Хмельницький національний університет
Факультет інформаційних технологій
Кафедра кібербезпеки

КВАЛІФІКАЦІЙНА РОБОТА

Яворський Олександр Віталійович

на здобуття ступеня вищої освіти магістра

Метод оцінювання інформаційної безпеки безпечного функціонування програмного забезпечення

Галузь знань 12 – Інформаційні технології
Спеціальність 125 – Кібербезпека та захист інформації
Освітня програма Кібербезпека та захист інформації

Шифр КРМКБЗІ. 2301154.23.01.22 ПЗ

Виконав студент 2 курсу група КБЗІм-23-1 Олександр ЯВОРСЬКИЙ
Керівник канд. техн. наук, доцент Володимир ДЖУЛІЙ
Нормоконтролер старший викладач Сергій МОСТОВИЙ

До захисту допускаю:
Завідувач кафедри кібербезпеки Юрій КЛЬОЦ

16 12 2024 р.

Хмельницький 2024

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Кібербезпеки

Рівень вищої освіти Магістр

Галузь знань 12 – Інформаційні технології

Спеціальність 125 – Кібербезпека та захист інформації

Освітня програма Кібербезпека та захист інформації

ЗАТВЕРДЖУЮ

Завідувач кафедри кібербезпеки

Юрій КЛЬОЦ

2 09 2024 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Яворському Олександру Віталійовичу

1 Тема роботи Метод оцінювання інформаційної безпеки безпечного функціонування програмного забезпечення

Керівник роботи канд.техн.наук, доцент Володимир ДЖУЛІЙ

Затверджено наказом ректора університету від 26 08 2024 № 60

2 Строк подання студентом кваліфікаційної роботи на кафедру 27.11.2024

3 Вихідні дані до роботи забезпечення безпеки безпечного функціонування програмного забезпечення за допомогою його оцінювання

4 Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Вступ. Дослідження актуальних методів та технологій оцінювання інформаційної безпеки програмного забезпечення. Модель процесу оцінювання інформаційної безпеки програмного забезпечення. Метод оцінювання інформаційної безпеки програмного забезпечення. Прикладне застосування методу оцінювання інформаційної безпеки програмного забезпечення. Висновки.

5 Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

6 Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7 Дата видачі завдання 2 09 2024 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
Ґрунтовне ознайомлення та дослідження предметної галузі	13.09.2024	Виконано
Визначення змісту, структури кваліфікаційної роботи	23.09.2024	Виконано
Підготовка першого розділу кваліфікаційної роботи	30.09.2024	Виконано
Підготовка другого розділу кваліфікаційної роботи	2.10.2024	Виконано
Підготовка третього розділу кваліфікаційної роботи	6.10.2024	Виконано
Підготовка статті/тези за темою кваліфікаційної роботи	8.11.2024	Виконано
Підготовка четвертого розділу кваліфікаційної роботи	18.11.2024	Виконано
Підготовка та оформлення ілюстративного матеріалу	22.11.2024	Виконано
Оформлення кваліфікаційної роботи	25.11.2024	Виконано
Попередній захист кваліфікаційної роботи	27.11.2024	Виконано
Захист кваліфікаційної роботи на засіданні ЕК	19.12.2024	Виконано

Студент Олександр ЯВОРСЬКИЙ

Керівник кваліфікаційної роботи Володимир ДЖУЛІЙ

ANNOTATION

Theme of qualification work: Information security assessment method for secure software operation

Author of the work: Yavorskyi Oleksandr Vitaliiovych

Mentor: Ph.D. Dzhulii Volodymyr Mykolayovych

Total volume of work: 87 pages, 24 figures, 2 appendices, 61 links.

Keywords: quantum channel, quantum key distribution, trusted nodes, secret key.

Modern software development systems face ever-increasing security threats that arise from the use of third-party dependencies, code errors, and architectural flaws. Traditional approaches to software security assessment are often fragmented and do not integrate static and dynamic analysis, which limits their effectiveness in identifying critical vulnerabilities.

The master's thesis solves the scientific problem of creating a method for assessing software information security that allows combining static, dynamic code analysis, and dependency checking. A methodology has been developed that automates the analysis process using tools integrated into a single CLI system.

The proposed solution is characterized by the ability to generate standardized security reports, adapt to different programming languages, and support scalability for large projects. This allows for significantly increasing the efficiency of security monitoring, minimizing the human factor in the analysis process. The results of the work can be used by developers, testers, and cybersecurity specialists to improve the processes of software development and security assurance.

02.12.2024

АНОТАЦІЯ

Тема кваліфікаційної роботи: Метод оцінювання інформаційної безпеки безпечного функціонування програмного забезпечення

Автор роботи: Яворський Олександр Віталійович

Керівник роботи: к.т.н., доц. Джулій Володимир Миколайович

Загальний обсяг роботи: 87 сторінок, 24 рисунків, 2 додатки, 61 посилання.

Ключові слова: оцінювання безпеки, аналіз програмного забезпечення, безпечне функціонування програмного забезпечення, інформаційна безпека.

Сучасні системи розробки програмного забезпечення стикаються з постійно зростаючими загрозами безпеці, які виникають через використання сторонніх залежностей, помилки в коді та недоліки в архітектурі. Традиційні підходи до оцінювання безпеки програмного забезпечення часто є фрагментованими та не інтегрують статичний і динамічний аналіз, що обмежує їх ефективність у виявленні критичних вразливостей.

У магістерській роботі вирішується наукова задача створення методу оцінювання інформаційної безпеки програмного забезпечення, який дозволяє поєднати статичний, динамічний аналіз коду та перевірку залежностей. Розроблено методику, яка забезпечує автоматизацію процесу аналізу за допомогою інструментів, інтегрованих у єдину CLI-систему.

Запропоноване рішення характеризується можливістю генерації стандартизованих звітів про безпеку, адаптацією до різних мов програмування та підтримкою масштабованості для великих проектів. Це дозволяє значно підвищити ефективність моніторингу безпеки, мінімізуючи людський фактор у процесі аналізу. Результати роботи можуть бути використані розробниками, тестувальниками та спеціалістами з кібербезпеки для покращення процесів розробки та забезпечення захисту програмного забезпечення.

02.12.2024

ЗМІСТ

ВСТУП

Сучасний світ стикається зі стрімким зростанням кількості кіберзагроз, що є особливо небезпечним для бізнесу, який активно використовує цифрові технології. Забезпечення інформаційної безпеки програмного забезпечення стає критично важливим у таких напрямках, як електронна комерція, фінанси, охорона здоров'я, промисловість, енергетика, транспорт та багато інших. Онлайн-магазини та маркетплейси, які працюють у сфері електронної комерції, обробляють значну кількість чутливих даних клієнтів, таких як персональні дані, номери кредитних карток, історія покупок. Для них ключовими загрозами є атаки на платіжні системи, викрадення даних клієнтів, фішингові кампанії та DDoS-атаки, які можуть паралізувати роботу платформи.

Фінансовий сектор, зокрема банки та платіжні системи, є однією з найбільш привабливих мішеней для хакерів. Величезні обсяги фінансових транзакцій та конфіденційних даних піддаються ризикам через атаки типу "людина посередині", ін'єкції SQL, зловмисне програмне забезпечення або атаки на мобільні додатки. Використання надійних методів оцінки безпеки дозволяє вчасно виявляти слабкі місця в інфраструктурі та уникати значних фінансових втрат.

У сфері охорони здоров'я цифрові системи зберігають медичні записи пацієнтів, результати діагностики, історії хвороб та іншу конфіденційну інформацію. Кіберзагрози, такі як витік даних, шантаж або маніпуляції з медичною інформацією, можуть мати катастрофічні наслідки для пацієнтів і репутації медичних установ. Розробка методів оцінки безпеки допомагає виявляти вразливості в системах зберігання та обміну даних.

Для промислових підприємств та енергетичних компаній, які все більше покладаються на автоматизовані системи управління, основними ризиками є атаки на інфраструктуру, саботаж або несанкціоноване втручання у виробничі процеси. Такі загрози можуть спричинити серйозні фінансові втрати, порушення технологічних циклів або навіть загрозу життю людей.

Забезпечення безпеки програмного забезпечення в цих секторах допомагає уникнути катастрофічних наслідків.

Транспортна галузь, яка активно впроваджує інтелектуальні системи управління, також стикається з проблемами безпеки. Загрози, такі як злом навігаційних систем, втручання в роботу автопілотів або атак на логістичні платформи, можуть призвести до серйозних збоїв у роботі інфраструктури. Надійні методи оцінки безпеки дозволяють забезпечити стабільність роботи систем і мінімізувати ризики збоїв.

Таким чином, забезпечення інформаційної безпеки програмного забезпечення є ключовим фактором стабільного функціонування цифрових систем у різних галузях. Розробка ефективних методів оцінки безпеки сприяє захисту чутливих даних, стабільності бізнесу та зменшенню ризиків, пов’язаних із кіберзагрозами.

Метою даної дипломної роботи є розробка методу оцінювання інформаційної безпеки програмного забезпечення, який дозволяє об'єктивно аналізувати ризики, виявляти вразливості та забезпечувати високий рівень безпечного функціонування систем. Основна увага приділяється інтеграції сучасних підходів до оцінки безпеки з використанням інструментів програмування, зокрема мови Go, яка дозволяє створювати ефективні та продуктивні рішення.

Для досягнення мети роботи визначено такі завдання:

– провести аналіз існуючих методів оцінювання інформаційної безпеки та виявити їхні основні недоліки;

– розробити метод оцінювання безпеки програмного забезпечення, яка буде зрозумілою, ефективною та адаптованою для автоматизації;

– реалізувати розроблений метод у вигляді програмного прототипу на мові програмування Go;

– провести тестування та аналіз результатів роботи розробленого прототипу;

– порівняти ефективність запропонованого методу з існуючими

підходами.

Існуючі методи оцінювання безпеки коду мають суттєві недоліки, які обмежують їхню ефективність у сучасних умовах. Традиційні підходи, такі як статичний аналіз коду, часто видають багато хибнопозитивних результатів, що ускладнює їх використання. Крім того, вони можуть не враховувати специфіку сучасних мов програмування, таких як Go, або нових архітектур, наприклад, мікросервісів.

Інструменти аналізу безпеки часто складні у налаштуванні та інтеграції. Це вимагає додаткових ресурсів і експертних знань, що є перешкодою для їхнього широкого впровадження в компаніях.

Багато існуючих методів не адаптовані для сучасних мов програмування. Наприклад, Go, яка є популярною завдяки своїй продуктивності та простоті, має специфіку (управління пам’яттю, конкурентність), яку часто ігнорують стандартні інструменти аналізу.

Наукова новизна дослідження полягає у створенні модульного підходу для інтеграції інструментів статичного аналізу безпеки програмного забезпечення, що забезпечує автоматизацію та спрощення процесу перевірки коду. Розроблене рішення дозволяє об’єднати результати роботи різних інструментів у єдиний формат, зручний для аналізу, та адаптувати систему для інтеграції в процеси CI/CD. Важливою складовою новизни є здатність системи масштабуватися для аналізу великих проектів і підтримувати різні мови програмування без значних змін архітектури.

Дослідження також включає порівняльний аналіз ефективності обраних інструментів на тестових даних, що дозволяє оцінити якість їхньої роботи в реальних умовах. Результати підтверджують актуальність розробленої системи для забезпечення інформаційної безпеки програмного забезпечення.

Таким чином, дана робота робить акцент на інтеграції та адаптації існуючих методів, пропонуючи зручний механізм для їх використання в реальних умовах. Це сприяє поглибленню практичної реалізації сучасних підходів до забезпечення інформаційної безпеки програмного забезпечення.

1 ДОСЛІДЖЕННЯ АКТУАЛЬНИХ МЕТОДІВ ТА ТЕХНОЛОГІЙ ОЦІНЮВАННЯ ІНФОРМАЦІЙНОЇ БЕЗПЕКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Аналіз поточних підходів до оцінки безпеки програмного забезпечення

Інформаційна безпека програмного забезпечення є однією з ключових проблем сучасного суспільства, яке все більше залежить від цифрових технологій. Програмні продукти використовуються в усіх сферах життєдіяльності: від фінансів і охорони здоров'я до транспортної інфраструктури та оборонної промисловості. Будь-яка уразливість у програмному забезпеченні може призвести до серйозних наслідків, включаючи витік конфіденційної інформації, фінансові втрати або навіть порушення функціонування критично важливих систем [1].

На сьогодні існують різноманітні методи забезпечення інформаційної безпеки, серед яких виділяються статичний і динамічний аналіз програмного забезпечення. Вони дозволяють виявляти потенційні уразливості як на етапі розробки, так і під час експлуатації програмного продукту. Проте, існуючі інструменти не завжди забезпечують повний охват усіх можливих загроз через високу складність аналізу, що пов'язано із зростаючою складністю сучасного програмного забезпечення та використанням новітніх технологій.

Практична значущість роботи полягає у впровадженні інтеграції з існуючими інструментами аналізу коду, що дозволяє знизити час на перевірку та поліпшити якість звітів про вразливості. Дослідження орієнтоване на розробку ефективного підходу, що забезпечить підвищену безпеку програмного забезпечення та стане основою для майбутніх розробок у цій сфері.

Статичний аналіз коду є одним із ключових методів оцінювання безпеки програмного забезпечення. Його сутність полягає у дослідженні вихідного коду без його виконання, що дозволяє ідентифікувати потенційні уразливості та помилки ще на етапі розробки. Це забезпечує можливість виявлення проблем на ранніх етапах життєвого циклу програмного забезпечення, що значно

знижує витрати на їх виправлення.

Цей підхід базується на автоматичному скануванні коду із застосуванням спеціалізованих алгоритмів і правил. Завдяки цьому можна виявляти небезпечні конструкції коду, аналізувати потоки даних для пошуку ін’єкцій чи витоків конфіденційної інформації, а також забезпечувати дотримання стандартів програмування. У більшості випадків результати аналізу подаються у вигляді детальних звітів, які містять вказівки на конкретні ділянки коду з описом виявлених проблем і рекомендаціями щодо їх усунення [2,3].

Однією з головних переваг статичного аналізу є його здатність виявляти уразливості до моменту виконання коду. Це дозволяє підвищити якість програмного забезпечення та зменшити трудовитрати на виправлення помилок. Інструменти статичного аналізу також можуть працювати з великою кодовою базою, інтегруючись у конвеєри CI/CD, що забезпечує автоматичну перевірку з кожною новою зміною в коді. Однак, цей метод має і свої обмеження. Він не здатний враховувати динамічний контекст виконання програми, що унеможливлює виявлення деяких специфічних вразливостей. Крім того, статичний аналіз часто видає хибні спрацювання, які вимагають додаткової перевірки вручну.

Незважаючи на певні обмеження, статичний аналіз коду є незамінним інструментом для виявлення таких поширених вразливостей, як SQL-ін’єкції, недостатня перевірка вхідних даних, використання застарілих криптографічних алгоритмів чи жорстко закодовані облікові дані. Завдяки автоматизації та високій швидкості перевірки цей метод активно використовується для забезпечення безпеки програмного забезпечення на всіх етапах розробки.

Статичний аналіз коду стає основою для побудови комплексної системи забезпечення безпеки, оскільки дозволяє систематично ідентифікувати найбільш критичні проблеми ще до виходу програмного продукту в експлуатацію. Його застосування сприяє значному зниженню ризиків, пов’язаних із впровадженням небезпечного коду в готовий продукт, і забезпечує якість та надійність розроблюваних рішень.

Динамічний аналіз є методом оцінювання безпеки програмного забезпечення, який здійснюється під час виконання програми. На відміну від статичного аналізу, цей підхід дозволяє виявляти проблеми, що проявляються лише в процесі роботи системи. Це включає поведінку програми в різних середовищах, взаємодію з іншими компонентами та реакцію на неочікувані вхідні дані.

Основна мета динамічного аналізу полягає у визначенні реальних загроз, які можуть бути використані зловмисниками. Наприклад, метод дозволяє виявляти уразливості, пов’язані з некоректною обробкою даних користувача, витоком пам’яті, некоректною автентифікацією чи неправильною роботою під час високих навантажень. У багатьох випадках результати такого аналізу є більш точними, ніж у статичних методах, оскільки вони враховують специфіку середовища виконання та взаємодії компонентів.

Процес динамічного аналізу зазвичай включає виконання тестів, які імітують реальні умови використання програми. Це можуть бути автоматизовані сценарії або ручне тестування, спрямоване на перевірку найбільш критичних функціональних блоків. Використання спеціалізованих інструментів дозволяє моніторити поведінку системи, фіксувати аномалії та оцінювати їхній вплив на безпеку [4,6,7].

Сильними сторонами динамічного аналізу є його здатність враховувати реальний контекст виконання, що дозволяє виявляти складні вразливості, які неможливо знайти за допомогою статичного аналізу. Це особливо актуально для програм, що взаємодіють із мережею, обробляють великі обсяги даних або працюють у багатокористувацькому середовищі. Проте метод має і свої недоліки. Він вимагає більше часу та ресурсів, оскільки потребує створення тестового середовища та виконання тестів. Крім того, аналіз охоплює лише ті аспекти, які були протестовані, тому існує ризик пропустити уразливості, які проявляються за інших умов.

Динамічний аналіз зазвичай застосовується разом зі статичним, що дозволяє поєднати переваги обох методів. Такий підхід забезпечує більш

глибоке розуміння безпеки системи, охоплюючи як теоретичні, так і практичні аспекти. У сучасній практиці динамічний аналіз є важливим інструментом для тестування веб-додатків, серверних систем, API та інших компонентів, що працюють у складних і змінних умовах.

Аналіз потоків даних є сучасним підходом до оцінки безпеки програмного забезпечення, який спрямований на відстеження руху даних від джерела до точки їх використання. Цей метод дозволяє ідентифікувати уразливості, що виникають внаслідок неправильного або небезпечного використання даних. Основною метою такого аналізу є забезпечення захисту від атак, що експлуатують слабкі місця в обробці вхідної інформації, таких як SQL-ін’єкції, XSS або витоки конфіденційних даних.

Суть методу полягає у створенні моделі програми, що відображає всі можливі маршрути передачі даних через її компоненти. Ця модель дозволяє аналізувати, як дані обробляються, перевіряються і змінюються, а також визначати, чи можуть вони бути використані зловмисниками. Наприклад, аналіз потоків даних може допомогти виявити, що дані з вхідного параметра передаються до SQL-запиту без належного екранування, створюючи ризик SQL-ін’єкції.

Метод аналізу потоків даних є особливо корисним у випадках, коли необхідно відстежити складні ланцюги передачі інформації, які можуть охоплювати кілька модулів або функцій. Це дозволяє виявляти проблеми, які неочевидні при поверхневому аналізі, наприклад, неправильну валідацію вхідних даних або передачу конфіденційної інформації через небезпечні канали.

Попри значну ефективність, аналіз потоків даних має свої обмеження. Його проведення потребує значних обчислювальних ресурсів, особливо для великих систем із великою кількістю модулів і складною архітектурою. Крім того, успішне застосування методу залежить від якості побудови моделі програми, що може вимагати додаткових зусиль з боку розробників [5,7].

Аналіз потоків даних зазвичай використовується в поєднанні з іншими

методами, такими як статичний і динамічний аналіз. Це дозволяє отримати більш повну картину безпеки системи. Метод є незамінним для виявлення вразливостей у сучасних додатках, які працюють із великими обсягами даних і піддаються численним загрозам, що пов'язані з обробкою зовнішніх запитів.

Таким чином, аналіз потоків даних виступає потужним інструментом у забезпеченні інформаційної безпеки програмного забезпечення. Його використання дозволяє ідентифікувати та усунути складні уразливості, що робить програмні продукти більш стійкими до атак і безпечними для користувачів.

Методологія "чорної скриньки" є одним із підходів до тестування безпеки програмного забезпечення, який базується на аналізі системи без доступу до її вихідного коду або внутрішньої архітектури. Цей підхід моделює поведінку зловмисника, який не має внутрішніх знань про систему, але намагається знайти і експлуатувати її вразливості через взаємодію із зовнішніми інтерфейсами.

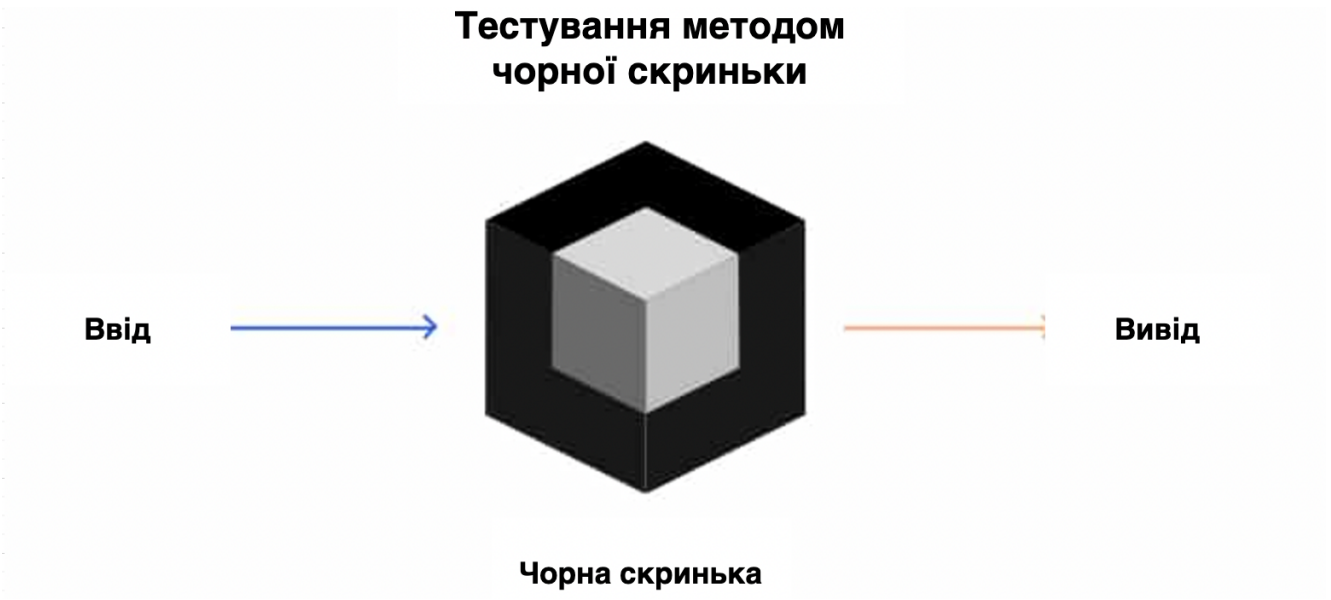


Рисунок 1.1 – Схема методу “чорної скриньки”

Суть методології полягає у тестуванні компонентів системи з точки зору кінцевого користувача або зовнішнього атакуючого. Для цього використовуються інструменти, які симулюють атаки на систему, генерують

великі обсяги запитів або аналізують поведінку системи у відповідь на аномальні дії. Наприклад, метод "чорної скриньки" дозволяє виявити уразливості в мережевих протоколах, веб-інтерфейсах, API або інших точках взаємодії, які можуть бути відкритими для атак.

Перевагою цього підходу є те, що він дозволяє оцінити реальну стійкість системи до атак без потреби у внутрішньому доступі до її коду. Це робить метод універсальним, адже він може застосовуватися до будь-яких програмних рішень, незалежно від їх архітектури чи мови програмування. Крім того, "чорна скринька" дозволяє тестувати систему в її робочому середовищі, враховуючи вплив мережевих затримок, конфігураційних помилок або інших зовнішніх факторів.

Однак, метод має і свої недоліки. Оскільки тестувальник не має доступу до внутрішньої логіки системи, деякі уразливості можуть залишитися непоміченими. Тестування охоплює лише ті аспекти, які були явно перевірені, що створює ризик пропустити проблеми в малодоступних або непопулярних функціях [8]. Крім того, метод "чорної скриньки" вимагає значних витрат часу і ресурсів, особливо якщо тестування проводиться вручну або потребує розробки складних сценаріїв.

У сучасній практиці методологія "чорної скриньки" широко використовується для тестування веб-додатків, мобільних додатків і API. Вона часто є основою для пентестів, коли команда спеціалістів-експертів виконує роль зловмисників, намагаючись знайти слабкі місця у захисті системи. Для підвищення ефективності тестування цей метод часто поєднують із іншими підходами, такими як статичний і динамічний аналіз, що дозволяє отримати більш повну картину загального рівня безпеки програмного забезпечення.

Таким чином, методологія "чорної скриньки" є важливим компонентом сучасного підходу до забезпечення інформаційної безпеки. Вона дозволяє оцінити рівень стійкості системи до атак і надати розробникам практичні рекомендації щодо вдосконалення захисту.

Ручний аудит коду є одним із найбільш надійних і ретельних методів

оцінки безпеки програмного забезпечення. Цей підхід передбачає детальне вивчення вихідного коду спеціалістами з інформаційної безпеки або розробниками з метою виявлення вразливостей, які можуть залишитися непоміченими автоматизованими інструментами. Унікальність методу полягає в його здатності враховувати специфічний контекст розробки та архітектури програмного забезпечення, що робить його надзвичайно ефективним для аналізу критично важливих систем.

Основною метою ручного аудиту є ідентифікація складних вразливостей, які не піддаються автоматичному виявленню. Наприклад, ручний аналіз дозволяє виявити логічні помилки у бізнес-логіці програми, некоректну обробку винятків або недоліки у реалізації криптографічних механізмів. Крім того, під час ручного аудиту експерти можуть враховувати специфічні вимоги до безпеки, які є важливими для конкретної галузі чи проекту.

Проведення ручного аудиту включає кілька основних етапів. Спочатку експерти аналізують архітектуру системи, щоб визначити ключові модулі, які потребують підвищеної уваги. Далі здійснюється перегляд коду з фокусом на виявлення небезпечних практик програмування, таких як недостатня валідація вхідних даних, використання застарілих бібліотек або функцій, а також неналежне управління пам'яттю. Результати аудиту документуються у вигляді звіту, який містить опис виявлених проблем, їхній вплив на безпеку системи та рекомендації щодо усунення.

Серед переваг ручного аудиту можна виділити його високу точність і здатність виявляти проблеми, які неможливо автоматизувати. Наприклад, це можуть бути складні уразливості, що виникають у результаті взаємодії кількох компонентів системи. Крім того, ручний аналіз дозволяє враховувати контекст і логіку роботи програми, що робить його незамінним для перевірки критично важливих функцій. Проте цей підхід має і свої недоліки. Він є доволі трудомістким і потребує значного часу, особливо для великих проектів. Крім того, ефективність аудиту залежить від кваліфікації експертів і їхнього розуміння архітектури системи.

У сучасній практиці ручний аудит коду часто застосовується у поєднанні з автоматизованими інструментами аналізу [9]. Такий підхід дозволяє спочатку автоматично виявити типові проблеми, після чого експерти зосереджуються на більш складних і специфічних аспектах. Це поєднання забезпечує як високу точність, так і ефективність перевірки.

Таким чином, ручний аудит коду є важливим інструментом у забезпеченні інформаційної безпеки програмного забезпечення. Його використання дозволяє виявляти складні уразливості, підвищувати якість коду та забезпечувати відповідність програмного продукту високим стандартам безпеки.

Аналіз поточних підходів до оцінки безпеки програмного забезпечення демонструє широкий спектр методів, кожен із яких має свої переваги, недоліки та специфічну сферу застосування. Статичний аналіз забезпечує швидке виявлення типових вразливостей на ранніх етапах розробки, тоді як динамічний аналіз дозволяє оцінити поведінку системи в реальному середовищі виконання. Аналіз потоків даних забезпечує глибоке розуміння взаємодії компонентів програми, що робить його незамінним для ідентифікації складних загроз. Методологія "чорної скриньки" дозволяє симулювати дії зловмисника, орієнтуючись на зовнішні точки доступу, а ручний аудит забезпечує найвищу точність при перевірці критичних функцій.

1.2 Огляд існуючих інструментів для аналізу коду

Інструменти для аналізу коду є важливим компонентом у забезпеченні інформаційної безпеки програмного забезпечення. Вони дозволяють автоматизувати процес виявлення вразливостей та підвищувати якість коду, що особливо важливо в умовах зростання складності сучасних систем. Існуючі інструменти поділяються на кілька категорій залежно від підходу до аналізу, серед яких найбільш поширеними є статичний аналіз, динамічний аналіз та

перевірка залежностей. Кожна з цих категорій має свої переваги і недоліки, що визначають їх застосування у конкретних умовах.

У цьому розділі розглянуто найбільш поширені інструменти для кожного з підходів, їхні особливості та роль у процесі забезпечення безпеки програмного забезпечення. Описані приклади ілюструють можливості кожного методу, а також їхнє місце у сучасних процесах розробки [10,14].

Статичний аналіз коду є одним із ключових підходів до забезпечення інформаційної безпеки програмного забезпечення. Він дозволяє досліджувати вихідний код програми без його виконання, забезпечуючи можливість виявлення вразливостей ще на етапі розробки. Завдяки цьому статичні аналізатори коду є незамінними в процесі побудови безпечних програмних рішень.

Одним із найпоширеніших інструментів для аналізу коду на Java є SpotBugs. Цей аналізатор орієнтований на виявлення багів у Java-коді, включаючи потенційно небезпечні конструкції, некоректне використання ресурсів та логічні помилки. SpotBugs інтегрується з іншими інструментами розробки, що робить його популярним серед команд, які працюють із масштабними проектами.

Для програмного забезпечення на базі PHP часто застосовується PHPStan. Цей інструмент виконує статичний аналіз коду, акцентуючи увагу на виявленні помилок у типізації, некоректному виклику методів та використанні застарілих функцій. PHPStan відомий своєю точністю і можливістю інтеграції в CI/CD-процеси, що забезпечує автоматичну перевірку кожної зміни у кодовій базі.

Для роботи з JavaScript і TypeScript ефективним рішенням є ESLint. Цей інструмент забезпечує перевірку коду на дотримання стандартів програмування, а також виявлення поширених помилок, які можуть стати джерелом вразливостей. Завдяки широкій підтримці плагінів ESLint адаптується до специфічних потреб проектів і дозволяє налаштовувати правила відповідно до вимог команди розробників.

Ще одним цікавим інструментом є Checkmarx, який підтримує широкий

спектр мов програмування і використовується для комплексного статичного аналізу. Checkmarx забезпечує глибоку перевірку коду на наявність критичних вразливостей, таких як SQL-ін’єкції, XSS або витоки конфіденційної інформації. Цей інструмент часто застосовується у великих компаніях, де вимоги до безпеки є особливо високими.

Попри численні переваги, статичні аналізатори мають свої обмеження. Вони не враховують динамічний контекст виконання програми, що може призводити до пропуску деяких типів вразливостей. Також значна кількість хибних спрацювань потребує додаткової ручної перевірки результатів.

Однак, завдяки швидкості роботи, можливості інтеграції у процеси розробки та детальним звітам, статичні аналізатори залишаються одним із основних інструментів для забезпечення безпеки програмного забезпечення. Їх застосування дозволяє знизити ризики на ранніх етапах розробки, забезпечуючи більш надійний та безпечний код.

Динамічний аналіз коду є важливим підходом до забезпечення безпеки програмного забезпечення, який дозволяє виявляти вразливості під час виконання програми. На відміну від статичного аналізу, цей метод враховує реальний контекст роботи системи, що забезпечує більш точне розуміння поведінки програми та її взаємодії з іншими компонентами. Динамічний аналіз спрямований на виявлення таких проблем, як неправильна обробка вхідних даних, конфігураційні помилки та потенційні витоки даних.

Одним із найпоширеніших інструментів для динамічного аналізу веб-додатків є OWASP ZAP (Zed Attack Proxy). Цей інструмент дозволяє тестувати веб-додатки на наявність вразливостей, таких як Cross-Site Scripting (XSS), SQL-ін’єкції та інші загрози. ZAP працює за принципом проксі-сервера, перехоплюючи запити між клієнтом і сервером, аналізуючи їх і генеруючи детальні звіти. Завдяки простоті використання та підтримці автоматизації ZAP широко застосовується як у невеликих командах, так і у великих організаціях.

Іншим популярним інструментом є Burp Suite, який надає потужний набір функцій для динамічного аналізу. Burp Suite дозволяє проводити ручне й

автоматизоване тестування веб-додатків, імітуючи атаки зловмисників. Інструмент підтримує виявлення широкого спектру вразливостей, таких як недостатня валідація даних або уразливі API. Його гнучкість та можливість інтеграції з іншими інструментами роблять його одним із лідерів у сфері безпеки веб-додатків.

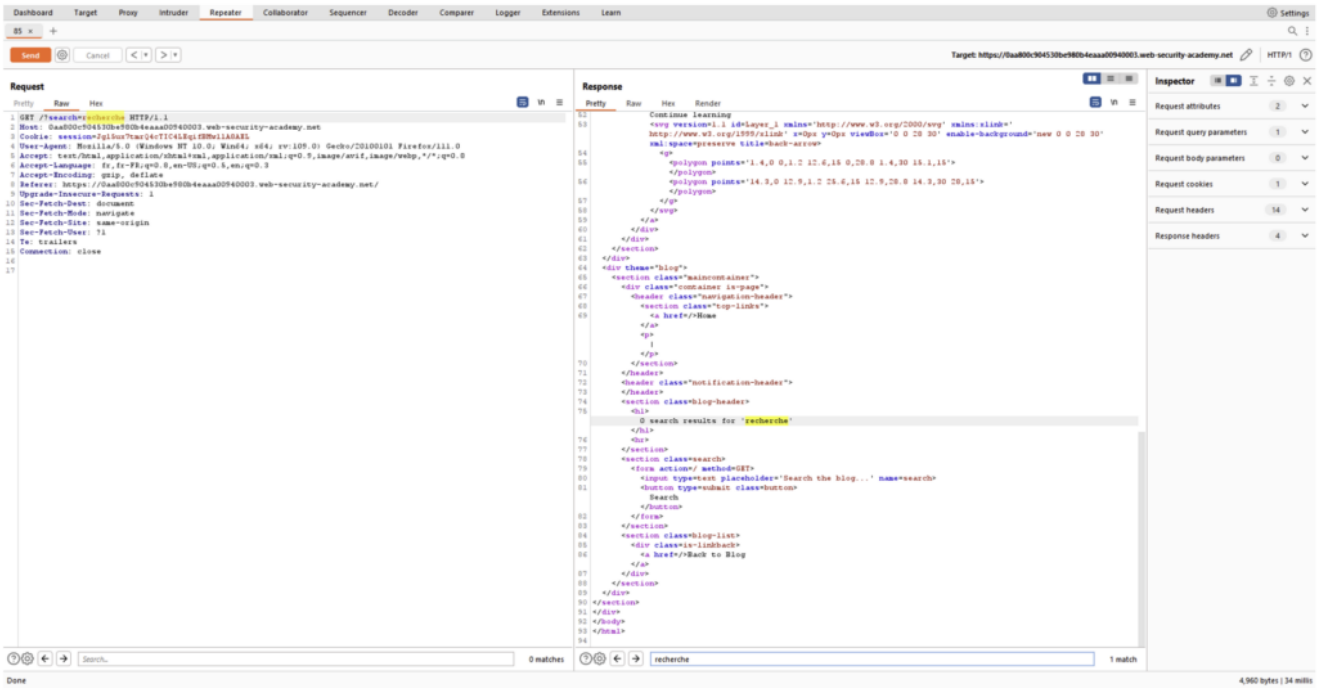


Рисунок 1.2 – Інтерфейс програми Burp Suite

Ще одним потужним рішенням є AppScan, яке орієнтоване на корпоративний сегмент. AppScan підтримує динамічний аналіз веб-додатків і API, забезпечуючи автоматизоване виявлення уразливостей та оцінку ризиків. Інструмент надає глибокі звіти з рекомендаціями щодо усунення знайдених проблем, що робить його корисним для великих організацій із високими вимогами до безпеки.

