***БАКАЛАВРСЬКА РОБОТА***

БР.ІП – 07.00.00.000 ПЗ

**Група ІП-22-1**

**Іванюк Ростислав**

**2026**

**Івано-Франківський національний технічний університет нафти і газу**

**Факультет інформаційних технологій**

**Кафедра інженерії програмного забезпечення**

**Іванюк Ростислав Ігорович**

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

# БАКАЛАВРСЬКА РОБОТА

**Методи тестування безпеки програмного забезпечення**
(назва роботи)

Інженерія програмного забезпечення

( назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

**Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:**

Здобувач освітнього ступеня _____Іванюк Р.І._____
(підпис, ініціали та прізвище здобувача)

Науковий керівник _____ Григорчук Любомир Іванович, доцент, к.п.н._____
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

**Допущено до захисту**
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

**Івано-Франківськ – 2026**

**Івано-Франківський національний технічний університет нафти і газу**

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

**ЗАТВЕРДЖУЮ:**
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____2026 р.

# ЗАВДАННЯ
## НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

**Іванюк Ростиславу Ігоровичу**

(прізвище, ім’я, по-батькові)

**1. Тема проекту (роботи) "Методи тестування безпеки програмного забезпечення "**

керівник проекту (роботи) Григорчук Л.І. доц., к.п.н.

затверджені наказом вищого навчального закладу від “ 21 ” квітня 2026 р. № 179/7

**2. Строк подання студентом проекту (роботи)** 10 червня 2026 р.

**3. Вихідні дані до проекту (роботи)** Результати і матеріали отримані під час проходження переддипломної практики

**4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)**

1. Аналіз вимог до програмного забезпечення
2. Аналіз вимог і постановка задачі
3. Проектування системи
4. Реалізація проекту
5. Тестування та впровадження

**5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)**

1 Життєвий цикл безпечної розробки програмного забезпечення (рис. 1.1, ст. 20)
2 Огляд критеріїв класифікації MBST (рис. 2.1, ст. 25)
3 Огляд основних завдань генетичного алгоритму (рис. 2.3, ст. 40)
4 Діаграма обробки CORAS (рис. 3.2, ст. 51)
5 Фаза тестування вбудованої безпеки (PEST) (рис. 3.5, ст. 59)

**6. Консультанти розділів проекту (роботи)**

| Розділ | Консультант | Підпис, дата | |
|---|---|---|---|
| | | Завдання видав | Завдання прийняв |
| | | | |
| | | | |

**2. Дата видачі завдання** **2026 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

# КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів дипломного проекту (роботи) | Строк виконання етапів проекту | Примітка |
|---|---|---|---|
| 1 | Визначення та обгрунтування теми роботи | 17.03.2026 | виконано |
| 2 | Огляд існуючих концепцій, рішень та сервісів в даній області | 21.04.2026 | виконано |
| 3 | Побудова моделі або алгоритму власного рішення | 01.05.2026 | виконано |
| 4 | Документування реалізації власного оригінального рішення вибраними засобами | 21.05.2026 | виконано |
| 5 | Оформлення пояснювальної записки бакалаврської роботи | 09.06.2026 | виконано |

Студент _____
(підпис)

Керівник роботи _____
(підпис)

# АНОТАЦІЯ

Бакалаврська робота містить 75 сторінок, 15 рисунків, 2 таблиці, список використаних джерел із 35 найменувань.

**Метою роботи** є дослідження методів тестування безпеки програмного забезпечення для виявлення вразливостей, оцінки рівня захищеності систем та підвищення надійності сучасних програмних рішень.

**Об'єкт дослідження:** процеси тестування та забезпечення безпеки програмного забезпечення.

**Предмет дослідження:** методи, моделі та інструменти тестування безпеки програмного забезпечення на різних етапах життєвого циклу розробки, включно зі статичним, динамічним та модельно-орієнтованим тестуванням.

**Результати дослідження**: систематизовано основні методи тестування безпеки, проаналізовано сучасні підходи до виявлення вразливостей та визначено ефективність їх застосування в реальних програмних системах.

**У першому розділі** проведено аналіз теоретичних основ тестування безпеки програмного забезпечення, сучасних загроз та концепції безпечного життєвого циклу розробки.

**У другому розділі** розглянуто методи класифікації, підходи та інструменти тестування безпеки, включаючи динамічне тестування.

**У третьому розділі** досліджено практичне застосування моделей у тестуванні безпеки, фазові підходи та сучасні галузеві практики оцінки безпеки.

**Висновок:** використання сучасних методів тестування безпеки програмного забезпечення дозволяє ефективно виявляти вразливості, підвищувати рівень захисту систем і зменшувати ризики кіберзагроз.

**КЛЮЧОВІ СЛОВА:** БЕЗПЕКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ТЕСТУВАННЯ БЕЗПЕКИ, ВРАЗЛИВОСТІ, SAST, DAST, SDLC, МОДЕЛЬНЕ ТЕСТУВАННЯ, КІБЕРБЕЗПЕКА, OWASP.

# ANNOTATION

The thesis consists of 75 pages, 15 figures, 2 tables, and a reference list with 35 sources.

**The aim of the work** is to study software security testing methods for vulnerability detection, system security assessment, and improvement of modern software reliability.

**Research object:** processes of software security testing and assurance.

**Research subject:** methods, models, and tools of software security testing across different stages of the software development life cycle, including static, dynamic, and model-based testing approaches.

**Research results:** the main software security testing methods were systematized, modern vulnerability detection approaches were analyzed, and their effectiveness in real software systems was evaluated.

**The first section** analyzes theoretical foundations of software security testing, modern threats, and the concept of secure software development lifecycle.

**The second section** describes classification methods, approaches, and tools of security testing, including dynamic testing techniques.

**The third section** examines practical application of models in security testing, phase-based approaches, and industry practices for security assessment.

**Conclusion:** the use of modern software security testing methods enables effective vulnerability detection, improves system protection, and reduces cybersecurity risks.

**KEY WORDS**: SOFTWARE SECURITY, SECURITY TESTING, VULNERABILITIES, SAST, DAST, SDLC, MODEL-BASED TESTING, CYBERSECURITY, OWASP.

# ЗМІСТ

# ВСТУП

У сучасному цифровому середовищі програмне забезпечення стає основою більшості бізнес-процесів, фінансових сервісів та критично важливих інформаційних систем. Зростання кількості кіберзагроз, уразливостей та атак на веб-додатки й корпоративні системи підкреслює необхідність забезпечення високого рівня безпеки ще на етапі розробки. Сучасні виклики, пов’язані з витоками даних, несанкціонованим доступом та складністю архітектур програмних систем, вказують на потребу у впровадженні ефективних методів тестування безпеки програмного забезпечення, які дозволяють виявляти вразливості до їх експлуатації.

**Актуальність теми** зумовлена необхідністю системного підходу до перевірки безпеки програмних продуктів, що включає статичне та динамічне тестування, моделювання загроз і застосування безпечного життєвого циклу розробки. Існуючі підходи до тестування часто застосовуються фрагментарно, що знижує їх ефективність у складних сучасних системах. У контексті стрімкого розвитку кіберзагроз створення та впровадження комплексних методів тестування безпеки є ключовим чинником підвищення надійності програмного забезпечення.

**Метою роботи** є дослідження методів тестування безпеки програмного забезпечення, їх класифікація, аналіз ефективності та особливостей застосування на різних етапах життєвого циклу розробки. Завданнями дослідження є аналіз сучасних підходів до тестування безпеки, визначення основних типів уразливостей, дослідження методів статичного та динамічного аналізу, вивчення безпечного життєвого циклу розробки, а також оцінка ефективності практичного застосування відповідних методів.

**Об’єктом дослідження** є процеси забезпечення та перевірки безпеки програмного забезпечення. Предметом дослідження є методи, моделі та інструменти тестування безпеки, що застосовуються для виявлення вразливостей

у програмних системах. Методи дослідження включають аналіз наукових джерел, порівняльний аналіз підходів до тестування безпеки, моделювання загроз, а також узагальнення практичних результатів використання сучасних інструментів безпеки.

Розроблюваний підхід до тестування безпеки передбачає комплексне застосування різних методів перевірки програмного забезпечення, що дозволяє підвищити рівень захищеності систем, зменшити ризики експлуатації вразливостей та забезпечити відповідність сучасним вимогам кібербезпеки. Очікується, що результати дослідження сприятимуть підвищенню якості та надійності програмних продуктів.

Бакалаврська робота містить 75 сторінки, 15 рисунків, 2 таблиці, 5 розділів, список використаних джерел із 30 найменувань.

# РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ТЕСТУВАННЯ БЕЗПЕКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## 1.1 Основи тестування безпеки програмного забезпечення

Сьогодні промисловість виробляє програмне забезпечення майже для кожної галузі, яка, як правило, є складною та величезною за розміром. Більшість програмного забезпечення відкрито використовується в розподіленому обчислювальному середовищі, що призводить до низки проблем безпеки в області додатків. Ці проблеми легко експлуатуються зловмисниками, тому кількість атак на програмні додатки швидко зростає. Оскільки кількість вразливостей надзвичайно зростає, у майбутньому можуть виникнути серйозні загрози та проблеми, які можуть призвести до серйозних наслідків для додатків. Загрози безпеці також виникають через розподілену та гетерогенну природу додатків, їхні багатомовні та мультимедійні функції, інтерактивну та адаптивну поведінку, постійно розвиваються сторонні продукти та швидкозмінні версії. Нещодавні звіти показують, що загрози безпеці, що оточують фінансові та банківські додатки, різко зросли та продовжують розвиватися.

Зараз вкрай важливо розробляти програмні додатки з високими атрибутами безпеки для забезпечення надійності. З огляду на поточні обставини, безпека є однією з головних проблем для багатьох програмних систем і додатків. Практики вважають, що встановлений периметр безпеки, такий як антивірус, брандмауер, є достатньо безпечним, і вони вважають їх ефективними заходами для вирішення аспектів безпеки додатка. Зі зростанням кількості продуктів безпеки все ще не вистачає уваги вирішенню проблем безпеки в системах онлайн-банкінгу, платіжних шлюзах, страхуванні тощо. Опитування IDG серед керівників IT-відділів та відділів безпеки показує, що майже 63% додатків залишаються неперевіреними на критичні вразливості безпеки [ 1 ].

Безпека – це процес, а не продукт. Її не можна додати або включити до програми під час її завершення, як додаткову думку. Тестування безпеки визначає тести для специфічних та чутливих вимог безпеки до програмного забезпечення [2]. Вимоги безпеки бувають як функціональними, так і нефункціональними, і тому вимагають іншого способу тестування порівняно з тими, що застосовуються для основних функціональних вимог. Тестування безпеки – це спроба виявити ті вразливості, які можуть порушувати цілісність стану, достовірність вхідних даних та логічну правильність, а також вектори кута атаки, які використовують ці вразливості. Крім того, тестування безпеки мінімізує ризик порушення безпеки та забезпечує конфіденційність, цілісність та доступність транзакцій клієнтів. Як тестувальнику, важче розробити вичерпні анти-безпекові вхідні дані та найважче довести, чи достатньо безпечна програма від цього нескінченного набору вхідних даних. Отже, необхідною передумовою є розуміння проблем безпеки та дискусійних питань, щоб зрозуміти існуючі методології та методи, доступні для тестування безпеки.

Сучасні IT-системи, засновані на таких концепціях, як хмарні обчислення, сервіси на основі місцезнаходження або соціальні мережі, постійно підключені до інших систем та обробляють конфіденційні дані. Ці взаємопов'язані системи піддаються атакам безпеки, які можуть призвести до інцидентів безпеки з високим рівнем серйозності, що впливають на технічну інфраструктуру або її середовище. Експлуатовані вразливості безпеки можуть призвести до значних витрат, наприклад, через простої або модифікацію даних. Значна частка всіх інцидентів безпеки програмного забезпечення спричинена зловмисниками, які використовують відомі вразливості [3]. Важливим, ефективним та широко застосовуваним заходом для підвищення безпеки програмного забезпечення є методи тестування безпеки, які виявляють вразливості та забезпечують функціональність безпеки.

У цій роботі тестування стосується активного, динамічного тестування, де поведінка тестованої системи (SUT) перевіряється шляхом застосування нав'язливих тестів, які стимулюють систему, спостерігають та оцінюють реакції

системи [4]. Тестування безпеки перевіряє вимоги до програмної системи, пов'язані з такими властивостями безпеки, як конфіденційність, цілісність, доступність, автентифікація, авторизація та невідмовність [5]. Іноді властивості безпеки є класичними функціональними вимогами, наприклад, «облікові записи користувачів вимикаються після трьох невдалих спроб входу», що наближається до однієї частини властивості авторизації та відповідає стандарту якості програмного забезпечення ISO/IEC 9126, який визначає безпеку як функціональну характеристику якості. Однак, здається бажаним, щоб тестування безпеки було безпосередньо спрямоване на вищезазначені властивості безпеки, на відміну від обходу функціональних тестів механізмів безпеки. Ця точка зору підтримується стандартом ISO/IEC 25010, який переглядає ISO/IEC 9126 та вводить Безпеку як нову характеристику якості, яка більше не включена до характеристики Функціональність.

Оскільки попередній тип (нефункціональних) властивостей безпеки описує всі виконання системи, цей вид тестування безпеки є складним за своєю суттю. Оскільки тестування не може показати відсутність помилок, одразу корисна перспектива безпосередньо розглядає порушення *цих* властивостей. Це призвело до розробки спеціальних методів тестування, таких як тестування на проникнення, яке імітує атаки з метою використання вразливостей. Тести на проникнення важко створити, оскільки тести часто не викликають безпосередньо спостережуваних експлойтів безпеки, а також тому, що тестувальники повинні мислити як зловмисник, що вимагає спеціальних знань. Під час тестування на проникнення тестувальники створюють ментальну модель властивостей безпеки, механізмів безпеки та можливих атак на систему та її середовище. Здається інтуїтивно зрозумілим, що тестування безпеки може отримати користь від визначення цих моделей тестування безпеки у чіткий та оброблюваний спосіб. Моделі тестування безпеки надають рекомендації для систематичної та ефективної специфікації та документування цілей тестування безпеки, тестових випадків безпеки, а також для їх автоматичної генерації та оцінки [6].

Варіант тестування, який спирається на явні моделі, що кодують,

інформацію про тестовану систему та/або її середовище, називається модельно-орієнтованим тестуванням (MBT) [7]. Особливо в контексті тестування безпеки процес розробки тестів, як правило, неструктурований, невідтворюваний, недокументований, не має детальних обґрунтувань для розробки тесту, а також залежить від винахідливості окремих тестувальників або хакерів, що мотивує використання явних моделей.

Модельно-орієнтоване тестування безпеки (MBST) – це модельно-орієнтоване тестування вимог безпеки. MBST – це відносно нова галузь досліджень, яка представляє великий інтерес для промислового застосування, де за останні роки було опубліковано багато підходів та визначних результатів. Щоб окреслити, які частини дослідницької області здаються добре вивченими та оціненими, а які ще ні, систематична класифікація існуючих робіт з MBST видається цінною для спрямування подальших досліджень та сприяння промисловому застосуванню.

Існуючі таксономії для модельного тестування зосереджені на функціональному тестуванні та недостатні для охоплення всіх конкретних аспектів модельного тестування безпеки [8]. Метою цієї роботи є визначення критеріїв класифікації для MBST та надання всебічного огляду сучасного стану в модельному тестуванні безпеки шляхом систематичного віднесення існуючих робіт до цих критеріїв. Визначені критерії класифікації для MBST доповнюють існуючі схеми класифікації для MBT за аспектами, специфічними для безпеки.

*Внесок.* У цій роботі наведено конкретні критерії класифікації для підходів до тестування безпеки на основі моделей, тобто критерії фільтрації та доказів. Схема розширює встановлені існуючі таксономії та класифікації тестування на основі моделей, але додатково враховує специфіку тестування безпеки та доказів підходів до тестування. Визначені критерії оцінюються шляхом систематичної класифікації існуючих підходів до тестування безпеки на основі моделей, описаних у літературі, відповідно до них. Для цієї мети було проведено систематичний пошук та відбір публікацій з тестування безпеки на основі моделей та класифіковано виявлені 119 статей. Для кожної статті можна

було однозначно класифікувати охоплений підхід MBST відповідно до визначених критеріїв фільтрації та доказів [9]. Це вказує на адекватність самої схеми класифікації та забезпечує характеристику сучасного стану тестування безпеки на основі моделей.

### 1.2 Сучасні виклики та загрози безпеці програмного забезпечення

Безпека – це важливий механізм, який обмежує зловмисників від використання програмного забезпечення. Сучасні програми стикаються з кількома проблемами безпеки, які, якщо їх не вирішити належним чином, можуть призвести до серйозних недоліків у програмі. Повідомляється, що дослідницька спільнота сприймає низку відкритих проблем безпеки як виклики. У цій роботі обговорюються проблеми безпеки, які мають значний вплив на програми, такі як складність програмного забезпечення, поширення стороннього коду та динамічні політики безпеки, про які повідомляють різні дослідники та практики [9] .

*Складність програмного забезпечення*

Програмне забезпечення, як процес, так і продукт, стає дедалі складнішим через панівну практику в галузі. Примітивне програмне забезпечення мало обмежені функціональні можливості та обмежену кількість користувачів, але завдяки розвитку технологій та зростанню залежностей від програмного забезпечення, воно є не тільки дуже складним, але й величезним за розміром та практично розширюваним операційним середовищем. Складне програмне забезпечення має більше рядків коду та більше взаємодій між модулями та з середовищем. Користувачам важче зрозуміти складні програми, що ускладнює їх тестування, що зрештою може призвести до більшої ймовірності появи малопомітних помилок у системі в непротестованих частинах. Чим більше помилок у системі, тим більше можливостей для появи вразливостей безпеки. Експерти стверджують, що складність програмного
забезпечення робить його більш вразливим [10].

