не відповідає очікуванням.

д) Автоматизація заходів з підтримки тестування: Автоматизація в тестуванні призначена не лише для виконання тестових скриптів щодо коду, але також може використовуватися для автоматизації інших тестових дій, таких як налаштування даних, перевірка результатів тестування та звітування про тести. Спритність вимагає частого розгортання коду, яке також можна автоматизувати. Це звільняє тестерів від буденних, повторюваних завдань, щоб вони могли більше зосередитися на тестуванні.

є) Вичерпне тестування: За допомогою автоматизації тестування можна повторити стільки разів, що дозволяє детально та вичерпно вивчити код. Це дуже корисно для забезпечення якості коду під час роботи в обмеженому вікні тестування.

Автоматизацію тестів найкраще розробляти поетапно, і її слід починати паралельно розробці, щоб час, який може бути використаний для тестування, не

витрачався даремно на автоматизацію. Автоматизація тестів повинна бути продуманим процесом, щоб вона була економічно ефективною та приносила достатню віддачу [1].

Тестування безпеки зростає швидше, ніж будь-який інший ринок безпеки, оскільки рішення AST адаптуються до нових методологій розробки та збільшують складність додатків. Керівники з питань безпеки та управління ризиками повинні інтегрувати AST у свої програми захисту програм. Потреба у забезпеченні безпеки додатків посилилась із збільшенням кількості ризиків та атак у віртуальному світі. Ось чому автоматизоване тестування безпеки взяло пріоритет, і ідея безперервного тестування та доставки також підтримується.

Зазвичай, тестування на безпеку проводиться після доставки продукту. Додаток перевірено на наявність недоліків безпеки та автентифікації, однак результати можуть бути неадекватними і в кінцевому підсумку можуть порушити роботу програми. DevSecOps еволюціонував, щоб збалансувати потреби тестування безпеки, включивши внутрішні сили DevOps в процес тестування безпеки. Ця модель пропонує основу для додавання перевірок безпеки в конвеєрах розробки та розгортання та робить кожного відповідальним за забезпечення безпеки [4].

Отже, автоматизовані тести вбудовані в цикл тестування, тримаючи модель DevOps у фокусі. Це призвело до появи різноманітних інструментів та технологій, що дозволяють підприємствам проводити тестування безпеки з перспективою DevOps.

Програмні додатки ускладнюються і потенційно можуть загрожувати через ринкові ризики та різні властиві вразливості. Отже, тестування має бути строгим та повторним. DevSecOps об'єднує сильні сторони DevOps, тестування безпеки та автоматизації. Основною метою DevOps є надання більшої потужності командам розробників для розгортання та моніторингу програми. Отже, впровадження автоматизованого тестування для отримання більш швидких результатів забезпечує кращу якість програм.

Рух DevSecOps все ще формується, і правила все ще залишаються на місці. Підприємства розуміють найкращий можливий спосіб автоматизації та впровадження тестування безпеки. Таким чином, Тестування безпеки стає сильнішим, повторним та набагато спритнішим для вирішення ринкових проблем.

Концепція все ще розвивається, але основи ті самі, що залишаються дуже близькими до автоматизації тестування та моделі DevOps. Включення аспекту безпеки є важливим. Постійне тестування та доставка становлять ядро моделі DevSecOps і роблять процес тестування та розробки більш спільним [5].

Найкращі практики автоматизації тестів безпеки подібні до найкращих способів реалізації будь-яких проектів автоматизованого тестування. Тільки тести безпеки повинні бути легко інтегровані в процес.

Для впровадження автоматизованого тестування безпеки в процес розробки програмного забезпечення, потрібно дотримуватись базових кроків, які допоможуть зробити це максимально ефективно і швидко.

1. Визначте вразливі місця. Виконання перевірки є абсолютно важливим. Рекомендується розбити додаток на частини / блоки та перевірити їх на наявність уразливостей. Це допомагає визначити шляхи збою та лазівки у всіх аспектах уразливостей програми. Багато вірусів та помилок у кіберпросторі, як правило, вникають у основні та найбільш непомічені вразливості системи безпеки [4].

Це може бути погана автентифікація, неефективні паролі або неадекватна політика безпеки. Існують сканери вразливостей для виявлення прихованої мережі та вразливостей на хості. Розбивши програму та запустивши автоматизовані тести для кожної функції, можна ефективно виявити вразливі місця. Це перший крок або найважливіший аспект, оскільки це дозволить командам вживати подальших дій та виконувати послідовно.

Насправді після виконання тестів групи можуть класифікувати вразливості відповідно до їх технічної серйозності, рекомендуючи єдине рішення безпеки або декілька виправлень та модернізацій.

2. Інтегруйте найкращі практики автоматизації з DevOps. Автоматизація тесту - це можливість для всього підходу DevOps. DevOps може бути успішним лише за умови успішної реалізації автоматизації. Концепція безперервного тестування та доставки працює з тим, що автоматизація тестів ефективно впроваджується в процесі. Концепція DevSecOps підсилює ідею автоматизації тестів безпеки через тестовий цикл [5].

Найкращий спосіб - інтегрувати найкращі практики автоматизації тестів та підходу DevOps до цілей тестування безпеки. Поки триває процес безперервного тестування, автоматизація тестів допомагає знаходити дефекти одночасно, а випуск програмного забезпечення відбувається постійно. Отже, на етапі розгортання тривають тести для перевірки безпеки програми.

3. Виберіть правильний інструмент. Як результат, на ринку існує безліч інструментів та технологій, що заохочують впровадження DevOps. Подібним чином, завдяки потужній комбінації автоматизації, тестування безпеки та DevOps, існує критична потреба у виборі правильного інструменту для реалізації [5].

Ви можете заморозити будь-яку структуру автоматизації тестування, але вона повинна добре організувати цілі проекту та вимоги безпеки. В ідеалі рекомендується вибрати інструмент, з яким команди розробників, операцій та безпеки знайомі та можуть ефективно інтегруватися в цикл тестування для отримання відчутних результатів.

4. Автоматизуйте тести безпеки. Тестування на безпеку вимагає особливого ставлення та підходів. Автоматизація тестів безпеки схожа на автоматизацію функціональних тестів або тестів продуктивності [5]. Автоматизуючи тести, тести безпеки можна розділити на функціональні тести безпеки, такі як автентифікація та генерація паролів, конкретні нефункціональні тести на відомі слабкі місця, сканування безпеки програми та інфраструктури та логіка додатків тестування безпеки.

Основною ідеєю має бути сегментація цілей тестування безпеки та автоматизація тестів для визначення критеріїв успіху. Отримання необхідних результатів та усунення вразливостей за допомогою необхідної автоматизації є важливим. Ніщо не можна розглядати як надмірну автоматизацію або недостатню автоматизацію, якщо виконуються критичні для бізнесу цілі.

5. Тест на спалах вразливості. Метою автоматизації тестів безпеки є підготовка програми до будь-яких можливих спалахів або масових атак. Визначаючи цілі та стратегію, важливо використовувати правильні інструменти та основи для спалаху [5]. Поточний сценарій лякає для будь-якої програми, і вразливість може виникнути всередині програми або зовнішньої. Розробка систем автоматизації для перевірки будь-якої такої атаки вразливості може бути гарною практикою.

Структури автоматизації покращуються завдяки кращим тестовим кейсам протягом певного періоду. Отже, інвестувати у створення надійної основи для тестування безпеки, безумовно, варто для підприємства / команди.