Попри високу ефективність, динамічний аналіз має свої обмеження. Процес тестування часто є більш ресурсозатратним, оскільки вимагає створення тестового середовища та запуску програми в реальних умовах. Крім того, динамічний аналіз обмежений лише тим, що може бути протестовано в рамках заданих сценаріїв, тому він може пропустити уразливості в

неактивованих модулях або функціях.

Однак, динамічні аналізатори залишаються важливим інструментом у забезпеченні безпеки програмного забезпечення, особливо для систем, які працюють у складних середовищах або взаємодіють із великою кількістю зовнішніх користувачів. Використання таких інструментів у поєднанні зі статичним аналізом дозволяє значно підвищити загальний рівень захисту, забезпечуючи як виявлення відомих проблем, так і адаптацію до нових загроз.

Сучасне програмне забезпечення значною мірою залежить від сторонніх бібліотек і модулів, які використовуються для прискорення розробки та впровадження нових функцій. Проте такі залежності можуть стати джерелом значних ризиків для безпеки, оскільки в них також можуть бути присутні вразливості. Інструменти для перевірки залежностей дозволяють автоматично ідентифікувати проблеми в зовнішніх бібліотеках, оцінювати їхній вплив на безпеку проекту та пропонувати шляхи вирішення.

Одним із найвідоміших інструментів для перевірки залежностей є Snyk, який підтримує широкий спектр мов програмування та середовищ. Snyk аналізує зовнішні бібліотеки, перевіряючи їх на наявність відомих вразливостей у базах даних безпеки. Інструмент не тільки виявляє ризики, а й пропонує автоматичне оновлення до безпечних версій залежностей, що значно спрощує процес управління безпекою в проекта.

Іншим потужним рішенням є Dependency-Check, розроблений OWASP. Цей інструмент виконує аналіз залежностей у проектах, визначаючи відомі вразливості на основі бази даних CVE (Common Vulnerabilities and Exposures). Dependency-Check підтримує інтеграцію з системами CI/CD, що дозволяє забезпечити безперервну перевірку безпеки під час розробки.

Для JavaScript-проектів популярним інструментом є npm audit, який є вбудованим у менеджер пакетів npm. Цей інструмент перевіряє встановлені залежності на наявність вразливостей і надає рекомендації щодо оновлення. Завдяки простоті використання npm audit часто використовується як перша лінія захисту в JavaScript-проектах.

Для Python доступний pip-audit, який працює за аналогічним принципом. Цей інструмент аналізує залежності, встановлені через pip, і перевіряє їх на наявність відомих проблем безпеки. Він використовує бази даних, такі як PyPI, для виявлення потенційних ризиків і пропонує оновлення до безпечних версій.

Попри значні переваги, інструменти для перевірки залежностей мають певні обмеження. Вони орієнтовані на виявлення вже відомих вразливостей, що означає, що нові або невідомі загрози можуть залишитися поза увагою. Крім того, для великих проектів із великою кількістю залежностей може знадобитися значний час на перевірку, особливо якщо потрібно оновити багато компонентів.

Незважаючи на це, інструменти для перевірки залежностей залишаються важливим елементом забезпечення безпеки програмного забезпечення. Їх використання дозволяє не лише знизити ризики, пов’язані з використанням небезпечних бібліотек, а й підвищити загальний рівень довіри до програмного продукту, особливо в умовах постійного зростання складності сучасних систем.

Універсальні інструменти для аналізу коду вирізняються своєю багатофункціональністю та здатністю поєднувати кілька методів перевірки, включаючи статичний і динамічний аналіз, а також перевірку залежностей. Вони призначені для комплексного забезпечення якості та безпеки програмного забезпечення, що робить їх популярними у великих проекта і середовищах із різними вимогами [11,12].

Одним із найвідоміших універсальних рішень є SonarQube, який підтримує аналіз десятків мов програмування та надає широкий спектр функцій для забезпечення якості коду. Інструмент аналізує код на предмет вразливостей, порушення стандартів програмування та технічного боргу. Крім цього, SonarQube пропонує інтеграцію з CI/CD-процесами, що дозволяє автоматизувати перевірки безпеки на кожному етапі розробки. Його модульна структура дозволяє додавати плагіни, що розширюють функціонал під конкретні потреби.

Ще одним прикладом універсального інструменту є Veracode, який забезпечує як статичний, так і динамічний аналіз коду. Veracode дозволяє

проводити комплексну оцінку безпеки, виявляючи вразливості на різних рівнях: від вихідного коду до виконуваного середовища. Інструмент також забезпечує перевірку залежностей і надає детальні звіти з рекомендаціями щодо усунення проблем.

Для організацій, що працюють з веб-додатками, часто використовується Acunetix. Цей інструмент об'єднує функції динамічного аналізу веб-додатків і перевірки залежностей. Acunetix може виявляти уразливості, пов'язані з конфігураціями серверів, небезпечними веб-інтерфейсами або слабкими механізмами автентифікації. Завдяки підтримці автоматизації Acunetix ідеально підходить для тестування великих веб-систем.

Універсальні інструменти часто поєднують переваги вузькоспеціалізованих рішень, забезпечуючи при цьому зручність використання та централізоване управління. Наприклад, вони можуть автоматично генерувати інтегровані звіти, які охоплюють різні аспекти аналізу, включаючи безпеку, якість та відповідність стандартам. Це значно полегшує роботу розробників і дозволяє швидше реагувати на виявлені проблеми [13,14].

Попри свої переваги, універсальні інструменти мають і певні недоліки. Їхнє впровадження може бути дорогим і потребувати складного налаштування. Крім того, через свою багатофункціональність такі інструменти іноді поступаються спеціалізованим рішенням у точності аналізу в певних сферах.

Проте універсальні інструменти залишаються одним із найефективніших способів забезпечення якості та безпеки програмного забезпечення, особливо в корпоративних середовищах. Їх використання дозволяє мінімізувати ризики, пов'язані з вразливостями, і створює базу для систематичного підвищення якості коду. Завдяки цьому вони є ключовими компонентами сучасних стратегій забезпечення безпеки в розробці програмного забезпечення.

Сучасні інструменти для аналізу коду відзначаються різноманіттям підходів і функціональних можливостей, які дозволяють вирішувати широкий спектр завдань у сфері забезпечення інформаційної безпеки. Однак, кожен тип інструментів має свої переваги, недоліки та специфіку використання, що

впливає на вибір найкращого рішення для конкретного проекту.

Статичні аналізатори, такі як SpotBugs, PHPStan або ESLint, чудово підходять для виявлення вразливостей у вихідному коді ще на етапі розробки. Їхня головна перевага — швидкість перевірки та здатність інтегруватися в CI/CD процеси, що дозволяє автоматизувати аналіз із кожною зміною у репозиторії. Однак статичний аналіз обмежується роботою з кодом без врахування динамічного контексту виконання, що іноді призводить до пропуску складних вразливостей.

Динамічні аналізатори, такі як OWASP ZAP, Burp Suite або AppScan, дозволяють досліджувати поведінку програми під час виконання. Вони враховують взаємодію компонентів, що робить їх ефективними для виявлення проблем, пов’язаних із конфігурацією, некоректною обробкою даних або недостатньою валідацією запитів. Проте динамічний аналіз потребує більше часу і ресурсів, а також створення відповідного тестового середовища.

Інструменти для перевірки залежностей, такі як Snyk або Dependency-Check, спеціалізуються на пошуку вразливостей у зовнішніх бібліотеках. Їхнє використання є критично важливим у сучасних проектах, які значною мірою залежать від сторонніх компонентів. Головним обмеженням цих інструментів є орієнтація на відомі проблеми, що залишає ризик для появи нових або невідомих загроз.

Універсальні рішення, такі як SonarQube або Veracode, пропонують багатофункціональний підхід, що об'єднує переваги статичного, динамічного аналізу та перевірки залежностей. Вони забезпечують комплексний контроль якості та безпеки програмного забезпечення, однак їхнє впровадження може бути складним і затратним.

Порівнюючи ці типи інструментів, можна зробити висновок, що кожен із них має свою сферу ефективності. Для великих проектів доцільним є комбінування кількох інструментів, що дозволяє охопити всі аспекти безпеки. Наприклад, статичний аналізатор може забезпечувати базову перевірку, тоді як динамічний інструмент виявляє проблеми, які проявляються лише під час

виконання програми, а перевірка залежностей гарантує безпеку зовнішніх бібліотек. Такий підхід дозволяє максимально знизити ризики та підвищити загальний рівень безпеки програмного забезпечення.

Огляд існуючих інструментів для аналізу коду демонструє, що сучасна практика забезпечення безпеки програмного забезпечення має широкий спектр методів і підходів. Статичні аналізатори, такі як SpotBugs, PHPStan і ESLint, забезпечують ефективне виявлення типових вразливостей на ранніх етапах розробки. Інструменти для перевірки залежностей, включаючи Snyk та Dependency-Check, допомагають ідентифікувати ризики, пов’язані з використанням зовнішніх бібліотек. Універсальні рішення, такі як SonarQube і Veracode, забезпечують комплексний підхід, об'єднуючи кілька методів перевірки в єдиній системі.

Кожна категорія інструментів має свої сильні сторони та обмеження. Статичний аналіз забезпечує швидкість і точність у межах вихідного коду, але не враховує динамічний контекст. Динамічний аналіз охоплює реальні умови роботи програми, проте потребує більше часу та ресурсів.

Поєднання цих інструментів дозволяє створити комплексний підхід до забезпечення безпеки, що є актуальним для сучасних вимог до програмного забезпечення. Запропоновані методи та інструменти можуть бути ефективно інтегровані у процеси розробки, знижуючи ризики та підвищуючи якість продукту. Це підкреслює важливість і необхідність створення інтегрованих рішень, які поєднують переваги різних підходів у єдиній системі.

1.3 Автоматизація безпекового аналізу в рамках CI/CD

У сучасному світі розробки програмного забезпечення процеси безперервної інтеграції та доставки (CI/CD) стали невід'ємною частиною ефективного та якісного створення продуктів. Вони дозволяють командам автоматизувати збірку, тестування та розгортання додатків, забезпечуючи

швидке впровадження змін і оновлень. Однак, зростаючий темп розробки підвищує ризики появи вразливостей у коді, що може призвести до серйозних проблем безпеки.

CI/CD — це підхід до розробки програмного забезпечення, який поєднує безперервну інтеграцію (Continuous Integration) та безперервне постачання/розгортання (Continuous Delivery/Deployment). Безперервна інтеграція передбачає часте злиття змін коду в головну гілку, де кожен коміт автоматично збирається та проходить через серію тестів для виявлення можливих помилок. Це забезпечує швидке виявлення та виправлення проблем, підтримуючи високу якість коду.

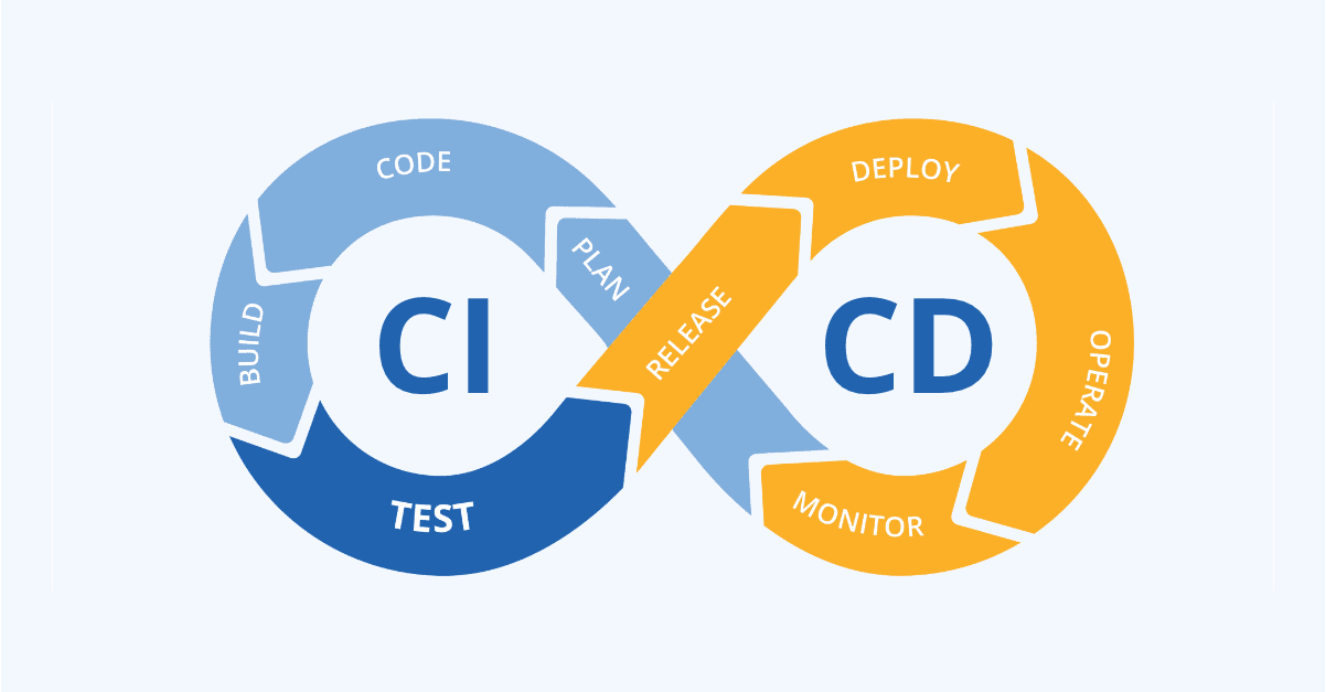


Рисунок 1.3 – Зображення схеми CI/CD

У рамках CI/CD процесів є декілька ключових етапів, на яких можуть бути інтегровані інструменти статичного та динамічного аналізу коду. Перш за все, на етапі безперервної інтеграції, статичні аналізатори коду використовуються для автоматичного сканування нових змін на предмет потенційних вразливостей, помилок або невідповідностей стандартам кодування. Це дозволяє розробникам отримувати миттєвий зворотний зв'язок та

виправляти проблеми ще до того, як код буде об'єднано з основною базою.

Далі, на етапі безперервного постачання, після успішного проходження статичного аналізу та інших автоматичних тестів, додаток може бути розгорнутий у тестовому або staging-середовищі. Тут вступають у дію інструменти динамічного аналізу, які тестують працюючий додаток на вразливості під час виконання. Це допомагає виявити проблеми, які не можуть бути знайдені статичним аналізом, такі як помилки конфігурації, проблеми з продуктивністю або безпекові вразливості, що проявляються лише під час реальної роботи додатка.

Імплементація цих інструментів у CI/CD конвеєр зазвичай здійснюється шляхом додавання відповідних стадій у конфігурацію конвеєра. Статичні аналізатори інтегруються на рівні системи контролю версій та автоматичних збірок, запускаючись при кожному коміті або pull request. Динамічні інструменти інтегруються на етапі розгортання, де вони автоматично запускають сценарії тестування після успішного деплою додатка в тестове середовище.

Для мінімізації цих ризиків важливо інтегрувати безпекові перевірки безпосередньо в CI/CD процеси. Статичні інструменти аналізу коду (SAST) стають ключовим елементом цього підходу, оскільки вони дозволяють виявляти потенційні вразливості та помилки на ранніх етапах розробки. Використання SAST в CI/CD забезпечує автоматичне сканування коду при кожному коміті або перед злиттям змін, що сприяє підтримці високого рівня безпеки без уповільнення розробки.

Імплементація статичних інструментів аналізу в CI/CD конвеєри зазвичай здійснюється шляхом інтеграції відповідних плагінів або скриптів у систему контролю версій, таку як Git, та сервери збірки, наприклад, Jenkins або GitLab CI. Це дозволяє автоматизувати запуск аналізу коду під час кожного коміту або пул-запиту, забезпечуючи регулярний та послідовний процес перевірки якості коду. Автоматизація цього процесу включає налаштування параметрів аналізу, визначення критеріїв успішності та інтеграцію результатів аналізу у зручному

для розробників форматі, наприклад, у вигляді детальних звітів або інтеграції з інструментами моніторингу та візуалізації даних. Виявлені вразливості можуть бути негайно передані відповідальним членам команди через системи відстеження помилок, такі як Jira або Trello, або через засоби комунікації, такі як Slack або електронна пошта, що забезпечує оперативне реагування на виявлені проблеми та їхнє швидке усунення. Таким чином, інтеграція статичних аналізаторів у CI/CD конвеєри сприяє підвищенню загальної безпеки та якості програмного забезпечення, знижуючи ризики виникнення критичних помилок у майбутньому та забезпечуючи більш стабільний, надійний та ефективний процес розробки програмних продуктів. Це також сприяє покращенню співпраці між членами команди та оптимізації робочих процесів, що в кінцевому підсумку веде до створення більш безпечного та якісного програмного забезпечення.

1.4 Постановка задачі

Розвиток сучасних технологій програмного забезпечення супроводжується постійним зростанням кількості вразливостей, які можуть бути використані зловмисниками для компрометації систем. Це створює значні ризики для безпеки даних, особливо в умовах широкого використання сторонніх бібліотек, складних архітектур та різних мов програмування. З урахуванням цього, актуальним завданням стає створення інструментів, які забезпечують автоматизацію аналізу безпеки програмного забезпечення та спрощують інтеграцію таких перевірок у процеси розробки.

Основною метою цієї роботи є розробка системи, яка дозволяє автоматизувати процес оцінки інформаційної безпеки програмного забезпечення шляхом інтеграції існуючих інструментів статичного та динамічного аналізу в єдину модульну платформу. Система повинна забезпечувати гнучкість для підтримки різних мов програмування, можливість

аналізу залежностей та інтеграцію з сучасними процесами CI/CD.

Для досягнення цієї мети необхідно вирішити такі завдання:

– дослідити сучасні підходи та інструменти для аналізу безпеки програмного забезпечення;

– розробити архітектуру системи, що підтримує модульність та інтеграцію кількох інструментів;

– забезпечити автоматизацію аналізу із можливістю генерації єдиних звітів про виявлені уразливості;

– провести тестування системи на прикладах програмного забезпечення, що містить як реальні, так і навмисно створені вразливості;

– оцінити ефективність запропонованого рішення та визначити його перспективи для подальшого розвитку.

Запропонована система покликана не лише зменшити трудомісткість аналізу безпеки, а й зробити процеси виявлення та усунення вразливостей доступними для широкого кола розробників, незалежно від їхнього рівня експертизи в сфері інформаційної безпеки. Це сприятиме підвищенню загального рівня захищеності програмного забезпечення та створенню більш надійних цифрових рішень.

2 МОДЕЛЬ ПРОЦЕСУ ОЦІНЮВАННЯ ІНФОРМАЦІЙНОЇ БЕЗПЕКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Моделювання процесу функціонування програмного забезпечення

Постійний розвиток сучасних технологій у сфері програмного забезпечення супроводжується безперервним збільшенням кількості вразливостей, які можуть бути використані зловмисниками для компрометації інформаційних систем. Цей тренд пов'язаний із зростанням складності програмних продуктів, їхньою інтеграцією у різноманітні сфери діяльності та розширенням функціональних можливостей, що, водночас, створює нові точки потенційних атак. Зловмисники постійно вдосконалюють свої методи та техніки, що дозволяє їм ефективніше виявляти та використовувати існуючі слабкі місця в програмному забезпеченні. Це породжує суттєві ризики для безпеки даних, особливо в умовах широкомасштабного застосування програмних продуктів у різних галузях, таких як фінанси, охорона здоров'я, державне управління, енергетика та інші критично важливі сектори.

Забезпечення безпечного функціонування програмного забезпечення включає в себе комплекс заходів, спрямованих на гарантування інформаційної безпеки в інформаційному середовищі під час експлуатації ПЗ. Ці заходи охоплюють різні аспекти, починаючи від розробки програмного забезпечення з урахуванням принципів безпеки (безпека за замовчуванням) і закінчуючи впровадженням ефективних механізмів моніторингу та реагування на інциденти. Крім того, важливо забезпечити якісну та надійну реалізацію програмним забезпеченням своїх функціональних можливостей, що включає ретельне тестування, аудит коду, регулярні оновлення та патчі для виправлення виявлених вразливостей. Комплексний підхід до безпеки програмного забезпечення також передбачає навчання та підвищення обізнаності розробників, користувачів та адміністративного персоналу щодо потенційних загроз та методів їх уникнення.

За своїм функціональним призначенням програмне забезпечення можна

класифікувати на декілька основних категорій: системне, прикладне та сервісне. Системне програмне забезпечення включає операційні системи, драйвери пристроїв, утиліти та інші компоненти, які забезпечують базову функціональність комп'ютера та управління його ресурсами. Прикладне програмне забезпечення охоплює програми, призначені для виконання конкретних завдань користувачів, таких як текстові редактори, електронні таблиці, графічні редактори, браузери та інші офісні або розважальні додатки. Сервісне програмне забезпечення включає програми та сервіси, які надають додаткові функції, такі як бази даних, сервери електронної пошти, веб-сервери та інші мережеві сервіси, що підтримують роботу інформаційної системи.

Програмне забезпечення є невід'ємним компонентом інформаційної системи, зокрема операційної системи, яка, у свою чергу, може бути частиною більшого програмно-апаратного комплексу або розподіленої обчислювальної мережі. Інформаційна система складається з різних взаємодіючих компонентів, включаючи апаратне забезпечення, мережеві ресурси, програмне забезпечення та дані, які разом забезпечують виконання певних функцій та завдань. Взаємозв'язок між цими компонентами визначає ефективність, надійність та безпеку всієї системи. Таким чином, якісне програмне забезпечення не лише виконує свої первинні функції, але й забезпечує стабільну та безпечну роботу всієї інформаційної системи, підтримуючи її інтегрованість та здатність адаптуватися до змінних умов експлуатації.

Оцінка надійності програмного забезпечення тісно пов'язана з можливістю впровадження в його код програмних закладок на етапах розробки та/або модифікації, за допомогою яких зловмисник може реалізувати створені вразливості. У зв'язку з цим, необхідність сертифікації та перевірки програмного забезпечення на відповідність вимогам чинних нормативних документів протягом усього його життєвого циклу є беззаперечною [4,5,10].

Для того щоб оцінити надійність програмних продуктів, які є невід'ємною складовою інформаційних систем, особливий інтерес представляють рандомізовані моделі та методи, що застосовуються в теорії надійності.

Розгляньмо основні типи моделей оцінки надійності програмних продуктів, детально зазначивши їхні переваги та недоліки.

Модель Джелінскі – Моранди ґрунтується на припущенні, що час до настання наступної відмови програмного забезпечення (наприклад, у випадку збою системи або кібератаки) розподілений експоненційно, а інтенсивність відмов ПЗ прямо пропорційна кількості невиправлених помилок, що залишаються в програмному коді.

Однією з ключових переваг цієї моделі є її висока простота та зручність, що робить її легкою у використанні навіть для користувачів з обмеженими знаннями в області моделювання та аналізу програмного забезпечення. Завдяки цій простоті, модель дозволяє швидко та ефективно виконувати необхідні розрахунки, що особливо важливо в умовах обмежених ресурсів часу та обчислювальних потужностей. Крім того, зручність моделі сприяє її широкому застосуванню у різних контекстах, від академічних досліджень до практичних завдань у сфері розробки програмного забезпечення [16].

Проте, незважаючи на зазначені переваги, ця модель має суттєвий недолік, який може вплинути на точність та надійність її результатів. Основним негативним аспектом є те, що при неточному або некоректному визначенні початкової кількості помилок, яка закладається в основу моделювання, інтенсивність відмов програмного забезпечення може набувати негативних значень. Така ситуація є абсолютно нелогічною та неприпустимою, оскільки інтенсивність відмов повинна бути невід'ємною величиною, що відображає реальні показники надійності програми. Негативні значення інтенсивності відмов свідчать про помилки в розрахунках або невірні припущення, які використовуються у моделі, що може привести до хибних висновків щодо рівня надійності програмного забезпечення. Це, в свою чергу, може мати серйозні наслідки для подальшого розвитку та підтримки програмних продуктів, оскільки неправильна оцінка надійності може призвести до недооцінки потенційних ризиків та необхідних заходів щодо їх усунення.

У контексті моделі Шумана, процес дослідження безпеки та надійності

програмного забезпечення організовано у кілька послідовних та ретельно продуманих етапів. Кожен етап передбачає виконання програмного забезпечення на певному наборі тестових даних, що дозволяє систематично виявляти потенційні помилки та вразливості. Під час проведення кожного етапу тестування всі виявлені помилки детально реєструються, однак їх не виправляють негайно. Такий підхід дозволяє здійснювати комплексний аналіз знайдених проблем без ризику їх непередбачуваного впливу на подальше тестування.

Після завершення кожного етапу тестування, усі виявлені помилки виправляються ретельно та систематично. Це включає внесення відповідних змін до коду програмного забезпечення, щоб усунути виявлені дефекти. Крім того, тестові набори коригуються з урахуванням виправлених помилок, що дозволяє забезпечити більш точне та ефективне тестування на наступних етапах. Такий підхід забезпечує поступове покращення якості програмного забезпечення, зменшуючи ймовірність виникнення нових помилок та підвищуючи загальну надійність системи [17].

Однією з головних переваг цієї моделі є можливість визначення всіх невідомих параметрів, необхідних для проведення точних та достовірних розрахунків, без потреби звертатися до додаткових або альтернативних моделей. Це значно спрощує процес моделювання та підвищує його ефективність. Проте недоліком є те, що модель передбачає: при коригуванні програмного забезпечення не вносяться нові помилки. У реальних умовах розробки та підтримки ПЗ це припущення часто не відповідає дійсності, оскільки внесення змін у код може супроводжуватися появою нових дефектів. Крім того, для виконання розрахунків за цією моделлю потрібна значна кількість зареєстрованих даних, що може бути складно забезпечити, особливо на початкових етапах розробки.

В основі моделі Шика – Волвертона лежить припущення, що частота виникнення помилок пропорційна не лише поточній кількості помилок у програмному забезпеченні, але й часу, витраченому на тестування. Іншими

словами, ймовірність виявлення помилок збільшується з плином часу, оскільки тестувальники отримують більше можливостей для виявлення існуючих дефектів. Такий підхід враховує динаміку процесу тестування та дозволяє більш точно моделювати процес виявлення помилок.

Ця модель має перевагу в тому, що вона дозволяє з високою точністю та надійністю розраховувати показники надійності програмного забезпечення, будучи при цьому простою та зручною у застосуванні. Вона не вимагає складних обчислень або спеціалізованих знань, що робить її доступною для широкого кола фахівців у сфері розробки та тестування ПЗ.

Однак недоліком моделі є те, що для розподілів помилок, відмінних від експоненціального, коли інтенсивність виникнення помилок змінюється з часом нерівномірно, необхідно використовувати умовні розподіли. У таких випадках умовна ймовірність для конкретного інтервалу тестування повинна відповідати проміжку часу від початку тестування до початку розглядуваного інтервалу. Якщо цього не дотримуватися, процес відновлення системи після усунення виявлених помилок може призвести до завищення ймовірностей безпомилкового функціонування на всіх етапах тестування. Це може спотворити результати оцінки надійності та призвести до неправильних висновків.

Модель Коркорена базується на припущенні про наявність у програмному забезпеченні множинних джерел потенційних програмних відмов, пов'язаних з різними типами помилок та вразливостей, а також враховує різну ймовірність їх виникнення. Такий підхід дозволяє більш детально та точно моделювати поведінку ПЗ в умовах реальної експлуатації, враховуючи різноманітність можливих проблем та їх внесок у загальний рівень надійності.

Перевагою цієї моделі є те, що вона використовує змінні ймовірності відмов для різних типів помилок, що дозволяє оцінити ймовірність безвідмовного виконання програмного забезпечення в конкретний момент часу. Однак недоліком є те, що для проведення оцінки необхідно враховувати апріорну інформацію або дані з попередніх періодів функціонування

аналогічних зразків ПЗ.

Модель IBM передбачає, що під час експлуатації користувачем поточної версії програмного забезпечення розробники займаються його активним супроводом — вносять покращення або виправлення без очікування запитів від користувача. Цей супровід може включати додавання нових функціональних можливостей. З певного моменту, коли розробник вважає свою задачу завершеною, починається пасивний супровід, при якому виправлення вносяться лише за запитом користувача. Під час супроводу в кожну нову версію ПЗ вносяться значні обсяги нових помилок разом із доопрацюваннями, змінами та виправленнями, що потребує коригувань і в наступних версіях.

Зокрема, у корпорації IBM доклали значних зусиль для прогнозування кількості необхідних виправлень програмного забезпечення з однієї версії на іншу. Для цього використовували великий обсяг експериментальних даних, що був зібраний упродовж багаторічного супроводу та підтримки операційних систем. Ці дані включали різноманітні метрики, такі як кількість виявлених помилок, час, витрачений на їх виправлення, а також характеристики кодової бази, що дозволило створити надійні математичні моделі для оцінки стабільності та якості програмного забезпечення.

Однією з таких моделей є модель Шнайдера, яка визначає взаємозв'язок між кількістю помилок у програмі, витратами на їх виправлення та ключовими характеристиками програмного коду. Зокрема, ця модель враховує витрати, виміряні у "людино-місяцях", кількість підпрограм у програмному продукті та загальну кількість тисяч операторів у коді. Такий підхід дає змогу оцінити масштабність проекту, а також спрогнозувати обсяги роботи, необхідні для усунення дефектів у майбутніх версіях.

Серед основних переваг моделі Шнайдера можна виділити її низьку обчислювальну складність, що робить її легкою для реалізації та використання навіть у проектах із обмеженими ресурсами. Крім того, її простота дозволяє швидко отримати результати без необхідності складних математичних обчислень або залучення вузькоспеціалізованих експертів.

Проте модель має і свої обмеження, які слід враховувати. Зокрема, одним із недоліків є те, що емпірично обране число виправлень, на якому базується модель, може бути необґрунтованим або недостатньо точним для конкретних випадків. Також модель не враховує вразливості нульового дня, які є критично важливими для сучасного програмного забезпечення, особливо в умовах зростаючої кількості кіберзагроз. Окрім цього, функціональні можливості програмного забезпечення не досліджуються детально, що може призводити до упущення важливих аспектів, пов'язаних із продуктивністю або користувацьким досвідом.

2.2 Модель безпечного функціонування програмного забезпечення

Запропонована модель безпечного функціонування програмного забезпечення спрямована на усунення недоліків, властивих раніше розглянутим моделям. На підставі результатів проведених досліджень цих моделей можна зробити наступні висновки щодо їхніх слабких сторін:

По-перше, деякі моделі при їх реалізації вимагають значного обсягу обчислювальних ресурсів. Це зумовлено як необхідністю накопичення та обробки великої кількості даних, так і високими вимогами до обчислювальних потужностей компонентів системи, які здійснюють аналіз безпеки.

По-друге, ймовірнісні моделі зазвичай ґрунтуються на припущеннях про те, що інтенсивність відмов або атак, а також кількість помилок у програмному забезпеченні заздалегідь відомі, і що відомий розподіл цих параметрів (наприклад, нормальний, пуассонівський або біноміальний розподіл). Однак такі припущення не завжди відповідають реальним процесам і системам, що може призводити до зниження точності моделювання [18, 19].

По-третє, відмови програмного забезпечення та кібератаки, викликані експлуатацією вразливостей у ПЗ, часто розглядаються як єдина сукупність. Внаслідок цього моделі не враховують важливої особливості роботи

шкідливого програмного забезпечення—а саме характеру його звернень до оперативної пам'яті.

По-четверте, жодна з розглянутих моделей не забезпечує комплексного уявлення про процес функціонування програмного забезпечення. Зокрема, відсутній погляд з боку інформаційної безпеки, що є критично важливим для повного розуміння потенційних загроз.

Для усунення вищезазначених недоліків пропонується покласти в основу моделі детальний опис взаємодії всіх сутностей, залучених у процесі функціонування програмного забезпечення. При цьому необхідно враховувати специфічні особливості поведінки шкідливого програмного забезпечення на комп'ютері або іншому обчислювальному пристрої, що дозволить більш точно моделювати потенційні загрози та розробляти ефективні стратегії їх нейтралізації [20].

Дослідження продемонстрували, що основними ознаками зараження комп'ютерних систем різними типами шкідливого програмного забезпечення є наступні фактори:

– суттєве уповільнення роботи комп'ютера або мережевого вузла. Зазвичай це обумовлено впровадженням шкідливого програмного забезпечення, яке під час виконання своїх шкідливих функцій споживає значні обчислювальні ресурси системи. У результаті цього на виконання легітимних обчислювальних завдань потрібно значно більше часу, що негативно впливає на загальну продуктивність системи;

– виконання нових функцій, не характерних для даного комп'ютера. Це може проявлятися у відправленні конфіденційних даних на сторонні веб-ресурси без відома користувача, запуску невідомих процесів, які раніше не були присутні в системі, та інших аномальних діях, що не відповідають нормальному функціонуванню комп'ютера;

– проблеми з відкриттям файлів, які раніше успішно відкривалися, та зміни в розширеннях файлів. Це може включати зникнення розширень файлів або, навпаки, появу подвійних розширень, що може свідчити про шифрування

файлів чи інші шкідливі дії з боку програмного забезпечення;

– раптове завершення роботи додатків або поява небажаних спливаючих вікон під час роботи з програмами. Такі симптоми потенційно сигналізують про наявність шкідливих процесів, які виконуються у фоновому режимі без згоди та відома користувача.

Шкідливе програмне забезпечення характеризується надмірним споживанням обчислювальних ресурсів комп'ютерної системи, що може призводити до зниження загальної продуктивності пристрою. Окрім цього, таке ПЗ виконує операції з оперативною пам'яттю, які суттєво відрізняються від дій, притаманних легітимним програмам. Це може включати нетипові звернення до різних областей пам'яті, які зазвичай не використовуються стандартними додатками, а також спроби переповнення буферів, що може спричинити несподівані помилки або збої в роботі системи [21].