Міністерства оборони зазначається: «Величезний функціонал та складність ІТ роблять їх легкими для використання та важкими для захисту, що призводить до створення цілі, яку можуть використовувати складні супротивники з національних держав ». Це показує, що складність насправді є ворогом безпеки. Складна система може мати лазівки, які можна використовувати, але важко діагностувати через складність. Mccabe Software стверджує, що складність може залишати лазівки, і зловмисники можуть імплантувати троянський код, який важко ідентифікувати через складність [11]. Це обговорення піднімає очевидне питання, як уникнути або мінімізувати складність, що зараз є одним з дискусійних питань на різних форумах.

У своїй відомій книзі під назвою «Немає чарівної кулі» Букс стверджує, що складність програмного забезпечення є суттєвою властивістю, а не випадковою. Ми не можемо уникнути складності програмного забезпечення, але воно може керувати нею, щоб мінімізувати її наслідки, оскільки складність усвідомлюється як результат кількох елементів, зокрема складності предметної області, складності процесу розробки та його управління, гнучкості, яка можлива завдяки програмному забезпеченню, та проблем характеристики поведінки дискретних систем [12].

*Код третьої сторони*

Сучасні додатки – це сукупність різних компонентів, зібраних з різних джерел, включаючи власні розробки, аутсорсинг, розробки з відкритим кодом, комерційні розробки та інші. Через тиск ринку щодо швидкого та низького виробництва програмного забезпечення, індустрії програмного забезпечення використовують значну частину коду, розробленого третіми сторонами. Згідно з дослідженням постачальника систем безпеки Veracode, присутність стороннього коду в додатках становить серйозну загрозу безпеці для компаній. Помилки в бібліотеці сторонніх розробників можуть призвести до вразливості хост-додатку в цілому [13]. Вразливості безпеки в сторонньому коді викликають серйозні занепокоєння, оскільки ці вразливості можуть впливати на
велику кількість додатків.

Було кілька випадків, коли компанії стикалися із серйозними загрозами безпеці через сторонній код. Нещодавно Twitter виявив недолік безпеки в сторонньому коді, в результаті якого на його сайті використовувалася помилка міжсайтового скриптингу. Сторонній код активує функцію JavaScript під назвою «onmouseover», яка може активувати спливаюче вікно. Цей недолік також можна було використати для перенаправлення користувача на заражений веб-сайт [14].

Очевидно, що сторонній код створює значні проблеми безпеки, але це не означає, що сторонній код взагалі не слід використовувати. Справжнє занепокоєння полягає в тому, чи були заходи безпеки впроваджені третіми сторонами та чи були вони належним чином протестовані незалежно та з додатками. Згідно зі звітом coverity, компанії використовують значну частину програмного коду від кількох третіх сторін, і цей код не тестується з такою ж метою та ретельністю, як програмне забезпечення внутрішньої розробки. Отже, висновок полягає в тому, що якщо використовується сторонній код, його необхідно належним чином протестувати на наявність недоліків безпеки [15].

*Динамічні політики безпеки*

Політики інформаційної безпеки визначають набір політик, прийнятих організацією для захисту своєї інформації, узгоджених з її керівництвом [16]. Вони чітко описують різні сторони з їхніми відповідними частинами інформації на відповідних рівнях. Документ політики інформаційної безпеки включає сферу дії політики, класифікацію інформації, мету управління безпечним поводженням з інформацією в кожному класі та інші.

У традиційних системах ці політики безпеки мають статичний характер, тобто програмісти можуть визначати політики безпеки лише під час компіляції. Однак сучасні системи взаємодіють із зовнішнім середовищем, де політики безпеки не можуть бути відомі заздалегідь та застосовані. Таким чином, політики безпеки мають динамічний характер для таких випадків, як, наприклад, визначення повноважень користувачів залежно від типу повідомлення, отриманого через зовнішнє середовище. Тому потрібен механізм, який дозволить приймати критично важливі рішення щодо безпеки під час виконання на основі

динамічних спостережень за середовищем [17].

*Значення тестування безпеки*

Виявлення вразливостей у програмному забезпеченні є важливою діяльністю, яка також сприяє мінімізації ризику порушення безпеки та забезпеченню конфіденційності, цілісності та доступності транзакцій клієнтів. Тестування безпеки більше не можна сприймати як другорядне завдання, яке потрібно додати або включити в останню хвилину. Тепер це повноцінна діяльність. Дослідники часто виступали за те, щоб його інтегрували в життєвий цикл розробки, і його слід суворо дотримуватися [18]. На жаль, це рідко практикується в галузі. Згідно зі звітом Trustwave про глобальну безпеку, 98% протестованих програм були визнані вразливими, що чітко свідчить про низький рівень серйозності в індустрії програмного забезпечення до тестування безпеки. Правильно виконане тестування безпеки з використанням відповідної методики принесе користь індустрії програмного забезпечення, її клієнтам, а також кінцевим користувачам, сприяючи зміцненню довіри до продукту. У наступних розділах обговорюється важливість тестування безпеки.

Галузь зазвичай вважає, що тестування безпеки не окупить інвестиції. Вони вважають, що завершили функціональне тестування, а середовище оснащене брандмауером та антивірусом, тому немає потреби інвестувати додаткові кошти в тестування безпеки. Але насправді все інакше. Нещодавнє дослідження показує, що три з чотирьох програм перебувають під загрозою атаки, і велика кількість цих атак здійснюється саме на програми, а брандмауери та SSL нічого не можуть зробити в цьому випадку [19]. Насправді, у довгостроковій перспективі галузь отримає такі переваги від тестування безпеки.

*Збережений імідж бренду*

Добре протестований застосунок має менше простоїв і покращує імідж бренду в галузі. Це додасть цінності бренду та опосередковано привабить інших клієнтів. Кірк Герат, головний спеціаліст з питань конфіденційності, помічник віце-президента та заступник головного юрисконсульта Nationwide Insurance у Колумбусі, штат Огайо, каже: «Будь-яке порушення має тенденцію значно

зменшувати ваші витрати на бренд» [20]. З іншого боку, шкода бренду може створювати серйозні проблеми для компаній. Вони можуть втратити довіру існуючих клієнтів, а нові клієнти неохоче використовуватимуть їхні продукти.

*Скорочений час виходу на ринок*

Неперевірені або неправильно протестовані програми потребують виправлення помилок на пізніших етапах, що може вимагати переробки програми. Це зрештою збільшить час виходу на ринок, оскільки переробка програми потребує впровадження, тестування та розгортання цих нових змін. Вплив цих нових змін також необхідно проаналізувати. Всі ці зусилля призведуть до затримки доставки продукту. Тестування безпеки може виявити кілька проблем безпеки на ранніх етапах розробки, які можна виправити з невеликими витратами та зусиллями.

*Нижчі витрати на розробку*

Правильно та ретельно протестовані застосунки зіткнуться з меншою кількістю збоїв; в іншому випадку витрати на виправлення застосунку на пізніших етапах будуть високими, а отже, і вартість розробки в цих випадках буде вищою. Наприклад, виправлення помилки вимоги, виявленої після розгортання застосунку, коштує в 10-100 разів дорожче порівняно з 10 разами, коли помилку виявляли на етапі тестування [21]. Аналогічно, виправлення помилки проектування, виявленої після розгортання застосунку, коштує в 25-100 разів дорожче порівняно з 15 разами, коли помилку виявляли на етапі тестування.

Немає сумнівів, що клієнт є кінцевим і прямим бенефіціаром добре перевіреного якісного продукту. Клієнтом може бути така галузь, як роздрібна торгівля, банківська справа та страхування. Неперевірене програмне забезпечення може бути використане зловмисниками, які матимуть прямий вплив на клієнта. Вони можуть втратити важливі дані та стати жертвою фінансових втрат. Тому клієнти найбільше стурбовані продуктами та постачальниками, які їх постачають. Добре перевірений продукт матиме велике значення для клієнта, зокрема, з таких причин.

*Продукт, стійкий до атак*

Неперевірений або неправильно протестований додаток стане жертвою кількох атак і забезпечить безпечне місце для зловмисників. Належним чином протестований додаток рідко зіткнеться з витоком даних, відмовою в обслуговуванні (DoS) та іншими проблемами безпеки. Під час фази тестування безпеки додаток буде перевірено на наявність широкого спектру вразливостей. Також важливо, щоб галузь залучала експертів з безпеки та використовувала ефективний процес тестування безпеки для створення продукту, стійкого до атак. Клієнт буде впевнений у своєму продукті, і його продукт не стане жертвою атаки.

*Програмне забезпечення кращої якості*

Очевидно, що включення засобів безпеки до продукту підвищить його якість. Тестування безпеки гарантує, що до продукту додано аспекти безпеки. Належним чином перевірений продукт на безпеку матиме мінімальний час простою в роботі та, отже, покращить його якість. З іншого боку, непротестований або неправильно перевірений продукт не матиме якісного аспекту.

*Мінімізує додаткові витрати*

Невеликий недолік у будь-якій частині програми може негативно вплинути на продуктивність критично важливих служб. Для компенсації цієї втрати може знадобитися інше обладнання або програмне забезпечення, що, у свою чергу, потребує більших інвестицій. Один недолік безпеки в програмі може коштувати клієнту мільйони. Дослідження, проведене Ponemon, показує, що порушення безпеки іноді коштує підприємствам в середньому 7,2 мільйона доларів за кожне порушення [21]. Це показує, що з одним збоєм у безпеці пов'язані величезні витрати. У деяких випадках галузі, можливо, доведеться виплатити компенсацію в мільйони доларів.

Кінцеві користувачі – це ті, хто отримає вигоду від галузевого IT-середовища, такі як клієнти онлайн-банкінгу. Якщо програмне забезпечення вразливе до атаки, то зловмисник може викрасти особисті дані, облікові дані та

інші дані. Кінцевий користувач може зіткнутися з перебоями в наданні послуг, а також може стати жертвою фінансових втрат. Тому дуже важливо, щоб кінцеві користувачі отримували належним чином захищені та добре протестовані програми. Нижче наведено значення добре протестованих програм з точки зору кінцевого користувача.

*Безперебійне обслуговування*

Незахищений продукт може стати жертвою DoS-атаки, яка може повністю заблокувати програми. Кінцеві користувачі не зможуть отримати доступ до програми через цей тип атаки. Зловмисники можуть використовувати більш складну атаку, розподілену відмову в обслуговуванні (DDoS). Відомі компанії, такі як Twitter та Facebook, у минулому стикалися з DoS-атаками [22]. Тестування безпеки гарантує, що програми не будуть використані для цих типів атак, і на дуже ранній стадії програми захищені від цих атак.

*Мінімізує ймовірність втрати персональних даних та облікових даних*

Зловмисники дуже зацікавлені в облікових даних користувачів, а також в особистих даних. Вони можуть використовувати ці дані для своїх злих намірів. Згідно з нещодавньою статтею CBS news, особисті дані понад 21 мільйона осіб були викрадені в результаті широкомасштабного витоку даних. Дуже важливо, щоб додаток захищався від таких порушень даних. Додаток має бути належним чином протестований та захищений від таких типів порушень.

## 1.3 Безпечний життєвий цикл розробки програмного забезпечення

Безпека — це не одноразовий підхід після впровадження. Однак, вона включає низку дій, що виконуються протягом усього SDLC, починаючи з самого першого і закінчуючи останньою фазою, що призводить до SSDLC. Безпеку слід сприймати як процес, а не як продукт. Тому її більше не можна розглядати як другорядну думку, яку можна додати або включити до програми в останню хвилину. Галузь, яка застосовує безпеку протягом усього SDLC, показує кращі результати порівняно з тими, хто робить це після розробки.

У 2010 році консалтингова компанія Forrester провела опитування 150 галузей, щоб зрозуміти поточні практики безпеки додатків, ключові тенденції та напрямки розвитку ринку [23]. У цьому опитуванні вони виявили, що галузь, яка практикує SSDLC зокрема, показали кращі результати рентабельності інвестицій порівняно із загальною популяцією. В іншому дослідженні повідомляється, що виправлення вразливостей безпеки, виявлених під час роботи продукту, приблизно в 6,5 разів дорожче, ніж виправлення тих, що знаходяться на ранній стадії SDLC. Експерти з безпеки запропонували кілька методів включення безпеки в SDLC, таких як життєвий цикл розробки безпеки (SDL) від Microsoft, точки контакту McGraw та комплексний процес безпеки легких додатків (CLASP) від OWASP [24]. На рисунку 1.1 зображено фази та набір завдань на кожній фазі SSDLC. Хоча фази зображені тут як традиційна каскадна модель, більшість організацій сьогодні дотримуються ітеративного підходу.

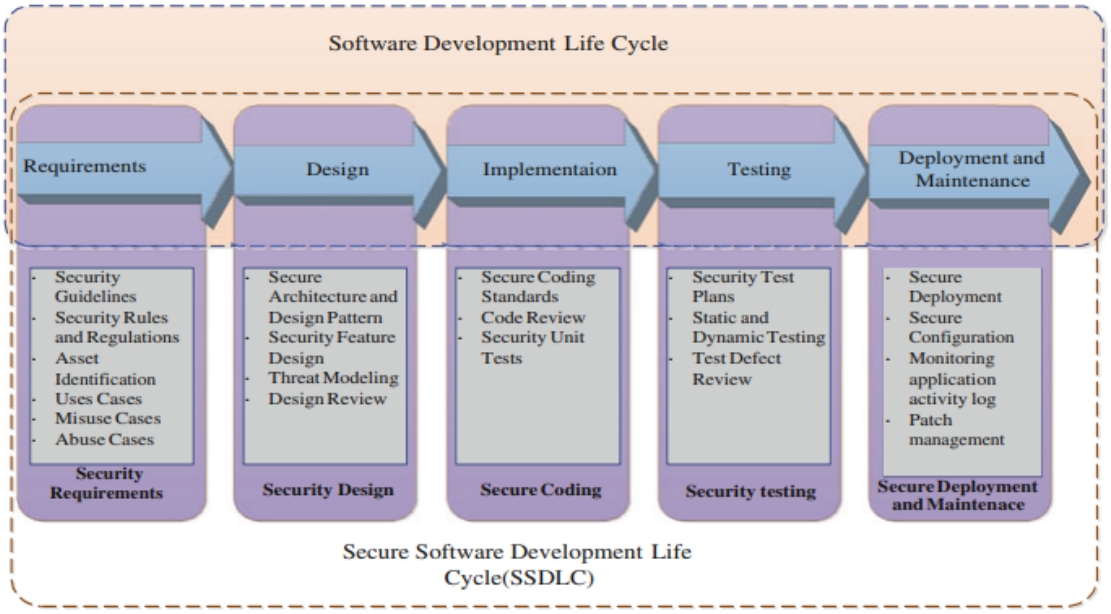


Рисунок 1.1 - Життєвий цикл безпечної розробки програмного забезпечення

SSDLC впроваджує безпеку на самому ранньому етапі життєвого циклу розробки. Він починається з фази вимог, де встановлюються вимоги безпеки, і продовжується фазою проектування, де використовуються безпечна архітектура та шаблони проектування. Далі йде фаза впровадження, де застосовуються стандарти безпечного кодування, потім фаза тестування, де перевіряються

функціональні та нефункціональні проблеми безпеки, і, нарешті, фаза розгортання та обслуговування, де програмне забезпечення розгортається в безпечному середовищі. Ці фази в SSDLC називаються вимогами безпеки, безпечним проектуванням, безпечним кодуванням, тестуванням безпеки та безпечним розгортанням та обслуговуванням. Ми коротко обговоримо кожну фазу SSDLC наступним чином.

*Вимога безпеки*

Спочатку встановлюються політики безпеки для всієї компанії, які забезпечують дозволи на основі ролей, контроль рівнів доступу, контроль паролів тощо. Специфікації вимог ретельно перевіряються, щоб визначити, де можна впровадити безпеку, що називається точками безпеки. На цьому етапі розглядаються рекомендації, стандарти, правила та положення безпеки, що мають значення для конкретної компанії, які разом із політиками безпеки та точками безпеки визначають повні вимоги до безпеки. Деякі зразки вимог безпеки наведено нижче:

- Додаток повинні використовувати лише законні користувачі.
- Кожен користувач повинен мати дійсний обліковий запис, а його рівні привілеїв повинні бути належним чином визначені.
- Додаток надсилає конфіденційні дані різним зацікавленим сторонам через мережу зв'язку, тому слід застосовувати метод шифрування.

Експерти запропонували кілька підходів до забезпечення безпеки на етапі розробки вимог, таких як [25]. Порівняльне дослідження різних методів розробки вимог безпеки].

*Безпечний дизайн*

На етапі проектування експерти з безпеки повинні визначити всі можливі атаки та відповідно спроектувати систему. Ці загрози можна систематично ідентифікувати за допомогою однієї з популярних методик, яка називається моделюванням загроз [26]. Це допомагає розробнику розробляти методи пом'якшення можливих загроз та спрямовує його до зосередження на тій частині системи, яка знаходиться під загрозою. Іншою технікою моделювання

проектування програмного забезпечення є діаграми уніфікованої мови розмітки (UML), які допомагають візуалізувати систему та включати дії, окремі компоненти та їх взаємодію, взаємодію між сутностями та інші. UMLsec, розширення UML для розробки безпечних систем, було запропоновано в літературі.