3 ПЛАТФОРМА KUBERNETES

3.1 Опис платформи Kubernetes

Kubernetes - це портативна, розширювана платформа з відкритим кодом для управління робочими навантаженнями та послугами в контейнерах, що полегшує як декларативну конфігурацію, так і автоматизацію. Він має велику, швидко зростаючу екосистему. Послуги, підтримка та інструменти Kubernetes широко доступні.

Назва Kubernetes походить від грецької, що означає керманич або льотчик. Google відкрив проект Kubernetes у 2014 році. Kubernetes поєднує в собі 15-річний досвід роботи у виробництві масштабних виробничих навантажень із найкращими в світі ідеями та практикою спільноти. Давайте подивимось, чому Kubernetes так корисний, повернувшись у минуле [7].

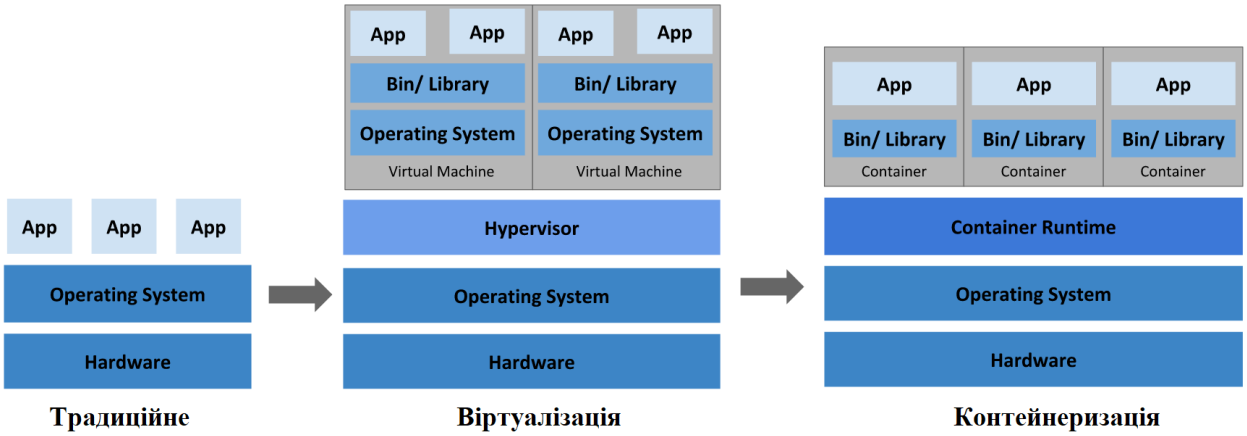

Рисунок 3.1 – Типи розгортання програмного забезпечення

Традиційна ера розгортання: на початку організації запускали програми на фізичних серверах. Не було можливості визначити межі ресурсів для додатків на фізичному сервері, і це спричинило проблеми з розподілом ресурсів. Наприклад, якщо на фізичному сервері працює кілька програм, можуть бути випадки, коли

одна програма забирала б більшу частину ресурсів, і як результат, інші програми мали б недостатню ефективність. Рішенням для цього було б запустити кожну програму на іншому фізичному сервері. Але це не змінилось, оскільки ресурси були недостатньо використані, і організаціям було дорого підтримувати багато фізичних серверів.

Ера віртуалізованого розгортання: як рішення було запроваджено віртуалізацію. Це дозволяє запускати кілька віртуальних машин (ВМ) на одному фізичному центральному процесорі. Віртуалізація дозволяє ізолювати програми між віртуальними машинами та забезпечує рівень захисту, оскільки інша програма не може вільно отримати доступ до інформації однієї програми.

Віртуалізація дозволяє краще використовувати ресурси на фізичному сервері та забезпечує кращу масштабованість, оскільки додаток можна легко додавати або оновлювати, зменшує апаратні витрати та багато іншого. За допомогою віртуалізації ви можете представити набір фізичних ресурсів як кластер одноразових віртуальних машин.

Кожна віртуальна машина - це повноцінна машина, що працює над усіма компонентами, включаючи власну операційну систему, поверх віртуалізованого обладнання.

Ера розгортання контейнерів: контейнери схожі на віртуальні машини, але вони мають розслаблені властивості ізоляції, щоб спільно використовувати операційну систему (ОС) серед програм, як показано на рисунку 3.1. Тому контейнери вважаються легкими. Подібно до віртуальної машини, контейнер має власну файлову систему, частку центрального процесора, пам’ять, простір процесів тощо. Оскільки вони відокремлені від базової інфраструктури, вони переносяться між хмарами та дистрибутивами ОС [7].

Контейнери стали популярними, оскільки вони надають додаткові переваги, такі як:

— Швидке створення та розгортання додатків: підвищена простота та ефективність створення образу контейнера порівняно з використанням зображень VM.

— Постійний розвиток, інтеграція та розгортання: забезпечує надійне та часте створення та розгортання зображень контейнера з швидкими та простими відкатами (завдяки незмінності зображення).

— Розділення проблем Dev та Ops: створюйте образи контейнера додатків під час збирання / випуску, а не часу розгортання, тим самим відокремлюючи програми від інфраструктури.

— Спостережливість не тільки відображає інформацію та показники на рівні ОС, але також стан роботи та інші сигнали.

— Екологічна узгодженість у розробці, тестуванні та виробництві: працює на ноутбуці так само, як у хмарі.

— Переносимість хмарних та ОС дистрибутивів: Працює на Ubuntu, RHEL, CoreOS, локально, у великих загальнодоступних хмарах та в будь-якому іншому місці.

— Управління, орієнтоване на програми: Підвищує рівень абстракції від запуску ОС на віртуальному обладнанні до запуску програми в ОС за допомогою логічних ресурсів.

— Слабко пов'язані, розподілені, еластичні, звільнені мікросервіси: програми розбиваються на менші, незалежні частини і можуть бути розгорнуті та керовані динамічно - а не монолітний стек, що працює на одній великій одноцільовій машині.

— Ізоляція ресурсів: передбачувана продуктивність програми.

— Використання ресурсів: висока ефективність та щільність.

3.2 Kubernetes, як середовище для життєвого циклу розробки програмного забезпечення

Контейнери - це хороший спосіб об’єднати та запустити ваші програми. У виробничому середовищі вам потрібно керувати контейнерами, в яких запущені програми, і переконатися, що немає простоїв. Наприклад, якщо контейнер опускається, потрібно запускати інший контейнер. Чи не було б простіше, якби з цією поведінкою працювала система?

Ось так на допомогу приходить Kubernetes. Kubernetes надає вам основу для стійкого запуску розподілених систем. Він дбає про масштабування та відновлення після відмови для вашої програми, надає схеми розгортання тощо.

Kubernetes визначає набір будівельних блоків, які в сукупності забезпечують механізми розгортання, обслуговування та масштабування додатків на основі процесора, пам'яті або користувацьких метрик. Kubernetes є вільно з'єднаним і розширюваним, щоб задовольнити різні навантаження. Ця розширюваність значною мірою забезпечується API Kubernetes [7], який використовується внутрішніми компонентами, а також розширеннями та контейнерами, які працюють на Kubernetes. Платформа здійснює свій контроль над обчислювальними та сховищними ресурсами, визначаючи ресурси як об'єкти, якими потім можна керувати.

Kubernetes слідує первинній архітектурі / копії. Компоненти Kubernetes можна розділити на ті, що управляють окремим вузлом, і ті, що є частиною площини управління.

Майстер нода Kubernetes є головним блоком управління кластером, керуючи його робочим навантаженням і спрямовуючи зв'язок по системі [7]. Площина управління Kubernetes складається з різних компонентів, кожен з яких має свій власний процес, який може працювати як на одному головному вузлі, так і на декількох ведучих, що підтримують кластери високої доступності.

