КВАЛІФІКАЦІЙНА РОБОТА

Група МІПЗд-22
Рекута В.В.

2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

на правах рукопису

Рекута Владислав Віталійович

УДК 004.4

Розробка та оцінка методів автоматизованого тестування безпеки програмного забезпечення

Спеціальність 121 – «Інженерія програмного забезпечення»

Кваліфікаційна робота на здобуття кваліфікації магістра

Нормоконтроль

_____Стисло О.В.

(підпис, дата, розшифрування підпису)

Студент

_____Рекута В.В.

(підпис, дата, розшифрування підпису)

Допускається до захисту

Завідувач кафедри

_____к.т.н., доц. Ващишак С.П.

(підпис, дата, розшифрування підпису)

Керівник роботи

_____к.ф.-м.н., доц. Остафійчук П.Г.

(підпис, дата, розшифрування підпису)

Івано-Франківськ – 2024

ЗВО «Університет Короля Данила»

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

Освітній ступінь: «магістр»

Спеціальність: 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____

« 19 » лютого 2024 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Рекута Владислав Віталійович

(прізвище, ім'я, по-батькові)

1. Тема кваліфікаційної роботи
Розробка та оцінка методів автоматизованого тестування безпеки програмного забезпечення
керівник роботи:
Остафійчук Петро Георгійович, кандидат фізико-математичних наук, доцент
затверджена наказом вищого навчального закладу від « 26 » червня 2024 року № 32/1с
2. Термін подання студентом роботи: 16 лютого 2024
3. Вихідні дані роботи: Методи автоматизованого тестування безпеки програмного забезпечення через аналіз існуючих рішень, моделей та методів тестування, а також розробку алгоритмів, технологій та програмної реалізації для вирішення цієї проблеми.
3. Зміст магістерської роботи (перелік питань, які потрібно розробити):
- Аналіз існуючих рішень безпеки програмного забезпечення
- Аналіз моделей і методів тестування безпеки програмного забезпечення
- Розробка алгоритмів та технологій, проектування ПЗ для вирішення проблем безпеки
- Програмна реалізація для вирішення проблеми безпеки програмного забезпечення

4. Дата видачі завдання: 29 червня 2023 року.

КОНСУЛЬТАНТИ РОЗДІЛІВ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Розділ	Консультант (прізвище, ініціали та посада)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1	Робота над 1 розділом	10 грудня 2023	Виконано
2	Робота над 2 розділом	20 грудня 2023	Виконано
3	Робота над 3 розділом	30 грудня 2023	Виконано
4	Робота над 4 розділом	15 січня 2024	Виконано
5	Написання вступу, загальних висновків, оформлення джерел посилань та додатків. Оформлення пояснювальної записки в цілому згідно вимог Методичних рекомендацій	20 січня 2024	Виконано
6	Перевірка Пояснювальної записки КРМ на плагіат	25 січня 2024	Виконано
7	Попередній захист КРМ на кафедрі	14 лютого 2024	Виконано
8	Доопрацювання (при необхідності) КРМ. Брошурування КРМ. Підготовка супровідних документів.	20 лютого 2024	Виконано
9	Захист КРМ на ЕК.	25-28 лютого 2024	

Студент _____ Рекута В.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Остафійчук П.Г.
(підпис) (прізвище та ініціали)

ПЕРЕЛІК ГРАФІЧНОГО МАТЕРІАЛУ

АНОТАЦІЯ

В даній роботі досліджується проблематика розробки та оцінки методів автоматизованого тестування безпеки програмного забезпечення.

У першому розділі проводиться аналіз літературних джерел, досліджуються відомі моделі, методи та засоби, що використовуються у даній галузі.

У другому розділі присвяченому розробці власних моделей та методів для вирішення конкретної задачі, пов'язаної із тестуванням безпеки програмного забезпечення, роботи описуються структура та принципи функціонування розроблених моделей.

У третьому розділі розглядається процес розробки алгоритмів та технологій, а також проектування програмного забезпечення для вирішення поставленої задачі. Розглядаються ключові аспекти проектування, архітектурні рішення та вибір технічних засобів.

У четвертому розділі презентується програмна реалізація розроблених методів, результати експериментів та їхній аналіз. Оцінюється ефективність розроблених моделей та методів, а також наводяться ключові висновки та рекомендації для подальших досліджень.

КЛЮЧОВІ СЛОВА: ТЕСТУВАННЯ БЕЗПЕКИ, МЕТОДИ ТЕСТУВАННЯ, АЛГОРИТМИ ТЕСТУВАННЯ, ПРОЕКТУВАННЯ ПЗ, ПРОГРАМНА РЕАЛІЗАЦІЯ, ЕФЕКТИВНІСТЬ ТЕСТУВАННЯ.