Крім того, шкідливе ПЗ може здійснювати інші дії, спрямовані на отримання несанкціонованого доступу до конфіденційних даних або ресурсів комп’ютера. Це може включати використання різноманітних технік обходу безпекових механізмів, інжекцію коду в процеси легітимних програм, а також маскування своєї діяльності під інші процеси для уникнення виявлення антивірусними засобами.

Наприклад, для шкідливого програмного забезпечення, яке експлуатує обчислювальні потужності комп'ютера для майнінгу криптовалют, характерними є наступні прояви в системі:

– значне та помітне уповільнення роботи обчислювальних компонентів.

– поява процесів, що споживають надмірну та невиправдано велику кількість ресурсів центрального процесора та графічного адаптера.

– несанкціоноване відправлення великого обсягу мережевого трафіку на сторонні веб-ресурси без відома і згоди користувача.

Таким чином, можна дійти висновку, що ключову роль у детальному та всебічному описі роботи програмного забезпечення відіграють кілька фундаментальних елементів. Перш за все, це функціональні можливості та

завдання, які виконує програмне забезпечення. До них належить чіткий опис операцій, які забезпечує програма, включаючи її основну функціональність, допоміжні процеси та будь-які додаткові можливості, що впливають на ефективність і зручність використання.

Не менш важливим є аспект змін і динаміки у використанні обчислювальних ресурсів системи. Це включає як процесорні ресурси, так і ресурси пам’яті. У межах цього елементу варто враховувати не лише поточне навантаження, але й те, як змінюється його рівень у різних режимах роботи програмного забезпечення — під час запуску, виконання інтенсивних обчислювальних завдань, обробки великих обсягів даних чи перебування у стані очікування [22].

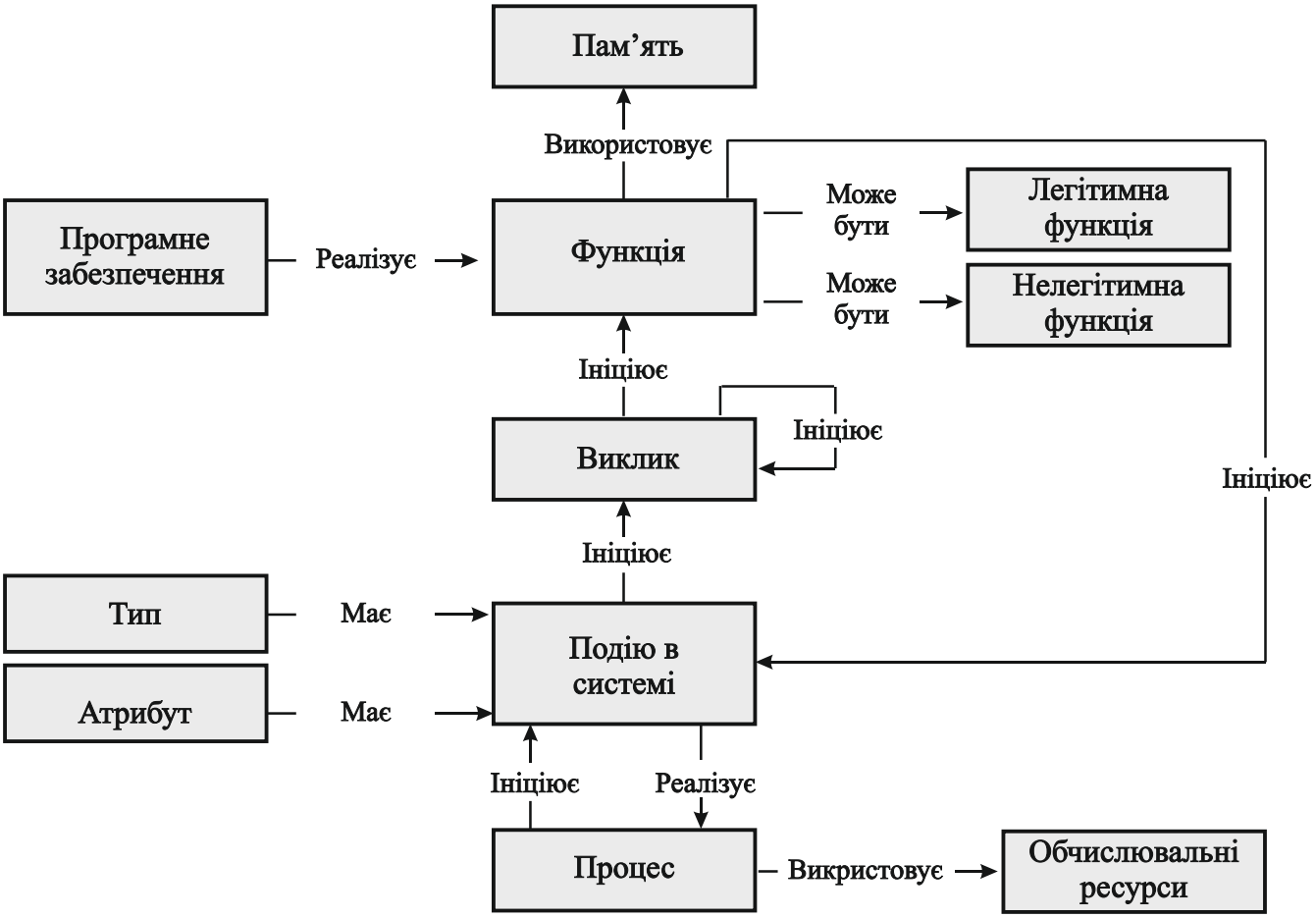


Рисунок 2.1 – Графічне представлення моделі безпечного функціонування програмного забезпечення

Окрему увагу слід приділити змінам, пов'язаним із роботою програмного забезпечення з оперативною пам’яттю. Це стосується не лише обсягів пам’яті, які використовуються програмою, але й специфіки управління цією пам’яттю: як ефективно програма виділяє та звільняє ресурси, наскільки добре реалізовані механізми уникнення витоків пам’яті, та чи забезпечується дотримання принципів безпеки під час роботи з оперативною пам’яттю.

Програмне забезпечення вважається безпечним у тому випадку, якщо його встановлення та подальше використання не призводять до порушення легітимних відносин і взаємодій у системі. Важливо також, щоб воно не мало потенційної можливості створювати такі порушення через наявність у своєму складі певних шкідливих функціональних можливостей або прихованих вразливостей. Для формалізації цього опису скористаємося математичним апаратом теорії множин, що дозволить чітко та строго представити всі необхідні елементи та взаємозв'язки. Ми формально опишемо модель безпечного функціонування програмного забезпечення, представивши його у вигляді кортежу: *Software=<F,Sign,R,M,Memory>*

де:

1. $F=\{f_1,f_2,\ldots,f_n\}$ - кінцева множина функцій, які виконує досліджуване програмне забезпечення, являє собою повний набір операцій та дій, що можуть бути виконані цим ПЗ. Функції, що входять до множини *F*, поділяються на деструктивні F_{destr} та легітимні функції F_{leg}: $F = F_{leg} \cup F_{destr}, F_{leg} \cap F_{destr} = \emptyset$. При цьому, в будь якій із підмножин F_{leg} і F_{destr} можуть бути присутні як документовані функції програмного забезпечення F_{doc}, так і не документовані, $F_{undoc}, F_{doc} \cap F_{undoc} = \emptyset$.

2. $Sign$–множина ознак, які характеризують виконання функцій програмного забезпечення, є ключовим елементом у детальному аналізі його роботи та поведінки. Маємо неоднозначне відображення множини функцій F на множину $Sign$: $F \to Sign$, це означає, що кожному елементу множини F може відповідати один або кілька елементів множини $Sign$, тобто одна функція може характеризуватися декількома ознаками, що відображають різні аспекти її

виконання та впливу на систему. Використовуючи відповідні індекси та підмножини, можемо задати такі відображення: $F_{leg} \to Sign_{leg}, F_{destr} \to Sign_{destr}$, $F_{doc} \to Sign_{doc}$, $F_{undoc} \to Sign_{undoc}$.

3. R – обчислювальні ресурси, необхідні програмному забезпеченню для реалізації своїх функцій, є критично важливим аспектом його ефективного функціонування. Маємо відображення $F \to R$, внаслідок якого кожній функції f_i призначаються три числові значення: $\{r_{r_{min}}, r_i, r_{max}\}$. Ці значення відповідають необхідним ресурсам центрального процесора, обсягу оперативної пам'яті та часу виконання відповідно. Очевидно, що оцінка необхідних обчислювальних ресурсів може змінюватися залежно від різних факторів, таких як навантаження на систему або специфіка виконуваних завдань. Проте в інформаційній системі для кожного програмного продукту повинен бути виділений певний та достатній обсяг обчислювальних ресурсів, щоб забезпечити її ефективне та стабільне функціонування. Це дозволить уникнути перевантаження системи та забезпечити оптимальну продуктивність усіх компонентів.

4. M – множина методів дослідження, які застосовуються для виявлення шкідливого програмного забезпечення, охоплює широкий спектр технік та підходів, призначених для ідентифікації, аналізу та нейтралізації потенційно небезпечних програмних компонентів [23,24].

5. $Memory$ – множина операцій над пам'яттю, які виконуються під час активації тієї чи іншої функції з множини F, є важливим компонентом для детального аналізу роботи програмного забезпечення. Таким чином, маємо відображення: $F \to Memory$, при якому кожній функції f_i зіставляється певний набір операцій над пам'яттю: $f_i \to \{memory_{i_1}, memory_{i_2}, ..., memory_{i_k}\}$.

Необхідно підкреслити, що використання методів M тільки в ідеальному випадку дозволить виявити повну та всебічну множину ознак $Sign$, які характерні для даного програмного продукту. У реальних умовах практичної діяльності слід говорити про визначення за допомогою методів M лише

підмножини ознак $Sign'$, $Sign' \in Sign$. Таким чином, на практиці вирішується складне завдання оцінки умовної ймовірності $P(A|B)$ настання події $A = \{Sign_{destr} = \varnothing \, and \, Sign_{undoc} = \varnothing\}$ за умови виконання події, тобто реалізації певного набору методів M', $M' \in M$, який дозволяє виявити достатню кількість ознак $Sign'_{destr}$ та $Sign'_{undoc}$, $B = \{M' \to \{ Sign'_{destr}, Sign'_{undoc}\} \neq \emptyset\}$.

Досягнення максимальної ефективності застосовуваних методів M є малоймовірним у реальних умовах, оскільки завжди існують різноманітні обмеження та фактори, що впливають на їх результативність. Проте, у такому ідеальному випадку, коли ефективність методів досягає свого максимуму, буде досягнуто рівність: $Sign_{destr} = Sign'_{destr}$, $Sign = Sign'_{undoc}$.

Сформулюємо критерії безпечного функціонування програмного продукту в термінах запропонованої моделі:

– програмний продукт не реалізує деструктивних та шкідливих функцій: $F_{destr} = \varnothing$ та $F_{undoc} = \varnothing$. Це означає, що в програмному забезпеченні відсутні як деструктивні функції, так і недокументовані деструктивні функції;

– обсяг обчислювальних ресурсів, задіяних для аналізу безпеки програмного продукту (за допомогою множини методів M), мінімізований: $M : R \to R_{\min}, R_{\min} = \{r1_{\min}, r2_{\min}, ..., r_{n_{\min}}\}$. Таким чином, кожна функція програмного продукту в ідеалі повинна використовувати мінімально допустимий обсяг обчислювальних ресурсів для своєї реалізації;

– не повинно бути обчислювальних ресурсів, що витрачаються на будь-яку множину функцій F', яка не перетинається з множиною $F : \exists F' \to R$, оскільки присутність шкідливого програмного забезпечення в системі часто призводить до активації нових, нетипових для роботи системи функцій;

– множина операцій над пам'яттю *Memory*, які виконуються програмним забезпеченням, повинна мати відповідне відображення на множину ознак $Sign$. Це зумовлено тим, що однією з ключових та важливих

ознак присутності шкідливого програмного забезпечення в системі є виконання нетипових, аномальних операцій з оперативною пам'яттю: $Memory \to Sign \neq \varnothing$;

– множина операцій з пам'яттю повинна мати відповідне відображення на множину ознак, оскільки важливою характеристикою наявності шкідливого програмного забезпечення в системі є виконання нетипових операцій з пам'яттю. Множина методів дослідження, що застосовуються для виявлення шкідливого програмного забезпечення, повинна використовувати результати цього відображення: $Memory \to Sign$. Це означає, що для ефективного вирішення поставленого завдання доцільно застосовувати саме ті методи, які враховують операції програмного забезпечення з пам'яттю;

– множина методів дослідження, які використовуються, повинна бути сформована таким чином, щоб умовна ймовірність $P(A|B)$ була максимізована до свого найвищого можливого значення. Іншими словами, для кожного елемента із множини має виконуватися умова: $M : P(A|B) \to 1 : (|) \to 1.$

Розроблена модель суттєво відрізняється від попередньо розглянутих моделей, призначених для оцінки безпеки, надійності та якості функціонування програмного продукту. Ця нова модель вводить інноваційні підходи та методології, які дозволяють більш глибоко аналізувати та оцінювати різні аспекти програмного забезпечення. Вона враховує не лише специфічні особливості роботи шкідливого програмного забезпечення на різних компонентах системи, таких як операційна система, мережеві сервіси, додатки користувача та інші критичні елементи, але й необхідність мінімізації використання обчислювальних ресурсів під час аналізу. Це означає, що модель розроблена з урахуванням ефективності ресурсів, що дозволяє проводити детальний аналіз без значного навантаження на апаратні засоби, що особливо важливо в умовах обмежених ресурсів або при необхідності одночасного обслуговування великої кількості систем.

Особливу увагу слід приділити питанням практичної реалізації цієї моделі, оскільки вона повинна поєднувати в собі високу ефективність

виявлення шкідливого програмного забезпечення з оптимальним та раціональним використанням обчислювальних ресурсів. Це включає в себе розробку ефективних алгоритмів аналізу, які здатні швидко і точно виявляти загрози, а також інтеграцію цих алгоритмів у існуючі системи без значної потреби в додаткових ресурсах. Крім того, важливо забезпечити масштабованість моделі, щоб вона могла адаптуватися до різних умов експлуатації та розширюватися відповідно до зростаючих вимог безпеки.

Програмне забезпечення, яке містить потенційно небезпечні можливості, може реалізовувати одну або декілька функцій, наслідки виконання яких здатні безпосередньо загрожувати порушенню безпеки даних або призводити до виконання шкідливого коду, що становить цю загрозу. Наприклад, таке програмне забезпечення може включати функції доступу до конфіденційних даних, можливості зміни або видалення важливої інформації, а також здатність виконувати команди віддаленого керування без відома користувача. Наслідки виконання цих функцій можуть бути різноманітними: від несанкціонованого доступу до особистих або корпоративних даних до повного контролю над системою, що може призвести до серйозних фінансових втрат, порушення бізнес-процесів або навіть загроз національній безпеці [25,27].

Крім того, шкідливе програмне забезпечення може використовувати різні техніки маскування та приховування своєї присутності в системі, що ускладнює його виявлення традиційними методами антивірусного захисту. Це можуть бути такі методи, як інжекція коду в легітимні процеси, використання полімерних технологій для зміни свого коду під час виконання, або ж шифрування своїх компонентів для уникнення аналізу. Відтак, ефективна модель оцінки безпеки повинна враховувати ці складні аспекти та забезпечувати багаторівневий підхід до виявлення і нейтралізації таких загроз.

Важливо також зазначити, що розроблена модель повинна бути гнучкою та адаптивною, здатною до постійного оновлення та вдосконалення у відповідь на нові види загроз та еволюцію шкідливих програм. Це включає в себе регулярне оновлення баз даних загроз, впровадження нових методів аналізу та

машинного навчання для прогнозування та виявлення нових видів шкідливого ПЗ. Такий підхід дозволяє забезпечити постійну актуальність та ефективність моделі у динамічному середовищі кібербезпеки.

Рисунок 2.№ – Програмне забезпечення з потенційно небезпечними можливостями

Розроблена модель є значним кроком уперед у галузі оцінювання безпеки, надійності та якості функціонування програмного забезпечення. Вона враховує унікальні аспекти поведінки шкідливих програм на різних компонентах інформаційних систем, забезпечуючи оптимальне використання обчислювальних ресурсів під час аналізу. Особливістю моделі є її орієнтація на

практичну реалізацію, що поєднує високу ефективність виявлення потенційних загроз із мінімальними витратами ресурсів.

Такий інтегрований підхід дозволяє не тільки значно посилити захист інформаційних систем, але й забезпечити їх стабільну, безперебійну роботу в умовах сучасних викликів кіберзагроз. Завдяки цьому, модель стає не лише інструментом для виявлення ризиків, але й надійною основою для побудови систем безпеки нового покоління [26].

Важливість систематизації цих можливостей випливає з особливостей роботи з оперативною пам'яттю. Зокрема, передача управління в область модифікованих даних є однією з таких можливостей, яка дозволяє перенаправити виконання програми до змінених сегментів пам'яті, що може призвести до непередбачуваних наслідків. Самомодифікація або зміна коду іншого програмного забезпечення в оперативній пам'яті чи на зовнішніх носіях інформації також становить значну загрозу, оскільки дозволяє шкідливому ПЗ змінювати власний код або втручатися в роботу інших програмних продуктів.

Крім того, самодублювання, підміна собою іншого програмного забезпечення або перенесення своїх фрагментів у області оперативної чи зовнішньої пам'яті, які не належать програмі, є серйозним порушенням безпеки системи, що може призвести до неконтрольованого розповсюдження шкідливого коду. Збереження інформації з областей оперативної пам'яті, що не належать даному програмному забезпеченню, дозволяє отримувати несанкціонований доступ до конфіденційних даних, що обробляються іншими програмами [28,29].

Спотворення, блокування або заміна інформації, яка є результатом роботи інших програм, може мати суттєвий вплив на цілісність даних. Такі дії здатні призвести до серйозних збоїв у роботі комп’ютерної системи або навіть до втрати важливої та критично необхідної інформації. Наприклад, спотворення даних може змінити результати обчислень або порушити правильність виконання програм, що, в свою чергу, може викликати непередбачувані помилки або збої в роботі програмного забезпечення. Блокування доступу до

певних файлів або ресурсів може перешкодити нормальній роботі системи, створюючи ситуації, коли користувачі або інші програми не можуть отримати доступ до необхідної інформації або функціональних можливостей.

Крім того, заміна інформації, яка генерується або використовується іншими програмами, може призвести до втрати важливих даних або до внесення некоректних даних у систему. Це може мати далекосяжні наслідки, особливо в критичних сферах, таких як фінанси, охорона здоров’я або державне управління, де точність та надійність даних є надзвичайно важливими. Наприклад, у фінансових системах подібні дії можуть призвести до неправильних обчислень балансу рахунків або порушення фінансових операцій, що може спричинити значні фінансові втрати.

Таким чином, важливість систематизації та аналізу цих можливостей обумовлена їхнім значним впливом на особливості роботи з оперативною пам’яттю та загальну безпеку функціонування програмного забезпечення. Робота з пам’яттю є одним із критичних аспектів забезпечення безпеки інформаційних систем, оскільки саме через неї здійснюється обробка та зберігання даних. Тому систематичний підхід до вивчення та усунення подібних вразливостей є необхідним для забезпечення стабільної та безпечної роботи інформаційних систем, а також для захисту важливих даних від можливих атак і втручань з боку зловмисників.

2.3 Висновок

У даній науковій роботі було здійснено систематизацію моделей, які забезпечують надійне та безпечне функціонування програмного забезпечення. Внаслідок проведеного ґрунтовного дослідження було виділено три основні типи моделей: аналітичні, статистичні та емпіричні.

Було розглянуто ряд найбільш часто застосовуваних моделей, а також виділено їхні недоліки та переваги з точки зору вирішення завдання опису

безпечного функціонування програмного продукту та розпізнавання шкідливого програмного забезпечення. За результатами проведених досліджень, розглянуті моделі мають переваги у плані простоти їхньої практичної реалізації. Проте водночас було виявлено наступні суттєві недоліки:

По-перше, деякі з розглянутих моделей при їх реалізації вимагають значного обсягу обчислювальних ресурсів для проведення аналізу безпеки та накопичення архівних даних.

По-друге, статистичні та ймовірнісні моделі базуються на припущеннях про те, що інтенсивність атак або відмов чи кількість помилок у програмному забезпеченні мають заздалегідь відомий розподіл (біноміальний, нормальний або пуассонівський), що не завжди відповідає реальним процесам і системам.

По-третє, відсутнє розділення між відмовами програмного забезпечення та виходом з ладу внаслідок кібератак; при цьому не враховуються вразливості нульового дня.

По-четверте, не аналізуються звернення досліджуваного програмного забезпечення до оперативної пам'яті, що могло б надати важливу інформацію про його легітимність або наявність шкідливих функцій.

І нарешті, жодна з розглянутих моделей не забезпечує комплексного представлення про процес функціонування програмного забезпечення; зокрема, аналіз з точки зору інформаційної безпеки відсутній.

З кожним роком задача ідентифікації та розпізнавання шкідливого програмного забезпечення стає все більш актуальною та складною. Це пов'язано з постійною цифровізацією різних галузей людської діяльності та інтенсивним використанням програмних продуктів для реалізації бізнес-логіки та управління технічними процесами в складних інформаційних системах. У результаті, зі збільшенням обсягу програмного забезпечення в системі, потенційно зростає і кількість помилок, які можуть у ньому міститися. Крім того, завдяки підключенню сучасних систем до глобальної мережі Інтернет, програмне забезпечення часто поширюється по мережі, що відкриває зловмисникам можливості створювати нові вектори кібератак на різноманітні

системи.

Запропонована модель безпечного функціонування програмного продукту спрямована на усунення недоліків, притаманних раніше розглянутим моделям. Вона долає зазначені проблеми завдяки врахуванню характерних особливостей прояву шкідливого програмного забезпечення на обчислювальних пристроях. Зокрема, модель враховує вплив шкідливого ПЗ на обчислювальні ресурси системи, такі як центральний процесор і оперативна пам'ять, а також специфіку його роботи з пам'яттю. Це дозволяє розробленій моделі одночасно враховувати як надійність функціонування програмного забезпечення, так і аспекти його безпеки від потенційних загроз.

У рамках цієї моделі були сформульовані чіткі критерії безпечного функціонування програмного забезпечення. Зроблено висновок, що для найбільш ефективної та практичної реалізації такої моделі доцільно використовувати статичний аналізатор коду. Це забезпечить оптимальну взаємодію між програмним забезпеченням та апаратними ресурсами, підвищуючи загальний рівень безпеки та надійності інформаційної системи.

3 МЕТОД ОЦІНЮВАННЯ ІНФОРМАЦІЙНОЇ БЕЗПЕКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Синтез методів оцінювання безпеки програмного забезпечення

З метою розроблення комплексного методу оцінювання рівня інформаційної безпеки програмного забезпечення, що базується на ретельному статичному аналізі вихідного коду та відповідає заздалегідь побудованій теоретичній моделі, ми здійснимо детальний аналіз і ґрунтовний синтез уже наявних рішень, підходів і практик. Такий підхід дасть змогу виявити найдоцільніші, перспективні й актуальні методи, які можуть виявитися корисними й суттєвими для подальшого використання у рамках даного дослідження.

У сфері статичного аналізу вихідного коду часто застосовуються спеціалізовані інструменти, які орієнтовані на конкретні мови програмування та визначені типи вразливостей. Серед них можна виділити два найпоширеніші: bandit, призначений для коду на Python, та gosec, що використовується у проектах на Go. Обидва аналізатори спроможні виявляти типові помилки безпеки, як-от некоректна робота із введенням користувача, використання застарілих або небезпечних функцій, неправильні налаштування криптографічних примітивів чи загальновідомі патерни, що спричиняють уразливості [30,31].

Gosec, як спеціалізований інструмент статичного аналізу коду на мові Go, має низку недоліків, на які варто зважати під час його використання. По-перше, в окремих випадках gosec може генерувати велику кількість хибно позитивних спрацювань, особливо без додаткового налаштування та фільтрації результатів. Це ускладнює ефективне використання інструменту, адже розробникам доводиться витрачати додатковий час на перевірку та відсіювання нерелевантних попереджень.

По-друге, через свою спрямованість на одну мову програмування, gosec має обмежений обсяг застосування. Якщо проект містить компоненти на інших

мовах або якщо потрібна комплексна оцінка безпеки у змішаному середовищі, доведеться залучати інші інструменти та інтегрувати їх результати вручну. Крім того, налаштування gosec під специфічні потреби чи додавання власних правил аналізу може виявитися нетривіальним завданням. Це вимагає додаткових зусиль і знань про внутрішню структуру інструменту, а також про те, як правильно налаштувати його для мінімізації хибно позитивів та підвищення точності.

Bandit, орієнтований на аналіз коду Python, має низку обмежень, які слід враховувати при його використанні. По-перше, як і в багатьох статичних аналізаторах, Bandit інколи генерує значну кількість хибно позитивних сповіщень. Унаслідок цього розробники змушені витрачати час на ручне відсівання нерелевантних попереджень, що може ускладнювати безперервний робочий процес [32,33].

По-друге, оскільки Bandit фокусується суто на Python-коді, він не здатний забезпечити комплексну оцінку безпеки у проектах зі змішаними технологічними стеками. Для повної картини безпеки може знадобитися інтегрувати результати Bandit з іншими інструментами, що ускладнює управління результатами та їх кореляцію.

Крім того, розширення бази правил або адаптація Bandit під специфічні вимоги проекту може вимагати додаткових зусиль. Це створює певні труднощі при налаштуванні інструмента з метою зменшення шуму та оптимізації процесу виявлення актуальних вразливостей.

Незважаючи на здатність розпізнавати поширені патерни небезпечного коду, обидва інструменти мають труднощі з ідентифікацією більш тонких або контекстно-залежних вразливостей, які вимагають розуміння логіки роботи програми. Це означає, що деякі помилки, пов’язані з бізнес-логікою або нестандартними патернами, можуть залишатися невиявленими.

Хоча і gosec, і Bandit надають певні параметри конфігурації, глибока кастомізація під складні чи нестандартні умови проекту може бути нетривіальною. Налаштування власних правил або систем фільтрації

результатів часом потребує досить значних зусиль, знань внутрішньої архітектури інструменту та вміння писати додаткові сценарії чи конфігурації.

Інструменти на кшталт gosec та Bandit зазвичай фокусуються на виведенні списку виявлених проблем, але не завжди надають розширені функції зведення інформації, візуалізації чи пріоритезації. Це може ускладнювати прийняття управлінських рішень щодо безпеки, особливо у великих командах чи проектах із розгалуженою архітектурою.

Швидкий розвиток мови Go, Python та пов’язаних із ними фреймворків іноді призводить до того, що інструменти не встигають оперативно відреагувати на нові патерни, конструкції або зміни у стандартних бібліотеках. Це може знижувати актуальність деяких правил чи здатність виявляти нові типи вразливостей, якщо інструмент оновлюється не досить часто чи повільно реагує на відповідні зміни в екосистемі.

Попри наявність у bandit та gosec досить широких баз правил, їхній самостійний запуск створює певні складнощі. Кожен із цих інструментів зосереджений на окремому сегменті коду й працює автономно, що унеможливлює пряме співставлення або об’єднання результатів аналізу без додаткових засобів. Відсутність єдиного підходу до агрегації даних призводить до фрагментованої оцінки безпеки, коли аналітик змушений переглядати кілька звітів, виконувати ручну кореляцію та пріоритезацію. До того ж у первинному вигляді bandit та gosec часто породжують чималу кількість хибно позитивних сповіщень, які ускладнюють швидке ухвалення рішень щодо виправлення реальних проблем [34].

Попри наявні недоліки, ці програмні інструменти все ж можуть ефективно функціонувати у взаємодії один з одним, створюючи підґрунтя для розроблення спеціалізованого застосунку. Такий застосунок, завдяки продумано спроектованій архітектурі, зможе об’єднувати різні аналізатори статичного коду в єдиному середовищі. Задля досягнення цієї мети варто впровадити гнучку модульну архітектуру, яка забезпечить легку інтеграцію нових інструментів статичного аналізу коду в майбутньому, уникаючи

технічних обмежень та спрощуючи процес налаштування.

Подібний підхід сприятиме тому, що ми зможемо водночас максимально використати переваги обох наявних інструментів та сформувати застосунок, котрий буде не лише функціональним, а й простим у розширенні та доповненні додатковими можливостями. Таким чином, ми не просто враховуємо поточні недоліки обраних рішень, а й закладаємо основу для їхнього систематичного вдосконалення та покращення методів оцінювання інформаційної безпеки під час подальшої розробки.

3.2 Метод оцінювання інформаційної безпеки безпечного функціонування програмного забезпечення

Приступимо до розроблення оригінального методу, що безпосередньо базується на попередньо сформульованій та побудованій математичній моделі. Під час цього процесу важливо не лише дотримуватися логіки моделі, а й комплексно враховувати всі вимоги, висунуті у попередньому розділі дослідження. Зокрема, це включає забезпечення належного рівня безпеки, стійкість до потенційних загроз, урахування продуктивності та масштабованості, а також гнучкість щодо подальшого розширення та адаптації рішення. Такий системний підхід дозволить створити метод, який органічно поєднає математичну теорію з практичними вимогами, забезпечуючи надійну та ефективну його реалізацію у реальних умовах експлуатації [35,48].

Розроблена система оцінювання інформаційної безпеки програмного забезпечення ґрунтується на ретельно спроектованій модульній архітектурі, що не лише забезпечує високу гнучкість та масштабованість рішення, але й значно спрощує його підтримку та оновлення у майбутньому. Основоположна концепція модульності полягає в розділенні загального функціоналу на чітко відокремлені компоненти, кожен із яких виконує спеціалізовану та логічно завершену задачу. Завдяки такому підходу можна оперативно реагувати на

зміну вимог, з легкістю підключати нові інструменти статичного аналізу коду або адаптувати систему до роботи з дедалі більшими обсягами вихідного коду, не вдаючись до кардинальних змін архітектури.

Архітектура даної системи передбачає наявність кількох ключових модулів, які спільно забезпечують увесь необхідний функціонал. Серед них особливо вирізняється модуль конфігурації – центральний елемент, відповідальний за завантаження, розбір та впорядкування параметрів роботи системи. Цей модуль використовує файл config.yaml, у якому містяться всі потрібні налаштування, включно з детальною інформацією про використані інструменти аналізу, специфічні параметри їхнього запуску, а також повні шляхи до вхідних та вихідних даних. Задіяна в цьому процесі структура Config уніфікує взаємодію між конфігураційними даними та іншими модулями системи, спрощуючи інтеграцію нових компонентів та забезпечуючи надійний, прогнозований і зрозумілий механізм налаштувань.

Другим ключовим компонентом модульної архітектури виступає модуль інтеграції з зовнішніми інструментами статичного аналізу, що реалізований за допомогою структури Analyzer. Цей модуль забезпечує тісну взаємодію системи з різноманітними аналізаторами, котрі можуть відрізнятися за методологією перевірки, типами виявлених вразливостей, а також мовами програмування, з якими вони працюють. У структурі Analyzer задаються команди запуску, набори прапорів (флагів), рівні критичності та пріоритетів для виявлених недоліків безпеки, а також усі необхідні параметри для коректного налаштування і стабільної роботи кожного окремого інструменту. Така гнучка архітектура уможливлює безболісне розширення функціоналу: можна без зайвих перешкод інтегрувати новий аналізатор під конкретні потреби проекту, розширювати підтримку різних мов програмування або вдосконалювати методи виявлення та класифікації вразливостей [36,40].

Наступний важливий елемент архітектури – це модуль обробки та інтерпретації результатів, який відповідає за систематизацію та упорядкування зібраних даних. Він виконує роль своєрідного «моста» між сукупністю

аналізаторів і кінцевим користувачем або інтегрованою системою контролю якості коду.

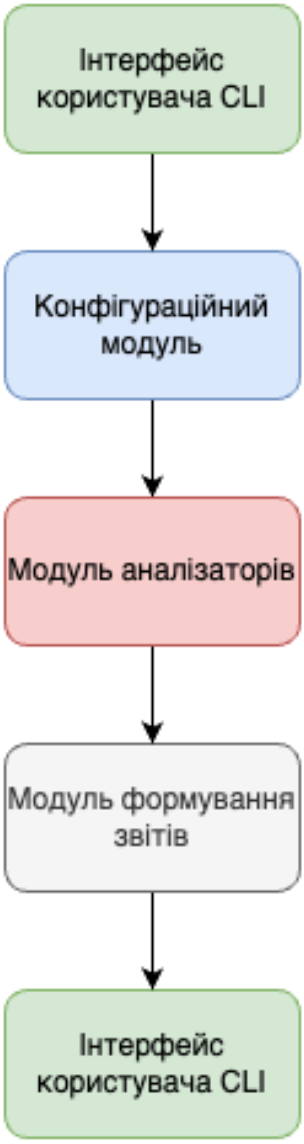


Рисунок 3.1 – Схема ієрархії модульної архітектури

Цей модуль отримує результати перевірок від усіх підключених аналізаторів, уніфікує їх та перетворює у зрозумілі формати – від простих текстових звітів до структурованих JSON-файлів, придатних для автоматизованої обробки. Завдяки модульності архітектури можна оперативно адаптувати формат і логіку звітності до поточних вимог проекту, зокрема інтегрувати результати в процеси безперервної інтеграції та розгортання

(CI/CD). Таким чином, кінцевий продукт не лише ефективно знаходить вразливості, а й забезпечує прозору звітність, зрозумілу як інженерам, так і менеджерам, та легко вбудовується в існуючі робочі процеси [37].

Модульний підхід також забезпечує зручність оновлення та обслуговування системи [38]. Наприклад, якщо один із компонентів потребує оновлення або заміни (наприклад, інтеграція нового аналізатора), це можна виконати без внесення значних змін до інших частин системи. Завдяки цьому система залишається стабільною, навіть якщо функціонал постійно розширюється.