Головні можливості, які Kubernetes надає [7]:

— Виявлення послуги та балансування навантаження. Kubernetes може виставити контейнер, використовуючи ім'я DNS або використовуючи власну IP-адресу. Якщо трафік до контейнера високий, Kubernetes може завантажувати баланс і розподіляти мережевий трафік, щоб розгортання було стабільним.

— Організація зберігання. Kubernetes дозволяє автоматично монтувати обрану вами систему зберігання, наприклад, локальні сховища, загальнодоступні хмарні постачальники тощо.

— Автоматизовані розгортання та відкоти. Ви можете описати бажаний стан для ваших розгорнутих контейнерів за допомогою Kubernetes, і він може змінити фактичний стан до бажаного стану з контрольованою швидкістю. Наприклад, ви можете автоматизувати Kubernetes для створення нових контейнерів для вашого розгортання, видалення існуючих контейнерів та прийняття всіх їх ресурсів до нового контейнера.

— Автоматична упаковка сміття. Ви надаєте Kubernetes кластер вузлів, який він може використовувати для запуску контейнерних завдань. Ви повідомляєте Kubernetes, скільки процесора та пам'яті (ОЗУ) потрібно кожному контейнеру. Kubernetes може вмістити контейнери на вузли, щоб найкраще використовувати ресурси.

— Kubernetes, що самовиліковується, перезапускає контейнери, які виходять з ладу, замінює контейнери, вбиває контейнери, які не відповідають на визначену користувачем перевірку стану здоров’я, та не показує їх клієнтам, поки вони не будуть готові до обслуговування.

— Управління секретом та конфігурацією. Kubernetes дозволяє зберігати та керувати конфіденційною інформацією, такою як паролі, маркери OAuth та ключі SSH. Ви можете розгортати та оновлювати секрети та конфігурацію програми, не відновлюючи образи контейнера та не відкриваючи секрети у конфігурації стека.

Kubernetes не є традиційною системою PaaS (платформа як послуга), що включає все. Оскільки Kubernetes працює на рівні контейнера, а не на

апаратному рівні, він надає деякі загальноприйнятні функції, загальні для пропозицій PaaS, такі як розгортання, масштабування, балансування навантаження та дозволяє користувачам інтегрувати свої рішення для ведення журналу, моніторингу та попередження. Однак Kubernetes не є монолітним, і ці рішення за замовчуванням є необов'язковими та підключаються. Kubernetes надає будівельні блоки для побудови платформ для розробників, але зберігає вибір та гнучкість користувачів там, де це важливо.

Щоб зрозуміти ключові відмінності платформи Kubernetes та як правильно її використовувати в середовищі життєвого циклу програмного забезпечення, потрібно визначити основні моменти, які не відносяться до даної платформи:

— Не обмежує типи підтримуваних програм. Kubernetes має на меті підтримувати надзвичайно різноманітні робочі навантаження, включаючи навантаження без громадянства, державні дані та обробку даних. Якщо програма може працювати в контейнері, вона повинна чудово працювати на Kubernetes.

— Не розгортає вихідний код і не створює вашу програму. Постійні процеси інтеграції, доставки та розгортання (CI / CD) визначаються організаційними культурами та уподобаннями, а також технічними вимогами.

— Не надає послуги на рівні додатків, такі як проміжне програмне забезпечення (наприклад, шини повідомлень), фреймворки обробки даних (наприклад, Spark), бази даних (наприклад, MySQL), кеші, а також системи кластерного зберігання (наприклад, Ceph) як вбудовані послуги. Такі компоненти можуть працювати на Kubernetes та / або мати доступ до них за допомогою програм, що працюють на Kubernetes за допомогою портативних механізмів, таких як Open Service Broker.

— Не диктує рішення щодо ведення журналу, моніторингу та попередження. Він надає деякі інтеграції як доказ концепції та механізми збору та експорту метрик.

— Не надає і не вказує мову / систему конфігурації (наприклад, Jsonnet). Він надає декларативний API, який може бути націлений на довільні форми декларативних специфікацій [7].

— Не забезпечує і не приймає жодної комплексної системи конфігурації, обслуговування, управління або самовідновлення.

— Крім того, Kubernetes не є простою системою оркестровки. Насправді це позбавляє від необхідності оркеструвати. Технічним визначенням оркестровки є виконання визначеного робочого циклу: спочатку виконайте A, потім B, потім C. На відміну від цього, Kubernetes включає набір незалежних, складових процесів управління, які постійно ведуть поточний стан до заданого бажаного стану. Не має значення, як ви переходите від A до C. Централізований контроль також не потрібен. В результаті виходить система, яка є простішою у використанні та потужнішою, надійнішою, стійкішою та розширюваною.

3.3 Kubernetes Оператор, як засіб автоматизації тестування

Kubernetes став золотим стандартом для організації контейнерів. Однак, коли справа стосується запуску програм, що потребують підтримки конфігурацій, йому потрібно отримати доступ до ряду залежностей та оперативних завдань, щоб він міг нормально функціонувати. Як можна уявити, якщо це робити з тисячами контейнерів, це забирає багато часу і займає багато ресурсів для розробників, саме тому потрібні Kubernetes Operators.

Оператор Kubernetes (зазвичай його просто називають Оператором) відноситься до методу упаковки, розгортання та управління програмою в Kubernetes [8].

Додаток в Kubernetes - це додаток, який розгортається на самому Kubernetes, і ним керується за допомогою API Kubernetes та інструментів kubectl. Щоб отримати максимальну віддачу від Kubernetes, вам потрібно мати доступ до цілісного набору API, який можна розширити для обслуговування та управління

вашим додатком, який працює на Kubernetes. У цьому випадку Оператори виступають в якості середовища виконання для управління цим видом додатків на Kubernetes.

Оператори розглядаються як продовження команди інженерів програмного забезпечення, оскільки вони допомагають контролювати ваше середовище Kubernetes та використовують його поточний стан або середовище для прийняття рішень протягом декількох мілісекунд [8]. По суті, Оператор приймає людські оперативні знання, а потім кодує їх у програмне забезпечення, щоб їх можна було легко запакувати та надати споживачам.

Оператори Kubernetes можуть обробляти цілий ряд оперативних завдань, починаючи від базової функціональності та виконуючи певну логіку для програми. Більш просунуті оператори можуть автоматично реагувати на несправності та виконувати оновлення [11].

Важливо зазначити, що не кожен Оператор однаковий; не існує універсальної моделі. Вони створені спеціально для запуску програми Kubernetes.

Виходячи з цього, можливості Оператора різняться залежно від того, наскільки логіка, параметри чи функціональність були впроваджені в самого Оператора. І це буде залежати від потреб та вимог бізнесу. Також, коли розробник розробляє Оператора, він, як правило, починає з малого, автоматизуючи установку додатків та надання послуг самообслуговування, а потім розвиває до більш складної автоматизації.

Незалежно від того, розробник ви або кінцевий користувач, Kubernetes Operator надає низку переваг [8].

Хоча Kubernetes став золотим стандартом для організації контейнерів, часто існує великий бар'єр для входу в розробку програм Kubernetes. Існує низка раніше існуючих залежностей та припущень, які потрібно взяти до уваги, і багато з них вимагають певного рівня знань та технічних знань. Крім того, розробники

повинні уважно ставитися до потреб своїх кінцевих користувачів; вони більше не хочуть, щоб їхні послуги були закритими та розрізненими.

Kubernetes Operator вирішує ці проблеми, об’єднуючи досвід та знання спільноти Kubernetes в одному проекті з відкритим кодом, щоб створити стандартний пакет програм, який розробники можуть використовувати для легкого створення своїх програм Kubernetes [8].