SUMMARY

This work considers the problems of developing and evaluating automated security testing methods in software engineering.

The first chapter provides a literature review, examining established models, methods, and tools common in the field.

The second chapter, devoted to the development of custom models and methods for solving specific security testing tasks, describes in detail the structure and principles of the created models.

The third chapter considers the process of developing algorithms and technologies, as well as the development of software solutions to achieve the set goals. The key aspects of design, architectural solutions, and selection of technical means are considered.

The fourth chapter presents the implementation of the developed methodology, the results of experiments, and their analysis. The effectiveness of the created models and methods was evaluated, and the main conclusions and recommendations for further research were provided.

KEYWORDS: SECURITY TESTING, TESTING METHODS, TESTING ALGORITHMS, SOFTWARE DESIGN, SOFTWARE IMPLEMENTATION, TESTING EFFICIENCY.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API - application programming interface (Прикладний програмний інтерфейс). набір визначень підпрограм, протоколів взаємодії та засобів для створення програмного забезпечення;

SDLC - Systems development life cycle (Життєвий цикл програмного забезпечення). Сукупність окремих етапів робіт, що проводяться у заданому порядку протягом періоду часу, який починається з вирішення питання про розроблення програмного забезпечення і закінчується припиненням використання програмного забезпечення;

RDP - Remote Desktop Protocol (Протокол віддаленого робочого стола). Протокол прикладного рівня, що використовується для забезпечення віддаленої роботи користувача із сервером, на котрому запущений сервіс термінальних з'єднань;

IoT - Internet of Things (Інтернет речей). Концепція мережі, що складається із взаємозв'язаних фізичних пристроїв, які мають вбудовані давачі, а також програмне забезпечення, що дозволяє здійснювати передачу і обмін даними між фізичним світом і комп'ютерними системами в автоматичному режимі, за допомогою використання стандартних протоколів зв'язку;

Fuzzing - Фаззінг.Техніка автоматизованого тестування програмного забезпечення, яка полягає в тому, що на вхід програми подаються недійсні, невідповідні або випадково згенеровані дані;

SAST - Static Application Security Testing (Статичне тестування безпеки додатків). Метод тестування білої скриньки. Він досліджує код, щоб знайти недоліки програмного забезпечення, такі, як ін'єкція SQL та інші, перераховані в OWASP Top 10;

DAST - Dynamic Application Security Testing (Динамічне тестування безпеки додатків). Метод тестування чорної скриньки, який перевіряє роботу програми, щоб знайти вразливості, якими може скористатися зловмисник;

AST - Interactive Application Security Testing (Інтерактивне тестування безпеки додатків). Термін для інструментів, що поєднують в собі переваги статичного тестування безпеки додатків (SAST) і динамічного (DAST);

SCA - Software Composition Analysis (Аналіз складу програмного забезпечення). Автоматизований процес, який ідентифікує програмне забезпечення з відкритим кодом у кодовій базі. Цей аналіз виконується для оцінки безпеки, відповідності ліцензії та якості коду;

Blackbox - Метод тестування програмного забезпечення, при якому перевіряється робота програми без знання її внутрішньої побудови та схеми роботи. Іншими словами, не маючи доступу до коду програми;

Whitebox - Метод тестування програмного забезпечення який полягає у перевірці внутрішньої структури елементів системи;

TaaS - Testing as a Service (Тестуваня як послуга). Аутсорсингова модель, в якій тестування, пов’язане з бізнес-діяльністю організації, виконується постачальником послуг, а не працівниками організації.

ВСТУП

Актуальність теми. У сучасну цифрову епоху програмне забезпечення є невід'ємною частиною нашого повсякденного життя, але кількість кібератак, що зростає, вимагає надійних заходів безпеки. Постійна поява складних загроз, від витоку даних до програм-вимагачів, створює значні ризики.

Автоматизовані методи тестування пропонують ефективні способи йти в ногу із загрозами, що розвиваються, забезпечуючи стійкість і безпеку програмного забезпечення. Суворі правила ще більше підкреслюють необхідність надійних заходів безпеки, які виконуються автоматизованими методами тестування, які відповідають галузевим стандартам.

Крім безпеки, ефективні методи тестування сприяють загальній надійності програмного забезпечення, зміцнюючи довіру користувачів. Постійний розвиток технологій і складність архітектури програмного забезпечення роблять традиційні підходи до тестування недостатніми, що вимагає інноваційних та автоматизованих рішень.

Мета і завдання дослідження. Метою цього дослідження є розробка та оцінка методів автоматизованого тестування безпеки програмного забезпечення з метою покращення їх ефективності та надійності. Завдання включають аналіз існуючих методів тестування, розробку нових алгоритмів та інструментів, їх імплементацію та експериментальне порівняння для виявлення найефективніших підходів.