*Безпечне кодування*

Вразливості безпеки застосунків можна загалом розділити на вразливості рівня проектування та вразливості рівня реалізації [27]. Вразливість рівня проектування виникає через недоліки в проектуванні, такі як використання невідповідної криптографії . Вразливості рівня реалізації існують через недоліки в кодуванні, такі як неправильне використання обробки помилок, необмеження небажаних вхідних даних тощо. Ці недоліки можуть призвести до таких вразливостей, як відмова в обслуговуванні, переповнення буфера. Під час фази безпечного кодування застосовуються методи безпечного кодування для обмеження вразливостей рівня реалізації. SEI CERT CMU надав набір стандартів безпечного кодування для таких мов, як C, C++, Java та інші. Ці стандарти можуть бути застосовані розробником для забезпечення безпеки кодування.

*Тестування безпеки*

Тестування безпеки є важливим етапом SSDLC, оскільки воно допомагає покращити безпеку програмного забезпечення. Його не слід розглядати як необов'язковий/другорядний компонент функціонального тестування, а слід розглядати як повноцінну діяльність. Тестування безпеки включає тестування функціональних аспектів безпеки, таких як тестування контролю доступу, автентифікації, а також тестування на основі ризиків, специфічних для програми, таких як відмова в обслуговуванні, переповнення буфера. У тестуванні безпеки застосовується кілька методів тестування, таких як випадкове тестування, тестування на основі пошуку, символічне виконання та інші, але випадкове тестування є найпопулярнішим, оскільки його легко дотримуватися та виявляє широкий спектр проблем безпеки [28].

*Безпечне розгортання та обслуговування*

Після успішного застосування засобів безпеки на кожному етапі продукт слід встановити в безпечному середовищі. Програмне забезпечення та його середовище необхідно постійно контролювати, і у разі виявлення проблем безпеки в продукті або середовищі слід прийняти рішення про необхідність встановлення виправлень [29]. Програмне забезпечення та його середовище також необхідно періодично оновлювати, щоб вони не постраждали від вразливостей безпеки. Зловмисники використовували застарілі або старіші версії програмного забезпечення, оскільки в цих випадках ймовірність вразливостей висока.

## 1.4 Висновок по розділу

У першому розділі досліджено теоретичні основи тестування безпеки програмного забезпечення. Розглянуто ключові принципи забезпечення інформаційної безпеки, сучасні кіберзагрози та вразливості, які можуть виникати на різних етапах розробки програмних систем. Особливу увагу приділено концепції безпечного життєвого циклу розробки програмного забезпечення, що передбачає інтеграцію заходів безпеки на всіх етапах створення продукту. Проведений аналіз підтвердив важливість систематичного тестування безпеки для підвищення надійності та захищеності програмних рішень.

# РОЗДІЛ 2. МЕТОДИ ТА ІНСТРУМЕНТИ ТЕСТУВАННЯ БЕЗПЕКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## 2.1 Критерії класифікації методів тестування безпеки на основі моделей

MBST – це тестування вимог безпеки на основі моделей. Тому підходи MBST можна класифікувати відповідно до будь-якої схеми класифікації для MBT. Зокрема, явно розглядають підходи MBST, оскільки вони класифікують безпеку як підтип нефункціональних вимог та збирають 16 підходів MBST. Однак жодна з існуючих схем класифікації MBT розглядається конкретні аспекти тестування безпеки, що обговорювалися раніше, включаючи тестування на основі ризиків, впровадження помилок, нечітке тестування, покриття вразливостей та докази. У цьому розділі визначено критерії для проблем безпеки в тестуванні на основі моделей з метою систематичної класифікації підходів MBST. Ці критерії доповнюють існуючі схеми класифікації MBT та узгоджуються з таксономією. Вони спрямовані на характеристику та порівняння існуючих, а також майбутніх підходів та тенденцій MBST.

Література показує, що технологія генерації тестових випадків не є предметно-орієнтованою, тобто не адаптована до тестування безпеки. Тому видається доцільним зосередитися на змодельованих аспектах безпеки для вибору (фільтрації) тестових випадків безпеки та наявних доказів підходів. Це призводить до появи двох груп критеріїв: *критеріїв фільтрації* та *критеріїв доказів.* Кожен критерій має кілька *значень* та відповідний *тип шкали* , який визначає, як значення призначаються певним підходам. Тип шкали - *одинарний вибір* або *множинний вибір.*

Критерії фільтрації включають специфікацію моделі *безпеки системи,* моделі *безпеки середовища* та *явні критерії вибору тестів.* У поєднанні ці моделі визначають специфічні для безпеки фільтри на моделі системи та визначають скінченну підмножину трас SUT, яка становить інтерес для цілей тестування

безпеки. Критерії фільтрації стосуються конкретних методів тестування вразливостей безпеки, наприклад, тестування на основі ризиків, впровадження помилок, нечітке тестування та покриття вразливостей. Критерії фільтрації детально обговорюються .

Критерії доказовості включають *зрілість оцінюваної системи*, *показники доказовості* та *рівень доказовості.* У поєднанні ці критерії оцінюють промислову застосовність та корисність підходів MBST, а також фактичний стан досліджень. Критерії доказовості детально обговорюються .

На рисунку 2.1 наведено огляд критеріїв MBST та їх вимірів, представлених у цьому розділі.

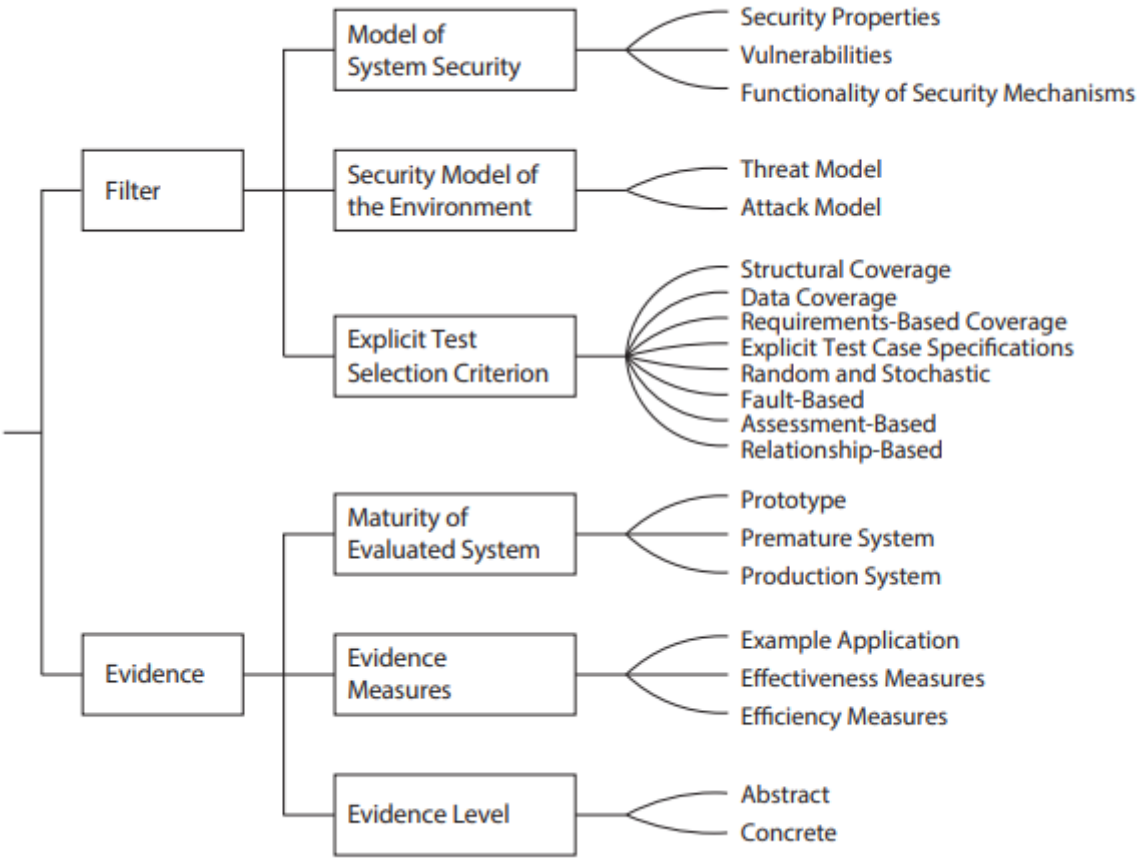


Рисунок 2.1 - Огляд критеріїв класифікації MBST

*Критерії фільтра*

Критерії фільтрації надають засоби для вибору відповідних тестових

випадків. Вони називаються фільтрами, оскільки визначають «цікаві» підмножини всіх трас SUT або моделі цієї SUT. Критерії фільтрації включають моделі безпеки системи та/або її середовища, а також явні критерії вибору тестів. Ці критерії формально визначають *цілі тестування безпеки* та описують , *що* моделюється: частини SUT, її середовище або, безпосередньо, основу для вибору тестових випадків. Далі критерії фільтрації «модель безпеки системи», «модель безпеки середовища» та «явні критерії вибору тестів», а також їх типи шкал та значення описані більш детально.

- *Властивості безпеки:* це моделі властивостей безпеки, таких як конфіденційність, цілісність, доступність, автентифікація, авторизація або невідмовність.
- *Вразливості:* це моделі характеристик системи, без яких певний клас порушень безпеки був би неможливим. Вразливості можна описати подібно до того, як несправності описуються моделями несправностей, зазвичай як відхилення від правильної програми або системи [30]. Прикладами моделей вразливостей є оператори мутації безпеки.
- *Функціональність механізмів безпеки*: це моделі функціональних заходів безпеки, таких як засоби контролю доступу або контролю використання. Модель тут зазвичай надається політикою, яка налаштовує відповідний механізм.
- *Модель системної безпеки* Моделі системної безпеки – це часткові моделі SUT, що зосереджені на аспектах безпеки. Ці моделі складаються з властивостей безпеки, вразливостей та функціональності механізмів безпеки. Оскільки ці моделі можна комбінувати, тип шкали цього критерію – *множинний вибір* . Зокрема, типи моделей системної безпеки такі:

*Модель безпеки середовища* Моделі безпеки середовища – це моделі аспектів безпеки технічного, організаційного та операційного контексту системи, яка потребує перевірки (SUT). Вони зосереджені на причинах та потенційних наслідках поведінки системи та інтерфейсів до SUT. Зверніть увагу на різницю між моделюванням вразливості, тобто частини (часткової) моделі

SUT, та моделюванням експлойтів цієї вразливості, тобто частин середовища SUT. Моделі середовища для тестування безпеки завжди пов'язані з конкретними вразливостями та описують, як орієнтуватися на конкретну вразливість (або клас вразливостей). Тип масштабу моделей безпеки середовища – це *багатовибірковий* зі значеннями моделі загрози та моделі атаки, які визначаються наступним чином:

- *Моделі загроз:* Ці моделі описують загрози, тобто потенційні причини інциденту, які можуть призвести до шкоди системам та організації [31].
- *Моделі атак:* Ці моделі описують послідовності дій для використання вразливостей. Загрози можуть розглядатися як їх причини. Моделі атак можуть бути стратегіями тестування, такими як фаззинг рядків або поведінки, можливо, на основі граматик, або синтаксичних моделей, що генерують тестові випадки.

Зв'язок між загрозами та атаками можна проілюструвати за допомогою дерев атак: Дерево атак являє собою концептуальне уявлення про те, як може бути атакований актив. Воно описує комбінацію загрози та відповідних атак. Зазвичай подія верхнього рівня (кореневий вузол) дерева атак представляє загрозу. Навпаки, кожен розріз дерева атак, з якого видалено кореневий вузол, є атакою. Кілька некореневих вузлів дерева використовують вразливості. Дочірні вузли вузла виражають умови, які необхідно виконати, щоб використати вразливість на батьківському вузлі. Якщо всі вузли в розрізі (включаючи кореневий вузол) є істинними, відповідна атака можлива та призводить до описаної загрози. Зауважте, що вразливості невід'ємно пов'язані з SUT, тоді як атаки невід'ємно пов'язані з середовищем SUT. Вибір тестів здійснюється з використанням обох.

*Явні* критерії вибору тестів Критерії вибору тестів визначають, як ще більше зменшити кількість можливих трас, визначених властивостями, механізмами безпеки, вразливостями та атаками. Тільки якщо ці набори достатньо малі, їх можна використовувати для цілей тестування. Вони є

моделями того, що робить траси SUT цікавими як тестові випадки. Типи явних критеріїв вибору тестів такі:

- *Структурне покриття:* Ці критерії використовують структуру моделі, таку як вузли та дуги моделі на основі переходів, або умовні оператори в моделі в нотації "до/після".
- *Охоплення даних:* Ці критерії стосуються того, як вибрати кілька тестових значень з великого простору даних.
- *Покриття на основі вимог:* ці критерії стосуються покриття неформальних вимог, що можливо, коли елементи моделі можна явно пов'язати з неформальними вимогами.
- *Явні специфікації тестових випадків:* Специфікації тестових випадків у певній формальній нотації використовуються для визначення того, які тести будуть згенеровані. Наприклад, вони можуть бути використані для обмеження шляхів через модель, які будуть тестуватися, для зосередження тестування на критичних сценаріях або для забезпечення тестування певних шляхів.
- *Випадкові та стохастичні:* ці критерії враховують прямо або опосередковано змодельовані ймовірності, такі як ймовірності дій, що моделюють профіль використання, або випадково згенеровані недійсні чи неочікувані вхідні дані.
- *на основі помилок:* Ці критерії стосуються явного покриття ненавмисно доданих та штучно введених помилок. Найпоширенішими критеріями на основі помилок є критерії *покриття мутацій* . Це передбачає мутацію моделі, а потім створення тестів, які розрізняють мутовану модель та вихідну модель. Мутанти схожі на вразливості, а згенеровані тести імітують атаки.
- *На основі оцінювання:* Ці критерії вибирають тестові випадки на основі розрахованих та/або оцінених значень, пов'язаних з тестовими випадками. Популярні критерії вибору тестових випадків на основі оцінювання базуються на значеннях ризику, які *визначають ймовірності та наслідки певних загроз,* що

розглядаються в тестових випадках. Інші показники для оцінювання можуть включати витрати, пріоритети або серйозність, а також розмір, складність або показники змін.

- *На основі зв'язків:* Ці критерії вибирають тести на основі зв'язків між моделями. Ці моделі можуть бути на різних рівнях абстракції, таких як модель рівня проектування та реалізації, або бути різними версіями однієї моделі у випадку *регресійного тестування* .

Оскільки ці моделі можна комбінувати, тип шкали цього критерію – *множинний вибір* . Наприклад, *нечітке тестування* [32] є ефективним методом пошуку вразливостей безпеки та надає приклад того, як поєднуються критерії вибору тестів. Традиційно інструменти нечіткого тестування застосовують випадкові мутації до добре сформованих вхідних даних програми та тестують отримані значення для виявлення вразливостей. Таким чином, нечітке тестування поєднує вибір тестів на основі помилок та випадковий вибір тестів.

Перелічені явні критерії вибору тестів базуються на вимірі вибору тестів таксономії MBT Уттінга та ін. Крім того, він містить додатковий критерій «на основі ризику», який має особливе значення для тестування безпеки та «на основі зв'язків», щоб охопити зв'язки між моделями різних рівнів абстракції та версій [33].

*Зв'язок* Вище було описано, як комбінація моделі безпеки системи, моделі безпеки середовища та критеріїв вибору тестів визначає *фільтри,* відповідно до яких вибираються або генеруються тести безпеки. Такий фільтр визначає відповідну скінченну підмножину та скорочення можливих трас виконання SUT і може бути досягнутий на основі кількох операцій:

- *Зв'язок властивості безпеки зі специфікацією SUT:* Властивості безпеки, виражені як твердження в будь-якій формі математичної логіки, можуть бути накладені шляхом обчислення їх логічного зв'язку з (формальною) специфікацією SUT. Зазвичай це інтерпретується як перетин наборів трас властивості та специфікації SUT. Шляхом перетину зі трасами властивості безпеки, набір трас SUT може лише зменшитися або залишитися ідентичним.

Для цілей тестування, спрямованих на порушення властивостей, часто використовуються заперечені або мутовані властивості.

- *Використання вразливостей:* Щоб використати вразливості, систему спочатку потрібно привести у стан, у якому можливо виконати експлойт. Тому моделі вразливостей обмежують кількість трас тими, що відвідують стани, в яких потенційно можуть бути використані вразливості.
- *Визначення функціональності механізмів безпеки:* Модель функціональності механізму безпеки, наприклад, механізму контролю доступу, зменшує можливі сліди, забороняючи ті системні переходи, які не дозволені політикою. Для цілей тестування часто використовуються заперечені або мутовані політики.
- *Складання моделі SUT з моделлю середовища:* це досягається шляхом з'єднання вхідних каналів моделі SUT з вихідними каналами моделі середовища, а також шляхом з'єднання вихідних каналів моделі SUT з вхідними каналами моделі середовища. Якщо модель середовища не дозволяє надсилати довільні послідовності вхідних даних до SUT - а це зазвичай є метою моделі середовища - тоді таке складання зменшує набір відповідних трас SUT до тих, які дозволені середовищем.
- *Явне визначення критеріїв вибору:* Явно визначені критерії вибору тестів визначають, як ще більше скоротити набір трас SUT. Наприклад, можна вказати, що виконуються лише команди, визначені як критичні, що фільтр, застосований до всіх трас, видає всі ті траси, які відповідають критерію покриття, або що повторення подій не може відбуватися.

У літературі загрози, атаки та вразливості кодуються всіма цими засобами та їх комбінаціями. Зауважте, що різні явні критерії вибору тестів можуть мати різну застосовність для поєднання з моделями системи або середовища. Наприклад, критерії на основі несправностей здебільшого застосовні до системних моделей, оскільки метою є пошук несправностей у SUT. Але випадкові та стохастичні критерії здебільшого застосовні до моделей

середовища, оскільки саме середовище описує моделі використання та вхідні дані SUT.