Для користувачів програм у гібридній хмарі надзвичайно важливо підтримувати їх актуальність, актуальність та безпеку, особливо коли поведінка споживачів постійно швидко змінюється. Kubernetes Operator враховує ці вимоги до кінцевих користувачів, допомагаючи розробникам створювати власні хмарні додатки, які легше використовувати, захистити та оновлювати [11].

Оператори спеціально створені для запуску програми Kubernetes. Вони створені спеціально з використанням оперативних знань. Хмарні функціональні можливості, кодовані в Операторі, можуть забезпечити вдосконалений інтерфейс користувача та можуть автоматизувати такі функції, як оновлення, резервне копіювання та масштабування. Все це можна досягти за допомогою стандартних інструментів Kubernetes, CLI та API.

Оператори можуть бути адаптовані до потреб вашої організації, і їх можна використовувати повторно в різних додатках [8]. Щоб дати змогу розробникам створювати власних операторів, вони можуть скористатися перевагами Operator Framework, що надає необхідні інструменти для побудови, управління та контролю ефективності ваших операторів.

В даній роботі для автоматизації процесу тестування безпеки програмного забезпечення в процесі розробки, буде реалізований Kubernetes оператор, який буде контролювати роботу інструментів тестування безпеки. Для роботи будуть обрані два інструменти з відкритим вихідним кодом та розроблений оператор, який зможе робити необхідні конфігурації, щоб забезпечити безперебійну та надійну роботу інструментів в процесі життєвого циклу розробки програмного забезпечення.

4 ПРАКТИЧНА ЧАСТИНА

4.1 Розгортання платформи Kubernetes

В рамках даної роботи, було розгорнуто платформу Kubernetes в малому масштабі (3 ноди) для демонстрації роботи платформи та подальшого тестування на її базі розробленого Kubernetes Оператора та створення тестового середовища розробки програмного забезпечення.

Для виконання даної задачі використовувався персональний комп’ютер з наступними характеристиками:

— ОС: Windows 10;

— Процесор: Intel Core i7-8665U CPU @ 1.90GHz 2.11 GH;

— Оперативна пам’ять: 32.0 ГБ;

— Тип системи: 64-бітна операційна система, x64 процесор;

Найбільш оптимальною операційною системою для розгортання Kubernetes є Linux, тому на нашій хостовій машині було піднято 4 віртуальні машину з операційною системою CentOS. Три машини безпосередньо для платформи Kubernetes по 2 ГБ оперативної пам’яті та одна для розгортання, з 1 ГБ оперативної пам’яті.

Платформа Kubernetes потребує досить тривалих ручних конфігурацій для повного розгортання, тому для наших цілей, було взято Ansible playbook для автоматичного розгортання платформи. Ми використали Kubespray [6], як один з найбільш популярних засобів автоматизації розгортання платформи Kubernetes.

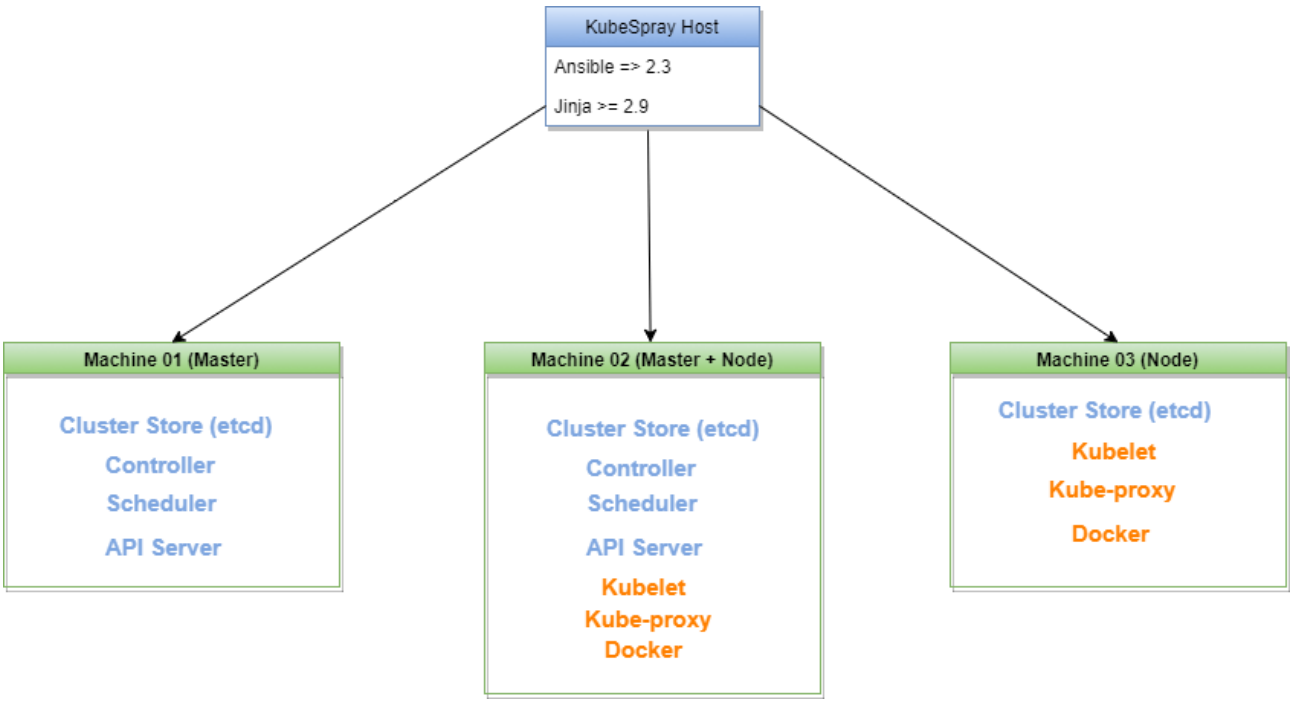

Рисунок 4.1 — Розгортання платформи Kubernetes з допомогою Kubespray

Для коректного функціонування Kubernetes було розгорнуто дві мастер ноди і дві робочих ноди для запуску додатків. Так як у нас розгорнута легка версія Kubernetes, то ми використали одну віртуальну ноду і для мастера і для робочої ноди. Kubernetes не обмежує нас в цьому, головне щоб потужностей самої віртуальної машини було достатньо. Також для платформи потрібна база даних, тому було розгорнуто кластерну версію розподіленої бази даних ETCD, саме її рекомендує використовувати офіційна документація.

4.2 Розробка Kubernetes Оператора

4.2.1 Мова програмування Golang

Мова програмування Go була розроблена компанією Google для вирішення проблем Google щодо розробки масштабного програмного забезпечення. Мова Go - це проект з відкритим кодом, який робить програмістів більш продуктивними. Його механізми одночасності дозволяють легко писати програми, які отримують максимум від багатоядерних та мережевих машин, тоді

як нова система типу дозволяє гнучку та модульну побудову програм. Go швидко компілюється до машинного коду, але має зручність збору сміття та потужність відображення під час виконання. Це швидка, статично набрана, компільована мова, яка нагадує динамічно набрану, інтерпретовану мову.

Мова Go (Golang) розроблялася як мова програмування для створення високоефективних програм, що працюють на сучасних розподілених системах і багатоядерних процесорах. Вона може розглядатися як спроба створити заміну мов C і C++ з урахуванням змінених комп'ютерних технологій і накопиченого досвіду розробки великих систем. За словами розробників, вона була розроблена для вирішення реальних проблем, що виникають при розробці програмного забезпечення в Google. В якості основних таких проблем вони виділяють:

— повільну збірку програм;

— неконтрольовані залежності;

— використання різними програмістами різних підмножин мови;

— труднощі з розумінням програм, викликані неудобочітаеми коду, поганим документуванням і так далі;

— дублювання розробок;

— високу вартість оновлень;

— несинхронні поновлення при дублюванні коду;

— складність розробки інструментарію;

— проблеми міжмовної взаємодії.