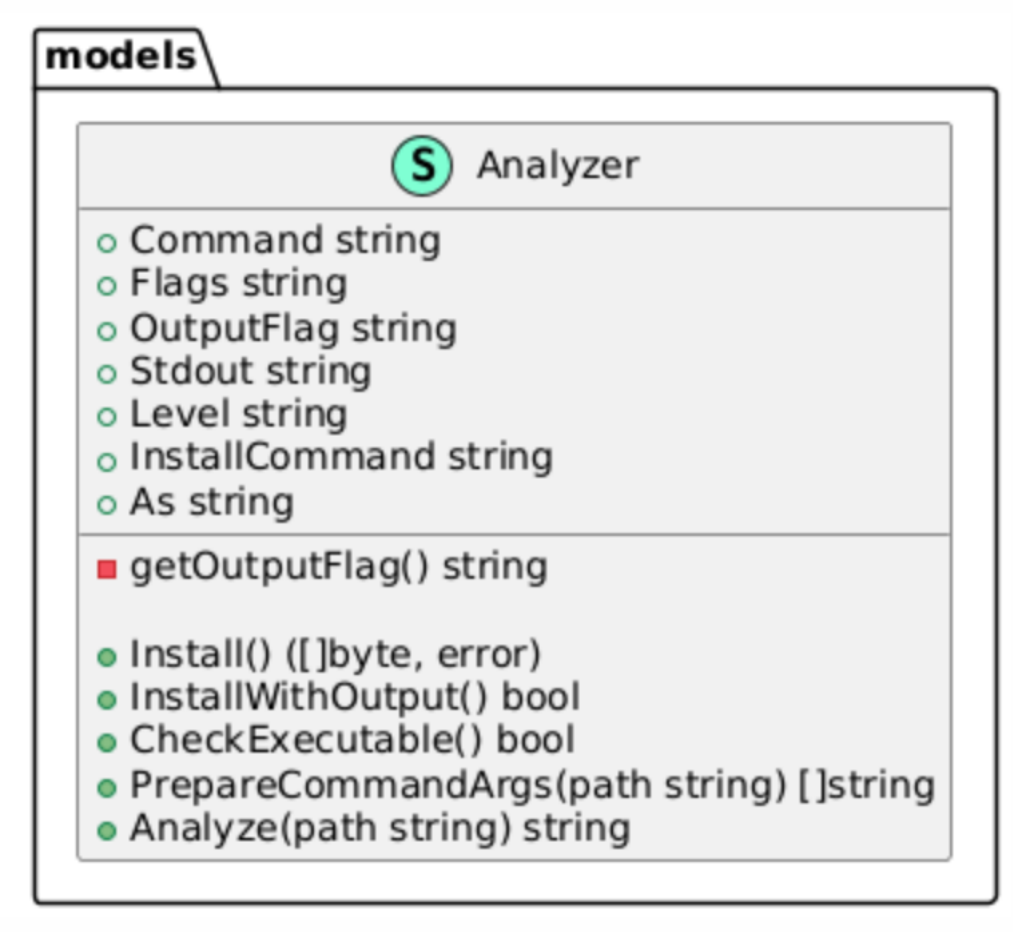


Рисунок 3.2 – Структура Analyzer з атрибутами та методами

Концепція модульності робить систему універсальною та ефективною для вирішення завдань аналізу безпеки програмного забезпечення. Вона дозволяє реалізувати гнучкий, адаптивний та масштабований підхід, що відповідає

сучасним вимогам до оцінювання інформаційної безпеки.

Масштабованість системи забезпечується її здатністю працювати з великими кодовими базами завдяки поетапній обробці вхідних даних. Кожен файл або модуль аналізується окремо, що дозволяє мінімізувати використання ресурсів та забезпечити стабільну роботу програми навіть у масштабних проектах. Крім того, реалізована можливість одночасного запуску кількох аналізаторів, що значно скорочує час аналізу. Для кожної мови програмування використовуються спеціалізовані інструменти, наприклад, аналізатор для Go-коду або аналізатор для Python, які працюють паралельно. Цей підхід дозволяє досягати високої ефективності під час виконання задач, що потребують комплексного аналізу [29, 59,60].

Адаптивність системи забезпечується її здатністю працювати з різними мовами програмування та легко інтегрувати нові інструменти аналізу. Завдяки використанню структури Analyzer всі необхідні параметри роботи інструментів задаються в конфігураційному файлі, що дозволяє користувачеві легко змінювати налаштування під конкретний проект.

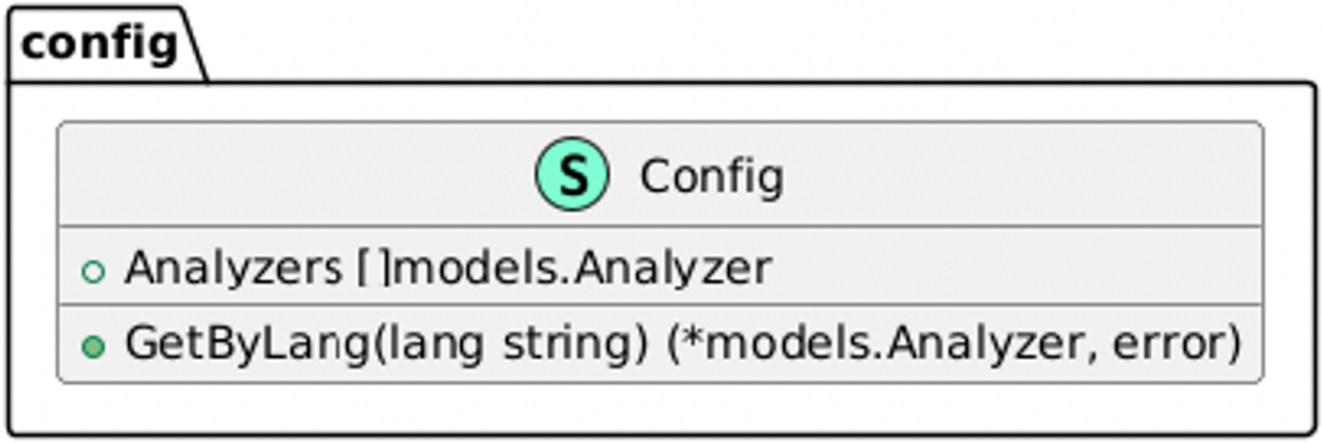


Рисунок 3.3 – Структура Config з атрибутом та методом

Система підтримує різні підходи до аналізу, адаптуючи свої процеси до вимог сучасного програмного забезпечення. Наприклад, для кожної мови програмування можна вказати власний набір параметрів, команд та прапорів, що дозволяє інтегрувати навіть ті інструменти, які раніше не були передбачені.

Завдяки такій гнучкості, система ефективно адаптується до змін у середовищі розробки чи до специфічних вимог проекту [39, 61].

Особливості масштабованості та адаптивності системи роблять її універсальним інструментом для аналізу програмного забезпечення різного розміру та типу. Вона не лише дозволяє працювати з великими проектами, але й забезпечує легкість конфігурації та розширення функціоналу, що відповідає сучасним потребам у сфері безпеки програмного забезпечення.

Рисунок 3.4 – Метод оцінювання інформаційної безпеки програмного забезпечення

Для реалізації системи оцінювання інформаційної безпеки програмного забезпечення були використані сучасні технології та інструменти, які забезпечують високу продуктивність, гнучкість і адаптивність. Основою розробки є мова програмування Go, яка була обрана завдяки її продуктивності, простоті інтеграції із зовнішніми інструментами та зручності роботи з паралельними процесами.

Мова Go забезпечує ефективну взаємодію з операційною системою для запуску зовнішніх інструментів аналізу через CLI-команди. Це дозволило легко

інтегрувати спеціалізовані аналізатори для різних мов програмування, наприклад, аналізатор для Go і аналізатор для Python. Інструменти аналізу запускаються із параметрами, які визначаються у конфігураційному файлі config.yaml, що дозволяє автоматизувати процес налаштування.

Взаємодія модулів у системі базується на чіткій послідовності дій, яка забезпечує зчитування конфігурацій, виконання аналізу та формування звітів. Завдяки модульній архітектурі кожен компонент виконує свою роль, а обмін даними між ними побудований на стандартизованих структурах, таких як Config та Analyzer. Алгоритм виконання модулі наступний:

– ініціалізація системи. На першому етапі система зчитує конфігураційний файл config.yaml. Його вміст завантажується у структуру Config, яка містить список аналізаторів, їхні параметри та налаштування. Це дозволяє системі зрозуміти, які інструменти аналізу використовуються, їхні команди запуску та прапори, а також місця для збереження результатів;

– підготовка аналізаторів. На основі даних зі структури Config створюються об'єкти Analyzer, які містять специфічну інформацію для кожного інструменту. Якщо аналізатор не встановлений у середовищі, система виконує його інсталяцію за допомогою команди, зазначеної у полі InstallCommand. Це забезпечує автоматичну готовність системи до роботи;

– обробка файлів для аналізу. Система ідентифікує вхідні файли або директорії, зазначені в конфігураційному файлі, та підготовлює їх для передачі в аналізатори. Для великих проектів реалізована поетапна обробка, що дозволяє мінімізувати навантаження на систему;

– виконання аналізу. Кожен аналізатор викликається окремо через CLI на основі команди, визначеної в полі Command структури Analyzer. Параметри передаються у вигляді прапорів (Flags), а результати зберігаються у файлах, вказаних у полях Stdout або OutputFlag. Цей процес забезпечує незалежність аналізаторів один від одного, дозволяючи виконувати їх паралельно.

– обробка результатів. Отримані результати стандартизуються у модулі обробки даних. Це дозволяє об’єднати вивід від різних аналізаторів у єдиний

формат. Дані класифікуються за рівнем критичності, типом вразливостей та місцезнаходженням у коді;

– формування звітів. На завершальному етапі система генерує звіти у визначеному користувачем форматі (наприклад, текстовий файл або JSON). Звіти містять детальну інформацію про знайдені уразливості, рекомендації щодо їх усунення та статистику про загальний стан безпеки проекту;

– вивід результатів. Готові звіти зберігаються у вихідній директорії, зазначеній у конфігураційному файлі. Користувач отримує доступ до детальних даних про виконаний аналіз, що дозволяє швидко прийняти рішення про подальші дії.

Таким чином, у результаті нашої роботи було створено метод оцінювання інформаційної безпеки програмного забезпечення, який повністю узгоджується з побудованою математичною моделлю та дотримується визначеного переліку вимог.

Створений метод відповідає головній меті цього дослідження, а саме – підвищенню рівня безпеки програмного забезпечення шляхом раннього виявлення та систематизації вразливостей. Застосування такого методу сприятиме формуванню зрілої й безпечної культури розробки, а також відкриває простір для наступних наукових досліджень та інженерних рішень, здатних підняти якість та надійність програмних продуктів на новий рівень.

3.3 Застосування створеного методу у системах програмного забезпечення

Створений нами метод демонструє надзвичайно високий рівень гнучкості й універсальності під час свого практичного застосування. Завдяки його модульній архітектурі та здатності легко адаптуватися до різних сценаріїв використання, даний метод можна ефективно інтегрувати як у локальне середовище розробки програмного забезпечення – наприклад, на робочих

станціях окремих розробників, – так і на сервері під час налагодження безперервних процесів інтеграції та доставки коду (CI/CD) [40,41].

Цей підхід надає можливість виявляти й усувати потенційні уразливості та недоліки безпеки безпосередньо на етапі розробки, що значно скорочує час відновлення й покращує загальну якість програмного продукту. Одночасно, його використання у CI/CD-конвеєрах дозволяє оперативно реагувати на нові зміни в коді, виконуючи автоматизований аналіз у реальному часі та підтримуючи незмінно високі стандарти якості й безпеки коду протягом усього життєвого циклу розробки. Такий підхід формує цілісну та гармонійну систему забезпечення надійності та захищеності програмних рішень.

Статичні аналізатори коду можуть бути органічно вбудовані у процес безперервної інтеграції та доставки (CI/CD) через створення окремих завдань у складі конвеєрів [42]. Завдяки цьому, кожен коміт чи злиття гілок автоматично супроводжуються запуском визначених інструментів перевірки безпеки та якості коду. Такі завдання виконуються на етапі, коли програмний продукт ще не перейшов до середовища виконання, що дає змогу виявити потенційні проблеми заздалегідь та запобігти їхньому потраплянню у продакшн. На рисунку 3.4 зображений алгоритм взаємодії розробника в процесі CI/CD.

Налаштування завдань у складових конвеєрів передбачає ретельно структурований підхід до визначення критеріїв запуску, середовища виконання та параметрів налаштування окремих статичних аналізаторів. Завдяки цьому, під час кожного автоматизованого запуску процесу розробки та інтеграції, статичні аналізатори методично сканують вихідний код, виявляючи типові та потенційно небезпечні вразливості, типові програмні помилки, а також будь-які відхилення від усталених політик і стандартів безпеки [43,58]. Отримані результати аналізу зберігаються у зрозумілому та стандартизованому форматі, після чого формуються аналітичні звіти з детальним описом знайдених недоліків [44,59].

Крім того, інтеграція аналізаторів до складу автоматизованих конвеєрів забезпечує виважене масштабування процесу оцінювання безпеки програмного

коду. Завдяки продуманій модульності та гнучкій архітектурі будь-які нові інструменти чи додаткові перевірки можна додавати або замінювати без ризику дестабілізації розробницького середовища. Такий підхід не просто підвищує зрілість і структурованість процесів розробки програмного забезпечення, а й дає командам змогу оперативно вдосконалювати якість та інформаційну безпеку кінцевого продукту.

Рисунок 3.4 – Алгоритм взаємодії розробка в процесі CI/CD

На рисунку 3.5 зображено алгоритм взаємодії розробника та сервера в процесі CI/CD. У цьому процесі беруть участь два основні суб’єкти: розробник та сервер CI/CD.

На початковому етапі розробник вносить необхідні зміни у вихідний код

програмного забезпечення, працюючи над його вдосконаленням, додаванням нових функціональних можливостей чи виправленням наявних недоліків. Після завершення роботи зміни комітяться до репозиторію, що виступає своєрідним центральним сховищем коду. Саме цей крок ініціює процес безперервної інтеграції та доставки (CI/CD) на спеціально налаштованому сервері [45,46].

Отримавши сигнал про новий коміт у репозиторій, сервер безперервної інтеграції та доставки (CI/CD) автоматично активує заздалегідь налаштований збірний конвеєр (pipeline). Цей конвеєр, зазвичай створений із урахуванням конкретних потреб проєкту та конфігураційних параметрів, складається з окремих завдань (jobs), кожне з яких відіграє строго визначену роль у процесі перевірки, складання та деплойменту програмного продукту.

На одному з початкових етапів роботи конвеєра виконується статичний аналіз коду – ключовий елемент комплексної оцінки інформаційної безпеки. Завдяки статичному аналізу можливо виявити широкий спектр потенційних проблем: починаючи від тривіальних синтаксичних похибок і відхилень від прийнятих стандартів кодування, закінчуючи серйозними вразливостями, що можуть загрожувати безпеці та надійності програмного забезпечення. Якщо під час аналізу будуть знайдені критичні загрози або суттєві недоліки, система автоматично зафіксує їх у детальному звіті, який негайно передається команді розробників. Отримавши такий звіт, інженери можуть оперативно вжити необхідних заходів щодо виправлення проблемних ділянок коду. Якщо ж аналіз не виявить жодних критичних помилок, процес продовжується без затримок.

Після успішного завершення етапу статичного аналізу система генерує білд – попередньо зібрану, готову до тестування версію програмного продукту [45,57]. Наступний крок – виконання серії автоматизованих тестів, що перевіряють коректність реалізованої функціональності, оцінюють продуктивність, сумісність та відповідність застосунку визначеним сценаріям використання. Якщо всі тести пройдуть успішно, білд автоматично буде розгорнуто у спеціалізованому тестовому середовищі, умови якого максимально наближені до реальної експлуатації. Такий багатоступеневий

підхід забезпечує високий рівень якості, надійності та безпеки програмного забезпечення, одночасно оптимізуючи робочі процеси команди розробників [22,47,48].

Такий ретельно спланований та багатоступеневий алгоритм забезпечує всебічний і комплексний підхід до перевірки вихідного коду, а також значно підвищує загальну якість розроблюваного програмного забезпечення. Поєднання автоматизованого аналізу безпеки з детальним та всебічним функціональним тестуванням, а також оперативним і ефективним деплойментом у спеціалізоване тестове середовище, створює синергію, яка дозволяє команді розробників миттєво реагувати на виявлені проблеми та недоліки. Це сприяє постійному підвищенню стандартів якості та надійності розроблюваних продуктів, адже кожен виявлений дефект або вразливість оперативно усувається, запобігаючи їхньому накопиченню та потенційному негативному впливу на кінцевий продукт.

Крім того, інтеграція цих етапів у єдиний автоматизований процес дозволяє мінімізувати потенційні ризики, що виникають протягом усього циклу розробки програмного забезпечення. Автоматизація рутинних завдань знижує ймовірність людських помилок, підвищує швидкість виконання перевірок та тестів, а також забезпечує послідовність і стандартизацію процесів [49, 56]. Це, в свою чергу, дозволяє розробникам зосередитися на більш критичних аспектах створення програмних продуктів, таких як архітектура, функціональність та інновації, забезпечуючи при цьому високу якість та безпеку кінцевого продукту.

Таким чином, впровадження такого комплексного алгоритму не лише оптимізує робочі процеси команди розробників, але й створює міцну основу для підтримки високих стандартів безпеки та якості програмного забезпечення. Це сприяє підвищенню загальної ефективності розробки, зменшенню часу виходу продукту на ринок та забезпеченню його надійності та безпеки у реальних умовах експлуатації, що є надзвичайно важливим у сучасному динамічному середовищі розробки програмних рішень.

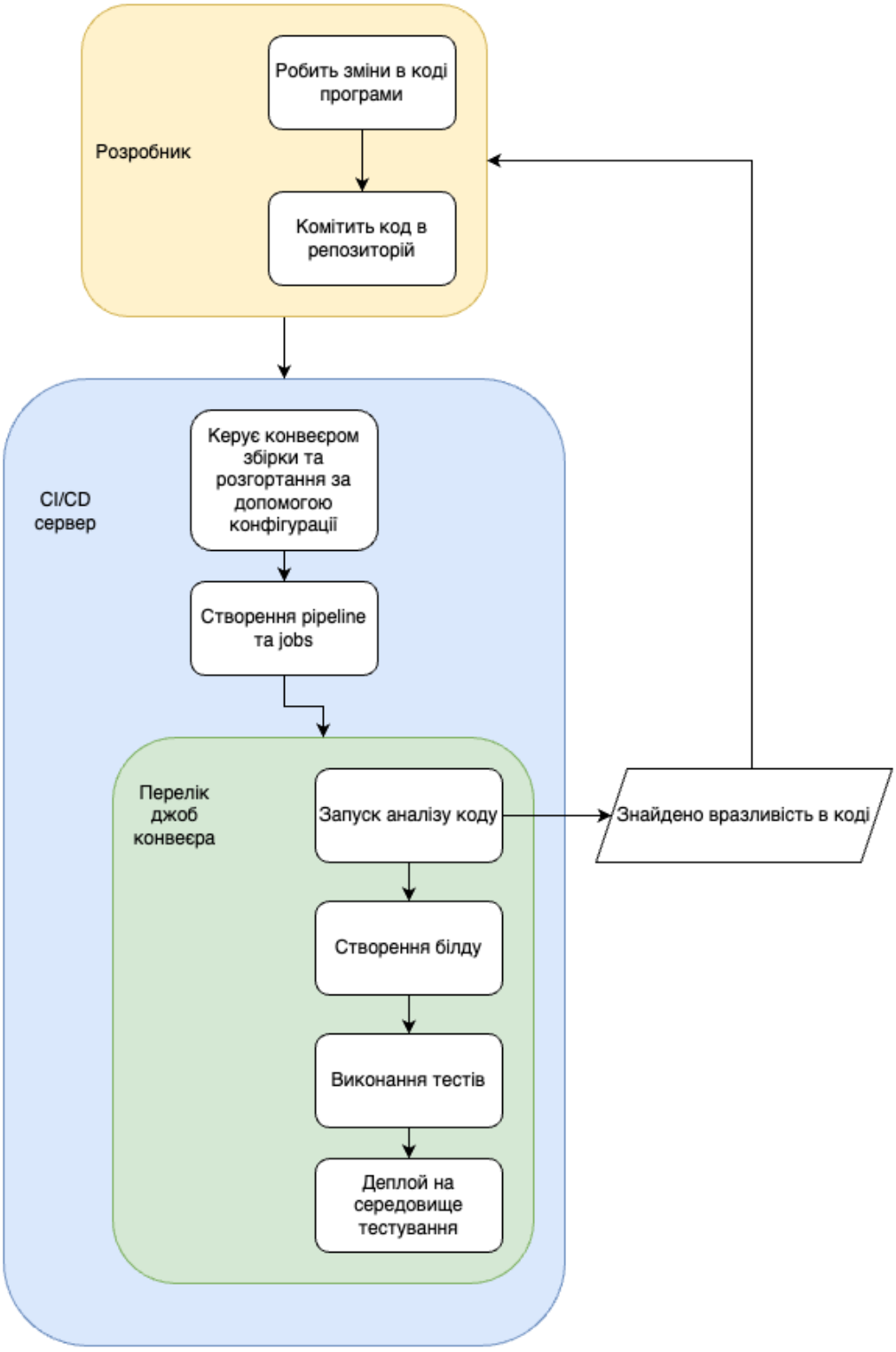


Рисунок 3.5 – Алгоритм роботи CI/CD за участі розробника та сервера

Автоматизація є ключовою особливістю розробленої системи, яка забезпечує ефективність, зручність використання та мінімізацію людського фактору під час оцінювання інформаційної безпеки програмного забезпечення.

Основна ідея полягає в автоматичному виконанні всіх етапів аналізу: від підготовки вхідних даних до формування звітності.

Алгоритм роботи завдання у конвеєрі CI/CD розпочинається з моменту активації процесу, яка зазвичай ініціюється після виконання нового коміту у репозиторії або примусового запуску пайплайну. Перш за все, система переходить до етапу ініціалізації проєкту, який включає ретельне зчитування конфігураційного файлу (наприклад, config.yaml) та налаштувань, визначених у ньому. На цьому ж етапі відбувається підготовка середовища виконання – від встановлення необхідних версій мов програмування й фреймворків до завантаження додаткових модулів та залежностей, без яких неможливе коректне функціонування аналізаторів коду. Цей алгоритм зображений на рисунку 3.6.

Після успішної ініціалізації проєкт переходить до наступного етапу – запуску команд для статичного аналізу коду. Тут система виконує заздалегідь налаштовані інструменти, перевіряючи вихідний код на наявність вразливостей, потенційних помилок і невідповідностей встановленим стандартам безпеки та кодування. У процесі роботи аналізатори можуть ідентифікувати широкий спектр проблем – від тривіальних недоліків стилю до критичних ризиків безпеки, які потребують невідкладної уваги [50,51].

Результати роботи інструментів збираються, нормалізуються та уніфікуються у відповідний формат для подальшого сприйняття й аналізу. Після завершення перевірок система створює детальний звіт, який може містити повний перелік виявлених вразливостей, їх критичність, потенційний вплив на безпеку продукту та рекомендації щодо виправлення. Такий звіт може бути надісланий команді розробників, спеціалістам з безпеки чи інтегрований у зовнішні системи керування якістю коду.

Завершальним етапом є формальне завершення роботи завдання. Після формування звіту система переходить у стан очікування наступного запуску. Це означає, що як тільки з’являться нові зміни у вихідному коді або буде ініційовано повторний запуск конвеєра, алгоритм знову активується,

забезпечуючи постійний контроль якості та безпеки програмного забезпечення у рамках процесів безперервної інтеграції та доставки.

Рисунок 3.6 – Алгоритм роботи завдання конвеєра

Система інтегрує зовнішні інструменти аналізу, такі як аналізатори для Go та Python, через прямі виклики CLI-команд. Усі команди для запуску інструментів, їх параметри та налаштування задаються в конфігураційному файлі config.yaml. Завдяки цьому процес інтеграції повністю автоматизований: користувачеві достатньо один раз налаштувати конфігураційний файл [52,53].

Для зручності та оптимізації процесів аналізу безпеки реалізовано автоматичний контроль наявності необхідних інструментів у середовищі

виконання. Якщо один із потрібних аналізаторів чи додаткових утиліт відсутній, система самостійно ініціює його встановлення за допомогою команди, попередньо визначеної у структурі Analyzer. Така функціональність дає змогу суттєво знизити ризик помилок під час запуску, забезпечити повноцінну роботу навіть у свіжому або нестандартному середовищі та позбавляє користувача необхідності вручну дбати про сумісність та доступність усіх компонентів [54, 56].

Автоматизація також охоплює процес формування підсумкових звітів. Система збирає результати з усіх задіяних інструментів аналізу, упорядковує та нормалізує їх, а потім створює звіти у заздалегідь визначеному форматі – наприклад, текстовому або у форматі JSON. Формат звітності та інші параметри можна гнучко налаштовувати через конфігураційний файл, адаптуючи під конкретні потреби проекту. Це забезпечує безперервний моніторинг якості та безпеки програмного забезпечення на кожному етапі його розробки та інтеграції з CI/CD-конвеєрами [55].

Сукупність описаних можливостей робить систему зручною, продуктивною та надійною у використанні. Автоматизація більшості рутинних завдань мінімізує людський фактор, зменшуючи ризики помилок, а також дозволяє спеціалістам зосередитися безпосередньо на результатах аналізу, їх інтерпретації та впровадженні потрібних змін. Універсальність і масштабованість системи сприяє її ефективному використанню розробниками, тестувальниками й інженерами з безпеки, підвищуючи загальну якість та інформаційну стійкість програмного продукту.

3.4 Висновок

У третьому розділі було здійснено аналіз методів оцінювання інформаційної безпеки програмного забезпечення на основі статичного аналізу коду, а також розроблено оригінальний підхід до впровадження такого методу.

Запропонована модель не лише підтвердила ефективність застосування статичних аналізаторів для виявлення вразливостей на ранніх етапах життєвого циклу розробки, а й продемонструвала здатність систематизувати отримані результати, забезпечуючи більш глибоке розуміння потенційних загроз та їхнього впливу на кінцеву якість програмного продукту.

За допомогою модульної архітектури вдалося досягти гнучкості та простоти інтеграції з різними інструментами статичного аналізу коду, що, своєю чергою, сприяє підвищенню рівня автоматизації та ефективності проведення перевірок.

Перспективи подальшого вдосконалення методу охоплюють інтеграцію з додатковими сторонніми інструментами, розширення підтримуваних мов програмування, а також використання новітніх концепцій машинного навчання для підвищення повноти та релевантності результатів. Такий підхід передбачає розбудову системи, здатної швидко адаптуватися до мінливих викликів безпеки у сучасному розробницькому середовищі, водночас забезпечуючи безперервне підвищення якості та надійності програмного забезпечення.

Таким чином, розроблений метод не лише закладає основу для подальших досліджень та впроваджень у сфері виявлення вразливостей, а й формує фундамент для побудови комплексних рішень, орієнтованих на поглиблений аналіз коду, розширені можливості інтеграції, а також оперативний та точний контроль інформаційної безпеки у динамічному процесі розробки програмних продуктів.

4 ПРИКЛАДНЕ ЗАСТОСУВАННЯ МЕТОДУ ОЦІНЮВАННЯ ІНФОРМАЦІЙНОЇ БЕЗПЕКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Реалізація методу оцінювання інформаційної безпеки для проведення аналізу коду на вразливість

Перш ніж розпочати процес розробки програмних прототипів, детально представимо перелік інструментів, які є необхідними для успішного та ефективного виконання поставленого завдання. Крім того, ми надамо докладний опис загальних методологій та підходів, що будуть застосовуватися протягом усього процесу реалізації проекту.

Для розробки програмного забезпечення для аналізу коду на вразливості було використано операційну систему macOS. macOS — це UNIX-подібна операційна система, розроблена компанією Apple Inc., яка відома своєю стабільністю, безпекою та зручністю для розробників. Вона надає потужний набір інструментів та середовищ для розробки, що робить її популярним вибором серед професіоналів у галузі програмування.

Однією з ключових переваг macOS є її сумісність з різними технологіями та мовами програмування, що дозволяє розробляти кросплатформені застосунки. Розроблене програмне забезпечення було спроектовано з урахуванням можливості виконання на системах Linux та macOS, що забезпечує широку доступність та гнучкість використання. Обидві ці операційні системи підтримують необхідні інструменти та бібліотеки для аналізу коду, що сприяє ефективному виявленню вразливостей.

Для розробки програмного забезпечення для аналізу коду на вразливості було використано мову програмування Go. Вона ідеально підходить для створення інструментів аналізу завдяки вбудованим можливостям паралельного програмування та статичної типізації.

У процесі розробки використовувалися середовища розробки GoLand та Visual Studio Code. GoLand — це спеціалізована інтегрована середовище розробки (IDE) від компанії JetBrains, оптимізована для мови Go. Вона надає

розширені функції автодоповнення, рефакторингу, налагодження та інтеграції з системами контролю версій, що значно підвищує продуктивність розробника.

Використання цих інструментів у поєднанні з macOS як основною операційною системою створило ефективне середовище для розробки кросплатформеного програмного забезпечення. Це дозволило забезпечити сумісність програми з операційними системами Linux та macOS, оскільки мова Go та використовувані IDE підтримуються на обох платформах.

Кожна мова програмування неможлива без залучення різноманітних бібліотек та пакетів, які вже реалізують певні логічні функції та механізми, що значно спрощують та оптимізують процес розробки програмного забезпечення. Для реалізації нашого проекту було використано наступні бібліотеки: urfave/cli/v2 та gopkg.in/yaml.v3.

Бібліотека urfave/cli/v2 застосовувалася для створення інтерфейсу командного рядка з підтримкою детального налаштування команд, прапорців та аргументів, що дозволило реалізувати гнучкий механізм обробки вхідних параметрів. Це спростило інтеракцію з користувачем і забезпечило можливість змінювати поведінку програми без необхідності редагування коду. Бібліотека gopkg.in/yaml.v3 використовувалася для парсингу та генерації конфігураційних файлів у форматі YAML, що дало змогу зберігати налаштування аналізу в структурованому та легко читаному вигляді. Це дозволило динамічно змінювати параметри аналізу, такі як список перевірок або виключень, без перекомпіляції програми.

У процесі розробки програмного забезпечення активно використовувалися система контролю версій Git та онлайн-платформа GitHub. Система Git забезпечувала детальне відстеження всіх внесених змін у коді, що дозволяло ефективно керувати різними версіями програми та координувати роботу кількох розробників над одним проектом. Платформа GitHub виконувала роль віддаленого репозиторію, забезпечуючи безпечне зберігання коду в хмарному середовищі та надаючи можливість доступу до нього з різноманітних пристроїв та місць. Використання Git та GitHub сприяло

оптимізації організації робочого процесу, спрощенню співпраці між членами команди, а також дозволяло швидко впроваджувати необхідні зміни або, за потреби, повертатися до попередніх версій коду. Це забезпечувало стабільність та надійність програмного продукту, а також дозволяло ефективно реагувати на виникаючі потреби та виклики під час розробки.

Система побудована на основі двох ключових структур — Config та Analyzer, які забезпечують налаштування, взаємодію з інструментами аналізу та обробку результатів. Ці структури формують ядро системи, реалізуючи гнучкість, масштабованість і модульність.

Config відповідає за зберігання та управління параметрами конфігурації системи. У ній міститься інформація, що визначає загальну роботу системи, включаючи список інструментів аналізу, шляхи до вхідних даних та формат звітності. На рисунку 4.1 зображено структуру Config.

Рисунок 4.1 – Структура Config

Роль Config у системі полягає у зчитуванні конфігураційного файлу config.yaml, який визначає параметри роботи, зберіганні налаштувань для запуску інструментів аналізу та передачі параметрів до інших компонентів системи, таких як модуль обробки команд або генерація звітів.

Analyzer відповідає за опис кожного інструменту аналізу, його параметрів, команд та способу інтеграції із системою. Кожен інструмент аналізу має власну конфігурацію, яка задається через цю структуру.

Роль Analyzer у системі полягає в описі кожного інструменту аналізу, що використовується, а також у формуванні команд для їх запуску з заданими параметрами. Крім того, Analyzer контролює збереження результатів у визначеному форматі та місці, а також автоматизує процес встановлення

інструментів за допомогою команди InstallCommand. На рисунку 4.2 зображено структуру Analyzer.

```
10
11  type Analyzer struct {  7 usages  Oleksandr Yavorskyi
12      Command        string `yaml:"command"`
13      Flags          string `yaml:"flags"`
14      OutputFlag     string `yaml:"output_flag"`
15      Stdout         string `yaml:"stdout"`
16      Level          string `yaml:"level"`
17      InstallCommand string `yaml:"install_command"`
18      As             string `yaml:"as"`
19  }
20
```

Рисунок 4.2 – Структура Analyzer

Взаємодія між структурами Config і Analyzer починається із завантаження конфігурації. Система зчитує файл config.yaml, перетворюючи його у відповідний об’єкт Config, який містить список аналізаторів у вигляді об’єктів Analyzer. На основі цих даних для кожного аналізатора виконується команда, зазначена у полі Command, із відповідними прапорами та параметрами, визначеними в конфігурації. Результати роботи аналізаторів зберігаються у заздалегідь визначених місцях: текстовий вивід направляється у файл, вказаний у полі Stdout, а звіти формуються у форматі, визначеному через OutputFlag. Така взаємодія забезпечує цілісність процесу від завантаження налаштувань до отримання стандартизованих результатів аналізу.

Файл config.yaml є конфігураційним файлом системи, який визначає всі основні параметри її роботи. Він служить центральним місцем для налаштування аналізаторів, шляху до вхідних даних, параметрів їх обробки та формату звітності. Завдяки простоті синтаксису YAML цей файл легко редагується і дозволяє адаптувати систему до потреб конкретного проєкту.

Файл, зображений на рисунку 4.3, містить розділ analyzers, у якому визначено список інструментів аналізу. Кожен аналізатор описаний окремим блоком, що включає параметри, необхідні для його роботи. Наприклад, у конфігурації вказуються команда для запуску інструмента, додаткові прапори, місце для збереження результатів і рівень критичності знайдених уразливостей. Для автоматизації роботи також додається команда для встановлення аналізатора, якщо він ще не інстальований.

config.yaml

```yaml
analyzers:
  - command: "analyzer-go"
    flags: "--verbose"
    output_flag: "-o"
    stdout: "output/go-report.txt"
    level: "critical"
    install_command: "go install analyzer-go"
    as: "GoAnalyzer"
  - command: "analyzer-python"
    flags: "--check-all"
    output_flag: "-o"
    stdout: "output/python-report.txt"
    level: "medium"
    install_command: "pip install analyzer-python"
    as: "PythonAnalyzer"
```

Рисунок 4.3 – Конфігураційний файл config.yaml

Файл config.yaml забезпечує централізоване налаштування системи, що спрощує її використання. Замість того щоб вручну передавати параметри кожного разу при запуску, користувач може визначити всі необхідні налаштування у конфігураційному файлі. Це дозволяє автоматизувати процеси,

підвищити зручність роботи та забезпечити гнучкість системи для роботи з різними інструментами та типами аналізу.

Реалізація процесу зчитування конфігураційних налаштувань, обробки файлів та виклику інструментів аналізу є одним із ключових технічних аспектів функціонування розробленої системи. Цей процес спроектований на основі принципів автоматизації та модульності, що дозволяє забезпечити високу ефективність роботи, гнучкість налаштувань та можливість масштабування системи відповідно до потреб різних проектів.