Об'єктом цього дослідження є сучасні автоматизовані методи тестування безпеки програмного забезпечення, зокрема спектр інструментів і методів, що застосовуються для виявлення та обробки вразливостей у програмних додатках.

Предметом дослідження є особливості та ефективність методів автоматизованого тестування безпеки програмного забезпечення. Дослідження зосереджується на розробці та оцінці різноманітних підходів та моделей, спрямованих на виявлення та усунення потенційних вразливостей в

програмному забезпеченні, що дозволить підвищити рівень їх безпеки та надійності.

Методи дослідження. У роботі застосовані різноманітні методи оцінки безпеки програмного забезпечення, включаючи статичний та динамічний аналіз вихідного коду, аналіз вразливостей та експлойтів. Дослідження також передбачає використання інструментів тестування безпеки програмного забезпечення, а також аналіз практичної цінності міжнародних стандартів та найкращих практик з безпеки.

Наукова новизна одержаних результатів. Наукова новизна дослідження полягає в розробці та оцінці нових автоматизованих методів тестування безпеки програмного забезпечення. Результати дозволяють глибше зрозуміти ефективність цих методів, пропонуючи потенційні покращення наявних методів тестування безпеки.

Практичне значення одержаних результатів. Практичне значення отриманих результатів полягає в підвищенні рівня безпеки програмного забезпечення. Практичні наслідки створюють матеріальну основу для посилення безпеки програмного забезпечення, оптимізації ресурсів і забезпечення відповідності стандартам.

Апробація результатів дослідження. Результати дослідження представлені на Міжнародній науковій конференції «Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення (випуск 85)» МНІК "Конференція онлайн". Крім того, підсумки роботи розміщено на сайті за наступним посиланням: http://www.konferenciaonline.org.ua/ua/article/id-1597/.

Структура. Кваліфікаційна робота викладена на 104 сторінках друкованого тексту, який складається із вступу, чотирьох розділів, висновків, списку та використаних джерел (42 найменувань) та додатків.

РОЗДІЛ 1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ БЕЗПЕКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1. Роль безпеки програмного забезпечення

Програмне забезпечення становить собою набір інструкцій, спрямованих на виконання конкретних завдань на комп’ютерах чи інших пристроях. Це код, написаний мовою програмування, який під час виконання дозволяє різним пристроям виконувати різноманітні функції. Програмне забезпечення виступає критичним компонентом в різних галузях, забезпечуючи функціональність пристроїв та інформаційних систем [1, с.7-13].

Життєвий цикл розробки програмного забезпечення (SDLC) представляє собою систематичний процес, що містить планування, створення, тестування, розгортання та підтримку програмного забезпечення [2, с.1-10].

Тестування програмного забезпечення є ключовим етапом у життєвому циклі його розробки з ряду об'єктивних причин:

- тестування сприяє виявленню дефектів, помилок у програмному забезпеченні;
- це необхідний компонент для забезпечення якості програмного забезпечення;
- під час тестування програмне забезпечення перевіряється на відповідність визначеним вимогам;
- тестування дозволяє виявити потенційні ризики та вразливості в програмному забезпеченні;
- високоякісне програмне забезпечення, ретельно протестоване, підвищує задоволеність користувачів;
- виявлення та усунення різноманітних дефектів, в тому числі вразливостей, на ранніх стадіях розробки економічно ефективніше, ніж їх

усунення на подальших етапах або після розгортання програмного забезпечення;

- тестування є ітеративним процесом, сприяючи постійному вдосконаленню продукту;

- у деяких галузях дотримання правових і нормативних стандартів є обов'язковим, і тестування допомагає переконатися у відповідності програмного забезпечення цим стандартам, уникаючи юридичних проблем, в тому числі можливих штрафних санкцій;

- тестування безпеки зосереджується на виявленні вразливостей та слабких місць у програмному забезпеченні, що можуть бути використані зловмисниками, забезпечуючи захист чутливої інформації;

- тестування допомагає уникнути системних збоїв і непередбачуваних ситуацій, забезпечуючи надійну роботу програмного забезпечення в різних умовах і середовищах.

Тестування безпеки програмного забезпечення є спеціалізованою формою тестування, призначеною для виявлення вразливостей і слабких місць з метою забезпечення їхньої стійкості до кіберзагроз і атак. Основна мета даного виду тестування полягає у виявленні та усуненні потенційних ризиків безпеки, що має велике значення в умовах сучасного цифрового середовища, насиченого кіберзагрозами, де захист інформації стає критично важливим аспектом [3, с.194; 4, с.72-75; 5, с. 8].