Основними вимогами до мови стали:

— Ортогональность. Мова повинна надавати невелике число засобів, які не повторюють функціональність один одного.

— Проста і регулярна граматика. Мінімум ключових слів, проста, легко розбирається граматична структура, легко читається код.

— Проста робота з типами. Типізація повинна забезпечувати безпеку, але не перетворюватися на бюрократію, лише збільшує код. Відмова від ієрархії типів, але зі збереженням об'єктно-орієнтованих можливостей.

— Відсутність неявних перетворень.

— Прибирання сміття.

— Засоби розпаралелювання, прості та ефективні.

— Підтримка рядків, асоціативних масивів і комунікаційних каналів.

— Чіткий поділ інтерфейсу і реалізації.

— Ефективна система пакетів з явною вказівкою залежностей, що забезпечує швидку збірку.

Ця мова була розроблена в розрахунку на те, що програми будуть транслюватися в об'єктний код і виконуватися безпосередньо, не вимагаючи віртуальної машини, тому одним з критеріїв вибору архітектурних рішень була можливість забезпечити швидку компіляцію в ефективний об'єктний код і відсутність надмірних вимог до динамічної підтримки.

Хоча для Go доступний і інтерпретатор, практично в ньому немає великої потреби, так як швидкість компіляції досить висока для забезпечення інтерактивної розробки.

4.2.2 Kubernetes Operator фреймворк

Operator Framework - це інструментарій з відкритим кодом для ефективного, автоматизованого та масштабованого управління власними програмами Kubernetes, які називаються Операторами. Це проект з відкритим кодом, який надає інструменти Kubernetes як для розробки, так і для виконання [8]. Більшість з цих інструментів доступні на GitHub.

Впровадження Kubernetes Operator фреймворка ознаменувало величезний крок для Kubernetes. Він забезпечує базову практику зниження бар'єру для розробки програм на Kubernetes. Структура, як показано на рисунку 4.2, включає:

— Комплект розробки програмного забезпечення оператора (SDK)

— Управління життєвим циклом оператора (Operator Lifecycle Management)

— Моніторинг оператора (Operator Metering)

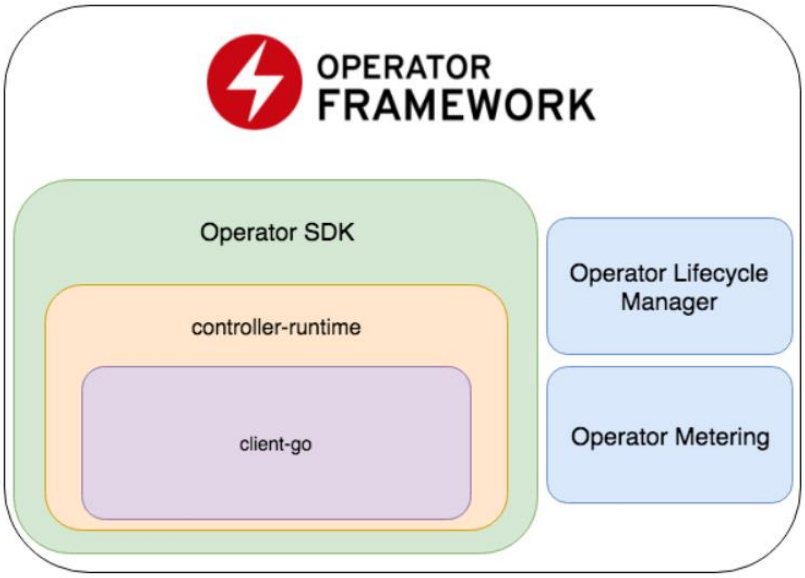

Рисунок 4.2 — Структура Kubernetes Operator фреймворка

Оператор SDK надає розробникам ключові інструменти для створення, тестування та пакування операторів. Коли було введено SDK оператора, він використовувався для передачі бізнес-логіки програми, будь то масштабування, оновлення чи виконання резервного копіювання, за допомогою API Kubernetes для виконання логіки [8]. Тепер SDK може дозволити інженерам зробити свої програми більш інтуїтивними та забезпечити взаємодію з користувачем, подібну до хмарної служби. Найкращі практики та шаблони коду, якими ділиться Оператор, включені до SDK.

Після побудови Операторів їх можна розгорнути на кластері Kubernetes. Тут з’являється Менеджер життєвого циклу оператора. Менеджер життєвого циклу оператора допомагає полегшити управління Операторами в кластері Kubernetes та його ресурсами. Адміністратори можуть контролювати, які оператори доступні в певному просторі імен, і вони можуть призначати, хто може взаємодіяти з запущеними операторами.

Моніторинг оператора це недавнє доповнення до Operator Framework, яке надає можливість вимірювати використання додатків. Це допоможе IT-командам

дотримуватися бюджету та вжити необхідних заходів для оптимізації додатків, що вимагають великих ресурсів.

Незалежно від того, розробник ви або кінцевий користувач, Operator Framework надає низку переваг, які ви повинні знати. Хоча Kubernetes став золотим стандартом для організації контейнерів, часто існує великий бар'єр для входу в розробку програм Kubernetes [8]. Існує низка раніше існуючих залежностей та припущень, які потрібно взяти до уваги, і багато з них вимагають певного рівня знань та технічних знань. Крім того, розробники повинні уважно ставитися до потреб своїх кінцевих користувачів; вони більше не хочуть, щоб їхні послуги були закритими та розрізненими.

Framework Operator вирішує ці проблеми, об’єднуючи досвід та знання спільноти Kubernetes в одному проекті з відкритим кодом, щоб створити стандартний пакет програм, який розробники можуть використовувати для легкого створення своїх програм Kubernetes.

Для користувачів програм у гібридній хмарі надзвичайно важливо підтримувати їх актуальність, актуальність та безпеку, особливо коли поведінка споживачів постійно швидко змінюється [8]. Operator Framework враховує ці вимоги до кінцевих користувачів, допомагаючи розробникам створювати власні хмарні додатки, які легше використовувати, забезпечити безпеку та бути в курсі подій.

4.2.3 Розробка Kubernetes оператора для тестування безпеки

Для роботи з кодом ми використовували Visual Studio Code, як один з найбільш зручних редакторів для написання коду. Він має необхідні плагіни для роботи з Go кодом та може інтегруватися з Kubernetes кластером.