Процес зчитування конфігураційних параметрів починається з файлу config.yaml, у якому детально визначено всі необхідні налаштування системи. До цих налаштувань належать перелік аналізаторів, параметри їхнього функціонування, а також шляхи для збереження отриманих результатів аналізу. Для завантаження та обробки конфігураційного файлу використовується бібліотека gopkg.in/yaml.v2, яка дозволяє перетворити вміст файлу у відповідну структуру даних Config. На початковому етапі запуску системи цей конфігураційний файл відкривається та зчитується, після чого отримані дані передаються відповідним об'єктам Config та Analyzer. Це забезпечує централізоване управління усіма параметрами системи, що значно спрощує подальшу роботу з ними та дозволяє легко вносити зміни до налаштувань без необхідності редагування основного коду програми.

Обробка файлів у системі здійснюється з використанням даних, отриманих з конфігураційного файлу, для запуску відповідних аналізаторів. Кожен аналізатор має свої специфічні параметри, які включають шляхи до вхідних файлів, додаткові прапори та місця для збереження вихідних даних. Система послідовно проходить через перелік аналізаторів, визначених у конфігурації, та підготовлені шляхи до файлів для проведення аналізу. Це дозволяє забезпечити гнучкість у використанні різних інструментів аналізу та налаштувань відповідно до специфіки проекту.

Система також забезпечує підтримку великих кодових баз завдяки покроковій обробці кожного окремого файлу або модуля програми. Такий

підхід дозволяє уникнути перевантаження оперативної пам'яті, що особливо важливо для масштабних та складних проектів, де обсяг вихідного коду може бути значним. Поступова обробка файлів дозволяє ефективно розподіляти ресурси системи, забезпечуючи високу продуктивність та стабільність роботи навіть при роботі з великими об'ємами даних.

Виклик інструментів аналізу реалізовано за допомогою стандартного пакету Go os/exec, який надає можливість запускати зовнішні програми з необхідними параметрами. На основі даних, отриманих зі структури Analyzer, формуються команди командного рядка (CLI-команди), які включають основну команду запуску аналізатора, додаткові прапори та параметри для визначення місць збереження вихідних даних. Кожен виклик інструменту супроводжується перевіркою результатів його виконання, що включає обробку можливих помилок та забезпечення коректного завершення процесу аналізу. Це дозволяє гарантувати надійність та стабільність роботи системи, а також оперативно реагувати на будь-які несподівані ситуації або помилки, що можуть виникнути під час виконання аналізу.

Таким чином, реалізація процесу зчитування конфігураційних параметрів, обробки файлів та виклику інструментів аналізу в рамках розробленої системи ґрунтується на сучасних принципах автоматизації та модульності. Цей підхід не лише забезпечує високу ефективність та гнучкість роботи системи, але й надає можливість її подальшого розширення та адаптації до різноманітних умов експлуатації та специфічних вимог проектів. Завдяки цьому система стає більш стійкою до змін, дозволяючи легко інтегрувати нові функціональні можливості без необхідності кардинальної перебудови існуючої архітектури.

Автоматизація процесів зчитування конфігураційних параметрів забезпечує швидке та точне налаштування системи відповідно до заданих вимог, мінімізуючи людський фактор і знижуючи ймовірність виникнення помилок. Модульність ж дозволяє розділити систему на окремі, незалежні компоненти, кожен з яких відповідає за виконання конкретних завдань. Це сприяє підвищенню надійності системи, оскільки проблеми в одному модулі не

впливають на роботу інших частин системи.

Завдяки таким підходам, створена система стає надійним та універсальним інструментом для оцінювання інформаційної безпеки програмного забезпечення. Вона може бути ефективно інтегрована у різні процеси розробки та підтримки програмних продуктів, незалежно від їхньої складності та масштабу. Це забезпечує високий рівень безпеки та якості кінцевого продукту, сприяючи зменшенню ризиків, пов'язаних з вразливостями та помилками в коді, а також підвищенню загальної продуктивності команди розробників.

Таким чином, використання сучасних принципів автоматизації та модульності в процесі зчитування конфігураційних параметрів, обробки файлів та виклику інструментів аналізу дозволяє створити потужний та адаптивний інструмент для оцінювання інформаційної безпеки програмного забезпечення. Це сприяє підвищенню загальної ефективності процесів розробки, забезпечуючи високу якість та безпеку програмних продуктів, а також дозволяє оперативно реагувати на нові виклики та вимоги ринку.

4.2 Функціональні можливості системи аналізу інформаційної безпеки програмного забезпечення

CLI-команда є основним інтерфейсом взаємодії користувача із системою. Вона забезпечує автоматизацію всіх ключових етапів аналізу, включаючи завантаження конфігурації, запуск аналізаторів і формування результатів у зручному форматі. Процес роботи CLI-команди побудований на основі чіткої послідовності дій, які виконуються автоматично після введення команди.

Щоб почати працювати над проектом, необхідно скомпілювати виконуваний файл. Після цього його можна буде використовувати повторно без необхідності повторної компіляції. Для цього потрібно виконати команду вказану на рисунку 4.4.

Якщо інструмент аналізу відсутній у середовищі, система автоматично виконує його встановлення, використовуючи команду з параметра InstallCommand. Це гарантує готовність кожного аналізатора до роботи без необхідності ручного втручання.

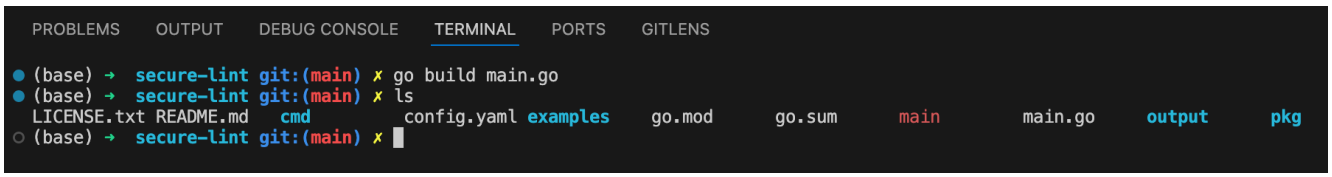


Рисунок 4.4 – Компіляція проекту у виконуючий бінарний файл

Під час виконання аналізу система логує кожен виклик, включаючи статус виконання, вивід та можливі помилки. Це дозволяє користувачеві відстежувати процес у режимі реального часу.

Після завершення роботи кожного аналізатора результати обробляються та зберігаються у файли, вказані у конфігурації. Вивід містить інформацію про знайдені вразливості, включаючи їх рівень критичності, місце у коді та рекомендації щодо виправлення. Формат результатів може варіюватися залежно від вимог користувача: текстові файли, JSON або інші формати, визначені у конфігурації.

CLI-команда забезпечує зручний спосіб взаємодії із системою, дозволяючи користувачеві автоматизувати процес аналізу без складних налаштувань. Інтуїтивно зрозуміла структура команди та автоматизація всіх етапів забезпечують простоту використання та високу продуктивність, роблячи систему ефективним інструментом для оцінювання безпеки програмного забезпечення.

На рисунок 4.5 демонструється команда, яка вводиться в командному рядку разом із її параметрами: --path, --lang та --help. Параметр --path призначений для вказання точного шляху до файлів, які необхідно піддати скануванню, забезпечуючи таким чином точне визначення місця розташування цільових файлів у файловій системі. Параметр --lang дозволяє користувачеві

визначити конкретні мови програмування, які мають бути враховані під час процесу сканування; у випадку, якщо жодна мова не вказана, команда автоматично ініціює сканування для всіх попередньо налаштованих мов програмування, забезпечуючи таким чином широку охопленість та універсальність процесу аналізу. Параметр --help слугує для виведення детальної інформації щодо використання команди та її опцій, що значно сприяє кращому розумінню користувачами доступних можливостей та правильній конфігурації процесу сканування відповідно до їхніх потреб. Така структура команди забезпечує високу ступінь гнучкості та зручності у використанні, дозволяючи адаптувати процес сканування під конкретні вимоги та специфічні потреби користувача, що, в свою чергу, підвищує ефективність та точність виконуваних завдань.

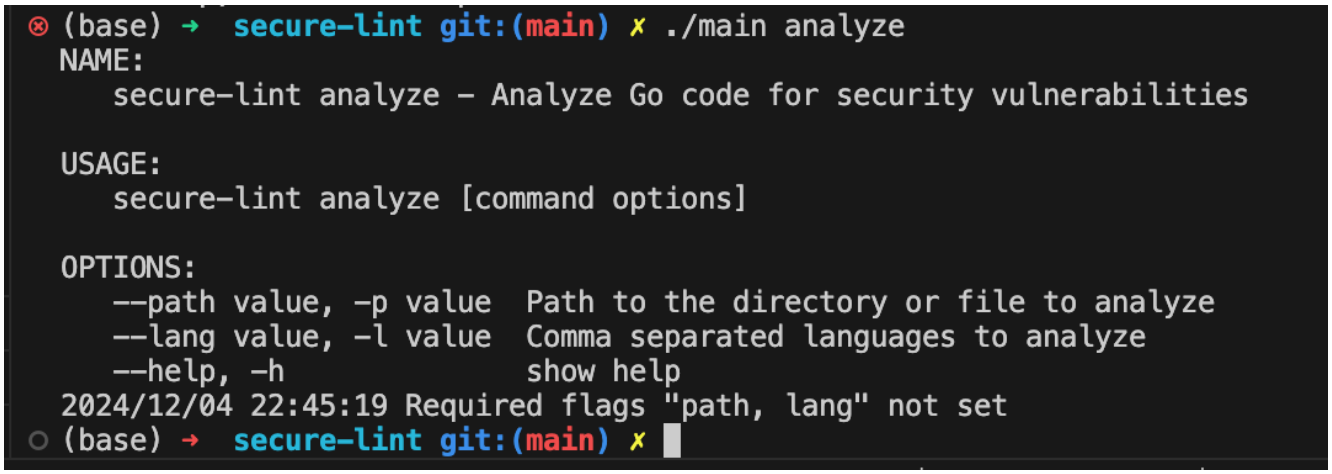


Рисунок 4.5 – Відображення параметрів команди в консолі

Для максимальної ефективності демонстрації функціональних можливостей створеного програмного забезпечення необхідно заздалегідь підготувати тестовий код, який умисно містить певні типові вразливості. У подальших графічних матеріалах наведено приклади такого програмного коду, застосування якого може суттєво знизити загальний рівень захисту програмного забезпечення, а також створити потенційні слабкі місця у системі. Ці недоліки можуть бути використані кіберзловмисниками для реалізації різноманітних атак чи отримання несанкціонованого доступу до

конфіденційних даних. Таким чином, наведені приклади акцентують увагу на важливості глибокого аналізу, виправлення подібних проблем і дотримання високих стандартів безпеки у процесі розроблення програмного забезпечення.

На рисунку 4.6 представлено приклад функції, яка включає в себе інтегровані облікові дані. Подібний підхід може створити серйозні ризики для безпеки, такі як несанкціонований доступ до ресурсів, труднощі з подальшим оновленням облікових даних, а також порушення загальноприйнятих стандартів інформаційної безпеки. Залишення подібних вразливостей у програмному коді може стати причиною значних проблем, включаючи витік конфіденційної інформації, підвищення вразливості до атак, а також ускладнення процесу підтримки і розвитку програмного забезпечення.

```
13
14  func HardcodedCredentials() {
15      username := "admin"
16      password := "supersecret" // Hardcoded password
17      fmt.Printf("Logging in as %s with password %s\n", username, password)
18  }
19
```

Рисунок 4.6 – Жорстко закодовані облікові дані в функції

На рисунку 4.7 продемонстровано атаку категорії SQL-ін’єкція, за допомогою якої кіберзлочинець здійснює введення шкідливих SQL-інструкцій через уразливі точки вводу даних у програмному забезпеченні. Цей вид атаки дозволяє проводити небажані маніпуляції із запитами до бази даних, що може стати причиною несанкціонованого отримання доступу до конфіденційної інформації, модифікації або повного видалення критично важливих даних, а також загальної компрометації інформаційної системи. Наслідки реалізації SQL-ін’єкції можуть охоплювати масштабні порушення інформаційної безпеки, суттєву втрату довіри кінцевих користувачів та вагомі фінансові збитки, які зазнає організація.

```
19
20  func SqlInjections() {
21      db, err := sql.Open("postgres",
22          "user=admin password=supersecret dbname=test sslmode=disable")
23      if err != nil {
24          panic(err)
25      }
26      defer db.Close()
27
28      userInput := "'; DROP TABLE users; --"
29      query := fmt.Sprintf(
30          "SELECT * FROM users WHERE username='%s'",
31          userInput) // SQL Injection
32      rows, err := db.Query(query)
33      if err != nil {
34          panic(err)
35      }
36      defer rows.Close()
37  }
38
```

Рисунок 4.7 – SQL-ін’єкція в функції

На рисунку 4.8 продемонстровано приклад використання застарілого або недостатньо надійного алгоритму хешування, де застосовується криптографічна хеш-функція, яка не забезпечує адекватного рівня захисту даних.

```
38
39  func WeakCryptography() {
40      data := []byte("sensitive data")
41      hash := md5.Sum(data) // Weak cryptographic hash
42      fmt.Printf("MD5 hash: %x\n", hash)
43  }
44
```

Рисунок 4.8 – Використання md5 хешування

Слабкі хеш-функції є вразливими до численних видів атак, зокрема грубого перебору (brute force) або експлуатації попередньо обчислених таблиць (rainbow tables), що дає змогу зловмисникам успішно відновити початкові дані з

отриманих хешів. Наслідки таких уразливостей включають компрометацію облікових даних користувачів, несанкціонований доступ до захищеної інформації, а також потенційну загрозу цілісності інформаційної системи. Використання ненадійних алгоритмів хешування значно збільшує ризик витоку даних, підриває довіру користувачів до безпеки системи та може призвести до масштабних фінансових втрат і серйозного репутаційного збитку для організації.

На рисунку 4.9 продемонстровано використання параметра InsecureSkipVerify, який створює критичну вразливість, дозволяючи встановлювати незахищені або ненадійно зашифровані з'єднання.

```
44
45  func InsecureTlsConfiguration() {
46      tr := &http.Transport{
47          TLSClientConfig: &tls.Config{InsecureSkipVerify: true},
48      }
49      client := &http.Client{Transport: tr}
50      resp, err := client.Get("https://example.com")
51      if err != nil {
52          panic(err)
53      }
54      defer resp.Body.Close()
55  }
56
```

Рисунок 4.9 – Використання InsecureSkipVerify в функції

Параметр InsecureSkipVerify вимикає перевірку серверних сертифікатів під час встановлення TLS/SSL з'єднання, що дозволяє підключатися до серверів із недійсними, самопідписаними або підробленими сертифікатами. Це призводить до серйозних ризиків, таких як атаки «людина посередині» (Man-in-the-Middle), коли зловмисники можуть перехоплювати, модифікувати або підслуховувати передані дані. У результаті конфіденційна інформація, зокрема паролі, особисті дані або фінансові транзакції, стає вразливою до

компрометації.

Застосування InsecureSkipVerify значно знижує рівень захищеності з’єднання, створюючи критичні прогалини у безпеці. Це не лише підриває довіру користувачів до системи, але й відкриває можливості для зловмисників отримати несанкціонований доступ, здійснювати викрадення даних або провокувати серйозні збитки, як фінансового, так і репутаційного характеру.

Ми визначили необхідну команду, а також способи її виконання у різних середовищах. Крім того, було встановлено програмний код, який підлягає детальному аналізу та перевірці. На цьому етапі проект переходить до тестувальної фази, яка включає проведення всебічних випробувань, ретельну оцінку ефективності розробленого рішення, а також виявлення потенційних недоліків. Тестування дозволить визначити, наскільки функціональність програми відповідає поставленим вимогам, забезпечити її надійність і стійкість до можливих загроз, а також оптимізувати роботу у випадку виявлених проблем.

4.3 Тестування та оцінка ефективності системи оцінювання інформаційної безпеки програмного забезпечення

Для виконання тестування файли, що містять код із навмисно створеними вразливостями, були збережені з відповідними розширеннями, які відповідають обраним мовам програмування або форматам даних. Усі ці файли були структуровано організовані та розміщені у спеціальній директорії з назвою "examples". Такий підхід дозволяє зручно управляти тестовими файлами, сприяє легкому доступу до потрібних прикладів під час аналізу й тестування, а також забезпечує швидке налаштування середовища для перевірки. Організація файлів у єдиному місці значно полегшує проведення випробувань, підвищуючи ефективність і точність роботи з тестовими сценаріями.

На рисунку 4.10 зображено процес виконання спеціалізованої команди

для ініціалізації тестування, а також відображено результати її виконання. У результатах представлено детальний список виявлених вразливостей, включаючи їх категорії, опис і типи, а також оцінку можливого впливу на безпеку програмної системи. Такий підхід дозволяє не лише побачити загальний стан безпеки аналізованого коду, але й забезпечує розуміння критичності знайдених проблем, що сприяє пріоритизації виправлення. Демонстрація результатів тестів є важливою складовою для забезпечення наочності, полегшення аналізу та прийняття відповідних заходів для усунення вразливостей.

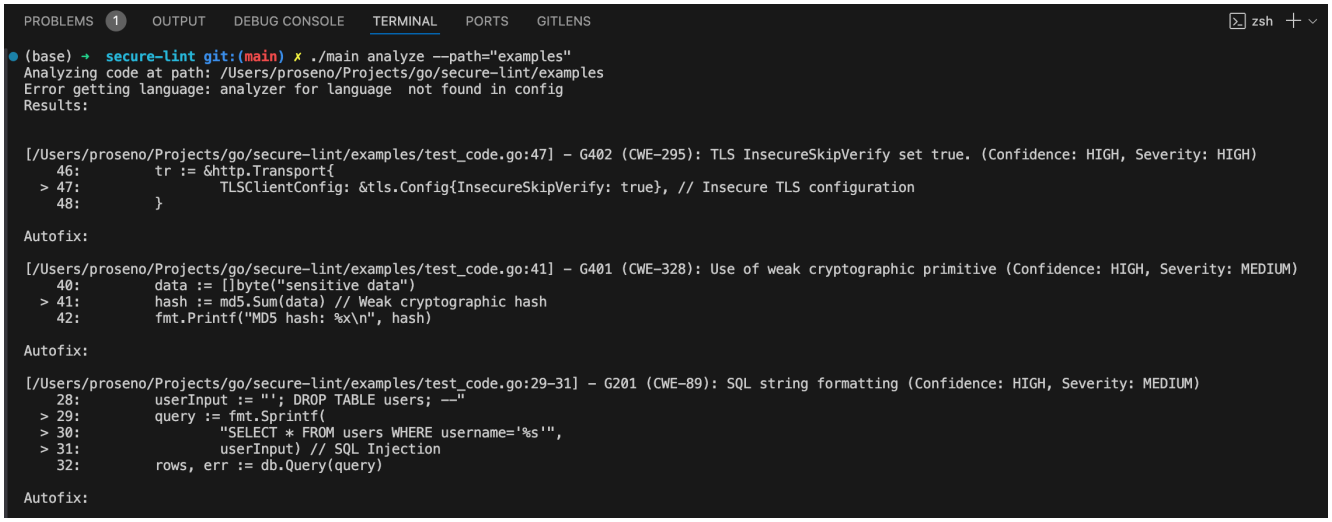


Рисунок 4.10 – Результат виконання команди для мови

Додатково команда надає другу частину звіту, яка містить узагальнений підсумок проведеного аналізу, показаний на рисунку 4.11. У цьому підсумку зазначено такі ключові дані:

– час виконання: загальна тривалість всього процесу тестування, що дозволяє точно оцінити ефективність та швидкість виконання аналізу, визначити можливі затримки та оптимізувати час на проведення подальших етапів;

– кількість помилок: загальна кількість виявлених вразливостей або помилок, що надає повне уявлення про масштаб потенційних ризиків у системі, їхній розподіл за типами та важливість для загальної безпеки програмного

забезпечення;

– розмежування помилок за рівнями: детальна класифікація помилок за їхньою критичністю, що допомагає визначити пріоритетність виправлень, забезпечуючи ефективне управління ресурсами та фокусування на найважливіших аспектах безпеки системи.

Ці ключові дані дозволяють команді розробників та фахівцям з безпеки більш обґрунтовано приймати рішення щодо необхідних заходів для покращення якості та безпеки програмного забезпечення, а також ефективно планувати подальші кроки в процесі розвитку проекту.

Таким чином, розширення тексту дозволяє більш детально описати кожен пункт, додаючи додаткові пояснення та контекст, що сприяє кращому розумінню ключових даних та їхнього значення для процесу аналізу безпеки програмного забезпечення.

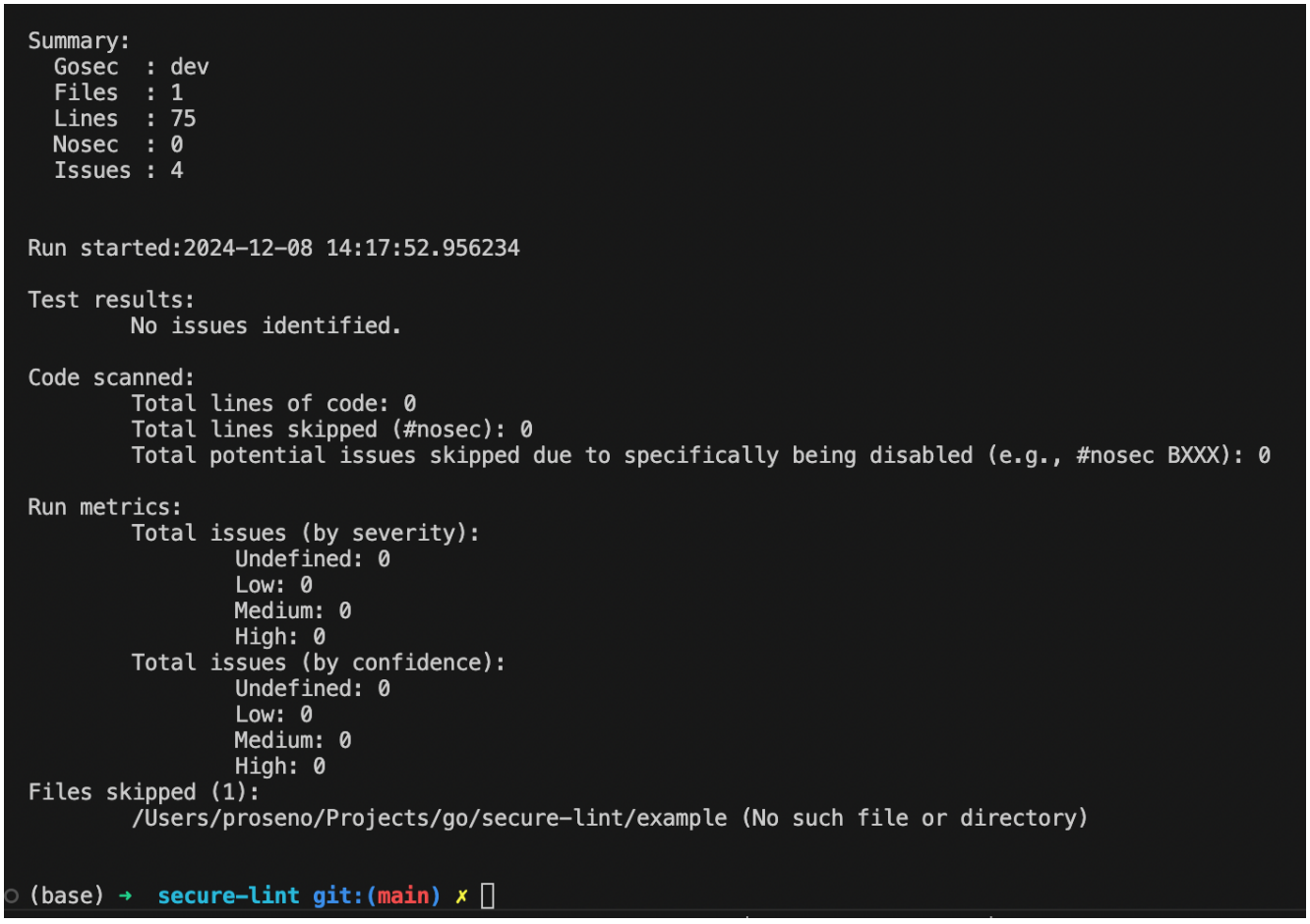


Рисунок 4.11 – Підсумок аналізу коду

За результатами звіту можна чітко побачити, що команда успішно виявила вразливості, описані в попередньому розділі. У звіті детально наведено номер рядка коду, в якому було виявлено потенційну вразливість. Крім того, надано сам код вразливості, її назву та детальний опис, що дозволяє глибше зрозуміти природу кожної проблеми.

Особливо важливою складовою звіту є детальна оцінка критичності кожної виявленої вразливості у програмному забезпеченні. Цей аспект дозволяє точно визначити пріоритетність заходів щодо їх усунення, ефективно зосереджуючи зусилля на найбільш небезпечних та потенційно шкідливих недоліках системи. Оцінка критичності враховує такі ключові фактори, як ймовірність використання вразливості зловмисниками, потенційний вплив цієї вразливості на загальну функціональність та безпеку системи, а також складність її усунення. Завдяки цьому підходу, команда розробників може більш обґрунтовано планувати свої дії, спрямовуючи ресурси на найбільш суттєві загрози та забезпечуючи тим самим більш високий рівень захисту інформаційних систем. Крім того, така оцінка допомагає вчасно виявляти критичні недоліки, що сприяє зменшенню ризиків у майбутньому та підвищенню загальної стійкості програмного забезпечення до потенційних атак.

Крім того, звіт містить детальні рекомендації щодо усунення кожної з виявлених вразливостей у програмному забезпеченні. Ці рекомендації включають конкретні пропозиції щодо модифікації коду, впровадження додаткових заходів безпеки, а також оптимізації процесів розробки. Такий всебічний підхід не лише допомагає ефективно виправити поточні недоліки та вразливості, але й сприяє запобіганню виникненню подібних проблем у майбутньому. Крім того, впровадження цих рекомендацій сприяє покращенню загальної якості програмного забезпечення, забезпечуючи більш безпечне та стабільне функціонування інформаційних систем у довгостроковій перспективі.

Додатково, у звіті представлено аналіз тенденцій та патернів, що дозволяє

виявити системні проблеми в архітектурі або методах розробки програмного забезпечення. Це забезпечує можливість впровадження стратегічних змін, спрямованих на підвищення загального рівня безпеки та надійності системи.

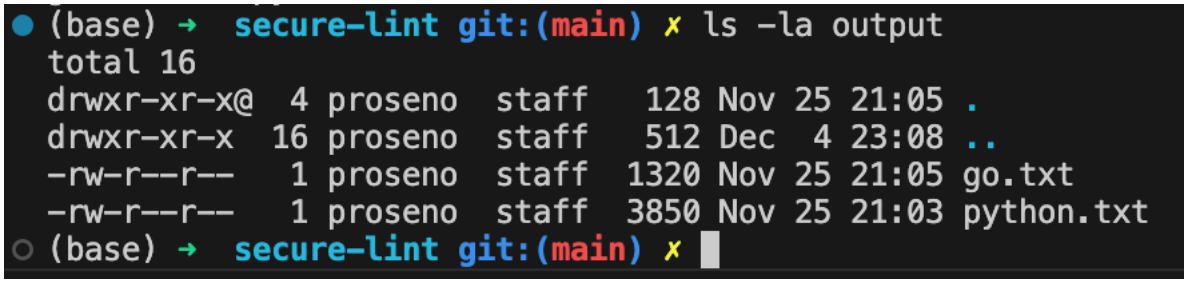


Рисунок 4.12 – Звіти збережені в файли

За замовчуванням усі згенеровані звіти автоматично зберігаються у спеціально створеній директорії під назвою "output", зображено на рисунку 4.12. Це рішення сприяє ефективному впорядкуванню та організації результатів тестування, що значно полегшує їх подальший аналіз і перегляд. Зберігання звітів у єдиній папці дозволяє уникнути хаотичного розміщення файлів у різних місцях, забезпечуючи швидкий і зручний доступ до необхідної інформації для всіх членів команди. Такий підхід також полегшує архівацію результатів тестувань, їх передачу або використання у майбутніх проектах, сприяючи підвищенню продуктивності та організованості робочого процесу.

4.4 Висновок

За результатами роботи над розділом, згідно з метою кваліфікаційної роботи, було виконано наступні завдання:

– розроблено прототип програми для аналізу коду на вразливості.

– створено тестовий код із вразливостями для демонстрації можливостей програми.

– проведено тестування роботи програми, результати якого підтвердили її здатність виявляти заздалегідь підготовлені вразливості.

– описано алгоритм роботи програми та детально розглянуто результати її функціонування.

– проаналізовано можливості подальшого вдосконалення прототипу, зокрема розширення функціоналу та оптимізації.

Розроблений прототип є основою для створення більш складних інструментів оцінки безпеки програмного забезпечення, які здатні ефективно реагувати на сучасні виклики у сфері кібербезпеки. Проте необхідно враховувати потенційні обмеження, зокрема залежність від якості початкових даних для аналізу, що вимагає регулярного оновлення та тестування інструменту.

Отримані результати підтверджують актуальність і важливість досліджуваного напрямку, а також необхідність продовження роботи у цьому полі для забезпечення стійкості та безпеки інформаційних систем.

ВИСНОВКИ

Моя робота присвячена розробці методу оцінювання інформаційної безпеки безпечного функціонування програмного забезпечення. У рамках роботи було проаналізовано сучасні підходи до статичного та динамічного аналізу коду, а також перевірки залежностей, і на їх основі створено інтегровану систему автоматизованого аналізу. Актуальність теми зумовлена зростаючими вимогами до безпеки програмного забезпечення, особливо в умовах постійних кіберзагроз і ризиків, пов’язаних із використанням зовнішніх залежностей.

– розроблено метод оцінки безпеки, яка об'єднує статичний і динамічний аналіз для забезпечення комплексного підходу;

– створено CLI-інструмент, який автоматизує процес перевірки коду та залежностей, а також формує стандартизовані звіти з результатами;

– проведено тестування інструменту на прикладах коду, що показало його ефективність у виявленні вразливостей;

– оцінено переваги та недоліки запропонованого підходу в порівнянні з існуючими методами.

Запропонований метод кардинально спрощує традиційний процес оцінювання безпеки програмного забезпечення, роблячи його більш системним, послідовним і автоматизованим. Завдяки використанню різноманітних інструментів статичного аналізу, упорядкованих за допомогою модульної архітектури, розробники та спеціалісти з безпеки отримують змогу оперативно виявляти критичні вразливості, оптимізувати робочі процеси й уникати зайвих витрат на усунення проблем на пізніх стадіях розвитку проєкту.

Подальший розвиток системи передбачає розширення її можливостей шляхом інтеграції нових, більш спеціалізованих аналізаторів, адаптації до дедалі ширшого спектра мов програмування й платформ, а також глибшої автоматизації процесів перевірки безпеки у CI/CD-конвеєрах. Такий підхід дасть змогу постійно вдосконалювати якість і надійність програмних продуктів,

відповідати зростаючим вимогам сучасних ринків і ефективно протидіяти новим загрозам, що виникають у результаті еволюції інформаційних технологій та кібернетичної криміналістики.

Отримані результати мають універсальну цінність для широкого кола фахівців, які беруть участь у розробленні та забезпеченні безпеки програмного забезпечення. До них належать розробники, тестувальники, DevOps-інженери, а також спеціалісти з кібербезпеки. Інструмент, створений у ході цієї роботи, легко інтегрується в процес розробки та може використовуватися як постійний моніторинговий механізм безпеки. Це дає змогу оперативно виявляти потенційні загрози й усувати їх на ранніх етапах, тим самим суттєво знижуючи ризик експлуатації вразливостей у майбутньому.

Запропоновані підходи й розроблений метод роблять вагомий внесок у вдосконалення автоматизованих систем аналізу безпеки програмного забезпечення. Ця робота пропонує сучасне й перспективне рішення актуальних викликів, пов’язаних зі зростаючою складністю програмних проєктів. Завдяки поєднанню гнучкої архітектури, модульності та інтегрованості з існуючими інструментами, запропонований метод дозволяє підтримувати високі стандарти якості та безпеки, а також забезпечує платформу для подальшого масштабування, розширення та адаптації до нових умов розробки.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Доктрина інформаційної безпеки України, затвердженої Указом Президента України від 25 лютого 2017 року № №47/2017, 15с.

2. Державний стандарт України Захист інформації. Технічний захист інформації. Основні положення. ДСТУ 3396.0-96 [Електронний ресурс]. – Режим доступу: http://www.dsszzi.gov.ua/dsszzi/control/uk/publish/article?art_id=38883&cat_id =38836.

3. Закон України «Про основні засади забезпечення кібербезпеки України» зі змінами. Відомості Верховної Ради (ВВР), 2017, № 45, ст.403, зі змінами від 28.07.2022 року. Режим доступу: https://zakon.rada.gov.ua/laws/show/2163.

4. Цибульник С. О. Технології розроблення програмного забезпечення. Частина 1. Життєвий цикл програмного забезпечення : підручник / С. О. Цибульник, К. С. Барандич. – К. : КПІ ім. Ігоря Сікорського, 2022.– 270 с.

5. Лисенко С.М. Аналіз методів виявлення шкідливого програмного забезпечення в компютерних системах. / С.М. Лисенко, Р.В. Щука - Вісник Хмельницького національно-го університету. 2020 № 2. (283). С. 101–107.

6. Ленков С.В. Метод прогнозування вразливостей інформаційної безпеки на основі аналізу даних тематичних інтернет-ресурсів / С.В. Лєнков, В.М. Джулій, А.М. Берназ, І.В. Муляр, І.В. Пампуха // Збірник наукових праць Військового інституту Київського національного університету імені Тараса Шевченка. – К.: ВІКНУ, 2023. – Вип. №78. – С. 123-134.

7. Ленков С.В. Метод протидії поширенню та виявлення шкідливої інформації в соціальних мережах/ С.В. Ленков, В.М. Джулій, Л.В. Солодєєва // Збірник наукових праць Військового інституту Київського національного університету імені Тараса Шевченка. – К.: ВІКНУ, 2022. – Вип. №77. – С. 103-117.

8. Ленков С.В. Модель безпеки поширення забороненої інформації в інформаційно-телекомунікаційних мережах / С.В. Ленков, В.М. Джулій, В.С. Орленко, О.В. Сєлюков, А.В. Атаманюк // Збірник наукових праць Військового інституту Київського національного університету імені Тараса Шевченка. – К.: ВІКНУ, 2020. – Вип. №68. – С. 53-64.

9. Крепич С. Я. Якість програмного забезпечення та тестування : базовий курс / С. Я. Крепич, Співак І. Я. – Тернопіль : ФОП Паляниця В. А., 2020. – 478 с.

10. Якість та тестування інформаційних систем : Навчальний посібник для самостійної роботи студентів вищих навчальних закладів / Золотухіна О. А., Негоденко О. В., Резник С. Ю., Разіна С. Я. – Київ : ННІІТ ДУТ, 2020. – 128 с.

11. Вишня В. Б. Основи інформаційної безпеки: навч. посібник / В. Б. Вишня, О. С. Гавриш, Е. В. Рижков. Дніпро : Дніпроп. держ. ун-т внутріш. справ, 2020. – 128 с.