В історії кібербезпеки можна відзначити безліч відомих негативних прикладів інцидентів безпеки, пов'язаних із використанням зловмисного, що негативно вплинули на окремих осіб, компанії та урядові структури. Розглянемо кілька найвідоміших з них [6, с.1-17; 7, с.194; 8, 72-75; 9, с.8]:

- WannaCry: В травні 2017 року сталася глобальна атака програм-вимагачів WannaCry, уразивши понад 200 000 інформаційних систем у 150 країнах. За допомогою вразливості в операційній системі Windows, відомої як EternalBlue, яку розкрила група хакерів The Shadow Brokers, атака призвела до

масштабних збоїв в різних секторах, таких як охорона здоров'я, транспорт і освіта;

- NotPetya: В червні 2017 року відбулася атака зловмисного програмного забезпечення NotPetya, що маскувався під програму-вимагач і націлювався на Україну та інші країни Європи та Азії. За використання того ж експлойта EternalBlue, що й WannaCry, атака завдала шкоди багатьом організаціям, включаючи банки, аеропорти й державні установи;
- CryptoLocker: З'явившись наприкінці 2013 року, CryptoLocker став однією з перших та найуспішніших атак програм-вимагачів. Він шифрував файли користувачів і вимагав викуп у біткоїнах, завдавши значні фінансові втрати та поширюючись до понад 500 000 комп'ютерів;
- Zeus: З 2007 по 2011 рік, троянській вірус Zeus викрадав облікові дані та особисту інформацію користувачів Windows. Використовуючи різні методи, такі як клавіатурний журнал. Zeus також створював ботнет для подальших атак, збитки від яких сягали мільйонів доларів;
- ILOVEYOU: У травні 2000 року шкідливе програмне забезпечення типу «хробак» ILOVEYOU заразив мільйони комп'ютерів у всьому світі через електронну пошту, використовуючи методи соціальної інженерії, завдавши збитків на суму 5,5 мільярдів доларів США.

Протягом останніх років кібератаки стали більш витонченими, використовуючи методи, що роблять їх складнішими для виявлення та захисту. Ось кілька прикладів цих методів [10, с.5-151; 11, с.1-82]:

- розширені стійкі загрози: це довгострокові та цілеспрямовані атаки, організовані добре фінансованими та кваліфікованими фахівцями держав або організованими злочинними групами. Такого типу атаки використовують передові методи проникнення та компрометації. Вони також використовують тактику для уникнення виявлення, таку як використання законних облікових даних, змішування зі звичайним мережевим трафіком, стирання слідів своєї діяльності та використання багатоетапних атак;

- вразливість нульового дня: це атаки, які використовують невідомі вразливості в програмному або апаратному забезпеченні, які не були виправлені або розголошені постачальниками. Вразливості нульового дня дають зловмисникам значну перевагу, оскільки вони можуть обійти заходи безпеки та отримати несанкціонований доступ до систем або даних до того, як вразливість буде ідентифікована власником (постачальником) програмного забезпечення. Ці атаки часто використовуються у довгострокових та цілеспрямованих атаках досвідченими фахівцями для фокусування на цінні інформаційні активи, такі як: критична інфраструктура, державні установи чи фінансові установи.

Кібератаки – це зловмисні дії, спрямовані на порушення конфіденційності, цілісності або доступності інформаційних систем або даних. Мотиви за кібератаками можуть бути різними, але основні категорії включають кримінальні, політичні та/або особисті:

- кримінальні: це тип кібератаки, де особи або група осіб, які спрямовані на фінансову вигоду шляхом крадіжки фінансів, даних або зриву бізнесу в цілому. Прикладами злочинних кібератак є програми-вимагачі, фішинг і т.п.;
- політичні: це кібератаки, які переслідують політичну чи ідеологічну мету, таку як: вплив на вибори, викриття корупції чи просування певних інтересів. Прикладами політичних кібератак є хактивізм, кібершпигунство та кібервійна;
- особисті: це кібератаки, викликані індивідуальними емоціями, такими як помста, цікавість або нудьга. Прикладами особистих кібератак є кіберпереслідування, кіберзалякування та кібервандалізм.

Зловмисне програмне забезпечення та програмне забезпечення-вимагач стали широко розповсюдженими засобами кібератак, що можуть серйозно піддавати ризику безпеку та функціональність інформаційних систем і даних. За звітом Всесвітнього економічного форуму за 2020 рік, кількість шкідливих

програм зросла на 358%, а програм-вимагачів — на 435%. Методи здійснення цих атак можуть відрізнятися, проте деякі з найпоширеніших включають:

- фішинг: це метод, який передбачає відправлення шахрайських електронних листів або повідомлень, які маскуються під відомі джерела, такі як банки, державні установи чи довірені контакти. Ціль полягає в тому, щоб скомпрометувати користувачів, намагаючись змусити їх натискати на шкідливі посилання чи вкладення, або надавати конфіденційну інформацію, таку як паролі чи номери кредитних карток;
- експлуатація протоколу віддаленого робочого столу: це метод, який використовує службу RDP, що надає користувачам віддалений доступ до комп'ютерів через Інтернет. Зловмисники можуть використовувати вкрадені облікові дані або використовувати вразливості у програмному забезпеченні RDP для отримання доступу до систем та реалізації зловмисного програмного забезпечення чи програм-вимагачі;
- експлуатація вразливостей програмного забезпечення: це метод, що передбачає використання відомих або слабких місць у програмному чи апаратному забезпеченні цільової системи, таких як операційні системи, програми або пристрої. Зловмисники можуть використовувати ці недоліки для уникнення заходів безпеки та реалізації шкідливого вихідного коду на цільовій інформаційній системі.

Вплив зловмисного програмного забезпечення і програм-вимагачів на окремих осіб і організації може бути серйозним і збитковим, включаючи втрату або крадіжку даних, пошкодження або збоїв в роботі системи та фінансові збитки:

- втрата або крадіжка даних: Зловмисне програмне забезпечення та програми-вимагачі можуть використовувати різні методи, такі як шифрування, видалення або крадіжка, для компрометації даних з цільової інформаційної системи. Це може охоплювати особисту інформацію, фінансові записи, інтелектуальну власність чи комерційну таємницю. Наслідком може бути крадіжка особистих даних, вимагання викупу або репутаційні втрати;

- пошкодження або порушення системи: Зловмисне програмне забезпечення та програми-вимагачі можуть завдати шкоди або вивести з ладу певну цільову систему. Це може включати пошкодження файлів, зміну налаштувань або блокування доступу. Наслідком може бути зниження продуктивності, втрата функціональності чи обмеження доступу до системи, що призводить до втрати контролю над нею;
- фінансові збитки або відповідальність: Зловмисне програмне забезпечення та програми-вимагачі можуть призвести до прямих або непрямих фінансових збитків для жертв. Це може включати виплату викупу за відновлення даних чи систем, компенсацію клієнтам чи партнерам, винесення справи до суду або сплату штрафів. Наслідком таких атак може бути значна втрата ресурсів та фінансових зобов'язань.

Безпека програмного забезпечення – це практика розробки, підтримки та захисту програмного забезпечення від зловмисних атак і ненавмисних помилок [12, с.20-27; 13, с.5-6]. Роль безпеки програмного забезпечення полягає в намаганні знизити ризики, такі як порушення конфіденційності даних, системних збоїв, фінансові втрати, репутаційні пошкодження та юридичні наслідки, що можуть виникнути внаслідок існуючих вразливостей програмного забезпечення.

Практики безпечної розробки програмного забезпечення, моделювання загроз і аналіз вихідного коду – це методи, що сприяють покращенню безпеки програмних продуктів і послуг. Ось короткий деяких із тих:

- практики безпечної розробки програмного забезпечення: Стандарти та вказівки для розробників з метою запобігання вразливостей безпеки;
- моделювання загроз: Процес ідентифікації, аналізу та визначення пріоритетів потенційних загроз і ризиків;
- аналіз вихідного коду: Перевірка вихідного коду програмної системи для виявлення та виправлення дефектів безпеки, вразливостей та слабких місць.

1.2. Тестування програмного забезпечення

Тестування безпеки програмного забезпечення охоплює різноманітні методи оцінок, спрямованих на виявлення вразливостей, слабких місць і потенційних ризиків безпеки програмного забезпечення (табл.1.1) [14, с.72-75]

Таблиця 1.1

Типи тестування ПЗ та опис

Тип тестування	Опис
Сканування безпеки	Автоматизований процес виявлення вразливостей шляхом сканування програмного забезпечення з метою знаходження потенційних слабких місць.
Тестування вразливостей	Спроби експлуатації вразливостей з метою виявлення слабких місць у програмному забезпеченні та реагування на можливі загрози.
Тестування аутентифікації та авторизації	Оцінка механізмів контролю доступу для перевірки їхньої ефективності та недопущення несанкціонованого доступу.
Тестування перехоплення та обробки даних	Виявлення вразливостей, пов'язаних із збором, передачею та обробкою даних, для запобігання несанкціонованому доступу та зловживанню інформацією.
Тестування безпеки API	Оцінка безпеки прикладних програмних інтерфейсів (API) для виявлення та усунення можливих загроз та атак.
Тестування шифрування	Перевірка ефективності та надійності застосованих методів шифрування для захисту конфіденційної інформації.