*Критерії доказів*

Критерії доказів визначають, скільки доказів доступно щодо корисності підходу MBST. Оскільки докази мають кілька аспектів, ці критерії включають *зрілість оцінюваної системи* , *міри доказів* , а також *рівень доказів* і оцінюються для різних застосувань підходу MBST. Докази не є специфічними для MBST, але стосуються MBT загалом. Це підтверджується тим фактом, що Діас-Нето та Травассос вже розглядають критерій класифікації «аналіз за типом експериментальних доказів» зі значеннями категорій «спекуляція», «приклад», «доказ концепції», «досвід/промислові звіти», а також «експериментування». Але особливо для тестування безпеки, де вибір та застосування відповідних методів тестування є дуже критичним та складним, документування доказів вимагає більш багатогранної схеми класифікації. Тому визначено наступні три *ортогональні* критерії доказів, які допомагають у цьому відношенні: зрілість оцінюваної системи, міри доказів та рівень доказів. Враховуючи не лише промислове застосування, але й докази, про які повідомлялося в дослідженнях, кожен із трьох критеріїв доказів є *необов'язковим*, оскільки конкретні результати можуть бути недоступні, як у спекулятивних підходах. Визначені критерії враховують відповідну схему класифікації для регресійного тестування безпеки, де підходи до регресійного тестування безпеки класифікуються з урахуванням критеріїв доказів оцінюваної системи, зрілості оцінюваної системи, а також заходів оцінки, та адаптують їх до тестування безпеки на основі моделей та рівня абстракції, необхідного для загальної таксономії. Далі обговорюються критерії зрілості оцінюваної системи, заходи доказів та рівень доказів, а також їх відповідні шкали та значення [34].

*Зрілість оцінюваної системи* Критерій зрілості оцінюваної системи відображає технічну обґрунтованість та ступінь розгортання найзрілішої системи, що тестується, що використовується для оцінки підходу MBST. Його шкала є *одноразовою* зі значеннями прототип, передчасна система та виробнича

система, які визначаються наступним чином:

- *Прототип:* це ранній зразок або реліз системи з метою її тиражування, вивчення або оцінки інших підходів. Прототипи мають деякі відомі обмеження (наприклад, щодо безпеки). Вони перебувають на стадії розробки або були розроблені без промислового чи технічного тиску.
- *Передчасна система:* це працююча система, яка ще не виконує завдання, цінні для зацікавлених сторін, на регулярній основі. Це проміжне значення вводиться для систем, які вже є більш зрілими, ніж прототип, але ще непродуктивними.
- *Виробнича система:* це працююча система, яка регулярно виконує завдання, цінні для зацікавлених сторін.

*Показники доказів* Показники доказів визначають якісні або кількісні критерії оцінки для оцінки підходу MBST у конкретному застосуванні. Ці показники є «прикладом застосування», а також, з підвищеною значущістю, «показниками ефективності», а також «показниками ефективності». Оскільки ці показники можна комбінувати, тип шкали цього критерію є *багатовибірковим* , а його значення є прикладом застосування, показниками ефективності та показниками ефективності, визначеними наступним чином [35]:

- *Приклад застосування*: Підхід було використано в задокументованому контексті, що вказує на доцільність застосування підходу, як правило, якісним способом.
- *Показники ефективності:* Загалом, вони вимірюють, чи досягається передбачуваний або очікуваний результат (ефект). У контексті тестування програмного забезпечення показники ефективності оцінюють вплив тестів на систему, що тестується. Загальним показником ефективності є кількість знайдених помилок; іноді також поділена на кількість виконаних тестових випадків. Для цілей тестування безпеки, особливо цікавить кількість знайдених вразливостей або мутацій.
- *Показники ефективності:* Загалом, вони вимірюють виконання або функціонування найкращим чином з найменшими витратами часу та зусиль. У

контексті тестування програмного забезпечення ефективність вимірюється шляхом співвіднесення артефактів, таких як помилки, тестові випадки або елементи протестованої моделі, з необхідним часом або витратами, в ідеалі шляхом порівняння з іншими підходами. Для цілей тестування безпеки, особливо цікавим є кількість знайдених вразливостей або мутацій відносно часу або витрат, необхідних для їх ідентифікації.

*Рівень доказовості.* Рівень доказовості визначає, чи оцінюється підхід на рівні невиконуваних абстрактних чи виконуваних конкретних тестових випадків [36]. Оцінювання на абстрактному рівні та на рівні виконуваних тестових випадків може бути комбінованим. Таким чином, шкала цього критерію є *багатовибірковою* зі значеннями абстрактний та виконуваний, які визначаються наступним чином:

- *Анотація:* На цьому рівні підхід оцінюється на основі невиконуваних тестових випадків. Наприклад, якщо згенеровані тестові випадки є послідовностями невиконуваних вузлів моделі на основі переходів, тоді вже можна виміряти докази (ефективність можна, наприклад, виміряти на основі моделі витрат або часу). Але конкретний вплив підходу на розгорнуту систему, що тестується, виміряти неможливо.

• *Виконуваний:* На цьому рівні підхід оцінюється на основі тестових випадків, які можна виконати на тестованій системі. Абстрактні тестові випадки часто генеруються як проміжний крок. Порівняно з абстрактним рівнем, набір можливих мір доказів не обмежений [37].

## 2.2 Підходи до тестування безпеки програмного забезпечення

Тестування безпеки – це важлива діяльність для виявлення вразливостей, використання яких може завдати шкоди системі. Це підвищує якість продукту та цінність бренду компанії. Щоденне збільшення використання програмного забезпечення та використання складних технологій значною мірою сприяють успіху зловмисників у їхніх злих намірах. Тому важливо належним чином

протестувати програмне забезпечення на наявність вразливостей безпеки, щоб захистити його від зловмисників .

У літературі дослідники запропонували кілька підходів до тестування безпеки для виявлення проблем безпеки. Тестування безпеки можна загалом розділити на статичне та динамічне залежно від складності програми та типу вразливостей, які потрібно знайти. Статичне тестування проводиться без запуску програмного забезпечення, таке як огляд коду, статичний аналіз, тоді як динамічне тестування виконується шляхом запуску програмного забезпечення. У цьому розділі ми обговорюємо ці широкі категорії підходів до тестування безпеки. Ми включаємо тематичні дослідження програм Java, кожне з яких належить до категорії статичного та динамічного тестування безпеки. Ці програми тестуються за допомогою інструменту FindBugs, інструменту статичного аналізу, та Symbolic PathFinder, конколічного виконавця. Хоча тут обговорювалися інші методи тестування в рамках цих категорій, ми застосували до програм Java лише інструмент статичного аналізу та інструмент конколічного виконання [38].

*Статичне тестування безпеки*

Статичне тестування безпеки автоматично виявляє вразливості безпеки в програмному забезпеченні без запуску програм. Деякі методи статичного тестування використовують вихідний код для аналізу, а інші використовують об'єктний код. Статичні методи можна застосовувати до дуже великих програм, але вони можуть страждати від генерації хибнопозитивних, хибнонегативних або хибнонегативних попереджень. Тут ми обговорюємо основні категорії статичного тестування безпеки, такі як перевірка коду, перевірка моделей та символічне виконання. Перевірка моделей та символічне виконання належать до методів статичного аналізу коду.

*Перевірка коду*

Перевірка коду передбачає тестування застосунку шляхом перевірки його коду. Її метою є пошук та виправлення помилок, допущених на початковому етапі розробки програмного забезпечення, і, отже, покращення загальної якості

програмного забезпечення. Перевірку коду можна виконувати як вручну, так і за допомогою автоматизованих інструментів. Ручна перевірка коду займає багато часу і потребує сканування коду рядок за рядком, тоді як автоматизована перевірка коду автоматично сканує вихідний код, щоб перевірити, чи було застосовано заздалегідь визначений набір найкращих практик.

*Перевірка моделі*

Перевірник моделі автоматично перевіряє, чи відповідає задана модель системи специфікації заданої системи. Специфікація може містити вимоги безпеки, захисту або інші вимоги, які можуть призвести до аномальної поведінки системи. Системи зазвичай моделюються за допомогою кінцевих автоматів, які виступають вхідними даними для перевірника моделі разом із набором властивостей, зазвичай виражених у вигляді формул часової логіки [39]. Перевірник моделі перевіряє, чи властивості виконуються системою, чи порушуються вони.

Перевірка моделі працює наступним чином. Припустимо, що *M* - це модель, тобто граф переходів станів, а *p* - властивість у темпоральній логіці. Отже, перевірка моделі полягає у знаходженні всіх станів s таких, що M має властивість p у стані s. Важливий інструмент Java Path Finder (JPF), заснований на перевірці моделі, є широко використовуваним дослідницьким інструментом для різних режимів виконання та розширень. У літературі було запропоновано кілька інструментів тестування на основі JPF, таких як Symbolic PathFinder, Exception Injector та інші. Повний список таких методів/інструментів можна знайти на сайті JPF.

*Символічне виконання*

Ця техніка генерує тестові випадки для кожного шляху шляхом генерації та розв'язання умови шляху (PC), яка фактично є обмеженням на вхідні символи. Кожна точка розгалуження програми має свою власну умову шляху, а PC на вищому рівні є агрегацією поточної гілки та гілок на попередньому рівні. Класичне символічне виконання внутрішньо містить два
компоненти:

- Генерація умови шляху : Припустимо, що оператор задано як *if(cond) then S1 else S2.* PC для гілки цього оператора буде *PC ! PC Л cond* (для справжньої гілки) та *PC ! PC Л - cond* (для хибної гілки).
- Розв'язувач умов шляху : Розв'язувач умов шляху або розв'язувач обмежень використовується для двох цілей: по-перше, для перевірки доцільності шляху, а по-друге, для генерації вхідних даних для програми, яка задовольняє ці обмеження. Розв'язання обмежень є основною проблемою символьного виконання, оскільки вартість розв'язання обмежень домінує над усім іншим, і причина полягає в тому, що розв'язання обмежень є NP-повною задачею.

Класичне символічне виконання забезпечує міцну основу для динамічного тестування, такого як *конколічне тестування.* У літературі запропоновано безліч робіт, заснованих на *конколічному тестуванні . Ми обговоримо конколічне тестування в наступному розділі.*

Таблиця 2.1

Помилки, знайдені за допомогою інструменту Findbugs

| **Назва Java-пакету** | **Кількість рядків коду (LOC)** | **Кількість класів** | **Всього знайдено помилок** |
|---|---|---|---|
| BareHTTP | 717 | 6 | 14 |

*Тематичне дослідження*

Щоб продемонструвати фактичну роботу інструменту тестування, ми розглянемо тематичне дослідження пакета Java, а саме BareHTTP [40]. Ми застосували популярний інструмент статичного тестування Findbugs, інструмент статичного аналізу коду Java на пакеті BareHTTP, який видає результати, наведені в таблиці 1. Findbugs приймає програми Java як вхідні дані та генерує різні типи помилок, такі як порожні паролі бази даних, вразливість XSS, що відображається в JSP, тощо.

Не всі помилки, знайдені за допомогою Findbugs, належать до категорії проблем безпеки. Ретельний аналіз звіту про помилки, згенерованого для

BareHTTP, показує, що деякі помилки виникають через те, що об'єкт потоку вводу-виводу не закритий. Це може призвести до витоку файлового дескриптора, а в крайньому випадку - до збою програм. Інструмент, про який йдеться, генерує широкий список помилок, і вони класифікуються за такими категоріями: погані практики, проблеми з продуктивністю, сумнівний код, вразливості шкідливого коду тощо. Усі помилки, створені Findbugs, не є фактичними помилками, і він може генерувати кілька хибнопозитивних результатів.

### 2.3 Методи динамічного тестування безпеки

Динамічне тестування безпеки включає тестування програм на безпеку в їхньому робочому стані. Воно може виявити недоліки або вразливості, які є занадто складними для статичного аналізу. Динамічне тестування виявляє реальні помилки, але може пропустити певні помилки через пропуск певних шляхів. Динамічне тестування включає використання різноманітних методів, серед яких найпопулярнішими є нечітке тестування, конколічне тестування, тестування на основі пошуку. Кожен з них цінний з певної точки зору, такої як ефективність тестування, генерація тестових випадків , покриття вразливостей. Кожен з цих методів описано далі разом з останніми дослідженнями в цих напрямках.

*Тестування нечіткості*

Фуз-тестування (випадкове тестування) – це тип тестування, при якому програма бомбардується тестовими випадками, згенерованими іншою програмою [12]. Програма, яка генерує тестові випадки, називається генератором фуз-тестування, який генерує випадкові, недійсні або неочікувані дані для тестування програми. Метою фуз-тестування є аварійне завершення роботи програми, щоб виявити діри в безпеці. Ранні методи фуз-тестування були простими та використовувалися для генерації випадкових чисел як вхідних даних для програми. Сучасні методи фуз-тестування є більш складними за своєю природою та генерують більш структуровані вхідні дані як тестові випадки.

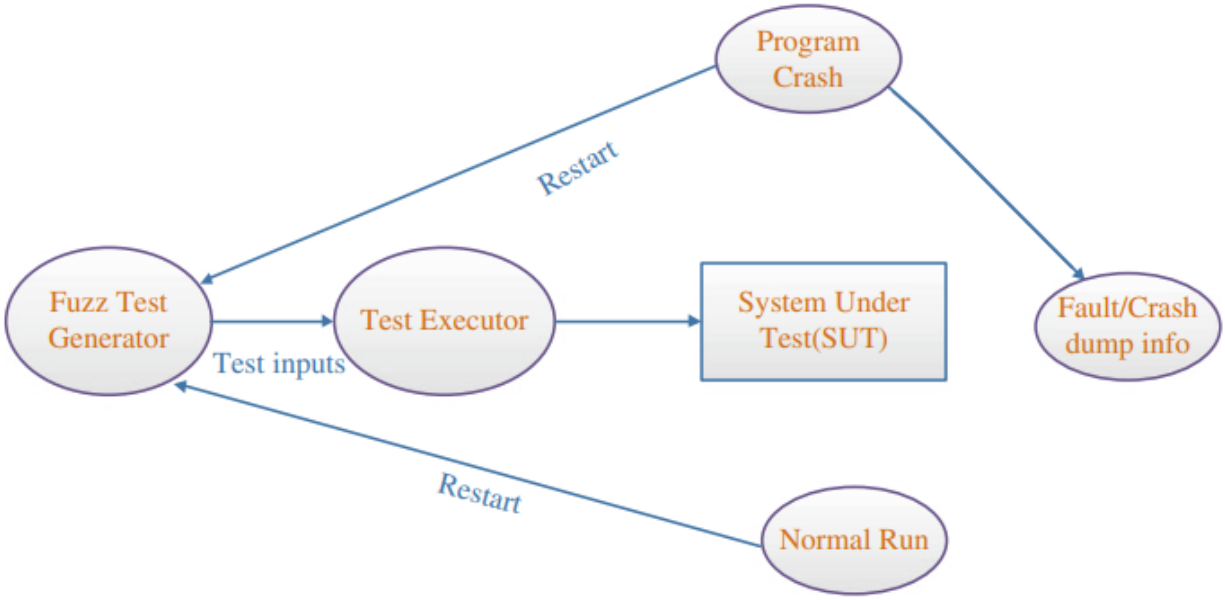


Рисунок 2.2 - Основна схема Fuzz-тестування

Фуз-тестування можна загалом розділити на фуз-тестування чорної скриньки та фуз-тестування білої скриньки. Фуз-тестування чорної скриньки генерує вхідні дані без знання внутрішніх механізмів програми, з іншого боку, фуз-тестування білої скриньки також враховує внутрішні механізми програми. Що стосується охоплення фуз-тестуванням, воно може виявити кілька проблем безпеки, таких як переповнення буфера, витоки пам'яті, DoS тощо [23]. Процес фуз-тестування можна краще зобразити на рис. 2.3. На цьому рисунку показано, що генератор фуз-тестів неодноразово генерує тестові вхідні дані, і програма виконується з цими вхідними даними за допомогою виконавця тестів. Також ведеться журнал для зберігання даних про помилки/збої.

Дослідники запропонували низку методів нечіткого тестування, таких як, для тестування програм на наявність помилок, що призводять до аварійного завершення роботи програми. Серед них популярними методами є JCrasher – автоматизований інструмент тестування, який перевіряє наявність неоголошених винятків під час виконання, виконання яких призведе до аварійного завершення роботи програми, та CnC – ще один метод випадкового

тестування, який поєднує статичне тестування та автоматичну генерацію тестів для пошуку сценарію аварійного завершення роботи програми.

*Конколічне тестування*

Конколічне тестування – це тип тестування, який під час виконання програми збирає обмеження шляху вздовж виконаних шляхів, і ці зібрані обмеження вздовж шляху розв'язуються для отримання нових вхідних даних для альтернативного шляху [19]. Цей процес повторюється, доки не будуть охоплені всі критерії покриття. По суті, це розширення класичного символічного виконання, яке динамічно генерує вхідні дані шляхом розв'язання обмежень шляху.

Конколічне тестування виявилося дуже успішним у тестуванні безпеки, оскільки воно виконує кожен шлях у програмі та гарантує, що програма не містить темних шляхів (неперевірених та забутих шляхів). Цей метод тестування автоматично виявляє критичні випадки, коли програміст неправильно обробляє винятки або не виділить пам'ять, що може призвести до вразливостей безпеки. Було розроблено кілька методів/інструментів з використанням конколічного виконання, таких як Cute та JCute, Symbolic JPF та інші. Ці методи дуже ефективні в тестуванні безпеки програм, оскільки вони тестують усі шляхи і, отже, уникають непровірених шляхів, які можуть призвести до вразливостей безпеки.