Спочатку потрібно створити базову стрктуру проекта для розробки оператора. Для цього ми використаємо Operator SDK, який є частиною Operator фреймворку. Для створення, як показано на рисунку 4.3, потрібно вказати назву нашого оператора, після цього SDK автоматично згенерую файли-шаблони, які

потрібні для нормального функціонування оператора.

```
vagrant@vagrant:~/diploma/src$ operator-sdk new security_diploma_operator
INFO[0000] Creating new Go operator 'security_diploma_operator'.
INFO[0000] Created go.mod
INFO[0000] Created tools.go
INFO[0000] Created cmd/manager/main.go
INFO[0000] Created build/Dockerfile
INFO[0000] Created build/bin/entrypoint
INFO[0000] Created build/bin/user_setup
INFO[0000] Created deploy/service_account.yaml
INFO[0000] Created deploy/role.yaml
INFO[0000] Created deploy/role_binding.yaml
INFO[0000] Created deploy/operator.yaml
INFO[0000] Created pkg/apis/apis.go
INFO[0000] Created pkg/controller/controller.go
INFO[0000] Created version/version.go
INFO[0000] Created .gitignore
INFO[0000] Validating project
```

Рисунок 4.3 – Створення нового проекту з допомогою Operator SDK

Наступним кроком потрібно створити API версію, яку буде використовувати наший оператор під час роботи на платформі Kubernetes, що можна побачити на рисунку 4.4. Також додатково вказується назва ресурсу, з допомогою якого можна буде створювати нові екземпляри оператора на платформі.

```
vagrant@vagrant:~/diploma/src/security_diploma_operator$ operator-sdk add api --api-version=app.security.kokhanevych.com/v1alpha1 --kind=SecurityTesting
INFO[0000] Generating api version app.security.kokhanevych.com/v1alpha1 for kind SecurityTesting.
INFO[0000] Created pkg/apis/app/group.go
INFO[0064] Created pkg/apis/app/v1alpha1/securitytesting_types.go
INFO[0064] Created pkg/apis/addtoscheme_app_v1alpha1.go
INFO[0064] Created pkg/apis/app/v1alpha1/register.go
INFO[0064] Created pkg/apis/app/v1alpha1/doc.go
INFO[0064] Created deploy/crds/app.security.kokhanevych.com_v1alpha1_securitytesting_cr.yaml
INFO[0064] Running deepcopy code-generation for Custom Resource group versions: [app:[v1alpha1], ]
INFO[0091] Code-generation complete.
INFO[0091] Running CRD generator.
INFO[0093] CRD generation complete.
INFO[0093] API generation complete.
```

Рисунок 4.4 – Додавання API версії для оператора

Так як основним завданням оператора є автоматичне корегування конфігурацій розгорнутих додатків, йому для цього потрібний окремий об’єкт, що буде займатися даним типом роботи. В Operator SDK для постійного контролю та моніторингу стану додатків існує ресурс Controller, в логіці роботи

якого, можна вказати необхідні специфікації та вказівки по моніторингу ресурсів. У Kubernetes контролери - це інструменти управління, які спостерігають за станом кластера, а потім вносять або вимагають змін, де це необхідно. Кожен контролер намагається перемістити поточний стан кластера ближче до бажаного стану. Для створення шаблонів необхідних конфігураційних файлів, потрібно виконати спеціальну команду, як продемонстровано на рисунку 4.5.

```
vagrant@vagrant:~/diploma/src/security_diploma_operator$ operator-sdk add controller --api-version=app.security.kokhanevy
ch.com/v1alpha1 --kind=SecurityTesting
INFO[0000] Generating controller version app.security.kokhanevych.com/v1alpha1 for kind SecurityTesting.
INFO[0000] Created pkg/controller/securitytesting/securitytesting_controller.go
INFO[0000] Created pkg/controller/add_securitytesting.go
INFO[0000] Controller generation complete.
vagrant@vagrant:~/diploma/src/security_diploma_operator$
```

Рисунок 4.5 – Додавання контроллера для оператора

На цьому процес створення необхідних файлів завершується, кінцеву файлову структуру проекту можна побачити на рисунку 4.7.

Як вже було зазначино, Operator SDK генерує необхідні файли-шаблони, які можна редугувати з метою додавання необхідної логіки роботи оператора.

Основним файлом для роботи є securitytesting_controller.go, в ньому будемо реалізовувати основну логіку роботи по моніторингу та підтримки роботи сканерів тестування безпеки програмного коду. Для роботи зі сканерами безпеки розгорнутими в кластері, в даному файлі ми реалізували декілька методів, які необхідні для контролю та автоматичної конфігураці даних екземплярів. Основним методом є Reconcile, який відповідає за постійний моніторинг та контроль стану ресурсу, частина коду показана на рисунку 4.6 . В ньому ми описали логіку та початковий стан ресурсів, до якого потрібно повертати конфігурацію сканерів в разі несправностей. Також додатково описані інші методи для роботи з ресурсами, а саме updateStatus, updateAvailableStatus, та updateInstanceStatus, які допомагають проводити конфігурацію та обновлення стану ресурсів з огляду на отримані під час моніторингу дані. Контроллер з

огляду на прописані інструкції, проводить постійний контроль та налаштування сканерів.

```
func (r *ReconcileSecurityTesting) Reconcile(request reconcile.Request) (reconcile.Result, error) {
    reqLogger := log.WithValues("Request.Namespace", request.Namespace, "Request.Name", request.Name)
    reqLogger.Info("Reconciling SecurityTesting")

    // Fetch the SecurityTesting instance
    instance := &appv1alpha1.SecurityTesting{}
    err := r.client.Get(context.TODO(), request.NamespacedName, instance)
    if err != nil {
        if errors.IsNotFound(err) {
            // Request object not found, could have been deleted after reconcile request.
            // Owned objects are automatically garbage collected. For additional cleanup logic use final
            // Return and don't requeue
            return reconcile.Result{}, nil
        }
        // Error reading the object - requeue the request.
        return reconcile.Result{}, err
    }

    // Define a new Pod object
    pod := newPodForCR(instance)
```

Рисунок 4.6 – Частина коду методу контролю стану сканерів

Також важливим є файл securitytesting_types.go в якому визначається опис нашого ресурсу, його основні поля та необхідні параметри для розгортання в кластері Kubernetes.

Після того як визначена основна логіка роботи контроллера та вказаний опис ресурсу для розгортання, необхідно підготувати файл для розгортання ресурсу в кластер. Розгортання в платформі Kubernetes відбувається з допомогою файлів-темплейтів в форматі YAML. Operator SDK вміє генерувати подібні файли, після генерації він розміщає їх в папці deploy. Ці файли не рекомендується редагувати вручну, так як SDK сам їх обновляє після редагування відповідних файлів з описом ресурсу.

Опис всіх необхідних параметрів для розгортання нашого ресурсу, можна побачити в файлі app.security.kokhanevych.com_securitytestings_crd.yaml. В ньому вказані всі можливі параметри, їх тип та короткий опис для чого вони потрібні. В тій же папці знаходиться файл з прикладом для розгортання

app.security.kokhanevych.com_v1alpha1_securitytesting_cr.yaml, який генерується для безпосереднього розгортання в кластер Kubernetes.

```
vagrant@vagrant:~/diploma/src/security_diploma_operator$ tree
.
├── build
│   ├── bin
│   │   ├── entrypoint
│   │   └── user_setup
│   └── Dockerfile
├── cmd
│   └── manager
│       └── main.go
├── deploy
│   ├── crds
│   │   ├── app.security.kokhanevych.com_securitytestings_crd.yaml
│   │   └── app.security.kokhanevych.com_v1alpha1_securitytesting_cr.yaml
│   ├── operator.yaml
│   ├── role_binding.yaml
│   ├── role.yaml
│   └── service_account.yaml
├── go.mod
├── go.sum
├── pkg
│   ├── apis
│   │   ├── addtoscheme_app_v1alpha1.go
│   │   ├── apis.go
│   │   └── app
│   │       ├── group.go
│   │       └── v1alpha1
│   │           ├── doc.go
│   │           ├── register.go
│   │           ├── securitytesting_types.go
│   │           └── zz_generated.deepcopy.go
│   └── controller
│       ├── add_securitytesting.go
│       ├── controller.go
│       └── securitytesting
│           └── securitytesting_controller.go
├── tools.go
└── version
    └── version.go

13 directories, 24 files
```

Рисунок 4.7 – Файлова структура проекту

4.3 Створення тестового середовища розробки програмного забезпення

Після створення оператора для тестування безпеки програмного коду, потрібно створити тестове середовище розробки програмного забезпечення де можна було б ефективно протестувати наший оператор. Найкращим інструментом для швидкого створення середовища розробки є CI інструмент Jenkins. Він дозволяє описати процес збору програми та її тестування з допомогою коду, що є дуже ефективно та зручно

Для нашого випадку ми створили тестовий процес життєвого циклу

програмного забезпеченя на прикладі тестового Java додатку. Для цього в нашому кластері Kubernetes ми розгорнули інструмент Jenkins [10] та провели необхідні базові налаштування його роботи, що передбачає встановлення додаткових плагінів для інтеграції з кластером, як показано на рисунку 4.8.