12. Соціальні мережі – реальні загрози віртуального світу. [Електронний ресурс]. – Режим доступу : http://ogo.ua/ articles/view/011–02–23/26490.htm.

13. Остапов С.Е. Технології захисту інформації: навчальний посібник / С.Е. Остапов, С.П. Євсеєв, О.Г. Король – Харків : Вид-во ХНЕУ, 2016. – 476 с.

14. Лєнков, С.В. Аналіз існуючих методів та алгоритмів виявлення атак в бездротових мережах передачі даних / С.В. Лєнков, В.М. Джулій, Н.М. Берназ, С.О. Божук // Збірник наукових праць Військового інституту Київського національного університету імені Тараса Шевченка. – К.: ВІКНУ, 2017. – Вип. № 56. – С.124-132.

15. Бурячок В.Л. Інформаційний та кіберпростори: проблеми безпеки, методи та засоби боротьби : посібник / [В. Л. Бурячок, С. В. Толюпа, В. В. Семко та ін.]. – К.: ДУТ-КНУ, 2016. – 178 с.

16. Рибальченко Л.В., Косиченко О.О. Проблеми безпеки персональних даних в Україні / Регіональна економіка / Запоріжжя. 2019. – с.57-62.

17. Джулій В.М. Метод класифікації додатків трафіка комп'ютерних мереж на основі машинного навчання в умовах невизначеності / В.М. Джулій, О.В. Мірошніченко, Л.В. Солодєєва // Збірник наукових праць Військового інституту Київського національного університету імені Тараса Шевченка. – К.: ВІКНУ, 2022. – Вип. №74. – С. 73-82.

18. Лавров Є.А. Математичні методи дослідження операцій: підручник / Є. А. Лавров, Л. П. Перхун, В. В. Шендрик – Суми: Сумський державний університет, 2017. – 212 с.

19. Гончар С.Ф. Оцінювання ризиків кібербезпеки інформаційних систем об'єктів критичної інфраструктури: монографія. / С. Ф. Гончар. – Київ, 2019. – 175 с.

20. Yemchuk L. Organizational Network Analysis as a Tool for Leadership Assessment in Software Development Team. Zhylinska O.; Chornyi A.; Dzhuliy V. – Institute of Electrical and Electronics Engineers (30 September 2020); INSPEC Accession Number: 20008165; DOI: 10.1109/ACIT49673.2020.

21. Сигнатура атаки. Wikipedia [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Сигнатура_атаки.

22. Яворський О.В. Модель інформаційної безпеки функціонування програмного забезпечення / О.В. Яворський , В.М. Джулій, С.В. Ленков, // SMART TECHNOLOGIES: Industrial and Civil Engineering, Issue 2(15), 2024, 31-45.

23. Фармагей О., Мельник Д., Петрик В., Карпович О., Остроухов В., Присяжнюк М., Чеховська М. Інформаційна безпека: навчальний посібник. – Київ: НАУ, 2018. – 320 с.

24. Моделі та методика оцінки інформаційної безпеки програмного забезпечення. Вісник Кременчуцького національного університету. – 2024. – № 3(79). – С. 45–55.

25. Метод оцінювання безпеки застосування програмних засобів. [Електронний ресурс] / [уклад.] В. С. Іванов. – Режим доступу: https://www.academia.edu.

26. Бурячок В. Л., Семко В. В. Кіберзахист інформаційних мереж:

методи та алгоритми: посібник. – Київ: ДУТ, 2019. – 198 с.

27. Моделі оцінювання безпеки інформаційних систем // Вісник Державного університету телекомунікацій. – 2022. – №2. – С. 52–63.

28. Фармагей О., Присяжнюк М., Остроухов В. Основи кібербезпеки: навчальний посібник. – Київ: НАУ, 2020. – 340 с.

29. Гончар С. Ф. Комплексна оцінка кіберзагроз в інформаційних системах: монографія. – Київ: КНУТД, 2021. – 210 с.

30. Anderson, R. Security Engineering: A Guide to Building Dependable Distributed Systems. – 3rd ed. – Wiley, 2020. – 1232 p.

31. Schneier, B. Click Here to Kill Everybody: Security and Survival in a Hyper-connected World. – W.W. Norton & Company, 2018. – 288 p.

32. Ross, S. J. Cybersecurity Threats, Malware Trends and Strategies. – Springer, 2020. – 225 p.

33. Stallings, W. Effective Cybersecurity: A Guide to Using Best Practices and Standards. – Addison-Wesley Professional, 2018. – 800 p.

34. Pfleeger, C. P., Pfleeger, S. L., Margulies, J. Security in Computing. – 5th ed. – Prentice Hall, 2018. – 944 p.

35. Мартін Р. Чистий код. Створення і рефакторинг за допомогою Agile / пер. з англ. І. Бондар-Терещенко. – Харків: Фабула, 2019. – 448 с.

36. Хамбл Дж., Фарлі Д. Неперервне постачання: автоматизація збірки, тестування та впровадження програмного забезпечення / пер. з англ. О. Коваленко. – Київ: Видавництво Жупанського, 2018. – 616 с.

37. Ван Мероде Г. Безперервна інтеграція (CI) та безперервна доставка (CD): практичний посібник з проєктування та розробки конвеєрів / пер. з англ. О. Сидоренко. – Київ: Yakaboo Publishing, 2020. – 320 с.

38. Шербінін О. DevOps: посібник для початківців. – Київ: Видавництво Жупанського, 2019. – 240 с.

39. Керріган Р. Аналіз коду: методи та інструменти / пер. з англ. М. Іваненко. – Львів: Видавництво Старого Лева, 2021. – 384 с.

40. NIX Solutions. CI/CD: що це, як пов’язано з DevOps, переваги, найкращі практики [Електронний ресурс]. – Режим доступу: https://www.nixsolutions.com/ua/blog/for-developer/ci-cd-shho-cze-yak-povyazano-z-devops-perevagy-najkrashhi-praktyky/.

41. PeerDH. Впровадження інструментів статичного аналізу коду в CI/CD процеси [Електронний ресурс]. – Режим доступу: https://peerdh.com/uk/blogs/programming-insights/implementing-static-code-analysis-tools-in-ci-cd-workflows-1.

42. Zomro. Знайомство з CI/CD: Аналіз Методології Розробки [Електронний ресурс]. – Режим доступу: https://zomro.com/ua/blog/faq/437-znakomstvo-s-cicd-analiz-metodologii-razrabotki.

43. DOU. CI vs CD vs CD: різниця підходів і чому вони важливі [Електронний ресурс]. – Режим доступу: https://dou.ua/forums/topic/47081/.

44. Highload.today. Що таке CI/CD, як працює та коли він знадобиться на проєкті [Електронний ресурс]. – Режим доступу: https://highload.today/uk/blogs/shho-take-ci-cd-yak-vin-pratsyuye-ta-koli-

znadobitsya-na-proyekti-lajfhaki-ta-bad-practices/.

45. Kim, G., Humble, J., Debois, P., Willis, J. The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations. – IT Revolution Press, 2021. – 480 p.

46. Ford, N., Parsons, P., Kua, P. Building Evolutionary Architectures: Support Constant Change. – O'Reilly Media, 2020. – 190 p.

47. Sadowski, C., Zimmermann, T. Rethinking Productivity in Software Engineering. – Apress, 2019. – 310 p.

48. Rahman, F., Devanbu, P. "How, and why, process metrics are better". – Proceedings of the 2018 International Conference on Software Engineering (ICSE).

49. Zeller, A., Bird, C., Zimmermann, T. "The Future of Static Code Analysis". – ACM Queue Journal, 2020.

50. Флорида, Р. Інноваційна екосистема: роль технологій у розвитку сучасного бізнесу. – Львів: Видавництво Львівської політехніки, 2019. – 320 с.

51. Семко, В. В., Толупа, С. В. Комп'ютерні системи та мережі: безпека, захист інформації. – Київ: НТУУ "КПІ", 2018. – 458 с.

52. Кучма, О. Ю., Левченко, О. В. Автоматизація розробки та тестування програмного забезпечення. – Харків: ХНУРЕ, 2020. – 280 с.

53. White, M., Torkura, K. Advanced Concepts in Security Automation. – Springer, 2021. – 412 p.

54. Шиманович, І. Г., Олійник, Т. В. Розробка інформаційних систем: безпека програмного забезпечення. – Вінниця: ВНТУ, 2019. – 310 с.

55. Іваненко О. В., Коваленко А. С. Безпека інформаційних систем: теорія та практика. – Київ: Видавництво НТУУ "КПІ", 2019. – 280 с.

56. Сидоренко Л. М. Автоматизовані системи контролю якості коду. – Львів: Видавництво Львівської політехніки, 2021. – 350 с.

57. Петрова Т. Ю., Григоренко В. В. Інформаційна безпека в сучасних мережах: методи захисту та аналіз загроз. – Харків: ХНУРЕ, 2020. – 300 с.

58. Бойко С. І. Інформаційна безпека програмного забезпечення: сучасні підходи та інструменти. – Дніпро: Видавництво ДНУ, 2018. – 250 с.

59. Taylor, R., Smith, J. Continuous Integration and Continuous Deployment with DevOps. – Packt Publishing, 2019. – 420 p.

60. Johnson, L., Miller, K. Automated Code Analysis for Enhanced Security. – Springer, 2020. – 350 p.

61. Морозенко О. В., Савченко І. П. Інтеграція CI/CD в процес розробки програмного забезпечення: методи та інструменти. – Київ: Видавництво Київського національного університету імені Тараса Шевченка, 2022. – 330 с.

ДОДАТОК А
(обов'язковий)
Фрагмент коду

```
package main

import (
	"log"
	"os"
	"secure-lint/pkg/config"
	"secure-lint/pkg/models"

	"github.com/urfave/cli/v2"
	"secure-lint/cmd"
)

func initProjectRoot() {
	projectRoot, err := os.Getwd()
	if err != nil {
		log.Fatal(err)
	}
	models.ProjectRoot = projectRoot
}

func main() {
	configData, _ := config.LoadConfig("config.yaml")
	initProjectRoot()
	app := &cli.App{
		Name:  "secure-lint",
		Usage: "A linting tool to analyze Go code for security vulnerabilities.",
		Commands: []*cli.Command{
			cmd.AnalyzeCommand(configData),
		},
	}

	if err := app.Run(os.Args); err != nil {
		log.Fatal(err)
	}
}

package analyzer

import (
   "fmt"
   "io/ioutil"
   "os"
   "secure-lint/pkg/config"
   "secure-lint/pkg/models"
)

type Issue struct {
   Description string
   Severity    string
}

func gatherReport() string {
   outputDirectory := models.ProjectRoot + "/output/"
   files, err := os.ReadDir(outputDirectory)
   if err != nil {
      fmt.Printf("Error reading directory: %v\n", err)
      return ""
```

```
    }

    var content string
    for _, file := range files {
        if file.IsDir() {
            fmt.Printf("Skipping directory: %s\n", file.Name())
            continue
        }

        filePath := outputDirectory + file.Name()

        fileContent, err := ioutil.ReadFile(filePath)
        if err != nil {
            fmt.Printf("Error reading file %s: %v\n", file.Name(), err)
            continue
        }
        content += string(fileContent) + "\n"
    }
    return content
}

func outputReport() {
    fmt.Println(gatherReport())
}

func runAnalyzer(analyzer *models.Analyzer, path string) {
    if analyzer.CheckExecutable() {
        if analyzer.Stdout != "" {
            result := analyzer.Analyze(path)
            fmt.Println(result)
        }
        analyzer.Analyze(path)
    }
}

func AnalyzeCode(path string, config *config.Config, langs []string) {
    if len(langs) == 0 {
        for _, analyzer := range config.Analyzers {
            runAnalyzer(&analyzer, path)
        }
    } else {
        for _, lang := range langs {
            analyzer, err := config.GetByLang(lang)
            if err != nil {
                fmt.Printf("Error getting language: %v\n", err)
                continue
            }

            runAnalyzer(analyzer, path)
        }
    }

    outputReport()
}

package cmd

import (
    "fmt"
    "secure-lint/pkg/analyzer"
    "secure-lint/pkg/config"
    "secure-lint/pkg/models"
    "strings"
```

```
    "github.com/urfave/cli/v2"
)

func getPath(c *cli.Context) string {
    return c.String("path")
}

func getLangs(c *cli.Context) []string {
    langString := c.String("lang")
    return strings.Split(langString, ",")
}

func AnalyzeCommand(config *config.Config) *cli.Command {
    return &cli.Command{
        Name:  "analyze",
        Usage: "Analyze Go code for security vulnerabilities",
        Flags: []cli.Flag{
            &cli.StringFlag{
                Name:     "path",
                Aliases:  []string{"p"},
                Usage:    "Path to the directory or file to analyze",
                Required: true,
            },
            &cli.StringFlag{
                Name:     "lang",
                Aliases:  []string{"l"},
                Usage:    "Comma separated languages to analyze",
                Required: false,
            },
        },
        Action: func(c *cli.Context) error {
            path := getPath(c)
            langs := getLangs(c)
            path = models.ProjectRoot + "/" + path
            fmt.Printf("Analyzing code at path: %s\n", path)
            analyzer.AnalyzeCode(path, config, langs)
            return nil
        },
    }
}

package config

import (
    "fmt"
    "secure-lint/pkg/models"

    "gopkg.in/yaml.v3"

    "os"
)

type Config struct {
    Analyzers []models.Analyzer `yaml:"analyzers"`
}

func LoadConfig(filePath string) (*Config, error) {
    file, err := os.Open(filePath)
    if err != nil {
        return nil, fmt.Errorf("failed to open file: %w", err)
    }
    defer file.Close()
```

```
    var config Config
    decoder := yaml.NewDecoder(file)
    if err := decoder.Decode(&config); err != nil {
        return nil, fmt.Errorf("failed to parse YAML: %w", err)
    }

    return &config, nil
}

func (c *Config) GetByLang(lang string) (*models.Analyzer, error) {
    for _, analyzer := range c.Analyzers {
        if analyzer.As == lang {
            return &analyzer, nil
        }
    }
    return nil, fmt.Errorf("analyzer for language %s not found in config", lang)
}
```

```
package report

import (
    "encoding/json"
    "os"
    "secure-lint/pkg/analyzer"
)

func GenerateJSONReport(issues []analyzer.Issue, outputPath string) error {
    file, err := os.Create(outputPath)
    if err != nil {
        return err
    }
    defer file.Close()

    encoder := json.NewEncoder(file)
    return encoder.Encode(issues)
}
```

```
package examples

import (
    "crypto/md5"
    "crypto/tls"
    "database/sql"
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
    "os/exec"
)

func HardcodedCredentials() {
    username := "admin"
    password := "supersecret" // Hardcoded password
    fmt.Printf("Logging in as %s with password %s\n", username, password)
}

func SqlInjections() {
    db, err := sql.Open("postgres",
        "user=admin password=supersecret dbname=test sslmode=disable")
    if err != nil {
        panic(err)
    }
```

```
    defer db.Close()

    userInput := "'; DROP TABLE users; --"
    query := fmt.Sprintf(
        "SELECT * FROM users WHERE username='%s'",
        userInput) // SQL Injection
    rows, err := db.Query(query)
    if err != nil {
        panic(err)
    }
    defer rows.Close()
}

func WeakCryptography() {
    data := []byte("sensitive data")
    hash := md5.Sum(data) // Weak cryptographic hash
    fmt.Printf("MD5 hash: %x\n", hash)
}

func InsecureTlsConfiguration() {
    tr := &http.Transport{
        TLSClientConfig: &tls.Config{InsecureSkipVerify: true}, // Insecure TLS configuration
    }
    client := &http.Client{Transport: tr}
    resp, err := client.Get("https://example.com")
    if err != nil {
        panic(err)
    }
    defer resp.Body.Close()
}

func FileInclusionVulnerability() {
    userInput := "../../etc/passwd" // Arbitrary file inclusion
    data, err := ioutil.ReadFile(userInput)
    if err != nil {
        fmt.Println("Error reading file:", err)
        return
    }
    fmt.Println("File contents:", string(data))
}

func CommandInjection() {
    userInput := "ls; rm -rf /" // Command injection
    cmd := exec.Command("sh", "-c", userInput)
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr
    if err := cmd.Run(); err != nil {
        fmt.Println("Command failed:", err)
    }
}
analyzers:
  -
    command: "gosec"
    output_flag: "-out"
    install_command: "go install github.com/securego/gosec/v2/cmd/gosec@latest"
    as: "go"
  -
    command: "bandit"
    flags: "-r"
    output_flag: "--output"
    install_command: "pip install bandit"
    as: "python"
```

ДОДАТОК Б
(обов'язковий)
КОПІЇ НАУКОВИХ ПУБЛІКАЦІЙ

Information technologies

Модель інформаційної безпеки функціонування програмного забезпечення

Сергій Лєнков[1], Володимир Джулій[2], Олександр Яворський[3], Костянтин Зацепін[4]

[1] Військовий інститут Київського національного університету імені Тараса Шевченка
Юлії Здановської, 81, Київ, Україна, 03189
[1] lenkov_s@ukr.net, https://orcid.org/0000-0001-7689-239X
[2,3,4] Хмельницький національний університет
вул. Інститутська, 11, Хмельницький, Україна, 29000
[2,3,4] dzhuliivm@khmnu.edu.ua, http://orcid.org/0000-0003-1878-4301

Анотація. В роботі виконано систематизацію моделей надійного та безпечного функціонування програмного забезпечення. В результаті проведеного дослідження виділено три типи моделей: аналітичні; статистичні; емпіричні [4,9].

Розглянуто ряд найчастіше застосовуваних моделей, виділено їх недоліки та переваги з погляду розв'язуваної задачі опису безпечного функціонування програмного продукту, та розпізнавання шкідливого програмного забезпечення. За результатами проведених досліджень розглянуті моделі мають переваги у плані простоти їх практичної реалізації, проте водночас виділено наступні недоліки: деякі з розглянутих моделей при реалізації вимагають великого об'єму обчислювальних ресурсів – для аналізу безпеки та накопичення архівних даних; використання статистичними та імовірнісними моделями припущень про те, що інтенсивність атак/відмов чи кількість помилок у програмному забезпеченню мають заздалегідь відомий розподіл (біноміальний, стандартний або пуасонівський), що не завжди вірно для реальних процесів і систем; немає поділу на відмови програмного забезпечення і вихід з ладу внаслідок кібератак, не враховуються також вразливості нульового дня; не аналізуються звернення досліджуваного програмного забезпечення до пам'яті, що могло б дати важливу інформацію про його легітимність чи наявність шкідливих функцій; жодна з розглянутих моделей не забезпечує комплексного представлення про процес функціонування програмного забезпечення, у тому числі, аналіз зі сторони інформаційної безпеки відсутній [1,8,16].

Завдання розпізнавання шкідливого програмного забезпечення з кожним роком стає дедалі актуальнішим і складнішим у зв'язку з цифровізацією галузей діяльності людини та використанням програмного забезпечення для виконання бізнес-логіки та технічних процесів у складних системах. Внаслідок цього, чим більший обсяг програмного забезпечення в системі, тим потенційно більше в ньому помилок, при цьому через підключення сучасних систем до мережі Інтернет, програмне забезпечення часто поширюється по мережі, що дозволяє зловмисникам створити нові вектори кібератак на системи [2,3,5,7].

Запропонована модель усуває наведені недоліки за рахунок того, що вона враховує характерні особливості прояву шкідливого програмного забезпечення на пристроях, а саме вплив шкідливого програмного забезпечення на обчислювальні ресурси системи та роботу з оперативною пам'яттю. Це дозволяє розробленій моделі враховувати як надійність функціонування програмного забезпечення, так і безпеку [9,11,19].

У термінах моделі сформульовані критерії безпечного функціонування програмного забезпечення, зроблено висновок про те, що для найбільш ефективної реалізації такої моделі на практиці має бути використаний гіпервізор [14,17,20].

Ключові слова: модель, інформаційна безпека, вразливості, атаки, конфіденційні дані, шкідливе програмне забезпечення, безпечне функціонування.

ВСТУП

Число кібератак на найважливіші галузі інфраструктури у всьому світі неухильно зростає з кожним роком. Так, за даними компанії Trellix, вже в першому кварталі 2023 року в Україні було зафіксовано на 22,5% більше кібератак, ніж у останньому кварталі 2022 року. У другому кварталі кількість кібератак зросла ще на 9%, порівняно з першим кварталом 2023 року [5,7].

Одним із найпоширеніших методів реалізації кібератак є використання шкідливого програмного забезпечення. Так, на початку 2023 року кількість кібератак, реалізованих цим методом, незначно поступилася лише кібератакам методами соціальної інженерії. Частка кібератак з використанням шкідливого програмного забезпечення, за даними Trellix, склала 71% для фізичних осіб та 60% для юридичних осіб [5,7,22].

У зв'язку з цим, актуальність завдання розпізнавання шкідливого програмного забезпечення щорічно зростає. Слід також враховувати, що зловмисники при реалізації кібератак комбінують різні типи шкідливого програмного забезпечення – використовують багатофункціональні трояни або завантажують на скомпрометовані об'єкти інфраструктури безліч різних за функціоналом шкідливих програм. Це значно ускладнює процес аналізу безпеки програмного забезпечення. Особливої гостроти проблема набуває останнім часом у зв'язку з об'єктивною необхідністю організації віддаленої роботи та доступу до інформаційних ресурсів компаній, систем управління та моніторингу технологічних процесів – що потребує вирішення завдання захисту «територіально розподіленого периметра». Зростає небезпека атак, що використовують прийоми як явно вираженої, так і прихованої соціальної інженерії, коли для впровадження шкідливих програм застосовуються довірені джерела [4,6,7,20].

Одним із каналів доставки шкідливих програм можуть виступати комплекти та компоненти легітимного програмного забезпечення. У разі відкритого коду користувачам надається можливість ознайомитися з вихідними текстами, та за необхідності виконати оцінку та аналіз як ефективності продукту, так і його безпеки. У разі програмного забезпечення із закритим кодом вихідні тексти або не надаються взагалі, або надаються частково або повністю державним сертифікаційним органам для винесення рішення про допуск до використання таких продуктів у певних мережах та системах. Більшість ліцензійних угод на використання програмного забезпечення містять повідомлення користувача про обмеженість гарантії та відмову від відповідальності при використанні продукту. Таким чином, програмний продукт без вихідного коду є «чорною» скринькою для користувача, і навіть у разі авторства відомого та сумлінного розробника та постачальника може містити: недокументовані можливості; шкідливі компоненти, впроваджені засоби розробки внаслідок атак на сховища; шкідливе програмне забезпечення і «чорні ходи», впроваджені на етапі розробки, компіляції програмного коду або поширення; помилки та недоробки, не усунені у процесі тестування; шкідливе програмне забезпечення та «чорні ходи», навмисно впроваджені третіми особами у процесі доставки [9,10,15].

Таким чином, завдання оцінки безпеки програмного забезпечення не вирішується довірою до програмного продукту, який розроблений відомою компанією, сертифікований та отриманий із джерел, які видаються довіреними.

АНАЛІЗ ОСТАННІХ ДОСЛІДЖЕНЬ ТА ПОСТАНОВКА ЗАДАЧІ

Виявити потенційні загрози можна на підставі аналізу вихідних текстів та динамічного аналізу поведінки у контрольованому середовищі. Однак в умовах відсутності вихідних текстів потрібне використання ще більш технічно складних процедур, у тому числі декомпіляції, реверсі інжинірингу і т.д. На практиці фахівцю із захисту інформації потрібно швидко і ефективно визначити можливість використання програмного забезпечення в інфраструктурі, що захищається, без порушення інфор-

маційної безпеки. При цьому час та обчислювальні ресурси обмежені, а повна автоматизація процесу аналізу безпеки неможлива у зв'язку з високою складністю предметної галузі [6,17,18].

На даний час не існує універсального та оптимального вирішення задачі виявлення шкідливого програмного забезпечення. Використання всього спектра наявних засобів дозволяє максимально наблизитися до вирішення завдання виявлення шкідливого коду, визначення його параметрів та поведінки, і навіть ідентифікації методів, а можливо навіть особистості зловмисника, проте вимагає докладання значних зусиль, тимчасових витрат і кваліфікації [5,7].

Для підвищення ефективності вирішення цього завдання в умовах обмежених часових та обчислювальних ресурсів, а також відсутності відкритих вихідних текстів програмного забезпечення, пропонується підхід, що полягає в автоматизації оцінки інформаційної безпеки програмного забезпечення без вихідних текстів та базується на моделі безпечного функціонування ПЗ та методики оцінки інформаційної безпеки [14,16].

Безпека ПЗ означає, що в результаті його включення в систему не відбувається порушення стану безпеки, тобто повною мірою зберігаються функціональні характеристики системи, не змінюються регламентовані взаємодії суб'єктів та об'єктів системи. Безпека також означає відсутність дефектів недокументованих можливостей, які можуть спричинити порушення безпеки як власне аналізованого ПЗ, так і системи, в яку воно встановлюється, у тому числі при взаємодії з іншими компонентами системи. Безпека ПЗ також передбачає відсутність у його складі шкідливого коду [4,10,20].

Характеристики шкідливого програмного забезпечення наведені в таблиці 1.

Таблиця 1. Характеристики шкідливого програмного забезпечення

Класи шкідливого програмного забезпечення, середовища оперування, область дії та поширення		Шпигунське	Рекламне	Куки файли	“ Чорний хід”	Троянець	Сніфери	Спам	Мережеві боти	Логічні бомби	Черв’ яки	Віруси
Технології створення ШПЗ	Патерни	+	+	+	+	+	+	+	+	+	+	+
	Обфузування	+	+	+	+	+	+	+	+	+	+	+
	Поліморфізм	+	+	+	+	+	+	+	+	+	+	+
	Спеціальні інструменти	+	+	+	+	+	+	+	+	+	+	+
Середовище виконання	Мережа	+	+	+	+	-	+	+	+	+	+	-
	Віддалене виконання	+	+	+	+	+	+	+	+	-	-	-
	Робоча станція	-	-	-	-	-	-	-	-	+	+	+
Засоби поширення, середовище доставки	Мережа	+	+	+	+	+	+	+	+	+	+	+
	Змінні диски	+	+	+	+	+	+	+	+	+	+	+
	Завантажувальні файли	+	+	+	+	+	+	+	+	+	+	+
Руйнівний вплив	Порушення конфіденційності	+	-	+	-	+	+	-	-	-	-	-
	Відволікання користувачів	-	+	-	-	-	-	+	-	-	-	-
	Відмова в обслуговуванні	-	-	-	+	-	-	+	+	+	+	+

Наведена характеристика порівняно умовна, оскільки шкідливе програмне забезпечення, що використовується для здійснення кібератак може мати комбінований функціонал. Найбільш небезпечними є засоби, націлені на приховану присутність у системі з метою викрадення інформації та перехоплення управління [5,7,8].

Регламентовані стандартом заходи та процедури також можуть бути використані для побудови системи оцінки безпеки програмного забезпечення, що пропонується сторонніми розробниками. Однак, безпосереднє застосування методик оцінки, побудованих на аналізі вихідного коду, не може бути ефективно здійснено у разі його відсутності. Більша частина програмного забезпечення для систем Windows, що пропонується користувачам, не постачається з вихідними текстами [2,9,13].

На етапі впровадження програмного забезпечення, завдання оцінки безпеки стоїть дуже гостро, і має вирішуватися експлуатаційним персоналом достатньо ефективно. В даний час немає єдиного методичного документа, який дозволяв би при його застосуванні персоналом без спеціальних знань приймати рішення про можливість застосування ПЗ у корпоративних мережах. Натомість існує великий інструментарій, безліч методів та підходів для дослідження поведінки ПЗ, шкідливий характер якого відомий, і завданням дослідника є визначення класу шкідливого ПЗ, характерних ознак та поведінки для подальшого використання в системах захисту [8,20].

Об'єктивні причини появи вразливостей у ПЗ полягають у надзвичайно високій структурній складності програмного коду, динамічності розвитку версій, що випускаються розробником, та легкості самодифікації програм при віддаленому оновленні. До цього можна додати проблему достовірної ідентифікації навмисно створених програмних закладок, недосконаліть нормативно-методичної та відставання інструментальної бази сертифікаційних випробувань [6,16,20].

Вразливі сигнатури у вихідних текстах ПЗ шукаються розробниками лише на етапах внутрішнього тестування, основною метою якого є виявлення власних помилок у розробленому програмному забезпеченню. При цьому про оцінку безпеки ПЗ найчастіше не замислюються [6,12,22].

Статичний аналіз вихідних текстів здійснюється без реального виконання досліджуваних програм. Залежно від інструмента, що використовується, глибина аналізу може змінюватись від визначення поведінки окремих операторів до дослідження, що включає весь наявний вихідний текст.

Динамічний аналіз вихідних текстів виконується за допомогою запуску програм на реальному чи віртуальному процесорі. Утиліти динамічного аналізу можуть вимагати завантаження спеціальних бібліотек, перекомпіляцію програмного коду. Для більшої ефективності динамічного аналізу потрібно подання тестованій програмі достатньої кількості вхідних даних, щоб отримати повніше покриття коду.

Статичний та динамічний аналізи дозволяють виконати наступні види контролю:

1. Контроль повноти та відсутності надмірності вихідних текстів. Він здійснюється експертом за допомогою програмних засобів, які виявляють можливо надлишкові функціональні об'єкти, що не використовуються.

2. Контроль відповідності вихідних текстів його об'єктному коду, полягає в компіляції вихідних текстів програм і порівнянні отриманих об'єктних (завантажувальних) кодів з аналогічними, наданими з дистрибутивом ПЗ, що перевіряється. В якості програмних засобів для здійснення цієї перевірки використовуються компілятор та програма підрахунку контрольних сум та порівняння файлів.

3. Контролює зв'язки функціональних об'єктів з управління. Ця перевірка полягає у побудові графа взаємодії функціональних об'єктів з управління (граф виклику функцій). Перевірка може бути виконана експертом без залучення додаткових програмних засобів, проте обсяг вихідних текстів та кількість функціональних об'єктів у сучасному програмному забезпечення досить велика.

4. Контроль зв'язків функціональних об'єктів за інформацією побудові графа взаємодії функціональних об'єктів за інформацією (граф передачі змінних між функціями та використання глобальних змінних усередині функцій).

5. Контроль інформаційних об'єктів передбачає побудову переліку інформаційних об'єктів (усі змінні, крім локальних, що не передаються інші функціональні об'єкти).

6. Контроль наявності заданих конструкцій у вихідних текстах полягає у пошуку певних конструкцій у вихідних текстах ПЗ. Пошук здійснюється за базою вразливостей, що містить сигнатури чи відмітні ознаки вразливості.

При контролі виконання функціональних об'єктів відбувається зіставлення фактичних маршрутів їх виконання та маршрутів, побудованих у процесі проведення статичного аналізу [5,7,15].

Лексичний аналіз передбачає пошук розпізнавання та класифікацію різних лексем об'єкта дослідження (програмного забезпечення), представленого у виконуваних кодах. У цьому лексемами є сигнатури. В даному випадку здійснюється пошук сигнатур наступних класів: сигнатури вірусів; сигнатури елементів; сигнатури (лексеми) «підозрілих функцій»; сигнатури штатних процедур використання системних ресурсів та зовнішніх пристроїв. Пошук лексем (сигнатур) реалізується за допомогою спеціальних інструментів – сканерів [10,14,16,22].

Синтаксичний верифікаційний аналіз передбачає пошук, розпізнавання та класифікацію синтаксичних структур, а також побудова структурно-алгоритмічної моделі самого продукту [11]. Семантичний аналіз передбачає дослідження функцій (процедур) продукту аспекті операційної середовища інформаційної системи. На відміну від попередніх видів аналізу, заснованих на статичному дослідженні, семантичний аналіз націлений на вивчення динаміки продукту - його взаємодії з довкіллям. Процес дослідження здійснюється у віртуальному операційному середовищі з повним контролем його дій та відстеженням алгоритму його роботи з структурно-алгоритмічної моделі. Семантичний аналіз є найбільш ефективним видом аналізу, але при цьому найтрудомісткішим. Узагальнена класифікація методів аналізу програмного забезпечення представлена на рисунку 1.

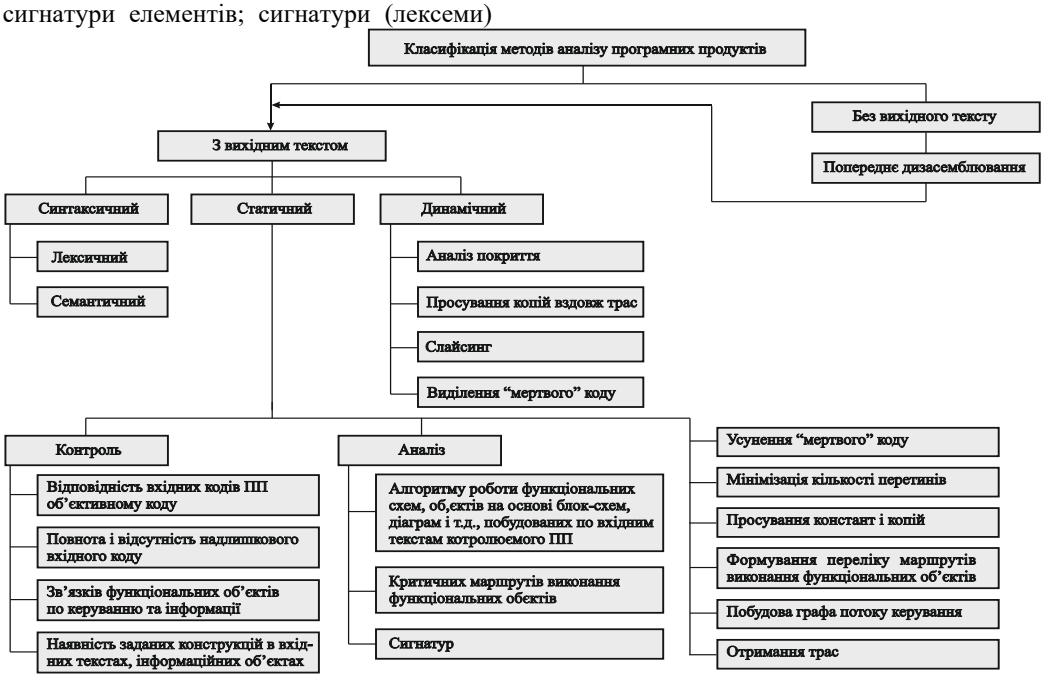


Рис.1. Узагальнена класифікація методів аналізу програмного забезпечення

Зі зростанням складності ПЗ динамічний аналіз стає нерозв'язним завданням і перетворюється на формальну процедуру. Багато програмних комплексів, які використовуються експертами у повсякденній роботі, мають списки потенційно небезпечних конструкцій, що робить неефективними методи сигнатурного аналізу. Не згадується сигнатурний аналіз у вимогах до випробувань ПЗ нижче другого рівня контролю (тобто, програм, що обробляють секретну та конфіденційну інформацію) [13]. Немає механізмів виявлення помилок кодування, пов'язаних із переповненням буфера, викликом функцій із чужого адресного простору, відсутнє очищення пам'яті тощо. Також відсутні вимоги щодо побудови переліку маршрутів під час виконання функціональних гілок програми, немає механізмів визначення повноти покриття коду та його достатності при динамічному аналізі.