*Тестування безпеки на основі пошуку*

Це застосування методу метаевристичної оптимізації пошуку для автоматизації завдання тестування. Він використовує критерій адекватності тесту, відомий як функція придатності, для автоматичної генерації тестових даних. Отже, метою є задоволення функції придатності, починаючи з початкових вхідних даних та постійно уточнюючи вхідні дані, які максимізують задовільність функції придатності. Хоча існує кілька алгоритмів на основі пошуку, які застосовувалися в тестуванні безпеки програмного забезпечення, генетичний алгоритм є найпопулярнішим серед них. На рисунку 4 представлено огляд основних завдань генетичного алгоритму.

Було розроблено кілька методів з використанням алгоритмів на основі пошуку для тестування проблем безпеки , таких як переповнення буфера, SQLI, збої програми та інші. В одній із цих статей було протестовано сценарій збою програми через виняток ділення на нуль. У їхньому методі програма спочатку перетворюється таким чином, що вираз, який призводить до збою, стає умовою. Використовуючи цю умову, тестові дані генеруються за допомогою генетичного алгоритму.

*Тематичне дослідження*

Ми розглянули випадок програми, наведений у лістингу, яка генерує неперехоплений виняток "ділення на нуль". Результати неперехоплених або неправильно оброблених винятків можуть призвести до серйозних проблем безпеки [23]. Це може дозволити зловмиснику спричинити збій програми, що зрештою призведе до DoS. Кілька випадків винятків "ділення на нуль" було зареєстровано в базах даних вразливостей, таких як OSVDB, NVD [25]. Згідно з записом бази даних, 9 таких випадків припадає на 2015 рік, а 15 випадків - на 2014 рік [36]. Наприклад, обробки вимірів, які може використовувати контекстно-залежний зловмисник [16].

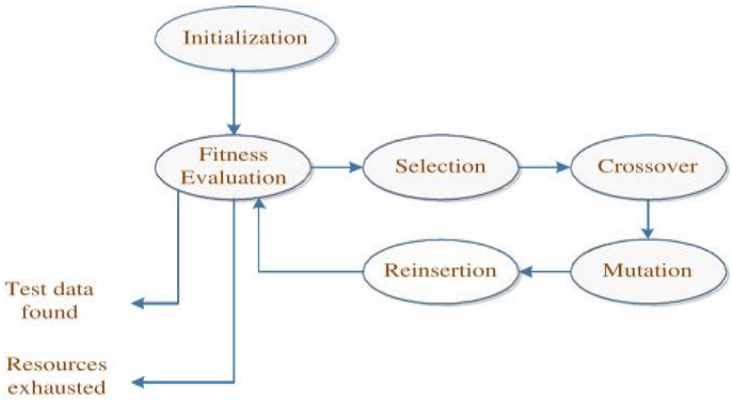


Рисунок 2.3 - Огляд основних завдань генетичного алгоритму

У програмі, згаданій у лістингу, виняток ділення на нуль не оброблено у рядках 7 та 12, що зрештою призведе до неперехопленого винятку. Цю програму було протестовано за допомогою Symbolic Java PathFinder [38]. Символічний Java PathFinder розроблено як розширення JPF, яке виконує програму на

символічних вхідних даних. Вхідні дані представлені як обмеження, згенеровані за допомогою умов у кодах, які згодом розв'язуються для генерації тестових вхідних даних.

```
1  class symexe{
2    int i,j,z;
3    void foo(int i,int j){
4      if (i>j)
5      {
6   System.out.println("i is greater than j");
7   z=i/(j-8);
8      }
9      else
10     {
11   System.out.println("j is greater than i");
12   z=j/(i-4);
13     }
14     if (z>5)
15   System.out.println("z is greater than limit");
16     else
17   System.out.println("z is less than the limit");
18   }
19
20     public static void main(String[] args){
21       symexe s=new symexe();
22       int p,k;
23       s.foo(12,6);
24     }
25   }
```

Рисунок 2.4 – Приклад перехоплення винятку

Якщо ми створимо Символічне дерево вищезгаданої програми, то вона згенерує шість шляхів, два з яких генеруватимуть неперехоплені винятки. Symbolic PathFinder генерує обмеження цих шести шляхів, одне з них наведено нижче:

```
(j 2 SYMINT / (i 1 SYMINT – CONST 4)) <= CONST 5 &&
(i 1 SYMINT – CONST 4) != CONST 0 && i 1 SYMINT <= j 2 SYMINT
```

Рисунок 2.5 – Генерація обмеження

Ці обмеження для кожного обмеження шляху послідовно розв'язуються за допомогою різних розв'язувачів обмежень, вбудованих у Symbolic Java PathFinder, і врешті-решт отримують такі результати:

Таблиця 2.2

Помилки, виявлені за допомогою символічного PathFinder

| Назва Java-програми | Кількість рядків коду (LOC) | Кількість згенерованих тест-кейсів | Всього знайдено помилок |
|---|---|---|---|
| DividebyzeroExample | 25 | 7 | 2 |

У звичайних умовах генерація значень для цих випадків зазвичай складна, оскільки програма може мати такі проблеми. Результати роботи Symbolic PathFinder у вищезгаданій програмі можна підсумувати в таблиці 2.2.

```
[foo(54,11)]
[foo(92,51)]
[foo(86,–77)]
[(expected = java.lang.ArithmeticException.class), foo(21,8), ##EXCEPTION## "jav
a.lang.ArithmeticException: div by 0..."]
[foo(21,8)]
[foo(11,54)]
[foo(0,0)]
[(expected = java.lang.ArithmeticException.class), foo(4,49), ##EXCEPTION## "jav
a.lang.ArithmeticException: div by 0..."]
[foo(4,49)]
```

Рисунок 2.6 – Тестові випадки foo

Тут, у наведених вище тестових випадках, два тестові випадки foo(21,8) та foo(4,49) генерують винятки ділення на нуль. Інші тестові випадки виконають решту шляхів, відмінних від тих, що генерували винятки. Цей тип тестування є дуже ефективним у тому сенсі, що для виявлення помилок у програмі потрібна невелика кількість тестових випадків. Ми взяли випадок неправильної обробки винятку «ділення на нуль» та протестували його за допомогою інструменту

конколічного тестування. Існують деякі інші типи винятків, що можуть призвести до збою програми, такі як виняток нульового вказівника, які не розглядалися в цьому дослідженні.

Методи тестування безпеки, далі класифіковані. У таблиці наведено короткий опис кожної методики, включаючи типи атак, які вона охоплює, такі як BOF, DoS та інші, згадані в таблиці.

### 3.4 Висновок по розділу

У другому розділі розглянуто методи та інструменти тестування безпеки програмного забезпечення. Проаналізовано критерії класифікації методів тестування на основі різних моделей оцінювання безпеки, а також досліджено основні підходи до виявлення вразливостей програмних систем. Окрему увагу приділено методам динамічного тестування, які дозволяють оцінювати поведінку програмного забезпечення під час його виконання та виявляти потенційні загрози. Результати розділу показали, що комплексне використання різних методів тестування забезпечує більш повне покриття можливих ризиків безпеки.

# РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ МЕТОДІВ ТЕСТУВАННЯ БЕЗПЕКИ

## 3.1 Використання моделей у тестуванні безпеки програмного забезпечення

Для тестування властивостей безпеки необхідна інформація з різних джерел, яка має бути систематично пов'язана одне з одним для підтримки відстеження та належного використання цієї інформації. Окрім функціональної специфікації системи та інформації про архітектуру системи, інформація про відомі або потенційні вразливості, потенційні атаки та ймовірності їх виникнення може дати вказівки щодо того, що та як тестувати. Крім того, ймовірності та оцінки серйозності потенційних атак можна підсумувати для формування аналізу ризиків, який вказує на загрози, які слід враховувати. Такий аналіз ризиків надає вказівки щодо упорядкування тестування та визначення пріоритетів тестування, а також підтримує управління тестуванням, вказуючи на потребу в рекомендованих тестових ресурсах.

Отже, тестування безпеки на основі моделей повинно базуватися на різних типах моделей, щоб охопити різні перспективи, що використовуються для забезпечення безпеки системи. Далі ми пропонуємо три різні категорії моделей, кожна з яких представляє окрему перспективу та може слугувати вхідними моделями для генерації тестів.

*Архітектурні та функціональні моделі*

Архітектурні та функціональні моделі SUT (тестової системи, що перевіряється) стосуються системних вимог щодо загальної поведінки та налаштування програмної системи. Основною перспективою цих моделей є структура та властивості тестованої системи. Моделі існують на різних рівнях абстракції та з різною деталізацією. Часто вони містять доповнення, які дозволяють зосередитися на конкретних властивостях системи, таких як властивості стійкості (наприклад, стани відмови) або властивості продуктивності

(наприклад, тривалість або пропускна здатність). Щодо тестування безпеки, нас головним чином цікавить визначення критичної функціональності системи відносно загальної архітектури програмного забезпечення та визначення критично важливих для безпеки інтерфейсів, які можуть бути точкою входу для зловмисника. Щодо критично важливих для безпеки інтерфейсів, моделі взаємодії або моделі протоколів (що включають моделі даних або моделі поведінки) становлять великий інтерес для тестування безпеки. Крім того, функціональні заходи безпеки (такі як засоби автентифікації або контролю доступу) можуть бути визначені у функціональних моделях та перевірені за допомогою підходів функціонального тестування.

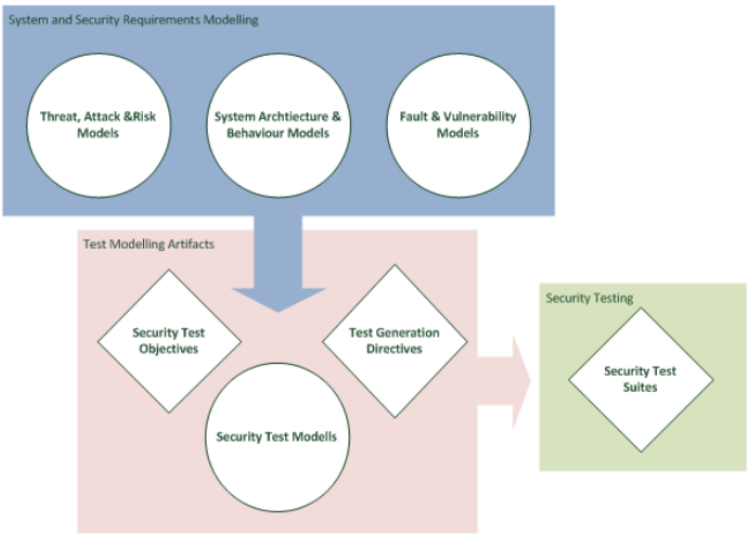


Рисунок 3.1 - Моделювання артефактів у MBST

*Моделі загроз, несправностей та ризиків*

Хоча архітектурні та функціональні моделі зазвичай описують очікувану конфігурацію та поведінку системи, методи моделювання ризиків, такі як підхід моделювання ризиків CORAS [13], зосереджуються на тому, що може піти не так. CORAS надає засоби для моделювання ризиків, загроз або несправностей та дозволяє ідентифікувати численні фактори ризику, описувати їхні взаємозв'язки та пов'язувати їх з ймовірностями виникнення та потенційними наслідками.

Окрім CORAS, існують інші підходи до моделювання несправностей та

атак. Добре відомими є аналіз дерева несправностей (FTA [20]) та причинно-наслідковий аналіз (CCA [17]). FTA розглядає несправності високого рівня та розкладає їх зверху вниз на базові події, які можна ідентифікувати та перевірити в системі.

Варіантом дерев відмов є так звані дерева атак [14]. Дерева атак безпосередньо пов'язані з ризиками безпеки. Вони починаються зі сценарію атаки високого рівня та розкладають його на конкретні базові взаємодії із системою.

Аналіз дерева подій (ETA [18]) працює знизу вгору. Він починається з ідентифікації небажаних системних подій та аналізує наслідки у разі виникнення небажаної події.

CCA (причинно-наслідковий аналіз [6] ) – поєднання концепцій FTA та ETA. Цей аналіз починається з нитки. Причини (зверху – місто) та наслідки (знизу – вгору) аналізуються одночасно.

*Моделі слабких та вразливих сторін*

Хоча моделі загроз, несправностей та ризиків зосереджені на причинах та наслідках системних збоїв, слабких місць або вразливостей, модель слабкості або вразливості описує саму слабкість або вразливість. Інформація, необхідна для розробки таких моделей, зазвичай надається базами даних, такими як Національна база даних вразливостей (NVD) або база даних загальних вразливостей та експозицій (CVE). Ці бази даних збирають відомі вразливості та надають інформацію розробникам, тестувальникам та експертам з безпеки, щоб вони могли систематично перевіряти свої продукти на наявність відомих вразливостей. Одна з проблем, яка ще не вирішена достатньою мірою, полягає в тому, як ці вразливості можна інтегрувати в системні моделі, щоб їх можна було використовувати для генерації тестів.

Одне з можливих рішень базується на ідеї тестування мутацій [22]. Як правило, тестування мутацій використовується для кваліфікації тестових наборів шляхом запуску тестів на мутацію тестованої системи. Якість тестового набору визначається щодо кількості мутантів, виявлених тестовим набором. Для

тестування безпеки моделі тестованої системи мутують таким чином, що мутанти представляють слабкі місця або відомі вразливості. Ці моделі слабких місць або вразливостей потім можна використовувати для генерації тестів за допомогою різних підходів MBT. Згенеровані тести використовуються для перевірки того, чи є тестована система слабкою або вразливою щодо слабких місць та вразливостей у моделі.

*Діяльність з тестування безпеки на основі моделей*

Тестування безпеки, як і будь-яке інше тестування, складається з низки дій та використовує артефакти, спрямовані на систематичне планування, проектування, специфікацію, реалізацію та виконання тестів, а також на оцінку результатів тестування та, за потреби, коригування планування тощо.

MBST демонструє незначні відмінності в цих видах діяльності, але дотримується основної послідовності дій. Головним завданням є надання властивостей системи, що розглядаються, конкретних тестових випадків (даних та поведінки), які представляють стимули для тестованої системи, та оцінку реакцій системи тестовими оракулами. Далі обговорюється, як генерувати тести безпеки за допомогою модельного підходу.

• Визначення цілей та методів тестування безпеки: Цілі тестування визначають загальні цілі тестування та пов'язують ці цілі з методами тестування, які дозволяють досягти цих цілей. Для тестування на основі моделей необхідно спланувати методи моделювання та стратегії генерації тестів. Особливо слід враховувати *моделі загроз, несправностей та ризиків,* щоб спрямувати або посилити ідентифікацію тестування щодо виявлених ризиків, загроз, несправностей та їх наслідків.

• Розробка функціональної тестової моделі: Тестова модель відображає або очікувані функціональні сценарії SUT (системна перспектива), або сценарії використання SUT (системна перспектива). Стандартні мови моделювання, такі як UML, можуть бути використані для формалізації точок контролю та спостереження SUT, динамічної поведінки під час взаємодії з системою, сутностей, пов'язаних з тестом у різних тестових конфігураціях, та

тестових даних, що застосовуються до системи. Тестові моделі повинні бути достатньо точними та повними, щоб дозволити автоматизоване виведення виконуваних тестів з цих моделей. Однак тестування безпеки зосереджується або на перевірці коректності функцій безпеки, або на перевірці стійкості до цілеспрямованого неправильного використання системи. Таким чином, *функціональні тестові моделі,* що використовуються для тестування безпеки, описують не лише типове середовище або використання системи, але й середовища зловмисників або нетипове використання, такі як атаки та спроби злому.

- Визначення критеріїв генерації тестів: Зазвичай існує нескінченна кількість можливих тестів, які можна згенерувати з моделі, тому розробники тестів обирають критерії генерації або вибору тестів, щоб обмежити кількість (згенерованих) тестів до скінченного числа, наприклад, вибираючи тести з найвищим пріоритетом, або щоб забезпечити специфічне покриття системних структур, поведінки тощо. Для тестування безпеки підходи, засновані на структурному покритті моделі, тобто визначення покриття елементів моделі згенерованими тестами, є недостатніми. Отже, використовуються підходи нечіткого тестування, які дотримуються інших видів критеріїв покриття та призводять до різних, але також більшої кількості згенерованих тестових даних або поведінки тестів. Однак також застосовні затверджені критерії генерації тестів, такі як покриття функціональних вимог безпеки або зважування функціональних вимог безпеки значеннями ризику.
- Генерація тестів: Генерація тестів у MBT зазвичай є повністю автоматизованим процесом отримання тестових випадків із заданої тестової моделі, визначеної обраними критеріями генерації тестів. Це також стосується MBST. Згенеровані тестові випадки – це послідовності високорівневих подій або дій до або від SUT, з вхідними параметрами та очікуваними вихідними параметрами та значеннями повернення для кожної події або дії. За потреби згенеровані тести додатково уточнюються до більш конкретного рівня або адаптуються до SUT для підтримки їх автоматичного виконання.

• Оцінка результатів тестування: Під час оцінювання результатів тестування та самої оцінки тестування якість SUT може бути оцінена відносно результатів тестування, а також якість тестів може бути оцінена відносно їх можливостей виявлення несправностей та вразливостей. Хоча для функціонального MBT існують вимірювання та метрики якості системи та тестування, це все ще є дослідницькою проблемою для MBST. Крім того, результати тестування необхідно аналізувати, якщо потрібні зміни до системних вимог, проектування системи, аналізу ризиків або самого процесу тестування. У цих випадках необхідно розпочати необхідні ітерації.