Продовження Табл.1.1 Типи тестування ПЗ та опис

Тип тестування	Опис
Тестування відновлення після інциденту	Оцінка системи відновлення після кіберінцидентів для забезпечення швидкого та ефективного відновлення роботи після атаки.
Тестування безпеки мобільних додатків	Аналіз і оцінка вразливостей, що впливають на безпеку мобільних додатків.
Тестування безпеки хмарних сервісів	Перевірка заходів безпеки, реалізованих у хмарних сервісах, для запобігання несанкціонованому доступу та втрати даних.

1.3. Типи автоматизованого тестування безпеки програмного забезпечення

Автоматизоване тестування безпеки програмного забезпечення – це процес, під час якого використовуються програмні інструменти для перевірки вразливостей та виявлення та усунення можливих недоліків чи помилок в безпеці програмного забезпечення.

Автоматизоване тестування безпеки програмного забезпечення допомагає розробникам і тестувальникам переконатися, що їхнє програмне забезпечення відповідає принципам конфіденційності, цілісності, автентифікації, авторизації, доступності та відмовостійкості.

Існує кілька видів автоматизованого тестування безпеки програмного забезпечення, таких як:

- статичне тестування безпеки додатків (SAST): Аналізує вихідний код для виявлення вразливих місць без її виконання;

- динамічне тестування безпеки додатків (DAST): Імітує реальні атаки для виявлення вразливих місць безпеки;
- інтерактивне тестування безпеки додатків (IAST): Поєднує методи SAST і DAST для виявлення вразливостей безпеки програми під час тестування або розробки.

Крім того, в ході даної роботи проаналізовано найбільш поширені типи автоматизованого тестування безпеки програмного забезпечення «дивись Додаток А» [15, с.5-141; 16, с.101-108; 17 с. 117-120].

В доповнення, в ході даної роботи проаналізовано можливості найбільш розповсюджених інструментів автоматизованого тестування безпеки програмного забезпечення «дивись Додаток Б» [18, с. 112-120; 19, с. 45-53; 20 с. 87-94].

На сьогодні існує декілька найкращих практик, рекомендацій, які важливо враховувати під час процесу автоматизованого тестування безпеки програмного забезпечення включає:

- визначення методології та технік тестування, які методології та техніки тестування найкраще підходять для програмного забезпечення, та його унікальних вимог безпеки;
- інтеграція тестування у процес розробки, використовуючи підхід DevSecOps. Важливо забезпечити виконання перевірок безпеки на кожному етапі життєвого циклу програмного забезпечення;
- використання автоматизованих інструментів тестування безпеки, які здатні проводити різні типи тестів, такі як SAST, DAST, IAST;
- оцінка та усунення вразливостей, а також визначення пріоритетності вразливостей безпеки, виявлених автоматизованими інструментами тестування, які необхідно усувати, із урахуванням їхнього потенційного впливу на безпеку;
- регулярне тестування та перевірка безпеки для забезпечення того, що новий чи оновлений код не вносить нових ризиків для безпеки системи.

Цифрова трансформація процесів та послуг розширила спектр методів і засобів кібератак, створюючи більше можливостей для кіберзлочинців. Декілька факторів сприяє цьому явищу [21, 8-9; 22, 3-4; 23, 6-8]:

- перехід до хмарних технологій обчислення, які пропонують значні переваги, такі як масштабованість, гнучкість та ефективність. Однак цей перехід також породжує нові проблеми безпеки, такі як спільна відповідальність, неправильна конфігурація та витік даних;
- впровадження нових технологій таких як штучний інтелект та машинне навчання можуть значно підвищити продуктивність та інновації, однак в той же час можуть збільшити складність та вимагати нових навичок та інструментів;
- віддалена робота, пандемія COVID-19 змусила багато організацій перейти на віддалену роботу, що збільшило використання особистих пристроїв, незахищених мереж та інструментів для співпраці. Це, в свою чергу, збільшило кількість певних ризиків витоку даних, фішингу і т.д.;
- розвиток Інтернету речей (IoT) відкриває нові можливості та сервіси, але разом з тим створює величезні обсяги даних і розширює периметр мережі, ускладнюючи моніторинг і захист.

1.4. Огляд сучасних стандартів та методологій забезпечення безпеки програмного забезпечення

Огляд сучасних стандартів та методологій забезпечення безпеки програмного забезпечення є актуальною та важливою задачею для розробників, адміністраторів та користувачів програмного забезпечення, оскільки вони визначають вимоги, рекомендації, критерії, процеси, техніки, інструменти, практики та інші аспекти, які сприяють підвищенню якості, надійності, продуктивності, захисту даних та інших характеристик програмних продуктів, що використовуються в різних сферах діяльності, таких як фінанси, медицина, освіта, оборона тощо. Безпека програмного забезпечення містить певні

механізми захисту від зловмисних атак, запобігання втрати даних, забезпечення конфіденційності, цілісності та доступності інформації, яка є цінним ресурсом для будь-якої організації, інституції, компанії або особи.