Дослідження вітчизняних та зарубіжних джерел дозволило виділити такі ключові методи, способи та підходи до оцінки інформаційної безпеки ПЗ без вихідних текстів:

1. Перевагою статичного аналізу є повне покриття коду, на відміну методів і способів динамічного аналізу. Виконуючи бінарний аналіз програмного забезпечення, що розробляється, є штатним для середовища функціонування, можна виконати перевірку сторонніх бібліотек, які були використані при розробці. Також можна виконати контрольну перевірку фінальної версії програмного забезпечення шляхом порівняння результатів аналізу вихідного коду та виконуваного коду. Також статичний аналіз може бути успішно використаний при дослідженні стороннього ПЗ, що отримується від постачальників або завантажується з Інтернету.

2. Динамічний аналіз із використанням «пісочниці». Такий вид аналізу являє собою підхід до виявлення підозрілої поведінки у віртуальному ізольованому середовищі. Важливо, що налаштування віртуальних середовищ може бути виконане відповідно до одного з двох підходів.

Оптимальний тут підхід, при якому запуск зразка програмного забезпечення виконується в такому віртуальному середовищі, яке повністю відповідає цільовій системі, якій призначений даний зразок. Однак у такому разі «пісочниця» повинна «на льоту» визначати цільове середовище вузла мережі та запускати потрібну віртуальну машину.

3. Динамічний аналіз із використанням відладчика. Відладники реалізують покрокове виконання програм та встановлення в них точок зупинки. Це може бути реалізовано лише на рівні центрального процесора, мікроконтролера, операційної системи. Ключові функції відладчика – запуск програми, встановлення точок її зупинки, відображення даних, що використовуються програмою, та внесення змін до програми, щоб перевірити, чи можливо таким чином усунути виявлені помилки.

4. Аналіз на імітаційному стенді із використанням форензики. Інтерактивне тестування безпеки додатків є методикою аналізу безпеки ПЗ, переважно спрямовану на дослідження поведінки веб-додатків під час їх роботи [14]. Рішення з інтерактивного тестування зазвичай працюють за рахунок розгортання спеціальних агентів у додатку, що працює. Ці агенти безперервно аналізують те, як взаємодіють додатки (зазвичай ініціювати автоматизованими тестами), щоб виявити вразливості. Деякі рішення щодо інтерактивного тестування безпеки можуть не тільки активно відстежувати вразливості (наприклад, SQL-ін'єкції), але й перевіряти їх, демонструючи їхню придатність для використання зловмисниками. Для аналізу безпеки в такому підході часто використовуються методи форензики, тобто всі методи, пов'язані з розслідуванням вже скоєних комп'ютерних інцидентів.

Таким чином, можна відзначити, що динамічний аналіз більш популярний, ніж статичний, при оцінці безпеки ПЗ без вихідних текстів, однак найкращий результат досягається при їх сукупному використанні, оскільки саме статичний аналіз з використанням дизассемблера дозволяє отримати повне покриття коду та усунути ключовий недолік, властивий методів динамічного аналізу [4,10,15].

Слід зазначити, що техніка динамічного аналізу є найбільш популярною, при цьому особливо ефективною вона є в поєднанні з

механізмами віртуалізації, оскільки це дає можливість дослідити поведінку зразка ПЗ у середовищі, наближеному за своїми характеристиками до цільової.

Особливої ефективності виявлення шкідливого програмного забезпечення слід очікувати при поєднанні статичного і динамічного аналізу, оскільки статичний аналіз забезпечує максимальне покриття коду. Однак в умовах відсутності вихідних текстів ПЗ це зробити важче, ніж у разі їх наявності, оскільки потрібно використовувати дизасемблер, що дозволяє приблизно відновити високорівневий код програми.

МОДЕЛЮВАННЯ ПРОЦЕСУ ФУНКЦІОНУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Завдання забезпечення безпечного функціонування ПЗ включає: забезпечення інформаційної безпеки для інформаційного середовища під час функціонування ПЗ; забезпечення якісної та надійної реалізації ПЗ своїх функцій.

Програмне забезпечення за функціональним призначенням можна поділити на системне, прикладне та сервісне. Програмне забезпечення є компонентом інформаційної системи (операційної системи, яка своєю чергою може бути частиною програмно-апаратного комплексу, розподіленої обчислювальної мережі).

Оцінка надійності ПЗ пов'язана з можливістю впровадження тексту програмних засобів на етапі розробки та/або модифікації програмних закладок, використовуючи які, зловмисник реалізує створені вразливості. У зв'язку з цим, необхідність сертифікації та перевірки ПЗ на відповідність вимогам чинних нормативних документів протягом усього його життєвого циклу, не викликає сумнівів [4,5,10].

Для оцінки надійності програмних продуктів, що є складовою інформаційних систем, окремий інтерес представляють рандомізовані моделі та методи теорії надійності. Розглянемо основні види моделей оцінки надійності програмних продуктів із зазначенням їх переваг та недоліків.

Модель Джелінскі – Моранди заснована на припущеннях, що час до наступної відмови ПЗ (у разі збою або кібератаки) розподілено експоненційно, а інтенсивність відмов ПЗ пропорційна кількості помилок, що залишилися в програмі.

Перевагою моделі є простота розрахунків, а основним недоліком те, що при неточному визначенні величини початкової кількості помилок інтенсивність відмов програми може стати негативною.

Модель Шумана, передбачається, що дослідження безпеки та надійності ПЗ проводиться в кілька етапів, кожен з яких являє собою виконання ПЗ набору тестових даних. Виявлені протягом етапу тестування помилки реєструються, але не виправляються. Після завершення етапу виправляються всі виявлені помилки, коригуються тестові набори та проводиться новий етап тестування.

Основною перевагою моделі є визначення всіх невідомих параметрів, необхідних для розрахунків, і відсутність необхідності звернення до інших моделей. Недоліком є припущення моделі про те, що при коригуванні програмного забезпечення не вносяться нові помилки. Також для розрахунків потрібна велика кількість зареєстрованих даних.

В основі моделі Модель Шика – Волвертона лежить припущення, за яким частота помилок пропорційна як кількості помилок в ПЗ, а й часу тестування, тобто, ймовірність виявлення помилок з часом зростає.

Модель дозволяє досить точно розраховувати надійність і проста в застосуванні.

Недолік моделі: для розподілів, відмінних від експоненціального, коли інтенсивність помилки змінюється з часом, необхідно використовувати умовні розподіли. При цьому умовна ймовірність для певного інтервалу тестування повинна відповідати проміжку часу від початку тестування до початку розгляду інтервалу тестування. Інакше цей процес відновлення за умови усунення виявлених помилок призведе до завищення ймовірностей безпомилкового функціонування на всіх ділянках тестування.

Модель Коркорена передбачає наявність у ПЗ багатьох джерел програмних відмов, пов'язаних з різними типами помилок та вразливостей, та різну ймовірність їх появи.

Перевагою моделі є те, що вона використовує ймовірності відмов, що змінюються, для різних типів помилок і оцінюється ймовірність безвідмовного виконання ПЗ на даний момент часу. Недоліком є виконання оцінки з урахуванням апріорної інформації чи даних попереднього періоду функціонування однотипних зразків ПЗ.

Модель IBM. Під час експлуатації користувачем поточної версії програмного забезпечення розробники, як правило, займаються його супроводом – вносять деякі покращення або виправлення в цю версію, не чекаючи, поки користувач цього вимагатиме. Супровід може включати також додавання нових функцій. З деякого моменту, коли розробник вважає своє завдання виконаним, починається пасивний супровід, коли виправлення вносяться вже за запитом користувача. При супроводі в кожну нову версію програмного забезпечення вноситься значна кількість нових помилок, разом із доопрацюваннями, змінами та виправленнями, що потребує виправлень також і в наступній версії.

Зокрема, в компанії IBM спробували передбачити кількість подібних виправлень від версії до версії, ґрунтуючись на велику кількість експериментальних даних, зібраних у ході супроводу операційної системи.

Модель Шнайдера пов'язує число помилок у програмі з витратами, виміряними в «людино-місяцях», числом підпрограм і загальним числом *тисяч* операторів у програмі.

Перевагою моделі є низька обчислювальна складність і простота реалізації, до недоліків можно віднести необґрунтованість емпірично обраного числа виправлень і те що, що вразливості нульового дня не враховуються, а функції ПЗ не досліджуються детально.

МОДЕЛЬ БЕЗПЕЧНОГО ФУНКЦІОНУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Запропонована модель безпечного функціонування ПЗ має усувати недоліки, властиві розглянутим моделям. За результатами проведених досліджень моделей, можна зробити наступні висновки про їх недоліки: деякі моделі при реалізації вимагають значного обсягу обчислювальних ресурсів, це пов'язано як із необхідністю накопичення та обробки великої кількості даних, так і з обчислювальними ресурсами компонентів системи, які здійснюють аналіз безпеки; ймовірнісні моделі, як правило, ґрунтуються на припущеннях про те, що інтенсивність відмов/атак чи кількість помилок у ПЗ заздалегідь відомі, відомий розподіл (стандартний, пуассонівський, біноміальний), що не завжди вірно для реальних процесів і систем; відмови ПЗ та кібератаки, викликані експлуатацією вразливостей у ПЗ, розглядаються як сукупність, внаслідок чого моделі не враховують важливої особливості роботи шкідливого програмного забезпечення – а саме характеру його звернень з пам'яттю; жодна з розглянутих моделей не забезпечує комплексного представлення про процес функціонування ПЗ, у тому числі, погляд з боку інформаційної безпеки відсутній.

Для усунення вищеописаних недоліків пропонується основою моделі покласти опис взаємодії сутностей, задіяних у процесі функціонування ПЗ. При цьому необхідно враховувати специфіку поведінки шкідливого програмного забезпечення на комп'ютері або іншому обчислювальному пристрої.

Дослідження показали, що з основних ознак зараження різного типу шкідливого програмного забезпечення наступні:

1. Уповільнення роботи комп'ютера або вузла мережі. Як правило, це викликано впровадженням шкідливого програмного забезпечення та виконанням своїх функцій, що вимагають обчислювальних ресурсів, внаслідок чого на виконання легітимних обчислювальних задач потрібно більше часу.

2. Виконання нових функцій, не властивих даному комп'ютеру: відправлення даних на сторонні веб-ресурси, запуск процесів, які раніше не виконувались, і т.д.

3. Проблеми з відкриттям файлів, які раніше успішно відкривалися на комп'ютері, а також зміни розширення файлів (зникнення розширень або, навпаки, поява подвійних розширень).

4. Раптове завершення роботи додатків або поява спливаючих вікон при роботі додатків, що потенційно сигналізує про шкідливі процеси, що виконуються фоном.

Для шкідливого програмного забезпечення характерне надмірне використання обчислювальних ресурсів, а також операції з пам'яттю, що значно відрізняються від операцій, що виконуються легітимним ПЗ.

Так, наприклад, для шкідливого програмного забезпечення, що використовує обчислювальні потужності середовища для майнінгу, характерним є такий прояв у системі, як уповільнення роботи обчислювальних компонентів системи, поява обчислювальних процесів, що використовують велику кількість ресурсів, а також відправлення мережевого трафіку на сторонні веб-ресурси.

Таким чином, ключову роль описі роботи програмногозабезпечення грають: функції, що виконуються програмним забезпеченням; зміни у використанні обчислювальних ресурсів; зміни, пов'язані з роботою програмного забезпечення з пам'яттю.

На рис. 2 наведено графічне представлення моделі як набору взаємодіючих сутностей. Опис взаємодії сутностей моделі (рис. 2), відбиває як надійність функціонування програмного забезпечення, що з виконанням заявлених функцій, а й безпеку функціонування програмного забезпечення.

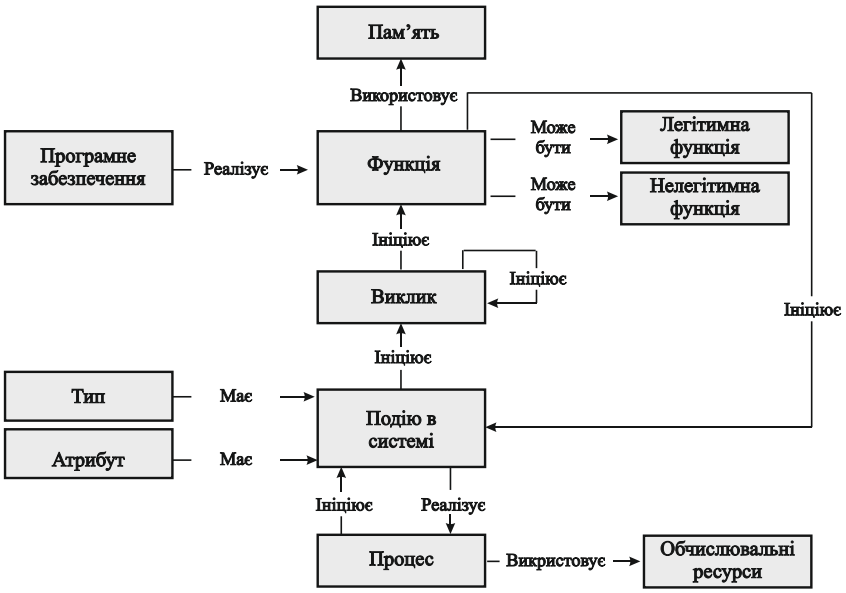


Рис. 2. Графічне представлення моделі безпечного функціонування програмного забезпечення

Безпечним буде вважатися ПЗ, встановлення якого не веде до порушення легітимних відношень у системі, але також і те, що не має такого потенціалу внаслідок наявності у своєму складі певного шкідливого функціоналу. Для формалізації опису скористаємося математичним апаратом теорії множин. Опишемо модель безпечного функціонування ПЗ формально, представивши ПЗ ( ) як кортеж:

$$Software =< F, Sign, R, M, Memory > \quad (1)$$

де:

1. $F = \{f_1, f_2, ..., f_n\}$ - кінцева множина функцій, які виконує програмне

забезпечення, що досліджується. Функції, що входять до множини F, поділяються на деструктивні F_{destr} та легітимні функції F_{leg} : $F = F_{leg} \bigcup F_{destr}$, $F_{leg} \bigcap F_{destr} = \varnothing$. При цьому, в будь якій із підмножин F_{leg} і F_{destr} можуть бути присутніми як документовані функції програмного забезпечення F_{doc}, так і не документовані F_{undoc} , $F_{doc} \bigcap F_{undoc} = \varnothing$.

2. $Sign$–множина ознак, що характеризують виконання функцій програмного забезпечення. Маємо неоднозначне відображення множини F на множину $Sign$: $F \to Sign$, тобто кожному елементу множини F відповідає один або кілька елементів множини $Sign$. Використовуючи відповідні індекси, задамо відображення $F_{leg} \to Sign_{leg}$, $F_{destr} \to Sign_{destr}$, $F_{doc} \to Sign_{doc}, F_{undoc} \to Sign_{undoc}$.

3. R – обчислювальні ресурси, необхідні програмному забезпеченню для реалізації своїх функцій. Маємо відображення $F \to R$, внаслідок якого кожній f_i призначаються три чисельні значення: $\left\{r_{i_{\min}}, r_i, r_{i_{\max}}\right\}$. Очевидно, що оцінка необхідних обчислювальних ресурсів може змінюватись, проте в системі для кожного програмного продукту повинен бути виділений певний обсяг обчислювальних ресурсів, щоб система ефективно функціонувала.

4. M – множина методів дослідження, що застосовуються для виявлення шкідливого програмного забезпечення.

5. $Memory$ – множина операцій над пам'яттю, що виконуються при активації тієї чи іншої функції з множини F . Таким чином, маємо відображення $F \to Memory$, за якого кожній функції f_i зіставляється набір операцій над пам'яттю:

$$f_i \to \left\{memory_{i_1}, memory_{i_2}, ..., memory_{i_k}\right\}.$$

Необхідно відзначити, що застосування методів M тільки в ідеальному випадку дозволить виявити множину ознак $Sign$, характерних для програмного продукту. На практиці слід говорити про визначення за допомогою методів M підмножини $Sign'$, $Sign' \in Sign$. На практиці вирішується завдання оцінки умовної ймовірності $P(A \mid B)$ події $A = \left\{Sign_{destr} = \varnothing \; and \; Sign_{undoc} = \varnothing\right\}$ при умові виконання події B реалізації деякого набору методів M', $M' \in M^{'}$, що дозволяє виявити достатнє число ознак $Sign'_{destr}$ та $Sign'_{undoc}$,

$$B = \left\{M' \to \left\{Sign'_{destr}, Sign'_{undoc}\right\} \neq \varnothing\right\}.$$

Тоді умовна ймовірність $P(A \mid B)$ визначиться як (1):

$$P(A \mid B) = P(A) \cdot P(B) = P\left\{Sign_{destr} = \varnothing \; and \; Sign_{undoc} = \varnothing\right\} \cdot \cdot P\left\{M' \to \left\{Sign'_{destr}, Sign'_{undoc}\right\} \neq \varnothing\right\} \quad (1)$$

Досягнення максимальної ефективності використовуваних методів M малоймовірно, однак у такому разі буде досягнуто рівність (2):

$$Sign_{destr} = Sign'_{destr}, \; Sign = Sign'_{undoc} \quad (2)$$

Сформулюємо критерії безпечного функціонування програмного продукту в термінах моделі:

1. Програмний продукт не реалізує деструктивних та шкідливих функцій: $F_{destr} = \varnothing$ та $F_{undoc} = \varnothing$.

2. Об’єм, задіяних для аналізу безпеки програмного продукту (за допомогою множини методів M) обчислювальних ресурсів мінімізовано:

$$M : R \to R_{\min}, R_{\min} = \left\{r_{1_{\min}}, r_{2_{\min}}, ..., r_{n_{\min}}\right\},$$

таким чином, кожна функція програмного продукту в ідеалі повинна використовувати для реалізації мінімально допустимого об’єму обчислювальних ресурсів;

3. Не повинно бути обчислювальних ресурсів, що витрачаються на будь - яку множину функцій F', що не перетинається

з множиною $F:\exists F' \to R$, оскільки присутність шкідливого програмного забезпечення в системі часто призводить до активації нових, нетипових для роботи системи функцій.

4. Множина операцій над пам'яттю *Memory* повинна мати відображення на множини ознак *Sign* , оскільки, важливою ознакою наявності шкідливого програмного забезпечення в системі є нетипові операції з пам'яттю: $Memory \to Sign \neq \varnothing$.

5. Множина методів дослідження, застосовуваних виявлення шкідливого програмного забезпечення, має використовувати результати відображення $Memory \to Sign$ означає, що для вирішення поставленого завдання доцільно застосовувати ті методи, які враховують операції програмного забезпечення з пам'яттю.

6. Множина використовуваних методів має бути такою, що умовна ймовірність $P(A|B)$ максимізована, тобто $M: P(A|B) \to 1: (|) \to 1$.

Розроблена модель відрізняється від розглянутих моделей оцінки безпеки, надійності та якості функціонування програмного продукту тим, що вона враховує як особливості функціонування шкідливого програмного забезпечення на компоненті системи, так і необхідність мінімізації обчислювальних ресурсів при аналізі. Особливу увагу слід приділити питанням практичної реалізації розробленої моделі, оскільки вона повинна поєднувати високу ефективність виявлення шкідливого програмного забезпечення та оптимальне використання обчислювальних ресурсів. Програмне забезпечення з потенційно небезпечними можливостями реалізує одну або кілька функцій, наслідки виконання яких можуть безпосередньо загрожувати порушенням безпеки даних, або призводити до виконання коду, що представляє цю загрозу. Види небезпечних можливостей представлені на рис. 3.

Рис. 4. Програмне забезпечення з потенційно небезпечними можливостями

Значимість їх систематизації виходить з особливостях роботи з пам'яттю, виділені можливості, наступні: передача управління в область модифікованих даних; самодифікація або зміна коду іншого програмного забезпечення в оперативній пам'яті або на зовнішніх носіях; самодублювання, підміна собою іншого програмного забезпечення або перенесення своїх фрагментів в область оперативної або зовнішньої пам'яті, що не належить програмі; збереження інформації з областей оперативної пам'яті, які не належать програмному забезпеченню; спотворення, блокування чи заміну інформації, що є результатом роботи інших програм; приховування своєї присутності у програмному середовищі.

Виявлення функцій, що здійснюють роботу з пам'яттю таким чином, що виникають потенційно небезпечні можливості, особливо значимо, тому що це основна характеристика поведінки шкідливого програмного забезпечення. Тому при реалізації моделі на практиці слід використовувати механізм, який забезпечує ефективний моніторинг звернень до пам'яті та її використання.

Процес у віртуальному операційному середовищі з повним контролем дій програмного забезпечення та відстеження алгоритму його роботи здійснюється більш ефективно. Таким чином реалізація розробленої моделі має базуватися на гіпервізорі, що дозволить віртуалізувати системні ресурси обчислювальних систем.

ВИСНОВКИ

В роботі виконано систематизацію моделей надійного та безпечного функціонування програмного забезпечення. В результаті проведеного дослідження виділено три типи моделей: аналітичні; статистичні; емпіричні.

Розглянуто ряд найчастіше застосовуваних моделей, виділено їх недоліки та переваги з погляду розв'язуваної задачі опису безпечного функціонування програмного продукту, та розпізнавання шкідливого програмного забезпечення. За результатами проведених досліджень розглянуті моделі мають переваги у плані простоти їх практичної реалізації, проте водночас виділено наступні недоліки: деякі з розглянутих моделей при реалізації вимагають великого обсягу обчислювальних ресурсів – для аналізу безпеки та накопичення архівних даних; використання статистичними та імовірнісними моделями припущень про те, що інтенсивність атак/відмов чи кількість помилок у програмному забезпеченню мають заздалегідь відомий розподіл (біноміальний, стандартний або пуасонівський), що не завжди вірно для реальних процесів і систем; немає поділу на відмови програмного забезпечення і вихід з ладу внаслідок кібератак, не враховуються також вразливості нульового дня; не аналізуються звернення досліджуваного програмного забезпечення до пам'яті, що могло б дати важливу інформацію про його легітимність чи наявність шкідливих функцій; жодна з розглянутих моделей не забезпечує комплексного представлення про процес функціонування програмного забезпечення, у тому числі, аналіз з боку інформаційної безпеки відсутній.

Завдання розпізнавання шкідливого програмного забезпечення з кожним роком стає дедалі актуальнішим і складнішим у зв'язку з цифровізацією галузей діяльності людини та використанням програмного забезпечення для виконання бізнес-логіки та технічних процесів у складних системах. Внаслідок цього, чим більший обсяг програмного забезпечення в системі, тим потенційно більше в ньому помилок, при цьому через підключення сучасних систем до мережі Інтернет, програмного забезпечення часто поширюється по мережі, що дозволяє зловмисникам створити нові вектори кібератак на системи.

Запропонована модель безпечного функціонування програмного продукту має усувати недоліки, властиві розглянутим моделям. Запропоновано модель усуває наведені недоліки за рахунок того, що вона враховує характерні особливості прояву шкідливого програмного забезпечення на пристроях, а саме вплив шкідливого програмного забезпечення на обчислювальні ресурси системи та роботу з

пам'яттю. Це дозволяє розробленій моделі враховувати як надійність функціонування програмного забезпечення, так і безпеку.

У термінах моделі сформульовані критерії безпечного функціонування програмного забезпечення, зроблено висновок про те, що для найбільш ефективної реалізації такої моделі на практиці має бути використаний гіпервізор. Вибір конкретного гіпервізора залежить від потреб, призначення та специфіки системи.

ЛІТЕРАТУРА

1. Доктрина інформаційної безпеки України, затвердженої Указом Президента України від 25 лютого 2017 року № №47/2017, 15с.
2. Державний стандарт України Захист інформації. Технічний захист інформації. Основні положення. ДСТУ 3396.0-96 [Електронний ресурс]. – Режим доступу: http://www.dsszzi.gov.ua/dsszzi/control/uk/publish/article?art_id=38883&cat_id =38836.
3. Закон України «Про основні засади забезпечення кібербезпеки України» зі змінами. Відомості Верховної Ради (ВВР), 2017, № 45, ст.403, зі змінами від 28.07.2022 року. Режим доступу: https://zakon.rada.gov.ua/laws/show/2163
4. **Цибульник С. О.** Технології розроблення програмного забезпечення. Частина 1. Життєвий цикл програмного забезпечення : підручник / С. О. Цибульник, К. С. Барандич. – К. : КПІ ім. Ігоря Сікорського, 2022.– 270 с.
5. **Лисенко С.М.** Аналіз методів виявлення шкідливого програмного забезпечення в компютерних системах. / С.М. Лисенко, Р.В. Щука - Вісник Хмельницького національного університету. 2020 № 2. (283). С. 101–107.
6. **Ленков С.В.** Метод прогнозування вразливостей інформаційної безпеки на основі аналізу даних тематичних інтернет-ресурсів / С.В. Лєнков, В.М. Джулій, А.М. Берназ, І.В. Муляр, І.В. Пампуха // Збірник наукових праць Військового інституту Київського національного університету імені Тараса Шевченка. – К.: ВІКНУ, 2023. – Вип. №78. – С. 123-134.
7. **Ленков С.В.** Метод протидії поширенню та виявлення шкідливої інформації в соціальних мережах/ С.В. Ленков, В.М. Джулій, Л.В. Солодєєва // Збірник наукових праць Військового інституту Київського національного університету імені Тараса Шевченка. – К.: ВІКНУ, 2022. – Вип. №77. – С. 103-117.
8. **Ленков С.В.** Модель безпеки поширення забороненої інформації в інформаційно-телекомунікаційних мережах / С.В. Ленков, В.М. Джулій, В.С. Орленко, О.В. Сєлюков, А.В. Атаманюк // Збірник наукових праць Військового інституту Київського національного університету імені Тараса Шевченка. – К.: ВІКНУ, 2020. – Вип. №68. – С. 53-64.
9. **Крепич** С. Я. Якість програмного забезпечення та тестування : базовий курс / С. Я. Крепич, Співак І. Я. – Тернопіль : ФОП Паляниця В. А., 2020. – 478 с.
10. Якість та тестування інформаційних систем : Навчальний посібник для самостійної роботи студентів вищих навчальних закладів / Золотухіна О. А., Негоденко О. В., Резник С. Ю., Разіна С. Я. – Київ : ННІІТ ДУТ, 2020. – 128 с.
11. **Вишня В. Б.** Основи інформаційної безпеки: навч. посібник / В. Б. Вишня, О. С. Гавриш, Е. В. Рижков. Дніпро : Дніпроп. держ. ун-т внутріш. справ, 2020. – 128 с.
12. Соціальні мережі – реальні загрози віртуального світу. [Електронний ресурс]. – Режим доступу : http://ogo.ua/articles/view/011–02–23/26490.htm.
13. **Остапов С.Е.** Технології захисту інформації: навчальний посібник / С.Е. Остапов, С.П. Євсеєв, О.Г. Король – Харків : Вид-во ХНЕУ, 2016. – 476 с.
14. **Лєнков, С.В.** Аналіз існуючих методів та алгоритмів виявлення атак в бездротових мережах передачі даних / С.В. Лєнков, В.М. Джулій, Н.М. Берназ, С.О. Божук // Збірник наукових праць Військового інституту Київського національного університету імені Тараса Шевченка. – К.: ВІКНУ, 2017. – Вип. № 56. – С.124-132.
15. **Бурячок В.Л.** Інформаційний та кіберпростори: проблеми безпеки, методи та засоби боротьби : посібник / [В. Л. Бурячок, С. В. Толюпа, В. В. Семко та ін.]. – К.: ДУТ-КНУ, 2016. – 178 с.
16. **Рибальченко Л.В., Косиченко О.О.** Проблеми безпеки персональних даних в Україні / Регіональна економіка / Запоріжжя. 2019. – с.57-62.
17. **Джулій В.М.** Метод класифікації додатків трафіка комп'ютерних мереж на основі машинного навчання в умовах невизначеності / В.М. Джулій, О.В. Мірошніченко, Л.В. Солодєєва // Збірник наукових праць Військового інституту Київського національного університету імені

Тараса Шевченка. – К.: ВІКНУ, 2022. – Вип. №74. – С. 73-82.
18. **Лавров Є.А**. Математичні методи дослідження операцій: підручник / Є. А. Лавров, Л. П. Перхун, В. В. Шендрик – Суми: Сумський державний університет, 2017. – 212 с.
19. **Гончар С.Ф.** Оцінювання ризиків кібербезпеки інформаційних систем об'єктів критичної інфраструктури: монографія. / С. Ф. Гончар. – Київ, 2019. – 175 с.
20. **Yemchuk L.** Organizational Network Analysis as a Tool for Leadership Assessment in Software Development Team. Zhylinska O.; Chornyi A.; Dzhuliy V. – Institute of Electrical and Electronics Engineers (30 September 2020); INSPEC Accession Number: 20008165; DOI: 10.1109/ACIT49673.2020.
21. Сигнатура атаки. Wikipedia [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Сигнатура_атаки
22. OPWNAI: Cybercriminals Starting to Use ChatGPT, January 6, 2023 [Електронний ресурс] – Режим доступу до ресурсу: https://research.checkpoint.com/2023/opwnai-cybercriminals-starting-to-usechatgpt.

REFERENCES

1. Doktryna informatsiinoi bezpeky Ukrainy, zatverdzhenoi Ukazom Prezydenta Ukrainy vid 25 liutoho 2017 roku № №47/2017, 15s.
2. Derzhavnyi standart Ukrainy Zakhyst informatsii. Tekhnichnyi zakhyst informatsii. Osnovni polozhennia. DSTU 3396.0-96 [Elektronnyi resurs]. – Rezhym dostupu: http://www.dsszzi.gov.ua/dsszzi/control/uk/publish/article?art_id=38883&cat_id =38836
3. Zakon Ukrainy «Pro osnovni zasady zabezpechennia kiberbezpeky Ukrainy» zi zminamy. Vidomosti Verkhovnoi Rady (VVR), 2017, № 45, st.403, zi zminamy vid 28.07.2022 roku. Rezhym dostupu: https://zakon.rada.gov.ua/laws/show/2163
4. **Tsybulnyk,S.O.**(2022)Tekhnolohii rozroblennia prohramnoho zabezpechennia. Chastyna 1. Zhyttievyi tsykl prohramnoho zabezpechennia : pidruchnyk / S. O. Tsybulnyk, K. S. Barandych. – K. : KPI im. Ihoria Sikorskoho.– 270 s.
5. **Lysenko, S.M.** (2020) Analiz metodiv vyiavlennia shkidlyvoho prohramnoho zabezpechennia v kompiuternykh systemakh. / S.M. Lysenko, R.V. Shchuka - Visnyk Khmelnytskoho natsionalnoho universytetu. № 2. S. 101–107.
6. **Lenkov S.V.** (2023). Metod prohnozuvannia vrazlyvostei informatsiinoi bezpeky na osnovi analizu danykh tematychnykh internet-resursiv / S.V. Lienkov, V.M. Dzhulii, A.M. Bernaz, I.V. Muliar, I.V. Pampukha // Zbirnyk naukovykh prats Viiskovoho instytutu Kyivskoho natsionalnoho universytetu imeni Tarasa Shevchenka. – K.: VIKNU -. №78. – С. 123-134.
7. **Lenkov S.V.** (2022). Metod protydii poshyrenniu ta vyiavlennia shkidlyvoi informatsii v sotsialnykh merezhakh/ S.V. Lenkov, V.M. Dzhulii, L.V. Solodieieva // Zbirnyk naukovykh prats Viiskovoho instytutu Kyivskoho natsionalnoho universytetu imeni Tarasa Shevchenka. – K.: VIKNU. – Vyp. №77. – С. 103-117.
8. **Lenkov S.V.** (2020). Model bezpeky poshyrennia zaboronenoi informatsii v informatsiino-telekomunikatsiinykh merezhakh / S.V. Lenkov, V.M. Dzhulii, V.S. ORLENKO, O.V. Sieliukov, A.V. Atamaniuk // Zbirnyk naukovykh prats Viiskovoho instytutu Kyivskoho natsionalnoho universytetu imeni Tarasa Shevchenka. – K.: VIKNU. – №68. – pp. 53-64.
9. **Krepych, S. Ya.** (2020) Yakist prohramnoho zabezpechennia ta testuvannia : bazovyi kurs / S. Ya. Krepych, Spivak I. Ya. – Ternopil : FOP Palianytsia V. A.. – 478 s.
10. Iakist ta testuvannia informatsiinykh system : Navchalnyi posibnyk dlia samostiinoi roboty studentiv vyshchykh navchalnykh zakladiv / Zolotukhina O. A., Nehodenko O. V., Reznyk S. Yu., Razina S. Ya. – Kyiv : NNIIT DUT, 2020. – 128 s.
11. **Vyshnia, V. B.** (2020) Osnovy informatsiinoi bezpeky : navch. posibnyk / V. B. Vyshnia, O. S. Havrysh, E. V. Ryzhkov. Dnipro : Dniprop. derzh. un-t vnutrish. sprav. – 128 s.
12. Cotsialni merezhi – realni zahrozy virtualnoho svitu. [Elektronnyi resurs]. – Rezhym dostupu: http://ogo.ua/ articles/view/011– 02–23/26490.htm
13. **Ostapov S. E.** (2016). Tekhnolohii zakhystu informatsii: navchalnyi posibnyk / S.E. Ostapov, S.P. Yevseiev, O.H. Korol–Kharkiv: Vyd-vo KhNEU. – 476 s.
14. **Lenkov S.V.** (2017). AnalIz Isnuyuchih metodiv ta algoritmiv viyavlennya atak v bezdrotovih merezhah peredachI danih / S.V. Lenkov, V.M. Dzhuliy, N.M. Bernaz, S.O. Bozhuk // Zbirnik naukovih prats Viyskovogo Institutu Kiyivskogo natsionalnogo universitetu imeni Tarasa Shevchenka. – K.: VIKNU. – Vip. No 56. – p.124-132
15. **Buriachok, V. L.** (2016). Informatsiinyi ta

kiberprostory: problemy bezpeky, metody ta zasoby borotby : posibnyk / V. L. Buriachok, S. V. Toliupa, V. V. Semko – K. : DUT-KNU – 178 s.