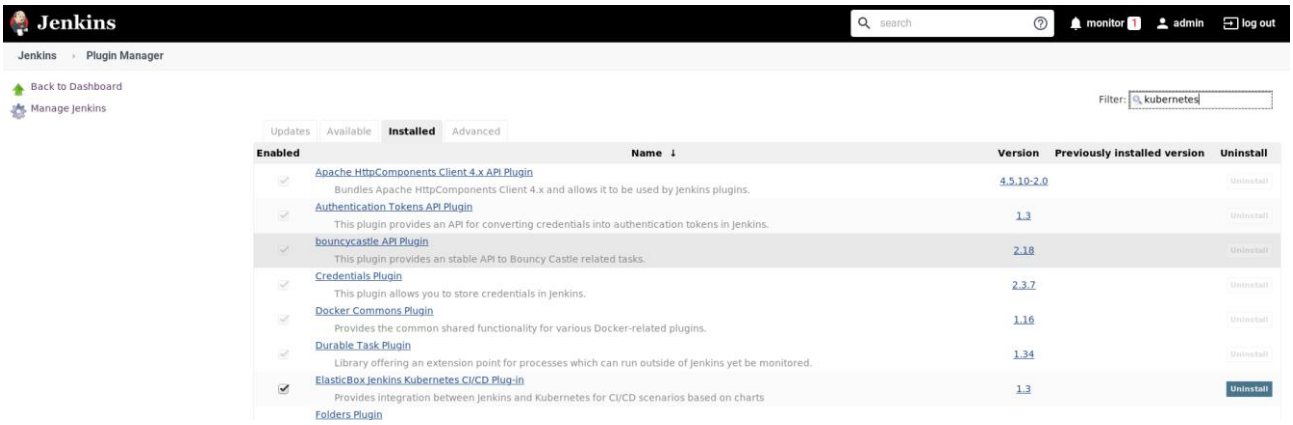

Рисунок 4.8 – Інструмент Jenkins та встановлені плагіни

Далі потрібно зробити опис процеса життєвого циклу програмного забезпечення. В Jenkins це робиться з допомогою пайплайнів написаних на мові програмування Groovy. В опис будуть включатися інструкції та команди для автоматизації циклу розробки програмного забезпечення. До його складу входить клонування програмного коду з github репозиторія, його компіляція, проведення тестування (unit, інтеграційні, перевірка безпеки), збір артефакта та безпосереднє розгортання готового додатку в кластері Kubernetes. Також Jenkins дозволяє відобразити роботу пайплайна, як продемонстровано на рисунку 4.9.

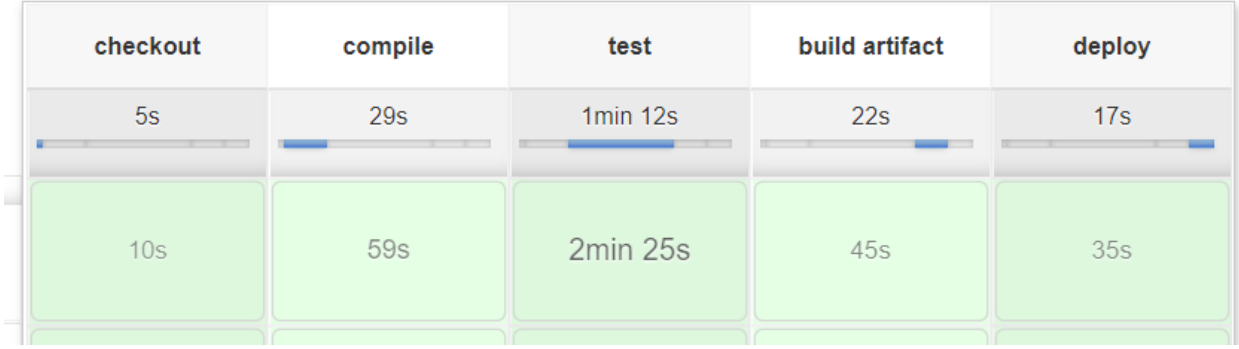

Рисунок 4.9 – Графічне відображення пайплайна в Jenkins

В рамках даної роботи був розроблений пайплайн на мові програмування Groovy згідно зі стандартами для роботи в інструменті Jenkins. Так як нам потрібно проводити розгортання в кластері Kubernetes, то потрібно описати контейнер в якому буде проводитися розгортання, що можна побачити на рисунку 4.10. Для зборки проекта нам потрібен інструмент для зборки Java, а саме Maven, як один з найбільш популярних. Також нам знадобиться контейнер з установленим Docker, тому що нам потрібно буде збирати контейнер з нашим додатком. Ну і для автоматичного розгортання використаємо інструмент розгортання ресурсів в кластерах Kubernetes Helm. З допомогою цього інструмента можна легко контролювати, які додатки були розгорнуті в кластері, а з допомогою автоматичного контролю версій можна легко керувати процесом випуску нових версій.

```
podTemplate(yaml: '''
apiVersion: v1
kind: Pod
metadata:
  labels:
    job: security-build-deploy
spec:
  containers:
  - name: maven
    image: maven:3.6.0-jdk-11-slim
    command: ["cat"]
    tty: true
  - name: docker
    image: docker:18.09.2
    command: ["cat"]
    tty: true
    volumeMounts:
    - name: docker-sock
      mountPath: /var/run/docker.sock
  - name: helm-cli
    image: alpine/helm:3.1.3
    command: ["cat"]
    tty: true
  volumes:
  - name: docker-sock
    hostPath:
      path: /var/run/docker.sock
''') {
```

Рисунок 4.10 – Код опису контейнерів для пайплайна в Jenkins

Етап тестування, як видно з рисунка 4.9, займає найбільше часу, тому що це не швидкий процес. Зазвичай на цьому етапі перевіряються функціональні та інтеграційні можливості програми, а в нашому випадку ще і проводиться тестування безпеки.

4.4 Тестуваннння створенного Kubernetes Оператора в середовищі розробки

Для тестування оператора, ми створемо кастомний ресурс для розгортання сканера безпеки, в нашому випадку це буде Spotbugs [9], та розгорнемо його в кластері Kubernetes. Spotbugs – це сканер статичного тестування безпеки програмного коду для мови програмування Java. Він досить просто інтегрується в процес розробки, але потребує постійного нагляду і налаштування. Тому ми автоматизуємо його підтримку з допомогою нашого Kubernetes оператора.