*Підходи до тестування в DIAMONDS*

Проект DIAMONDS (Розробка та промислове застосування технологій тестування безпеки для багатьох доменів) під керівництвом Fraunhofer FOKUS, Берлін, розробляє ефективні та автоматизовані методи тестування безпеки для критично важливих для безпеки мережевих систем у різних промислових галузях, таких як промислова автоматизація, банківська справа та телекомунікації. DIAMONDS розробляє методи проектування об'єктивних, прозорих, повторюваних та автоматизованих тестів безпеки , що зосереджені на системних специфікаціях та пов'язаних з ними ризиках. Цілі проекту включають визначення методів моделювання помилок та вразливостей безпеки, визначення каталогу шаблонів тестування безпеки, розробку методів MBST та визначення методології MBST. DIAMONDS досліджує вразливості мережевих систем у розглянутих областях, щоб вивести загальні принципи, методи та засоби, що дозволяють проводити ефективне тестування безпеки промислового значення. З урахуванням результатів тематичних досліджень методологія тестування безпеки DIAMONDS буде оцінена та оптимізована. Результати проекту надаються зацікавленим сторонам, а також через внески до стандартизації в ETSI та інших органах стандартизації.

Особлива увага в DIAMONDS приділяється (1) ризик-орієнтованому MBST та (2) модельному нечіткому тестуванню.

*Тестування безпеки на основі ризиків*

Тестування на основі ризиків можна загалом запровадити з двома різними цілями. З одного боку, підходи до тестування на основі ризиків можуть допомогти оптимізувати загальний процес тестування. Результати аналізу ризиків, тобто результати аналізу загроз та вразливостей, використовуються для керівництва ідентифікацією тестів і можуть доповнювати результати розробки вимог систематичною інформацією щодо загроз та вразливостей системи. З іншого боку, моделювання атаки полягає у пошуку відхилень SUT від її специфікації, що призводять до вразливостей, оскільки недійсні вхідні дані не відхиляються, а обробляються SUT. Такі відхилення можуть призвести до невизначених станів SUT і можуть бути використані зловмисником, наприклад, для успішного виконання відмови в обслуговуванні.

Комплексна оцінка ризиків додатково вводить поняття значень ризику, тобто оцінку ймовірностей та наслідків для певних сценаріїв загроз. Ці значення ризику можуть бути додатково використані для зважування сценаріїв загроз і таким чином допомогти визначити, які сценарії загроз є більш релевантними, а отже, ідентифікувати сценарії загроз, які потребують ретельнішого розгляду та тестування.

Крім того, підходи до тестування на основі ризиків можуть допомогти оптимізувати аналіз ризиків та саму оцінку ризиків. Аналіз ризиків та оцінка ризиків, подібно до інших видів діяльності з розробки на ранніх етапах проекту, в основному базуються на припущеннях щодо самої системи. Тестування є одним із найважливіших засобів проведення реальних експериментів із системою, і таким чином дозволяє отримати емпіричні докази існування вразливостей, застосовності та наслідків сценаріїв загроз, а також якості контрзаходів. Таким чином, результати тестування на основі ризиків можуть бути використані як форма доказів припущень, зроблених під час оцінки ризиків та аналізу ризиків.

Зокрема, тестування на основі ризику може допомогти в

- надання доказів функціональної коректності контрзаходів,

- надання доказів відсутності відомих вразливостей, та
- виявлення невідомих вразливостей,
- оптимізація аналізу ризиків шляхом виявлення нових факторів ризику та переоцінки значень ризику.

Мова CORAS [7] інтегрує різні деревоподібні підходи до моделювання ризиків. Це графовий підхід до моделювання, який наголошує на моделюванні сценаріїв загроз та надає формалізми для анотації сценаріїв загроз значеннями ймовірності та формалізми для обґрунтування цих анотацій.

На рисунку 3.2 показано просту модель ризику CORAS, яка зображує сценарій загрози для несанкціонованого доступу до бази даних. Мова CORAS дозволяє пов'язувати сценарії загроз зі зловмисниками (так звані загрози, наприклад, «хакер» або «скрипт-кідді» на рисунку 3.2) та з потенційними вразливостями. Вразливість позначається символом розблокованого замка (див., наприклад, «SQL-ін'єкція»). І останнє, але не менш важливе: сценарій загрози пов'язаний з небажаними інцидентами, наприклад, «Записи бази даних змінені неавторизованими особами». Ця модифікація шкодить активу «вміст бази даних» (див. «коричневий мішок з грошима» на рисунку 3.2) і, отже, може негативно вплинути на дохід активу «»(див. «білий мішок з грошима»).

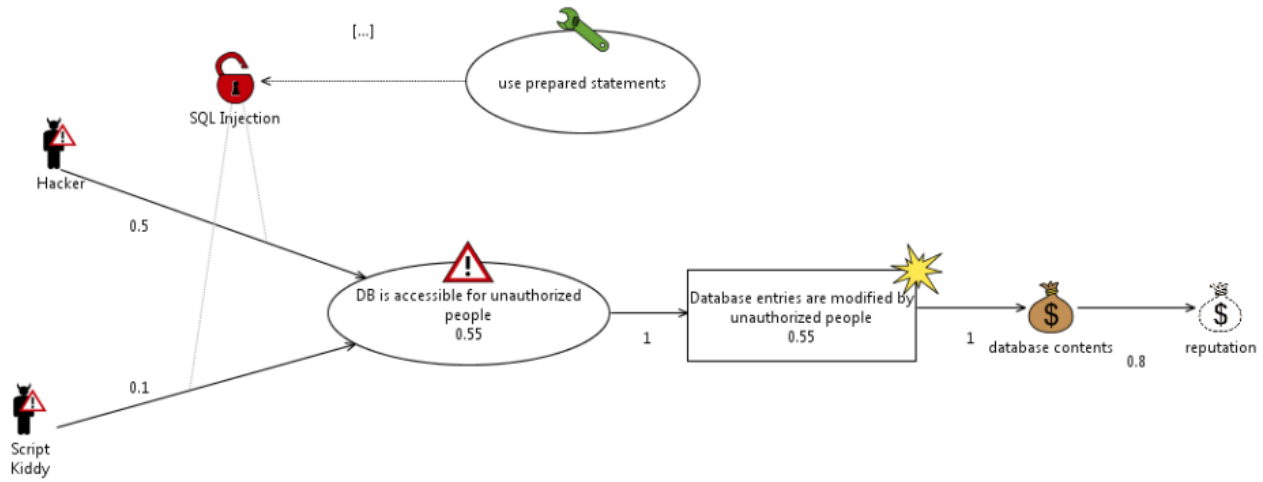


Рисунок 3.2 - Діаграма обробки CORAS

Обробка «використання підготовлених операторів» замість рядків, що містять SQL-запити, зображена на цьому рисунку зеленими плоскогубцями. За винятком того, що загрози, вразливість, активи та лікування – всі елементи мають анотації, що позначають ймовірність або частоту переходу або частоту, наприклад, сценарію загрози.

Такі моделі ризиків можна використовувати по-різному для підтримки тестування. Метою *ідентифікації тестів на основі ризиків* є покращення дизайну тесту таким чином, щоб охоплювати області SUT з високим рівнем ризику та водночас оптимально використовувати ресурси тестування, зосереджуючись спочатку на найвищих ризиках.

При ідентифікації тестів на основі ризику, для окремих факторів ризику розробляються цілі тестування. Ми розглядаємо ціль тестування як, перш за все, неформальну специфікацію, яка визначає, який аспект певної системи, функціональності чи протоколу тощо слід протестувати. Подібно до вимог в інженерії вимог, цілі тестування являють собою вимоги до тестування, які можна уточнювати та розкладати під час розробки тесту. Далі ми опишемо зв'язок між цілями тестування та елементами, що використовуються в аналізі ризиків. Кількісні оцінки пов'язаних ризиків можна використовувати для зважування цілей тестування.

Цілі тестування для

- небажаний інцидент описує, які методи тестування можна застосувати для ініціювання та виявлення небажаного інциденту, а також для характеристики його наслідків.
- Сценарії загроз описують, які методи тестування можна застосувати для ініціювання сценарію загрози та характеристики його наслідків.
- Вразливості описують, які методи тестування можна застосувати для виявлення вразливості.
- Сценарії лікування описують, які методи тестування можна застосувати для характеристики зрілості та ефективності сценарію лікування.

Іншим способом використання моделей ризику в тестуванні є *вибір*

*тестів на основі ризику.* Він використовується для пошуку оптимального набору тестових випадків відповідно до певної стратегії вибору. Стратегія вибору враховує доступні тестові ресурси та оптимізує вибрані тести відповідно до обраних критеріїв покриття. У функціональному тестуванні покриття часто описується покриттям вимог або покриттям елементів системи чи тестової моделі, таких як стани, переходи або рішення. У тестуванні на основі ризику ми прагнемо охопити системні ризики. Критерії розробляються шляхом врахування значень ризику з оцінки ризику для встановлення пріоритетів для генерації тестів або для впорядкування виконання тестів під час тестового запуску. Вибір тестів може бути виконаний або на існуючих тестових випадках для вибору тестів для тестового запуску, або під час генерації тестів, щоб забезпечити спрямовану, цілеспрямовану генерацію тестів.

І останнє, але не менш важливе, тестування безпеки підтримує *контроль ризиків* . Контроль ризиків стосується перегляду результатів оцінки ризиків шляхом корекції припущень щодо ймовірностей, наслідків або зрілості сценаріїв обробки, або ж повного завершення результатів аналізу ризиків шляхом інтеграції вразливостей і, таким чином, потенційних загроз, сценаріїв загроз та небажаних інцидентів. Результати тестування можуть відображати та перевіряти припущення, зроблені під час аналізу ризиків. Результати тестування можуть бути використані для коригування результатів аналізу ризиків шляхом введення нових або переглянутих вразливостей або переглянутих оцінок ризиків на основі виявлених дефектів. Результати тестування, інформація про покриття тестуванням та переглянутий або підтверджений рівень оцінки ризиків можуть надати вагомі аргументи щодо рівня безпеки системи, оскільки результати тестування стосуються.

- ризики, які розглядаються та охоплюються відповідними тестовими випадками.
- сценарії обробки або загрози, які мають бути перевірені за допомогою тестових випадків.
- активи, пов'язані з ними ризики перевіряються відповідними

тестовими випадками.

- вразливості, які були виявлені та розташовані у відповідних тестових випадках.

*Модельно-орієнтований фаззинг*

Хоча походження фаззингу базується на повністю рандомізованому підході , блочні та модельні фаззери використовують свої знання про структуру повідомлень для систематичної генерації повідомлень, що містять недійсні дані серед дійсних даних [19].

Систематичні підходи часто є більш успішними, оскільки структура повідомлення зберігається, і таким чином зростає ймовірність того, що згенероване повідомлення буде прийнято тестовим обладнанням (SUT). Використання принципів фуз-тестування не лише для генерації тестових даних, але й для генерації тестової поведінки доповнює традиційні підходи до фуз-тестування. Фазинг поведінки не лише відображає генерацію нетипових повідомлень, але й змінює типовий вигляд і порядок повідомлень. Наприклад, дійсну та схвалену послідовність повідомлень можна перетворити на нетипову та невідому послідовність шляхом перестановки повідомлень, їх повторення та видалення або просто зміни типу повідомлення.

Фазинг поведінки спрямований на пошук недоліків у дизайні та вразливостей у системах, які не можна просто виявити шляхом застосування недійсних вхідних даних. Він зосереджений на зловживаннях на вищому рівні функціональності. Наприклад, вимога безпеки визначає, що завантаження може бути розпочато лише після успішної автентифікації. У вразливій системі завантаження може бути розпочато додатково без будь-якої автентифікації. Таку помилку можна виявити за допомогою типових вхідних даних, але нетипової поведінки, наприклад, шляхом простого пропуску автентифікації .

DIAMONDS розробляє підходи до фаззингу на основі моделей, які використовують, наприклад, оператори фаззингу на сценарних моделях, що задаються діаграмами послідовностей. Далі наведено спрощений приклад з банківської сфери для ілюстрації того, як оператори фаззингу застосовуються до

діаграм послідовностей. Для зручності розуміння більшість параметрів опущено.

Діаграма послідовності на рисунку 3.3 описує, як клієнт банку може виконати доручення на переказ. Клієнти можуть замовити як національний, так і міжнародний переказ (повідомлення 1). Після цього клієнт надсилає ім'я одержувача доручення на переказ, суму до переказу (повідомлення 2), а також інформацію про національний банківський рахунок одержувача (повідомлення 3) у випадку національного доручення на переказ або інформацію про міжнародний банківський рахунок одержувача (повідомлення 4) у випадку міжнародного доручення на переказ. Доручення на переказ має бути авторизоване клієнтом, який надіслав дійсний номер транзакції TAN (повідомлення 5). Якщо клієнт випадково надіслав недійсний TAN, наприклад, через помилку, він може спробувати ввести дійсний TAN до двох разів ще раз (повідомлення 7, цикл комбінованих фрагментів).

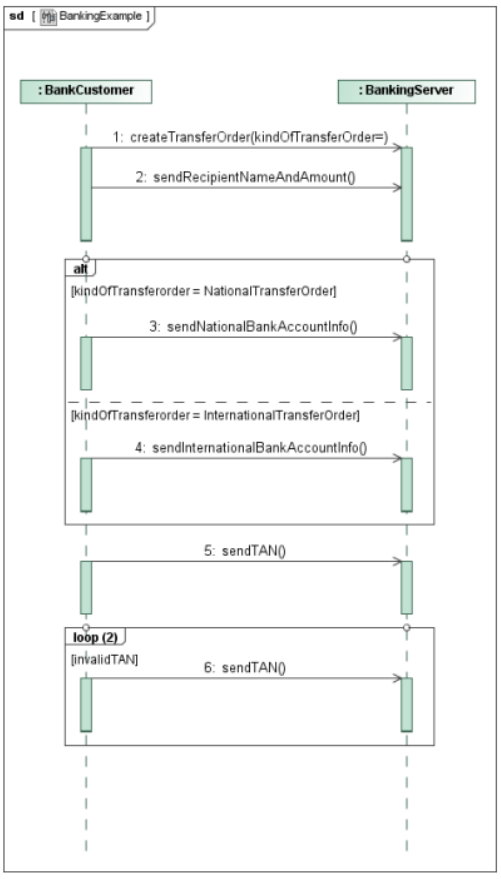


Рисунок 3.3 - Послідовність переказу

Застосовуючи оператори фаззингу до діаграми, повідомлення можна переміщувати, видаляти, повторювати, вставляти або змінювати тип повідомлення для отримання недійсної послідовності. Оператори фаззингу виконують мутацію на діаграмі, що призводить до недійсної послідовності порівняно з оригіналом. Один оператор фаззингу виконує лише одну мутацію діаграми послідовності. Наприклад, оператор фаззингу може переміщувати повідомлення 5 після повідомлення 2.

Інший оператор фаззингу може генерувати недійсну послідовність повідомлень, заперечуючи обмеження взаємодії операндів взаємодії. Заперечуючи обмеження взаємодії об'єднаного фрагмента циклу, послідовності, згенеровані з результуючої діаграми послідовностей, містять щонайменше два дійсні номери транзакцій, що надсилаються на банківський сервер (три, якщо другий заданий TAN є дійсним). На рисунку 3.4 показано результати використання операторів фаззингу, наведених вище.

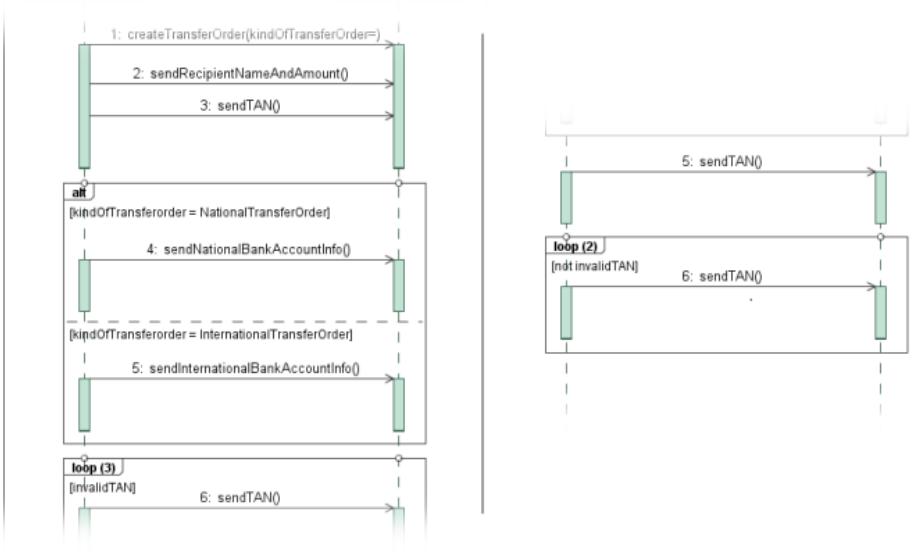


Рисунок 3.4 - Нечітка послідовність переказів

Виконання вищезгаданих операторів фаззингу призводить до різних діаграм послідовностей, які є основою для подальшої генерації тестових випадків. Однак, головна ідея підходів до фаззингу загалом полягає в можливості автоматичної генерації великої кількості тестових випадків. Це досягається

шляхом застосування не лише одного оператора фаззингу до діаграми послідовності, а й набору операторів фаззингу кілька разів, наприклад

шляхом застосування одного оператора фаззингу до кількох елементів моделі діаграми послідовностей або шляхом застосування кількох, можливо, різних операторів фаззингу, один за одним. Комбінація операторів фаззингу дозволяє генерувати велику кількість тестових випадків.

### 3.2 Фазове тестування вбудованих механізмів безпеки

У попередніх розділах ми обговорювали, що безпека - це не одноразовий підхід після впровадження, а серія дій протягом життєвого циклу розробки. Подібно до цього, як безпека є поетапною діяльністю, ми запропонували проводити тестування безпеки з самого першого етапу, тобто від етапу вимог до етапу розгортання та обслуговування.