Сучасні стандарти та методології забезпечення безпеки програмного забезпечення базуються на таких принципах, як:

- ризикоорієнтований підхід, за яким безпека програмного забезпечення визначається відповідно до рівня ймовірності та наслідків потенційних загроз, таких як вразливості, помилки, атаки, витік даних тощо. Цей підхід передбачає використання таких методів, як аналіз ризиків, оцінка ризиків, управління ризиками, зниження ризиків, перевірка ризиків, моніторинг ризиків тощо, які дозволяють ідентифікувати, класифікувати, оцінювати, запобігати, мінімізувати, контролювати та відстежувати ризики безпеки програмного забезпечення;
- застосування криптографічних механізмів, які дозволяють захищати конфіденційність, цілісність та автентичність даних та комунікацій, а також запобігати несанкціонованому доступу, модифікації або псуванню інформації. Криптографічні механізми включають використання симетричних та асиметричних алгоритмів шифрування, хеш-функцій, цифрових підписів, цифрових сертифікатів, протоколів захищеної передачі даних, систем розподіленого довірчого обліку тощо, які дозволяють забезпечити конфіденційність, цілісність та автентичність даних та комунікацій, а також встановити та перевірити ідентичність та повноваження сторін, що беруть участь у взаємодії;
- розробка безпечного коду, який відповідає вимогам, стандартам, нормам та найкращим практикам безпеки програмного забезпечення, а також вільний від вразливостей, помилок, дефектів, багів, вірусів, шпигунських програм, рекламних програм, шкідливих програм тощо, які можуть призвести до порушення безпеки програмного забезпечення. Розробка безпечного коду передбачає використання таких методів, як безпечне програмування, безпечний дизайн, безпечна архітектура, безпечні фреймворки, безпечні бібліотеки,

безпечні шаблони, безпечні стандарти кодування, перевірка вхідних та вихідних даних, валідація даних, шифрування даних, управління сесіями, автентифікація, авторизація, ведення журналу реєстрації подій, аудит, обробка помилок, резервне копіювання тощо, які дозволяють створювати код, що відповідає вимогам, стандартам, нормам та кращим практикам безпеки програмного забезпечення;

- тестування та аудит безпеки, які дозволяють перевіряти відповідність програмного забезпечення встановленим вимогам.

Висновки до розділу 1

У рамках дослідження, присвяченого аналізу наявних рішень у сфері безпеки програмного забезпечення, було виявлено ряд ключових аспектів, які визначають роль та важливість безпеки у цифровому середовищі.

Виділення тестування програмного забезпечення як вирішального етапу у життєвому циклі розробки підкреслює важливість вчасного та ефективного виявлення вразливостей та помилок. Автоматизоване тестування безпеки стає ключовим інструментом для забезпечення високого рівня надійності програм.

Розгляд різних типів автоматизованого тестування безпеки, таких як SAST, DAST, IAST, дозволяє краще розуміти їхні переваги та обмеження. Вибір конкретного методу повинен бути обґрунтованим та враховувати контекст конкретного проєкту.

Огляд сучасних стандартів та методології забезпечення безпеки програмного забезпечення підкреслює важливість використання належних підходів на різних етапах розробки. Впровадження таких стандартів сприяє створенню більш надійних програм.

Здобуті знання дозволять покращити ефективність тестування безпеки програмного забезпечення та підвищити рівень захисту у цифровому середовищі.

РОЗДІЛ 2. АНАЛІЗ МОДЕЛЕЙ І МЕТОДІВ ТЕСТУВАННЯ БЕЗПЕКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1. Сучасні методи розробки безпечного програмного забезпечення

Сучасні методи розробки безпечного програмного забезпечення - це ті, які враховують всі можливі загрози та вразливості, які можуть виникнути під час створення, розгортання та використання програмних продуктів, що мають високі вимоги до надійності, продуктивності, захисту даних та інших характеристик. Сучасні методи безпеки програмного забезпечення базуються на наукових дослідженнях, міжнародних стандартах, кращих практиках та досвіді провідних компаній, що займаються розробкою програмного забезпечення. Деякі з цих методів включають:

- застосування стандартів та кращих практик безпеки на всіх етапах життєвого циклу розробки програмного забезпечення (SDLC), від аналізу вимог до тестування та підтримки. Це передбачає використання таких стандартів, як ISO/IEC 27001, ISO/IEC 27034, ISO/IEC 15408, NIST SP 800-53, OWASP Top 10, SANS Top 25 тощо, які визначають вимоги, принципи, процеси, методи, техніки, ролі, артефакти та інші аспекти безпечного програмного забезпечення. Кращі практики безпеки включають такі рекомендації, як використання безпечних фреймворків, бібліотек, шаблонів,стандартів кодування, перевірки вхідних даних, валідації вихідних даних, шифрування даних, управління сесіями, аутентифікації, авторизації, журналу реєстрації подій, аудиту, обробки помилок, резервного копіювання тощо;
- використання інструментів та методик для ідентифікації, оцінки та усунення потенційних дефектів безпеки, таких як статичний аналіз коду, динамічне тестування безпеки, аудит коду, перевірка моделей тощо. Ці інструменти та методики дозволяють виявляти та усувати вразливості,