Спочатку потрібно розгорнути кастомний ресурс app.security.kokhanevych.com_v1alpha1_securitytesting_cr.yaml створенний в процесі розробки оператора.

```yaml
apiVersion: app.security.kokhanevych.com/v1alpha1
kind: SecurityTesting
metadata:
  name: example-securitytesting
spec:
  version: "3.1.1"
  image: "devsecopsat/spotbugs"
  initImage: "busybox"
  volumes:
    - name: "data"
      storageClass: "gp2"
      capacity: "1Gi"
  plugin: maven
  reportType: file
```

Рисунок 4.11 – Код кастомноого ресурсу для розгортання Spotbugs

Розгортання проводиться з допомогою стандартних засобів в Kubernetes. Після розгортання ми отримаємо працюючий екземпляр сканера Spotbugs, який буде на постійному моніторигу та контролю в нашого оператора, результат розгортання в кластері Kubernetes можна побачити на рисутнку 4.12.

```
vagrant@vagrant:~/diploma/src/security_diploma_operator$ kubectl get pods
NAME       READY   STATUS    RESTARTS   AGE
jenkins    1/1     Running   1          4d23h
spotbugs   1/1     Running   0          9m26s
```

Рисунок 4.12 – Працюючі екземпляри Spotbugs та Jenkins

Наступним кроком потрібно інтегрувати перевірку безпеки з допомогою Spotbugs в наший пайплайн. Для цього на етап тестування додамо ще один крок по перевірці безпеки. Так як ми використовуємо Maven як інструмент для роботи з Java кодом, то ми можемо використати його можливості по інтеграції з різними сканерами. З допомогою плагіна для Maven виконаємо сканування безпеки нашого програмного коду сканером Spotbugs, код етапа тестування в Jenkins пайплайні можна побачити на рисунку 4.13.

```
stage ('test') {
  container('maven') {
    try {
        sh """
            mvn compile -B --settings ${vars.mavenSettings}
            $SPOTBUGS_HOME/bin/spotbugs -textui -pluginList $SPOTBUGS_HOME/plugin/findsecbugs-plugin-1.8.0.jar \
            -xml:withMessages -output $DOJO_API_FILE -low -progress -effort:max -bugCategories SECURITY .
        """
        archiveArtifacts artifacts: 'spotbugs-result.html'
    } catch (Exception ex) {
        println("[JENKINS][ERROR] SPOTBUGS stage has failed. Reason - ${ex}")
    }
  }
}
```

Рисунок 4.13 – Код етапа тестування безпеки в Jenkins пайплайні

Після сканування Spotbugs генерує звіт з результатами тестування в форматі HTML. Робота сканера ніяк не залежить від роботи Jenkins і повністю контролюється оператором Kubernetes.

	Severity	Name	CWE	Date	Age	SLA	Reporter	Status
☐	High	MD2, MD4 and MD5 are weak hash functions	328	Sept. 4, 2020	96	66	Admin User	Active, Verified
☐	Medium	Potential Path Traversal (file read)	22	Sept. 4, 2020	96	6	Admin User	Active, Verified
☐	Medium	Potential Path Traversal (file read)	22	Sept. 4, 2020	96	6	Admin User	Active, Verified
☐	Medium	Potential Path Traversal (file read)	22	Sept. 4, 2020	96	6	Admin User	Active, Verified
☐	Medium	Potential Path Traversal (file read)	22	Sept. 4, 2020	96	6	Admin User	Active, Verified
☐	Medium	Potential CRLF Injection for logs	117	Sept. 4, 2020	96	6	Admin User	Active, Verified
☐	Medium	Potential Path Traversal (file read)	22	Sept. 4, 2020	96	6	Admin User	Active, Verified

Рисунок 4.14 – Результат сканування безпеки сканером Spotbugs

Було проведено декілька запусків пайплайна для отримання коректних результатів тестування, сканування проводилося для декількох різних додатків написаних на мові програмування Java, з результатами сканування безпеки можна ознайомитися в таблиці 4.1, там приведена кількість вразливстей, які були

знайдені під час нашого тестування та їх критичність.

Таблиця 4.1 – Результати сканування безпеки з допомогою Spotbugs

№ додатку	Висока критичність	Середня критичність	Низька критичність	Загальна к-сть вразливостей
1	1	15	17	33
2	3	14	21	38
3	2	18	15	35
4	0	24	12	36

По результатам тестування можна зробити висновок, що розроблений нами оператор працює достатньо ефективно. З допомогою Kubernetes API оператор має змогу керувати конфігураціями сканера безпеки, що дає нам можливість досить просто та швидко інтегрувати його в наший процес розробки програмного забезпечення. Від розробника не вимагається спеціальних знать по роботі та конфігураціям сканера, так як всі налаштування робить оператор, тому це надає змогу використовувати тестування безпеки більш широко і швидко.

Як варінт подальшої роботи над проектом, можна розширювати набір інструментів тестування безпеки для нашого оператора. Для автоматичного вибору потрібного сканера можна використати інструмент автоматичного розгортання в Kubernetes кластер Helm.

ВИСНОВКИ

В процесі написання атестаційної роботи було створено засіб автоматизації тестування безпеки програмного забезпечення з допомогою Kubernetes оператора та проведено його тестування в життєвому циклі програмного зебезпечення. В результаті роботи отримано готовий до роботи інструмент автоматизації тестування безпеки програмних додатків на платформі Kubernetes. Також було створено середовище розробки програмного забезпечення на базі інструмента Jenkins та проведено тестування створеного Kubernetes оператора.

Метою даної роботи було проведення аналізу проблеми автоматизації тестування безпеки програмних додатків та створення ефективного інструмента автоматизації на базі платформи Kubernetes.

Під час виконання атестаційної роботи були виконанні наступні дії:

— проведено огляд життєвого циклу програмного забезпечення;

— розглянуто місце тестування в процесі розробки програмних продуктів;

— розглянуто тестування безпеки, як частину процесу безпечної розробки програмного забезпечення;

— проведено аналіз проблем автоматизації тестування безпеки в процесі SDLC;

— розглянуто платформу Kubernetes, її структуру та особливості роботи;

— проведено дослідження Kubernetes операторів, як засобу автоматизації тестування безпеки програмного забезпечення;

— розгорнуто платформу Kubernetes;

— розроблено Kubernetes оператор для автоматизації роботи інструменту статичного тестування безпеки програмного забезпечення;

— проведено розгортання оператора на платформі Kubernetes;

— створено середовище розробки програмного забезпечення на базі інструмента Jenkins;

— проведено тестування роботи створеного оператора в процесі розробки програмного забезпечення;

— зроблені висновки та надані рекомендації, щодо майбутніх покращень в роботі оператора.

ПЕРЕЛІК ПОСИЛАНЬ

1. Glenford J. Myers. The Art of Software Testing. – 2011. – 256 с.

2. Nayan B. Ruparelia. Software development lifecycle models. International scientific conference. – 2010. – 6 с.

3. Nirali Honest. Role of Testing in Software Development Life Cycle. International Journal of Computer Sciences and Engineering. – 2019. – 6 с.

4. OWASP Testing Guide 4.0 [Електронний ресурс] – Режим доступу: https://owasp.org/www-pdf-archive/OTGv4.pdf

5. OWASP Software Assurance Maturity Model [Електронний ресурс]: Режим доступу: https://owaspsamm.org

6. Офіційний сайт інструмента Kubespray [Електронний ресурс]: Режим доступу: https://kubespray.io

7. Офіційний сайт платформи Kubernetes [Електронний ресурс]: Режим доступу: https://kubernetes.io/docs/home/

8. Офіційна документація Kubernetes Operators [Електронний ресурс]: Режим доступу: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

9. Офіційний сайт інструмента Spotbugs [Електронний ресурс]: Режим доступу: https://spotbugs.github.io

10. Офіційний сайт інструмента Jenkins [Електронний ресурс]: Режим доступу: https://www.jenkins.io

11. Коханевич Є.Г., Федюшин О.І Автоматизація аналізу безпеки програмного коду за допомогою платформи Kubernetes // Восьма міжнародна науково-технічна конференція «Проблеми інформатизації». Зб. матеріалів форуму. – Харків: ХНУРЕ. 2020. – С. 95.