16. **Rybalchenko L.V., Kosychenko O.O.** (2019). Problemy bezpeky personalnykh danykh v Ukraini / Rehionalna ekonomika / Zaporizhzhia – s.57-62
17. **Dzhulii V.M.** (2022). Metod klasyfikatsii dodatkiv trafika kompiuternykh merezh na osnovi mashynnoho navchannia v umovakh nevyznachenosti / V.M. Dzhulii, O.V. Miroshnichenko, L.V. Solodieieva // Zbirnyk naukovykh prats Viiskovoho instytutu Kyivskoho natsionalnoho universytetu imeni Tarasa Shevchenka. – K.: VIKNU. – Vyp. №74. – pp. 73-82.
18. **Lavrov Ye. A**. (2017.). Matematychni metody doslidzhennia operatsii: pidruchnyk / Ye. A. Lavrov, L. P. Perkhun, V. V. Shendryk – Sumy: Sumskyi derzhavnyi universytet – 212 p
19. **Honchar S. F.** (2019). Otsiniuvannia ryzykiv kiberbezpeky informatsiinykh system obiektiv krytychnoi infrastruktury: monohrafiia. / S. F. Honchar. – Kyiv – 175 s.
20. **Yemchuk L.** Organizational Network Analysis as a Tool for Leadership Assessment in Software Development Team. Zhylinska O.; Chornyi A.; Dzhuliy V. – Institute of Electrical and Electronics Engineers (30 September 2020); INSPEC Accession Number: 20008165; DOI: 10.1109/ACIT49673.2020.
21. Syhnatura ataky. Wikipedia [Elektronnyi resurs] – Rezhym dostupu do resursu: https://uk.wikipedia.org/wiki/Syhnatura_ataky.
22. OPWNAI: Cybercriminals Starting to Use ChatGPT, January 6, 2023 [Elektronnyi resurs] – Rezhym dostupu do resursu: https://research.checkpoint.com/2023/opwnai-cybercriminals-starting-to-usechatgpt.

Information security model of functioning software

Serhii Lienkov, Volodymyr Dzhuliy, Oleksandr Yavorskyi, Kostyantyn Zatsepin

Abstract. The paper systematizes the models of reliable and safe functioning of the software. As a result of the research, three types of models were identified: analytical; statistical; empirical.

A number of the most frequently used models are considered, and their disadvantages and advantages are highlighted from the point of view of solving the problem of describing the safe functioning of a software product and recognizing malicious software. According to the results of the research, the considered models have advantages in terms of the simplicity of their practical implementation, but at the same time, the following disadvantages are highlighted: some of the considered models require a large amount of computing resources when implemented - for security analysis and accumulation of archival data; the use of statistical and probabilistic models of assumptions that the intensity of attacks/failures or the number of errors in software have a pre-known distribution (binomial, standard or Poisson), which is not always true for real processes and systems; there is no division into software failures and failures due to cyber attacks, zero-day vulnerabilities are also not taken into account; memory accesses of the investigated software are not analyzed, which could provide important information about its legitimacy or the presence of malicious functions; none of the considered models provides a comprehensive representation of the process of software functioning, including, there is no analysis from the information security side.

The task of recognizing malicious software is becoming more and more relevant and difficult every year in connection with the digitalization of human activities and the use of software for the execution of business logic and technical processes in complex systems. As a result, the larger the volume of software in the system, the more errors there are potentially, and due to the connection of modern systems to the Internet, the software is often distributed over the network, which allows attackers to create new vectors of cyber attacks on systems.

The proposed model of safe functioning of the software product should eliminate the shortcomings inherent in the considered models. The proposed model eliminates the mentioned shortcomings due to the fact that it takes into account the characteristic features of the manifestation of malicious software on devices, namely the impact of malicious software on the computing resources of the system and working with RAM. This allows the developed model to take into account both the reliability of software operation and security.

In terms of the model, the criteria for the safe functioning of the software are formulated, it is concluded that for the most effective implementation of such a model in practice, a hypervisor should be used.

Keywords: model, information security, vulnerabilities, attacks, confidential data, malware, secure operation.

ДОДАТОК В

Презентація кваліфікаційної роботи

~ proseno# show slide_1_presentation.pptx

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ЯВОРСЬКИЙ Олександр Віталійович

Метод оцінювання інформаційної безпеки безпечного функціонування програмного забезпечення

Науковий керівник
д.т.н., доцент Джулій В.М.

кафедра кібербезпеки

~ proseno#

~ proseno# show slide_1_start.pptx

Тема Метод оцінювання інформаційної безпеки безпечного функціонування програмного забезпечення

Мета магістерської роботи - Підвищення якості оцінки інформаційної безпеки безпечного функціонування програмного забезпечення

Наукова задача – розробка методу модульної інтеграції статичних інструментів аналізу безпеки програмного забезпечення для автоматизації процесу перевірки коду, об'єднання результатів у стандартизований формат та забезпечення масштабованості і адаптивності системи до різних мов.

Об'єкт дослідження: Системи оцінювання безпеки безпечного функціонування засобів захисту інформації.

Предмет дослідження: Методи оцінювання інформаційної безпеки безпечного функціонування програмного забезпечення.

Задачі досліджень у роботі формулюються наступним чином:

1. Провести аналіз існуючих методів оцінювання інформаційної безпеки і виявити їхні недоліки.
2. Розробити метод оцінювання безпеки програмного забезпечення, яка буде зрозумілою, ефективною та адаптованою для автоматизації.
3. Реалізувати розроблений метод у вигляді програмного прототипу на мові програмування Go.
4. Провести тестування та аналіз результатів роботи розробленого прототипу.
5. Порівняти ефективність запропонованого методу з існуючими підходами.

~ proseno#

~ proseno# show slide_2_tasks(1).pptx

Наукова новизна полягає у створенні модульного підходу для інтеграції інструментів статичного аналізу безпеки програмного забезпечення, що забезпечує автоматизацію та спрощення процесу перевірки коду. Розроблене рішення дозволяє об’єднати результати роботи різних інструментів у єдиний формат, зручний для аналізу, та адаптувати систему для інтеграції в процеси CI/CD. Важливою складовою новизни є здатність системи масштабуватися для аналізу великих проектів і підтримувати різні мови програмування без значних змін архітектури

Методи дослідження. Для вирішення задач у магістерській роботі застосовувалися методи: теоретичний аналіз існуючих методів оцінювання інформаційної безпеки програмного забезпечення, методи моделювання для розробки архітектури системи оцінки безпеки, експериментальні методи для тестування та оцінки ефективності розробленої системи, статистичний аналіз отриманих результатів для порівняння з існуючими підходами.

Практична цінність кваліфікаційної роботи полягає у підвищенні ефективності оцінки інформаційної безпеки програмного забезпечення за рахунок інтеграції статичних інструментів аналізу, автоматизації процесу перевірки коду, а також мінімізації обчислювальних ресурсів і трудомісткості процесу виявлення загроз.

Апробація роботи. Наукові результати і основні положення магістерської роботи доповідались і обговорювались на всеукраїнських та міжнародних науково-технічних конференціях,

Публікації. За темою дипломної роботи ОКР «Магістр» опубліковано 1 теза доповідей, 1 фахова стаття.

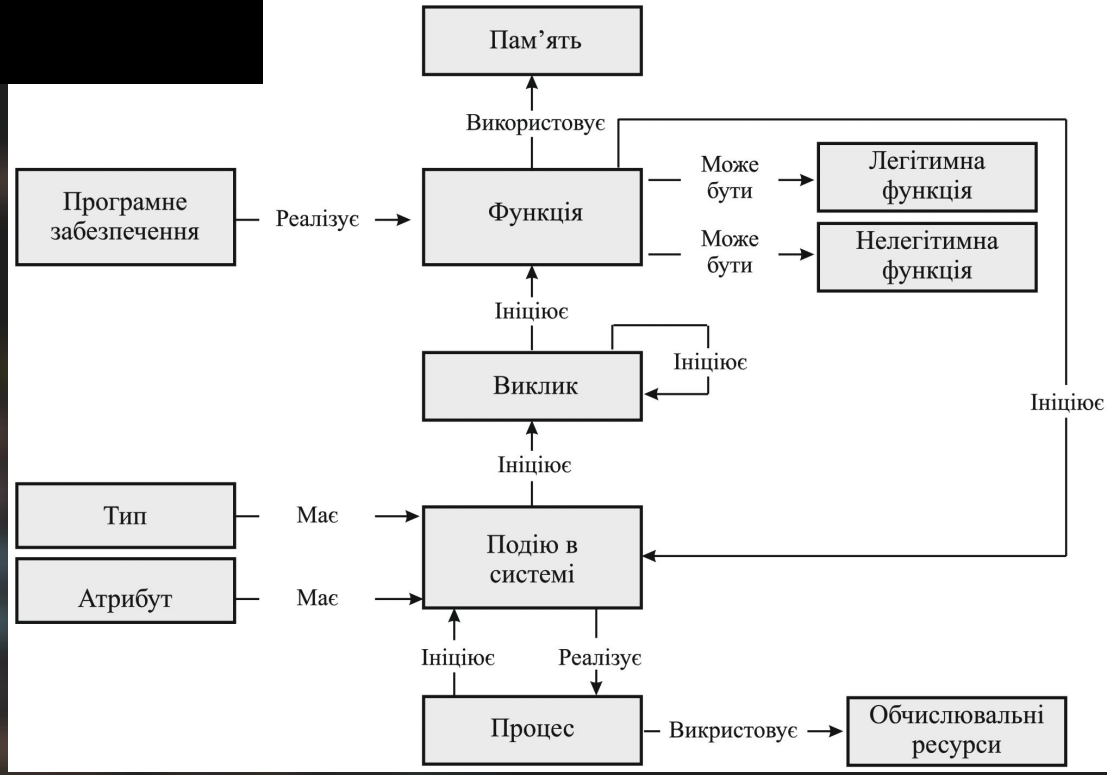
~ proseno#


```
~ proseno# show clide_analysis.pptx

Види потенційно небезпечних можливостей

~ proseno#
```

~ proseno# show clide_431_diagram4.pptx

Графічне представлення моделі безпечного функціонування програмного забезпечення

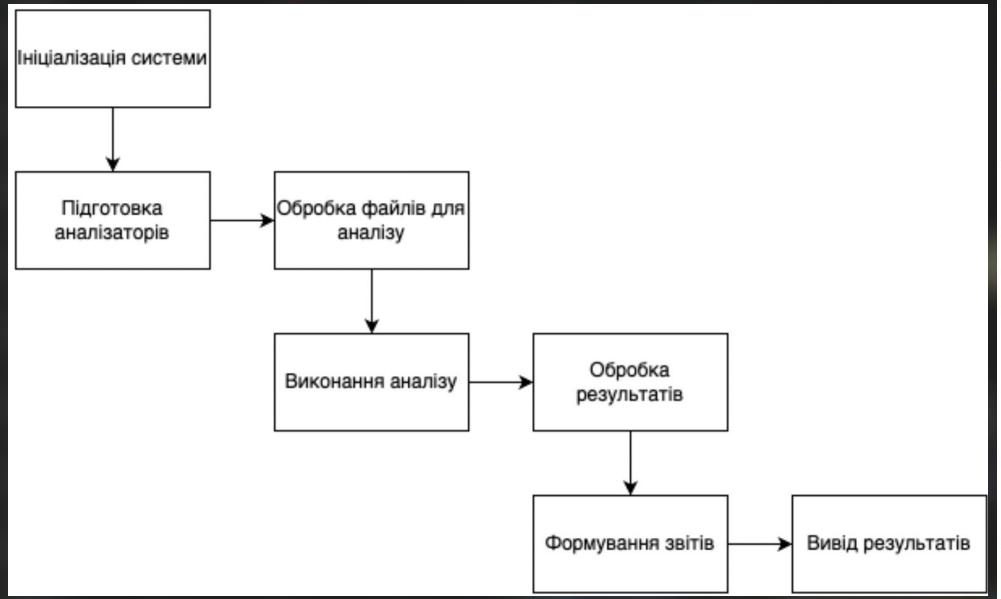
~ proseno#

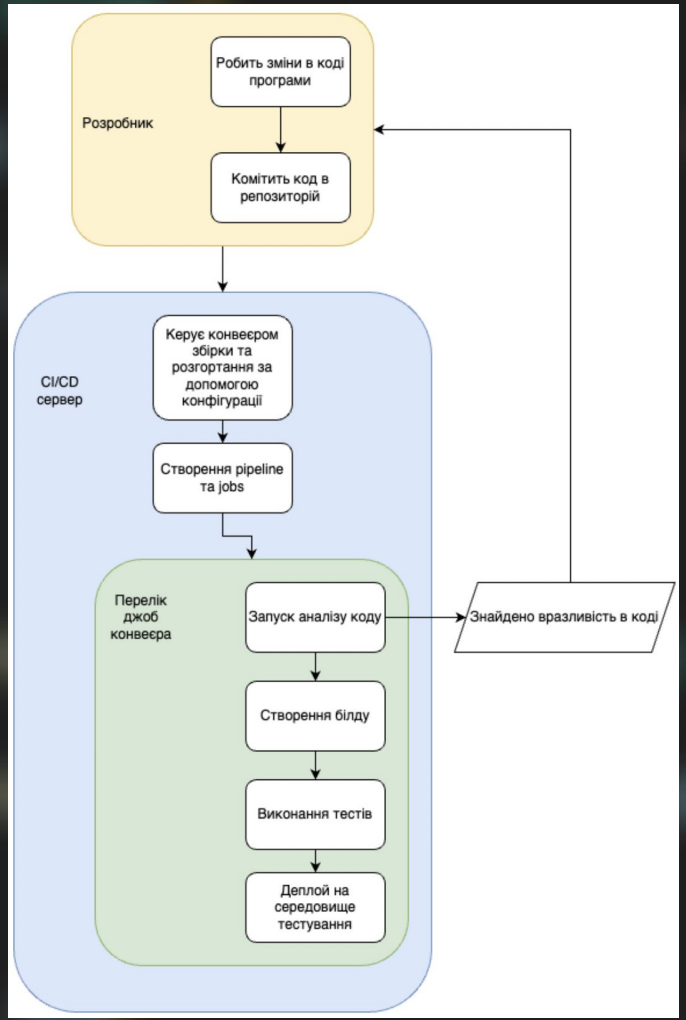


~ proseno# show clide_method.pptx

Метод оцінювання безпеки безпечного функціонування програмного забезпечення

~ proseno#



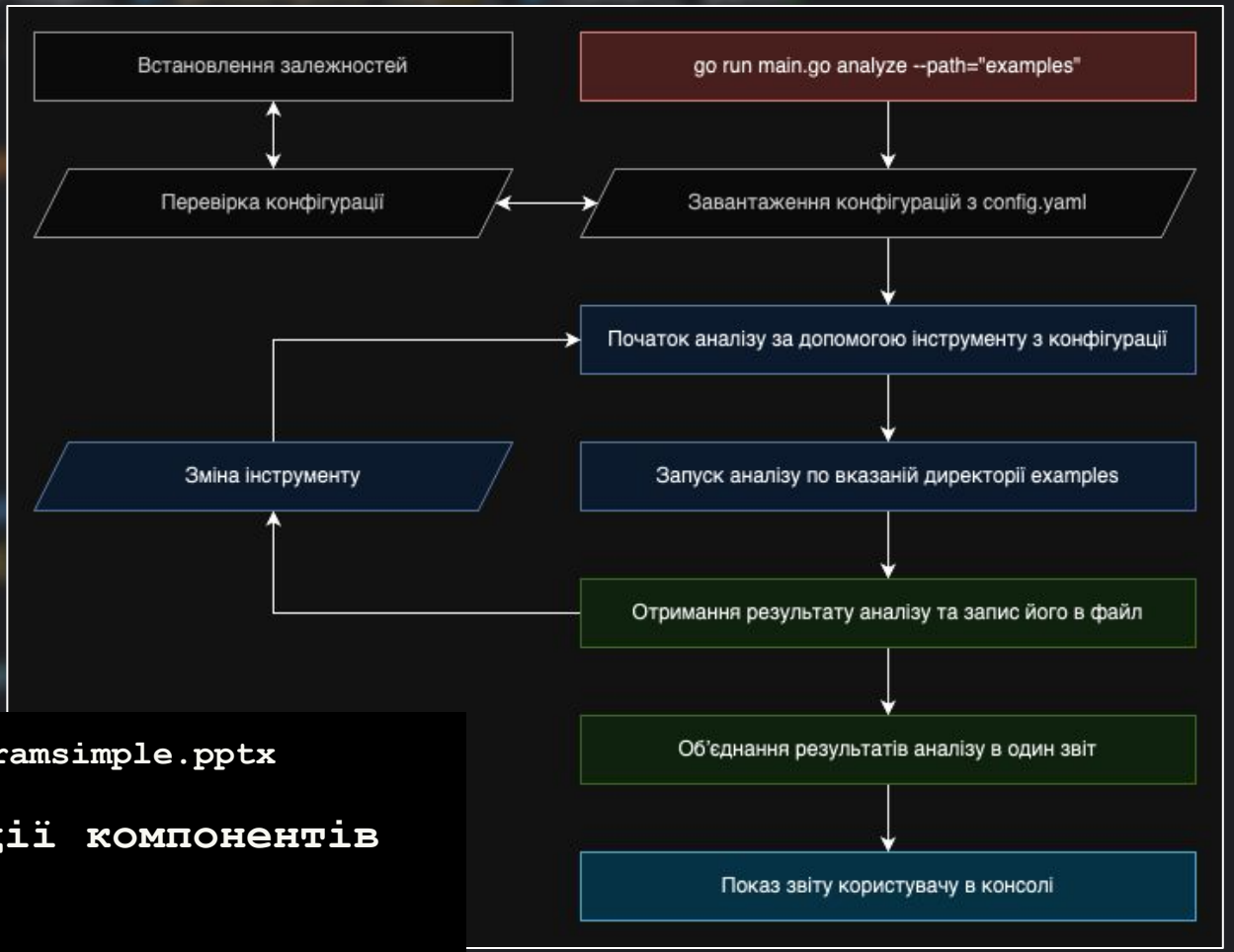


```go
func SqlInjections() {  no usages  Oleksandr Yavorskyi *
    db, err := sql.Open( driverName: "postgres",
        dataSourceName: "user=admin password=supersecret dbname=test sslmode=disable")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    userInput := "'; DROP TABLE users; --"
    query := fmt.Sprintf(
        format: "SELECT * FROM users WHERE username='%s'",
        userInput) // SQL Injection
    rows, err := db.Query(query)
    if err != nil {
        panic(err)
    }
    defer rows.Close()
}
```

```python
# Hardcoded credentials vulnerability
def hardcoded_credentials():
    username = "admin"  # Hardcoded username
    password = "supersecret"  # Hardcoded password
    print(f"Connecting with username: {username} and password: {password}")

```

```go

func InsecureTlsConfiguration() {  no usages  Oleksandr Yavorskyi
    tr := &http.Transport{
        TLSClientConfig: &tls.Config{InsecureSkipVerify: true}, // Insecure TLS configuration
    }
    client := &http.Client{Transport: tr}
    resp, err := client.Get( url: "https://example.com")
    if err != nil {
        panic(err)
    }
    defer resp.Body.Close()
}
```

```python
# Weak cryptographic hash function
def weak_hash(data):
    hash_value = hashlib.md5(data.encode()).hexdigest()  # Weak MD5 hash
    print(f"MD5 hash of {data}: {hash_value}")

```




```
~ proseno# show clide_8_diagramsimple.pptx
```

Спрощена схема взаємодії компонентів

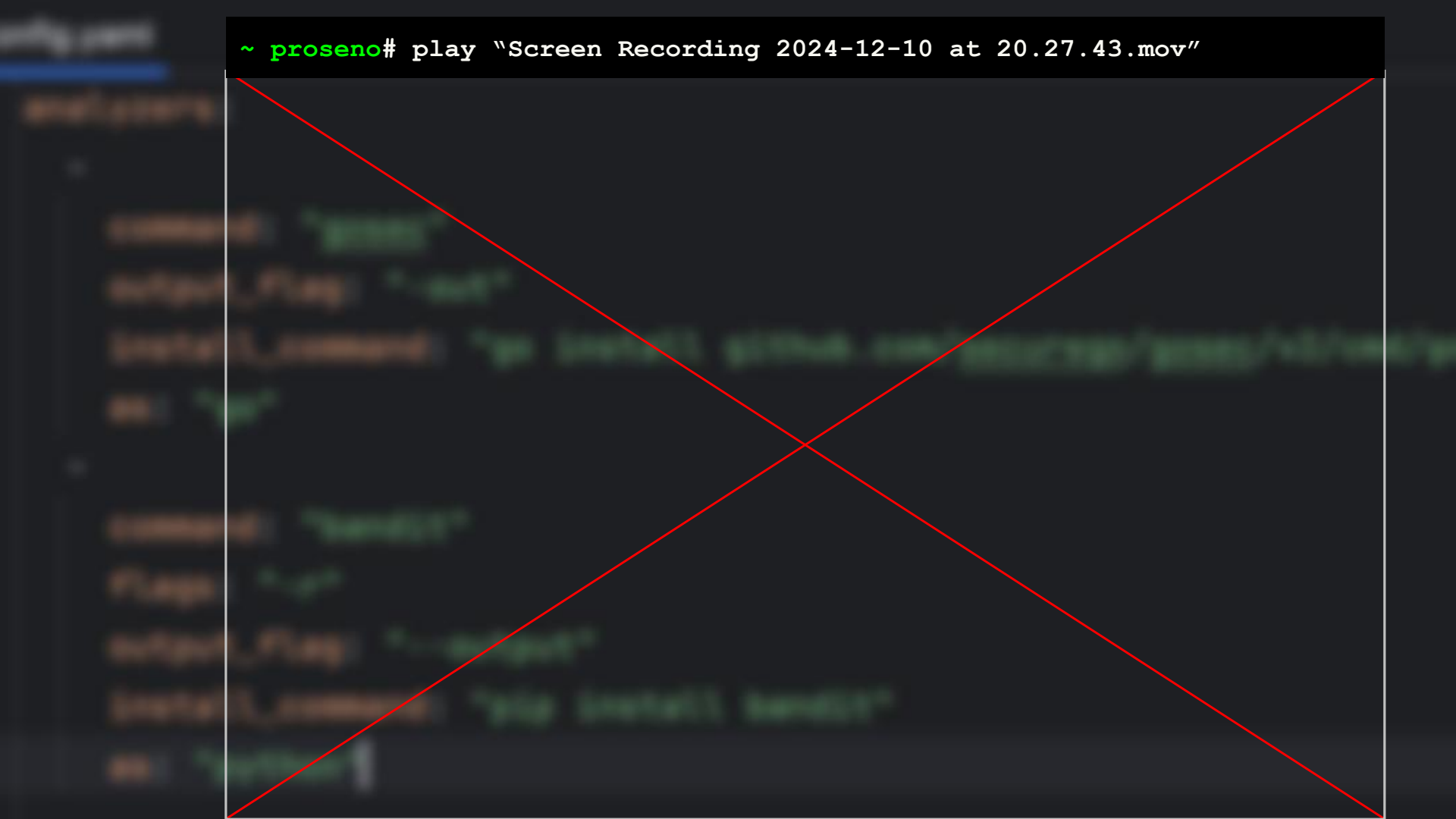
```
~ proseno#
```

```
~ proseno# go run main.go analyze --lang=python --path=examples

Аналіз python коду: /Users/Projects/go/secure-lint/examples

~ proseno#
```

```
13  --------------------------------------------------
14  >> Issue: [B105:hardcoded_password_string] Possible hardcoded password: 'supersecret'
15     Severity: Low   Confidence: Medium
16     CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
17     More Info: https://bandit.readthedocs.io/en/1.7.10/plugins/b105_hardcoded_password_string.html
18     Location: /Users/proseno/Projects/go/secure-lint/examples/test_python.py:10:15
19  9        username = "admin"  # Hardcoded username
20  10       password = "supersecret"  # Hardcoded password
21  11       print(f"Connecting with username: {username} and password: {password}")
22
23  --------------------------------------------------
24  >> Issue: [B608:hardcoded_sql_expressions] Possible SQL injection vector through string-based query construction.
25     Severity: Medium   Confidence: Low
26     CWE: CWE-89 (https://cwe.mitre.org/data/definitions/89.html)
27     More Info: https://bandit.readthedocs.io/en/1.7.10/plugins/b608_hardcoded_sql_expressions.html
28     Location: /Users/proseno/Projects/go/secure-lint/examples/test_python.py:17:12
29  16       cursor = conn.cursor()
30  17       query = f"SELECT * FROM users WHERE username = '{user_input}'"  # Vulnerable to SQL Injection
31  18       cursor.execute(query)
32
```

```
~ proseno# go run main.go analyze --lang=go --path=examples

Аналіз golang коду: /Users/Projects/go/secure-lint/examples

~ proseno#
```

```
[/Users/proseno/Projects/go/secure-lint/examples/test_code.go:47] - G402 (CWE-295): TLS InsecureSkipVerify set true. (Confidence: HIGH, Severity: HIGH)
    46:     tr := &http.Transport{
  > 47:         TLSClientConfig: &tls.Config{InsecureSkipVerify: true}, // Insecure TLS configuration
    48:     }

Autofix:

[/Users/proseno/Projects/go/secure-lint/examples/test_code.go:41] - G401 (CWE-328): Use of weak cryptographic primitive (Confidence: HIGH, Severity: MEDIUM)
    40:     data := []byte("sensitive data")
  > 41:     hash := md5.Sum(data) // Weak cryptographic hash
    42:     fmt.Printf("MD5 hash: %x\n", hash)

Autofix:

[/Users/proseno/Projects/go/secure-lint/examples/test_code.go:29-31] - G201 (CWE-89): SQL string formatting (Confidence: HIGH, Severity: MEDIUM)
    28:     userInput := "'; DROP TABLE users; --"
  > 29:     query := fmt.Sprintf(
  > 30:         "SELECT * FROM users WHERE username='%s'",
  > 31:         userInput) // SQL Injection
    32:     rows, err := db.Query(query)
```


~ proseno# play "Screen Recording 2024-12-10 at 20.27.43.mov"

~ proseno# show Slide_conclusion_final.pptx

Висновки

Для досягнення мети виконана та поставлена задача, що містить наступні результати дослідження:
- розроблено метод оцінки безпеки, яка об'єднує статичний і динамічний аналіз для забезпечення комплексного підходу;
- створено CLI-інструмент, який автоматизує процес перевірки коду та залежностей, а також формує стандартизовані звіти з результатами;
- проведено тестування інструменту на прикладах коду, що показало його ефективність у виявленні вразливостей;
- оцінено переваги та недоліки запропонованого підходу в порівнянні з існуючими методами.

Запропонований метод кардинально спрощує традиційний процес оцінювання безпеки програмного забезпечення, роблячи його більш системним, послідовним і автоматизованим. Завдяки використанню різноманітних інструментів статичного аналізу, впорядкованих за допомогою модульної архітектури, розробники та спеціалісти з безпеки отримують змогу оперативно виявляти критичні вразливості, оптимізувати робочі процеси й уникати зайвих витрат на усунення проблем на пізніх стадіях розвитку проекту.

Отримані результати мають універсальну цінність для широкого кола фахівців, які беруть участь у розробленні та забезпеченні безпеки програмного забезпечення. До них належать розробники, тестувальники, DevOps-інженери, а також спеціалісти з кібербезпеки. Інструмент, створений у ході цієї роботи, легко інтегрується в процес розробки та може використовуватися як постійний моніторинговий механізм безпеки. Це дає змогу оперативно виявляти потенційні загрози й усувати їх на ранніх етапах, тим самим суттєво знижуючи ризик експлуатації вразливостей у майбутньому.

~ proseno#

```
~ proseno# show Slide_8_final.pptx

 Дякую за увагу

Process finished with exit code 0
~ proseno#
```

Завідувачу кафедри кібербезпеки
к.т.н., доц. Кльоцу Ю.П.
Яворського Олександра Віталійовича
ПІБ здобувача вищої освіти

Студента ФІТ, 2 курсу, групи КБЗІм-23-1

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності у Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів дисциплінарної та академічної відповідальності, ознайомлений (а). Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений (а) та надаю свою згоду на обробку й збереження університетом моєї роботи в інституційному репозитарії Хмельницького національного університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-обчислювального комплексу StrikePlagiarism та/або програмнотехнічного засобу Anti-Plagiarism) і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення текстових збігів в роботах.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

09.12.2024
дата

підпис

Протокол аналізу звіту подібності науковим керівником

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Олександр Яворський

Співавтор:

Назва: Метод оцінювання інформаційної безпеки безпечного функціонування програмного забезпечення

Науковий керівник: Володимир Джулій

Підрозділ: Кафедра кібербезпеки

Коефіцієнт подібності 1:0.8%

Коефіцієнт подібності 2:0%

Мікропробіли: 0

Заміна букв: 0

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2024-12-13 13:50:23.0

Після аналізу Звіту подібності констатую наступне:

☑ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обґрунтування:

Дата 13.12.24 експерт

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 0.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 5%**

	Документ		Сумарний збіг по Базі Даних	
ID: 158624 Назва: Метод оцінювання інформаційної безпеки безпечного функціонування програмного забезпечення Додано в БД: 2024-12-13 Автора: Яворський Олександр Керівники: Джулій В.М. Консультанти: Опоненти:	Символи	Лексеми	Символи	Лексеми
	125726	1891	744 (1%)	12 (1%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ

КАФЕДРИ КІБЕРБЕЗПЕКИ

ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:
Назва: Метод оцінювання інформаційної безпеки безпечного функціонування програмного забезпечення

Автор: Яворський Олександр Віталійович

Спеціальність: 125 – Кібербезпека та захист інформації

Освітня програма: Кібербезпека та захист інформації

Науковий керівник: Джулій Володимир Миколайович

Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом (далі – зазначаються підстави віднесення запозичень до правомірних). Робота приймається до захисту.	відповідає
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи (далі – зазначаються детальні та аргументовані підстави віднесення запозичень до правомірних). Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи.	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Оригінальність тексту роботи за результатами перевірки системою StrikePlagiarism складає 99,2%, оригінальність тексту роботи за результатами перевірки системою Anti-Plagiarism v-15.257 складає 100%.

Згідно з правилами чинного Положення «Про систему забезпечення академічної доброчесності у хмельницькому національному університеті» від 24.09.2024, авторська робота, обсяг оригінального тексту у відсотках до загального обсягу матеріалу в якій складає 90-100 %, визнається роботою з високою унікальністю тексту і допускається до захисту.

Керівник роботи _____ Володимир ДЖУЛІЙ

Гарант ОП _____ Віра ТІТОВА

Завідувач кафедри кібербезпеки _____ Юрій КЛЬОЦ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

освітньо-кваліфікаційного рівня «магістр»

Студент Яворський Олександр Віталійович

Тема: «Метод оцінювання інформаційної безпеки безпечного функціонування програмного забезпечення»

Галузь знань 12 «*Інформаційні технології*»

Спеціальність 125 «*Кібербезпека та захист інформації*»

Освітня програма «*Кібербезпека та захист інформації*»

Обсяг дипломної роботи освітньо-кваліфікаційного рівня «магістр»: кількість сторінок записки 87;

1. Короткий зміст КР та прийнятих рішень Кваліфікаційна робота присвячена дослідженню питань, пов'язаних із розробкою та впровадженням методу оцінювання інформаційної безпеки програмного забезпечення з використанням інтеграції автоматизованих інструментів аналізу. Метою роботи є підвищення якості та ефективності процесу оцінювання безпеки за рахунок розробки комплексної методики, що об'єднує статичний та динамічний аналіз коду, а також перевірку залежностей. Для досягнення поставленої мети було проведено аналіз існуючих методів та інструментів оцінювання безпеки програмного забезпечення; розроблено методику, яка дозволяє автоматизувати процес аналізу та мінімізувати людський фактор; реалізовано систему, яка інтегрує різні інструменти аналізу та формує стандартизовані звіти. Оцінено ефективність запропонованого методу шляхом тестування на реальних прикладах коду та порівняння з існуючими рішеннями.

2. Висновок про відповідність КР завданню Кваліфікаційна робота у повній мірі відповідає поставленому завданню як в теоретичній так і у практичній частині роботи.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У вступі обґрунтовується актуальність теми роботи, її зв'язок з галуззю знань «Інформаційні технології» та спеціальністю «Кібербезпека», формулюється мета та основні завдання кваліфікаційної роботи. У першому розділі було проведено аналіз існуючих способів оцінки безпеки програмного забезпечення, що дозволило виявити проблеми та завдання, що потребують вирішення; застосування принципів побудови систем оцінювання ефективності безпеки стало основою для постановки задачі і проектування архітектури системи. У другому розділі було проаналізовано вимоги та потреби бізнесу з урахуванням специфіки, визначено необхідні компоненти та функціонал системи оцінювання ефективності безпеки, побудована модель оцінювання безпеки безпечного функціонування програмного забезпечення. У третьому розділі наведено опис процесу реалізації системи оцінювання ефективності безпеки на підприємстві. У четвертому розділі було розроблено програму за методом, протестовано та наведено результати її роботи.

4. Позитивні сторони кваліфікаційної роботи Запропонований метод оцінювання інформаційної безпеки програмного забезпечення інтегрує статичний та динамічний аналіз коду, що забезпечує комплексний підхід до виявлення вразливостей. На відміну від традиційних методів, система автоматизовано зчитує конфігураційні налаштування, запускає аналізатори та формує стандартизовані звіти, що суттєво знижує трудомісткість процесу та мінімізує людський фактор. Метод дозволяє ефективно працювати з великими кодовими базами, забезпечуючи високу продуктивність та масштабованість системи. Автоматизація процесів забезпечує швидке виявлення актуальних загроз, що підвищує оперативність реагування на кіберзагрози. Запропоновані заходи дозволяють враховувати всі аспекти оцінки безпеки, забезпечують високу точність аналізу та можливість масштабування під потреби різних користувачів, виключаючи необхідність залучення висококваліфікованих спеціалістів для проведення аналізу.

5. Негативні сторони проекту: З роботи не зовсім ясно, яким чином у автоматизованому режимі формується перелік актуальних загроз інформаційній безпеці системи, а також як саме мінімізуються обчислювальні ресурси процесу виявлення цих загроз.

6. Оцінка графічного оформлення та пояснювальної записки роботи.

7. Відгук про роботу в цілому Загалом, кваліфікаційна робота заслуговує на високу оцінку. Матеріал роботи добре структурований, зрозумілий і логічно послідовний. Усі розділи узгоджені між собою, що забезпечує чітке сприйняття викладеного матеріалу в межах обраної тематики. Пояснювальна записка містить багато наочних пояснень. Графічні матеріали надають можливість візуалізувати доцільність та ефективність рішень, які були обрані для досягнення поставлених завдань у процесі проектування.

8. Інші зауваження -

9. Оцінка дипломної роботи Розглянувши позитивні та негативні сторони представленої кваліфікаційної роботи, можна зробити висновок, що робота заслуговує оцінки «відмінно/ A (5,0)».

РЕЦЕНЗЕНТ (прізвище, ім'я, по батькові, посада, місце роботи)

Підченко Сергій Костянтинович, завідувач кафедри ТМІТ, доктор, технічних наук, професор, Хмельницького національного університету

« 16 » грудня 2024 р. _____ (підпис)