Безпека програмного забезпечення повинна застосовуватися як дворівневий підхід. По-перше, безпека повинна здійснюватися протягом усього життєвого циклу розробки, а на другому рівні тестування безпеки має бути вбудоване на кожному етапі SDLC. Рисунок 3.5 детально демонструє цю концепцію. На першому етапі збираються та формалізуються вимоги безпеки, такі як правила безпеки, стандарти, політики та цілі. Створюються та аналізуються тестові випадки відповідно до цих вимог безпеки.

На другому етапі тестові випадки безпеки виводяться зі сценарію атаки та тестуються функції безпеки. Одним із таких сценаріїв атаки може бути пошук шляхів збою програми. Верифікація дизайну на основі моделі також корисна для тестування безпечного дизайну. Під час тестування безпеки на етапі кодування аналізуються методи безпечного кодування та застосовується статичне тестування безпеки для виявлення проблем безпеки. На етапі тестування функціональні та нефункціональні проблеми безпеки тестуються за допомогою статичних або динамічних методів або їх комбінації.

Таблиця 3.1

Методи тестування безпеки та охоплення типів атак

| Категорія | Метод тестування | Опис та характеристики | Покриття типів атак |
|---|---|---|---|
| **Статичне** *(Static)* | **Аналіз коду** *(Code review)* | • Аудит вихідного коду на наявність уразливостей безпеки, що виконується вручну або за допомогою інструментів.<br>• Здатний точно визначати проблеми безпеки. Економічно вигідний, але малоефективний для великих додатків. | Усі види атак, залежно від кваліфікації/досвіду експерта, який проводить аудит. |
| | **Перевірка моделей** *(Model checking)* | • Перевірка відповідності моделі системи заданій специфікації безпеки.<br>• Легко повторно генерувати тест-кейси у відповідь на зміни в системі.<br>• Сильний зв'язок тестів із вимогами або архітектурним дизайном. | SQLi (ін'єкції), BOF (переповнення буфера), DoS, XSS. |
| **Динамічне** *(Dynamic)* | **Фаззінг-тестування** *(Fuzz testing)* | • Тестування програми за допомогою величезних масивів випадкових або згенерованих вхідних даних.<br>• Велика глибина покриття окремих шляхів керування програми, але втрачається широта покриття всіх можливих шляхів. | DoS (відмова в обслуговуванні). |
| | **Тестування на основі пошуку** | • Тестування з використанням метаевристичних методів оптимізації пошуку для автоматизації перевірок безпеки.<br>• Працює добре, якщо евристика надає чіткі орієнтири для пошуку. | SQLi, BOF, DoS, XSS. |
| | **Конколічне тестування** *(Concolic testing)* | • Одночасне виконання одного конкретного шляху та генерація вхідних даних для альтернативного шляху.<br>• Широке покриття шляхів керування програми, але може бракувати глибини покриття.<br>• Складно вирішувати складні обмеження, а обробка зовнішніх функцій є неефективною. | SQLi, BOF, DoS, XSS. |

Нарешті, тестування безпеки реального середовища проводиться шляхом тестування безпечної конфігурації, середовища та платформи, що

використовується. Модулі, що постачаються як патч, також повинні бути протестовані на наявність проблем безпеки. Виробниче середовище необхідно належним чином протестувати, оскільки це гарантуватиме захист програми від атак у реальному часі в реальному середовищі.

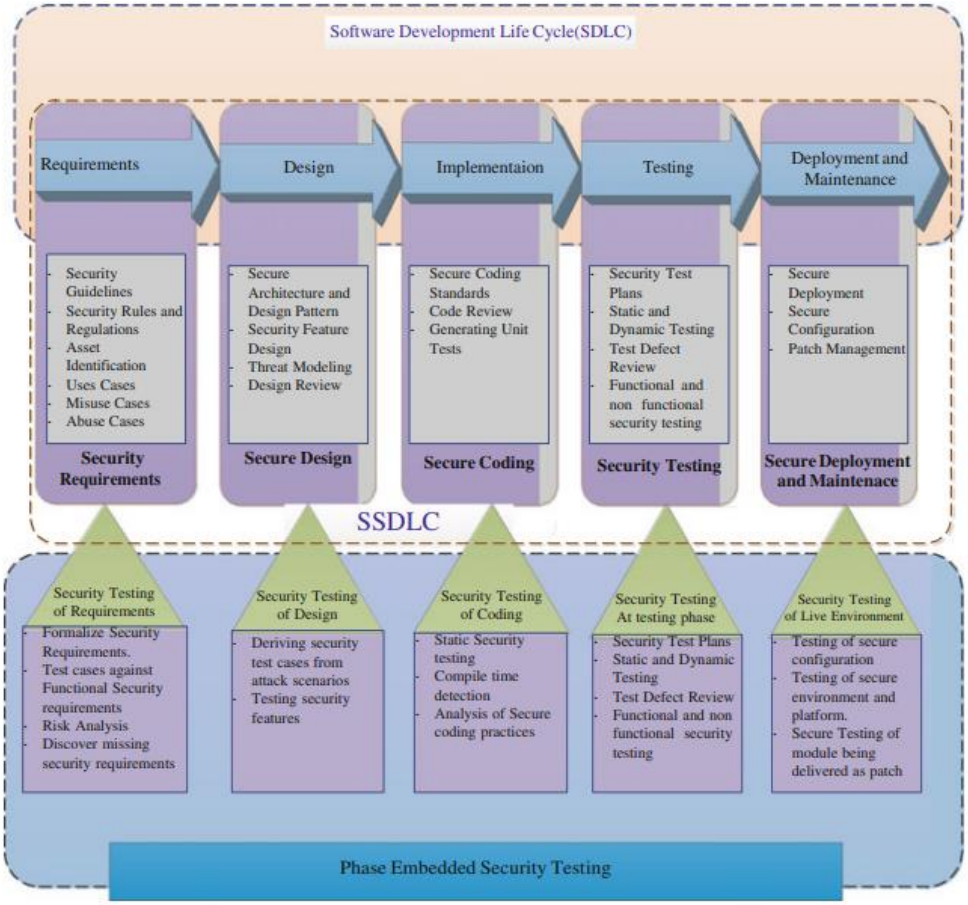


Рисунок 3.5 - Фаза тестування вбудованої безпеки (PEST)

Як приклад, ми розглядаємо організацію, яка сформувала команду розробників програмного забезпечення для створення веб-додатку. Розуміючи важливість безпеки, пов'язаної з додатком, команда розробників програмного забезпечення доклала всіх зусиль, щоб врахувати безпеку в додатку. Експерти з безпеки були залучені до команди розробників від початкового етапу збору вимог до розгортання.

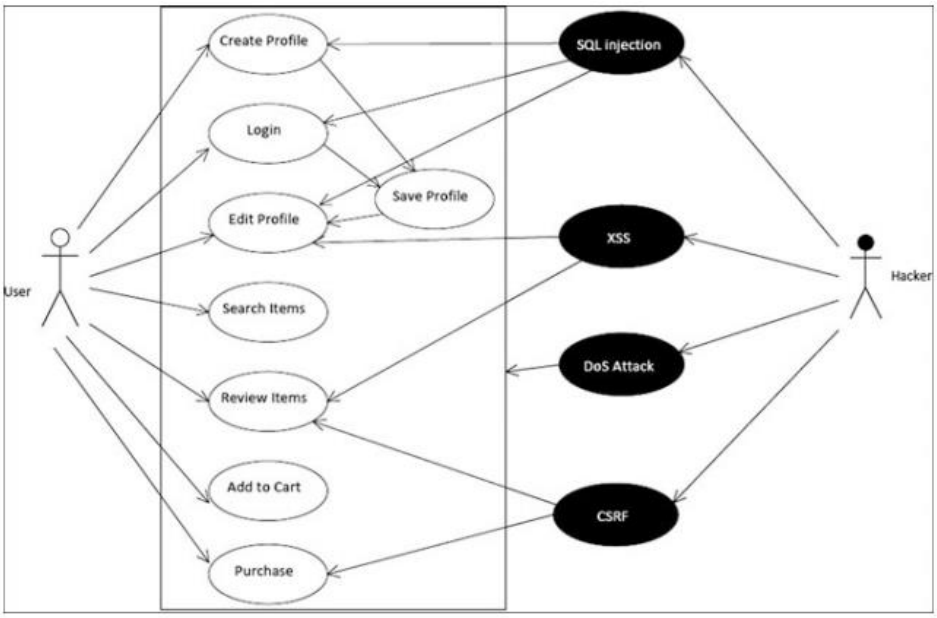


Рисунок 3.6 - Діаграма випадків неправильного використання для веб-застосунку для онлайн-шопінгу

Технічне обслуговування. Заходи з тестування безпеки проводилися протягом усього циклу розробки, щоб гарантувати, що безпека була створена та ретельно протестована. На кожному етапі були виконані наступні заходи під час PESC для застосунку для онлайн-шопінгу.

• Тестування безпеки вимог: Спочатку зібрані вимоги безпеки моделюються з використанням сценарію неправильного використання. Ми намалювали сценарій неправильного використання застосунку для онлайн-покупок, як показано на рис. 3.6. Випадок неправильного використання надає корисну інформацію для перевірки вимог безпеки експертами з домену та безпеки. Нижче ми показали сценарій неправильної перевірки вхідних даних, що може призвести до XSS.

- Хакер вводить дані для входу
- Система відображає меню продуктів
- Хакер вибирає найпопулярніший продукт
- Система відображає детальну інформацію про продукт
- Хакер потрапляє на перевірку з посиланням на шкідливий веб-сайт
- Хакер виходить з системи.

В іншому прикладі SQLI було використано наступний тестовий сценарій.

- Відкрити веб-застосунок
- Почніть перегляд веб-сайту та перейдіть на сторінку входу
- Введіть ім'я користувача
- Введіть пароль: або 1 = 1

Перевірте, чи було обійдено вхід.

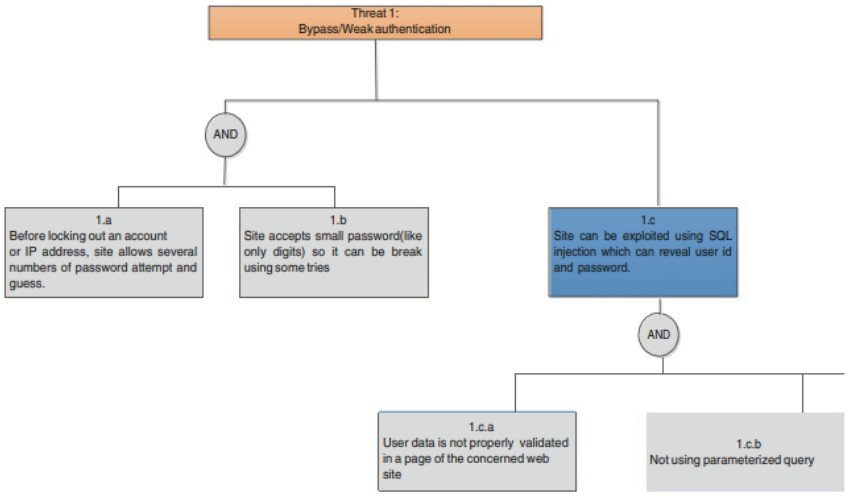


Рисунок 3.7 - Діаграма моделі загрози для обходу автентифікації.

Аналогічно, було створено кілька сценаріїв для перевірки шкідливої діяльності. Вимоги безпеки також можна формалізувати для автоматичної перевірки вимог. Однак у цьому випадку ми не формалізували вимоги безпеки, але кілька досліджень використовували цей метод [18].

Тестування безпеки дизайну : Тестування на цьому етапі стосується виявлення вразливостей безпеки, які можуть потрапити в додаток через помилки дизайну, такі як неправильна обробка помилок та слабка автентифікація. Ми змоделювали загрози безпеці за допомогою схеми моделювання загроз, яка допомагає експертам виявляти загрози, які можуть потрапити в систему. Ми включили модель загрози обходу/слабкої автентифікації, як показано на рис. 3.7. Це допомагає експертам визначити, чи зазнає додаток для онлайн-шопінгу цих загроз. Подібно до цієї моделі загроз, було створено кілька компонентних моделей загроз для перевірки будь-яких вразливостей.

Тестування безпеки коду: Тестування безпеки на цьому етапі проводиться для виявлення будь-яких відсутніх практик безпечного кодування або поширених шаблонів вразливостей. Експерти з безпеки переглянули весь код, щоб дізнатися про будь-які недоліки безпеки або відсутні практики безпеки. Ми застосували інструменти статичного аналізу, такі як find-security-bugs, для виявлення будь-яких відомих вразливостей. Цей інструмент можна додати як плагін до Findbugs, і його можна використовувати для тестування XSS, SQLi, DoS та інших у певній програмі. Він дуже ефективний для пошуку потенційних XSS, таких як наведено в коді 2 нижче рисунок 3.8.

```
<%
String taintedInput = (String)request.getAttribute("input");
%>
 [...]
<%= taintedInput %>
```

Рисунок 3.8 - Плагін до Findbugs.

У наведеному вище коді було виявлено потенційну XSS-вразливість. Цей вразливий код може бути використаний для виконання шкідливого JavaScript у браузері клієнта. Список інструментів статичного аналізу можна знайти на сайті OWASP [11].

- Тестування безпеки на етапі тестування : Після огляду та тестування безпечного кодування наступним кроком є застосування будь-якого інструменту динамічного тестування безпеки для виявлення вразливостей під час виконання програми. Ці інструменти знаходять потенційні вразливості в програмах, такі як перевірка вводу/виводу, проблеми специфічних програм та інші. Інструмент динамічного тестування безпеки, такий як IBM Security AppScan, можна застосувати для виявлення проблем безпеки. Ми завантажили пробну версію цього інструменту з сайту IBM та протестували фіктивний веб-сайт « http://www.altoromutual.com » на наявність проблем безпеки. Цей інструмент створив кілька тестових випадків, і деякі з них здатні використовувати веб-сайт

через XSS, SQLi та інші вразливості.

У дослідженні було застосовано аналогічні інструменти, засновані на інших методах тестування безпеки, таких як тестування безпеки на основі пошуку та конколічне тестування. Список таких інструментів можна знайти на сайті OWASP [13].

- Тестування безпеки розгортання та обслуговування : Після розгортання програми необхідно періодично проводити перевірки справності програми та її платформи, щоб переконатися у відсутності нових загроз безпеці в системі. Якщо системі потрібні оновлення або нові виправлення, їх також слід належним чином перевірити на наявність вразливостей безпеки. Експерти запропонували кілька методів моніторингу часу виконання для моніторингу програм у робочому стані. Огляд методів моніторингу часу виконання можна знайти в роботі [14].

Ми розглянули реальний випадок веб-застосунку Virtual Freer версії 1.57, в якому нещодавно було виявлено вразливість SQLi. Згідно з даними лабораторії вразливостей [15], в офіційній системі керування контентом (CMS) Virtual Freer версії 1.57 було виявлено вразливість обходу сеансу автентифікації. Віддалені зловмисники можуть використовувати цю вразливість для доступу до панелі адміністратора або веб-інтерфейсу користувача. Вразливість, знайдена у файлі login.php, дозволяє віддаленим зловмисникам використовувати корисне навантаження SQL 1 = 1, що призводить до обходу скрипта перевірки на сторінці login.php. Таким чином, здійснюється доступ до панелі адміністратора без дійсного облікового запису.

Якби розробники дотримувалися підходу PEST, як обговорювалося вище, то на дуже ранній стадії вони могли б протестувати цю вразливість. Ми обговорили, як можна створити тестовий сценарій для вразливості SQLi. У випадку, якщо вона не пройшла тестування вимог, то на етапі проектування це було б перевірено за допомогою моделі загроз, яку ми показали під час тестування безпеки проекту. Отже, ідея PEST полягає у виявленні та виправленні проблем безпеки на ранніх стадіях. Здається, що ці дії призведуть до накладних

витрат, але в довгостроковій перспективі такі дії виявляться перевагою або ціннісними активами, що дозволить компанії мати передову перевагу на конкурентному ринку.

### 3.3 Аналіз та обговорення сучасних галузевих практик тестування безпеки

Сьогодні програмне забезпечення використовується майже в кожній сфері, будь то розваги, готельний бізнес, фінанси та страхування, роздрібна торгівля чи інші. Зловмисники націлені на їхнє IT-середовище, таке як електронна комерція, точки продажу (POS) та корпоративні/внутрішні мережі. Нещодавнє дослідження показало, що у 2014 році фінансова та страхова галузі зіткнулися з 57% своїх порушень у корпоративних/внутрішніх мережах та 43% в електронній комерції [17]. Це чітко вказує на те, що додатки повинні бути належним чином захищені та протестовані на наявність проблем безпеки. Що стосується безпеки додатків, веб- та мобільні додатки схильні до серйозних недоліків безпеки. Згідно зі звітом Trustwave global security [19], динамічне тестування безпеки додатків (DAST) тестованих веб-додатків виявило 17 748 вразливостей, і 98% з них мали одну або декілька вразливостей безпеки. Серед цих додатків 35% мали вразливість до витоку інформації, 20% мали вразливість XSS, окрім інших. Це дослідження показує, що кількість інцидентів безпеки зростає з кожним днем, і серед усіх виявлених тверджень додатки є найпопулярнішою ціллю для зловмисників.

У тому ж дослідженні Trustwave представила результати тестування безпеки мобільних додатків, проведеного у 2014 році. Вони виявили щонайменше одну вразливість у 95% мобільних додатків та 6,5 медіанної кількості вразливостей на додаток. Вони спостерігали 26% зростання критичних вразливостей порівняно з 2013 роком. Оскільки мобільні додатки продовжують зростати, зловмисники все більше зосереджуються на мобільних додатках. Неадекватні заходи безпеки під час розробки призведуть до серйозних невдач

для користувачів. Тому дослідники більше стурбовані належним тестуванням безпеки мобільних додатків.

Зазвичай, вразливості безпеки тестуються за допомогою продуктів для тестування безпеки застосунків (AST). AST включає кілька підходів [20], таких як статичний AST (SAST), динамічний AST (DAST), інтерактивний AST (IAST), що поєднує SAST та DAST, та мобільний AST, що поєднує традиційні SAST та DAST разом з поведінковим аналізом. Постачальники пропонують ці підходи як інструмент або через службу підписки. Згідно з дослідженням Gartner, більшість підприємств впроваджують AST, але вони відрізняються адаптацією та зрілістю підходів. DAST, а потім SAST, є найбільш бажаною комбінацією, тоді як IAST та мобільний AST з'явилися нещодавно.

*Вимоги галузі та майбутні тенденції*

Зі зростанням потреби у веб-додатках та мобільних додатках, галузі промисловості більше стурбовані механізмами захисту. Оскільки кількість інцидентів безпеки зростає з кожним днем, галузі промисловості прагнуть розробки комплексної платформи тестування безпеки додатків. Існує потреба розширити функціональність виявлення безпеки додатковою функціональністю, а саме захистом безпеки додатків. Наразі галузі практикують SaaS:Security as a Service на основі вимог для SAST та DAST, і вони прагнуть підтримувати вимоги корпоративного класу за допомогою керування доступом на основі ролей (RBAC) для консолідації звітності на основі ризиків [26]. Галузі промисловості приділяють більше уваги безпеці мобільних додатків, оскільки мобільні додатки революціонізують спосіб ведення бізнесу в галузях промисловості та дуже швидко підключаються до широкої аудиторії. Забезпечення безпеки в мобільних додатках є складним завданням, оскільки галузі промисловості використовують різні мови та платформи для їх розробки.

На основі нещодавнього опитування спостерігається така майбутня тенденція:

- Експерти розглядають гібридні методи для створення автоматизованого набору інструментів для експлуатації. Одним із таких

гібридних методів може бути символьне виконання та аналіз графів виконання коду за допомогою методів штучного інтелекту (ШІ) [22].

- Дослідники планують поєднати машину станів протоколу з тестуванням на основі моделей, а також намагаються впровадити методи зворотного проектування для автоматичного вилучення опису вхідного формату [27].

- Дослідники працюють над новою технологією, а саме технологією самозахисту програм під час виконання (RASP), яка пропонує розширення можливостей виявлення та захисту від загроз під час виконання. Вона контролює виконання зсередини програми, керує нею за потреби та, нарешті, ініціює заходи захисту [26].

- Основними постачальниками на цьому ринку тестування безпеки є IBM, HP, Accenture, Cisco, McAfee, Applause, Veracode та WhiteHat Security. Очікується, що світовий ринок тестування безпеки подвоїться до 2019 року, з розрахунковим сукупним річним темпом зростання (CAGR) 14,9, а Північна Америка, як очікується, стане найбільшим ринком послуг тестування безпеки [18].

Швидке зростання впровадження високофункціональних програмних застосунків, їхньої складності та загроз безпеці вже є реальністю. Така тенденція, у свою чергу, неминуче відкриє нові напрямки дослідження щодо тестування з різних точок зору. Ринок, здається, формує іншу та неминучу роль у формі послуг з тестування безпеки, але, звичайно, не без відповідних питань та викликів. Чітко визначене розуміння проблем та викликів обов'язково приведе нас у правильному напрямку та до мети ефективно та результативно. Більше того, сучасний стан техніки в тестуванні безпеки, здається, не достатній для перевірки першопричини, тобто прихованих вразливостей, і, отже, для забезпечення безпеки. Таким чином, пропонується основа для поетапного тестування вбудованої безпеки як безперервна діяльність, а не фрагментарний та продуманий підхід.

Ця робота корисна для фахівців з безпеки для тестування своїх програм

на наявність вразливостей безпеки та відповідного застосування захисту. Це також забезпечить майбутній напрямок для дослідників тестування безпеки, оскільки методи, обговорені тут, є гарною темою для роботи, зокрема, тестування безпеки на основі пошуку та гібридних методів.

Фактично, загрози безпеці, що оточують фінансові програми, зросли в геометричній прогресії та продовжують драматично розвиватися. Отже, зараз вкрай необхідно зробити програмні програми високобезпечними та надійними. Тестування безпеки – це спроба виявити ті вразливості, які можуть порушувати цілісність стану, достовірність вхідних даних та логічну правильність, а також вектори кутів атаки, які можуть використовувати ці вразливості . Це мінімізує ризики порушення безпеки та забезпечує конфіденційність, цілісність та доступність транзакцій клієнтів. Сучасні програми стикаються з кількома викликами безпеки, серед яких складність програмного забезпечення, сторонній код та динамічні політики безпеки. Ці виклики часто суттєво впливають на безпеку програм.

З іншого боку, програмне забезпечення стає надзвичайно складним, величезним за розміром та практично розширюваним операційним середовищем. Таким чином, важче категорично зрозуміти програми, важко діагностувати проблеми та тестувати, що призводить до можливості помилок у системі, які можуть призвести до вразливостей безпеки. Через тиск ринку та суворі терміни для швидкого та низького виробництва та доставки програмного забезпечення, сторонній код широко використовується в індустрії програмного забезпечення. Загальновідомим та очевидним фактом є те, що сторонній код створює багато проблем, пов'язаних з безпекою, які можуть призвести до серйозного ризику для компаній. Тому, якщо використовується сторонній код, його необхідно належним чином та ефективно протестувати на наявність недоліків безпеки. У цьому підрозділі зроблено спробу розробити такий механізм для тестування та забезпечення високого рівня безпеки на кожному етапі за допомогою PEST.

Політики інформаційної безпеки визначають набір політик, прийнятих

організацією для захисту своєї інформації, узгоджених із зацікавленими сторонами та керівництвом. Вони включають сферу застосування політики, класифікацію інформації, мету управління безпечним обробленням інформації в кожному класі та деякі інші. Сучасні системи взаємодіють із зовнішнім середовищем, де політики безпеки не можуть бути відомі заздалегідь та застосовані. Таким чином, політики безпеки мають динамічний характер для таких випадків, як, наприклад, визначення повноважень користувачів залежно від типу повідомлення, отриманого через зовнішнє середовище. Тому по суті необхідний механізм, який може дозволити приймати критично важливі рішення щодо безпеки під час виконання на основі динамічних спостережень за середовищем.

Безпека є одним з головних аспектів якості програмного забезпечення через обставини, що склалися в індустрії програмного забезпечення. Програмне забезпечення може бути функціонально коректним, але при цьому не мати якості з точки зору стабільності, безпеки або зручності використання. Технології розвиваються неймовірно швидкими темпами та вражаюче впливають майже на кожну сферу. Тому вкрай важливо усвідомити необхідність та важливість визначення механізму з комплексними можливостями тестування, який адаптується до динамічної та гетерогенної природи веб-домену, що змінюється. Зростаюча кількість порушень безпеки зробила тестування безпеки життєво важливою частиною життя програмного забезпечення. Тестування безпеки програмного забезпечення відіграє важливу роль у виявленні можливих вразливостей та пов'язаних з ними загроз. Воно гарантує, що програма відповідає поточним потребам безпеки, коли вона піддається впливу зловмисних атак. Усі теорії марні, доки не будуть запропоновані кращі інструменти та методи. Практика повинна починатися з самого початку, на рівні вимог, і продовжуватися до остаточної розробки програми . Це дозволить виявити вразливості саме в той момент, коли вони з'являються.

Інструменти автоматизації тестування безпеки потребують постійного оновлення з урахуванням постійно розвиваючихся та мінливих технологій. Це

одна з головних проблем та інструментів, які потребують належної інтеграції в існуючі робочі процеси розробки. Постійний моніторинг та оцінка, безумовно, захистять від можливих майбутніх загроз. У розділі стверджується та завершується тим фактом, що тестування безпеки залежить від технологій. Отже, зараз необхідно дослідити методи тестування, які можуть відстежувати всі можливі проблеми, одночасно розробляючи ретельний дизайн та стратегії тестування. Майбутній напрямок вимагає того, як ми адаптуємося до безпечної динамічної та всепроникної природи Інтернету.

З огляду на поточну ситуацію та прогнози, світле майбутнє вимальовується для сфери забезпечення якості та тестування програмного забезпечення, такої як автоматизоване, символічне, формальне, поетапне тестування безпеки, а не демонстраційне традиційне тестування. Це, у свою чергу, призведе до створення надійних програмних компонентів у їхньому справжньому дусі, щоб відповідати майбутнім вимогам до програмного забезпечення та безпечним функціям. Тестувальники програмного забезпечення повинні бути готові скористатися можливостями, що з'являються в багатомільярдній індустрії тестування безпеки програмного забезпечення.

### 3.4 Висновок по розділу

У третьому розділі досліджено практичні аспекти застосування методів тестування безпеки та оцінено їх ефективність. Розглянуто використання моделей безпеки для планування та проведення тестування, а також фазовий підхід до перевірки вбудованих механізмів захисту програмного забезпечення. Проведено аналіз сучасних галузевих практик тестування безпеки та визначено їх переваги для забезпечення стійкості програмних систем до кіберзагроз. Отримані результати підтвердили, що впровадження сучасних методів тестування безпеки дозволяє своєчасно виявляти вразливості, зменшувати ризики та підвищувати загальний рівень захищеності програмного забезпечення.

# ВИСНОВКИ

У ході виконання роботи було досліджено методи тестування безпеки програмного забезпечення, спрямовані на виявлення вразливостей, оцінку рівня захищеності систем та підвищення загальної надійності програмних рішень. У роботі проаналізовано сучасні підходи до безпекового тестування, їх особливості, переваги та обмеження, а також роль інтеграції тестування безпеки у процес розробки програмного забезпечення. Розглянуто ключові категорії методів, зокрема статичне, динамічне та модельно-орієнтоване тестування, а також їх застосування для виявлення різних типів вразливостей.

Процес дослідження охопив аналіз актуальних кіберзагроз, які виникають у сучасних програмних системах, включаючи помилки конфігурації, вразливості автентифікації, ін’єкційні атаки та порушення контролю доступу. Було визначено, що зростання складності програмного забезпечення та використання розподілених архітектур значно підвищує ризики виникнення безпекових проблем, що робить тестування безпеки обов’язковим етапом життєвого циклу розробки.

Ми розпочали з аналізу теоретичних основ тестування безпеки програмного забезпечення, визначивши базові принципи забезпечення захисту інформаційних систем та концепцію безпечного життєвого циклу розробки (SDLC). Було розглянуто, як інтеграція безпекових перевірок на ранніх етапах проєктування дозволяє значно знизити витрати на виправлення помилок та мінімізувати ризики експлуатації вразливостей у готовому продукті.

Далі було досліджено методи класифікації тестування безпеки, що дозволило систематизувати існуючі підходи за рівнем доступу до системи, способом аналізу коду та етапом виконання тестування. Окрему увагу приділено підходам black-box, white-box та gray-box тестування, які застосовуються залежно від доступної інформації про систему та цілей перевірки. Також було розглянуто інструментальні засоби, що автоматизують процес виявлення вразливостей та підвищують ефективність тестування.

Практична частина роботи охопила дослідження методів динамічного тестування безпеки, які дозволяють виявляти вразливості під час виконання програмного забезпечення. Було проаналізовано підходи до моделювання атак, перевірки поведінки системи у реальному часі та використання автоматизованих сканерів безпеки. Встановлено, що динамічне тестування є особливо ефективним для виявлення вразливостей, які неможливо визначити лише шляхом аналізу вихідного коду.

Окремим етапом дослідження стало вивчення використання моделей у тестуванні безпеки програмного забезпечення. Було показано, що модельно-орієнтовані підходи дозволяють формалізувати поведінку системи та автоматизувати процес генерації тестових сценаріїв, що підвищує точність і повноту перевірок. Також розглянуто фазове тестування вбудованих механізмів безпеки, яке передбачає поетапну перевірку захисту на рівні компонентів, інтеграції та системи в цілому.

У роботі проаналізовано сучасні галузеві практики тестування безпеки, які застосовуються в реальних проєктах розробки програмного забезпечення. Було визначено, що найбільш ефективним підходом є комбіноване використання різних методів тестування, включаючи автоматизовані інструменти, ручний аналіз та безперервну інтеграцію безпекових перевірок у процес розробки. Це дозволяє своєчасно виявляти критичні вразливості та зменшувати ризики кібератак.

Загалом, результати роботи підтверджують, що застосування сучасних методів тестування безпеки програмного забезпечення є ключовим елементом забезпечення кіберзахисту. Їх використання дозволяє ефективно виявляти та усувати вразливості на різних етапах життєвого циклу розробки, підвищувати якість програмних систем та зменшувати потенційні ризики експлуатації. У перспективі подальші дослідження можуть бути спрямовані на вдосконалення автоматизованих методів тестування та інтеграцію штучного інтелекту для виявлення складних атак.

## СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА


1. OWASP Foundation. (2024). OWASP Web Security Testing Guide. https://owasp.org/www-project-web-security-testing-guide/
2. OWASP Foundation. (2024). OWASP Top 10 Web Application Security Risks. https://owasp.org/www-project-top-ten/
3. NIST. (2024). Application Security Testing Guide (SP 800-115 updates). https://csrc.nist.gov/publications
4. NIST. (2023). Secure Software Development Framework (SSDF). https://csrc.nist.gov/publications/detail/sp/800-218/final
5. Microsoft. (2025). Security Testing in DevOps. https://learn.microsoft.com/en-us/security/
6. Microsoft. (2025). Threat Modeling Tool Overview. https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool
7. Microsoft. (2025). Secure Development Lifecycle (SDL). https://www.microsoft.com/en-us/securityengineering/sdl
8. SANS Institute. (2024). Security Testing & Assessment Guidelines. https://www.sans.org/white-papers/
9. NIST. (2023). Guide to Computer Security Log Management. https://csrc.nist.gov/publications/detail/sp/800-92/rev-1/final
10. PCI Security Standards Council. (2024). Payment Application Security Testing. https://www.pcisecuritystandards.org/
11. ISO/IEC. (2023). ISO/IEC 27001 Information Security Management. https://www.iso.org/isoiec-27001-information-security.html
12. ISO/IEC. (2023). ISO/IEC 27034 Application Security. https://www.iso.org/standard/44378.html
13. IEEE. (2024). Software Security Engineering Standards. https://standards.ieee.org/
14. MITRE. (2024). CWE - Common Weakness Enumeration. https://cwe.mitre.org/

15. MITRE. (2024). ATT&CK Framework for Adversary Modeling. https://attack.mitre.org/

16. PortSwigger. (2025). Web Security Academy. https://portswigger.net/web-security

17. OWASP. (2024). Security Testing Guide (SAST, DAST, IAST). https://owasp.org/

18. OWASP. (2024). Software Assurance Maturity Model (SAMM). https://owasp.org/www-project-samm/

19. Veracode. (2025). Application Security Testing Guide. https://www.veracode.com/security/application-security-testing

20. Checkmarx. (2025). SAST and DAST Security Testing Approaches. https://checkmarx.com/

21. Synopsys. (2024). Software Security Testing Best Practices. https://www.synopsys.com/

22. Fortify. (2024). Application Security Testing Solutions. https://www.opentext.com/products/fortify

23. IBM. (2025). Application Security Testing Guide. https://www.ibm.com/security/application-security

24. AWS. (2025). Security Testing in Cloud Applications. https://docs.aws.amazon.com/security/

25. Google Cloud. (2025). Security Testing Best Practices. https://cloud.google.com/security

26. Red Hat. (2024). DevSecOps Security Testing Practices. https://www.redhat.com/en/topics/security/devsecops

27. GitHub. (2025). Security Code Scanning Documentation. https://docs.github.com/en/code-security

28. SonarSource. (2025). Static Application Security Testing (SAST). https://www.sonarsource.com/

29. ZAP Proxy. (2025). OWASP ZAP Security Testing Tool. https://www.zaproxy.org/

30. Burp Suite. (2025). Web Application Security Testing Tools. https://portswigger.net/burp

31. Acunetix. (2024). Web Vulnerability Scanner Guide. https://www.acunetix.com/

32. Qualys. (2024). Security Testing & Vulnerability Management. https://www.qualys.com/

33. Rapid7. (2024). Application Security Testing Solutions. https://www.rapid7.com/

34. Imperva. (2025). Web Application Security Testing Practices. https://www.imperva.com/

35. Cloud Security Alliance. (2024). Security Testing Guidelines. https://cloudsecurityalliance.org/

36. Hacking Articles. (2023). Web Application Security Testing Methods. https://www.hackingarticles.in/

37. HackerOne. (2025). Bug Bounty & Security Testing Insights. https://www.hackerone.com/

38. Bugcrowd. (2025). Crowdsourced Security Testing Guide. https://www.bugcrowd.com/

39. OWASP Juice Shop. (2024). Vulnerable Web App for Security Testing Practice. https://owasp.org/www-project-juice-shop/

40. Microsoft Security Blog. (2025). Modern Application Security Testing Practices. https://www.microsoft.com/en-us/security/blog/

# БІБЛІОГРАФІЧНА ДОВІДКА

**Тема бакалаврської роботи:** "Методи тестування безпеки програмного забезпечення"

Обсяг пояснювальної записки: 74 аркушів

Дата закінчення бакалаврської роботи 10 червня 2026р.

Підпис студента _____