помилки, атаки, витоки даних та інші проблеми безпеки на різних рівнях абстракції, від програмного коду до архітектури, від дизайну до вимог. Деякі з найпоширеніших інструментів та методик для безпеки програмного забезпечення є: SAST (Static Application Security Testing), DAST (Dynamic Application Security Testing), IAST (Interactive Application Security Testing), SCA (Software Composition Analysis), AST (Automated Security Testing), Code Review, Threat Modeling, Penetration Testing, Vulnerability Scanning, Fuzz Testing, Security Auditing тощо;

- забезпечення навчання та освіти розробникам, менеджерам проєктів та іншим зацікавленим сторонам щодо принципів та практик безпечного програмного забезпечення, а також про актуальні загрози та контрзаходи. Це включає проведення тренінгів, семінарів, вебінарів, курсів, сертифікацій, конференцій, виставок, конкурсів, хакатонів та інших форм навчання та освіти, які сприяють підвищенню рівня знань, навичок, компетенцій, свідомості та відповідальності учасників процесу безпеки програмного забезпечення. Також до цього належить створення та підтримка культури безпеки в організації, яка спонукає до постійного вдосконалення та інновацій у сфері безпечного програмного забезпечення;
- застосування принципу найменших привілеїв, за яким кожен компонент програмного забезпечення має доступ лише до тих ресурсів, які йому необхідні для виконання своїх функцій, а також обмеження можливостей для зловмисників. Цей принцип полягає в тому, що кожен процес, користувач, роль, модуль, функція, змінна, об’єкт, файл, папка, база даних, мережа, пристрій тощо має мати мінімальний набір прав, дозволів, привілеїв, ресурсів, операцій, функцій, які необхідні для його роботи, а також мінімальний час, протягом якого вони використовуються. Це дозволяє зменшити поверхню атаки, запобігти ескалації привілеїв, ізолювати компоненти, захистити важливі дані, забезпечити принцип розділення обов’язків та інші переваги;
- використання криптографічних методів для захисту конфіденційності, цілісності та автентичності даних та комунікацій, а також для

запобігання несанкціонованому доступу, модифікації або псуванню інформації. Криптографічні методи включають використання симетричних та асиметричних алгоритмів шифрування;

- регулярне оновлення програмного забезпечення для усунення виявлених помилок безпеки та покращення його функціональності та продуктивності. Це передбачає використання таких механізмів, як автоматичне оновлення, патчі, сервісні пакети, гарячі фікси тощо, які дозволяють оперативно вносити зміни до програмного забезпечення, виправляти вразливості, додавати нові функції, підтримувати сумісність з іншими системами та пристроями. Також до цього належить проведення регулярних перевірок безпеки програмного забезпечення, використання антивірусних програм, фаєрволів, детекторів вторгнень та інших засобів захисту від зовнішніх та внутрішніх загроз.

2.2 Оцінка методів автоматизованого тестування безпеки програмного забезпечення

Для автоматичної генерації тестових вхідних даних існує різноманітність методів, починаючи від випадкового тестування Blackbox в одному випадку до символічного виконання Whitebox в іншому. Випадкове тестування Blackbox і символьне виконання Whitebox є двома протилежними методами автоматичної генерації тестових вхідних даних. Ми плануємо розглянути ці два підходи та висвітлити їхні відмінності.

Існує низка методів автоматичної генерації тестових вхідних даних, включаючи випадкове тестування чорного ящика на одному кінці та символічне виконання білого ящика на іншому.

Значно розширюючи область, статичне тестування безпеки програми (SAST) аналізує вихідний код або двійковий файл програми для виявлення вразливих місць без її виконання. Цей метод спрямований на раннє виявлення потенційних проблем під час розробки.

У контексті динамічного тестування безпеки додатків (DAST) проводиться імітація реальних атак на запущену програму для виявлення вразливих місць в її поведінці та конфігурації.

Інтерактивне тестування безпеки додатків (IAST) поєднує методи SAST і DAST для виявлення вразливостей під час тестування або розробки, забезпечуючи деталі щодо вразливостей та їх контексту.

Оцінка контролю безпеки (SCA) визначає ефективність засобів контролю безпеки, реалізованих у програмі чи системі, на основі визначених стандартів або вказівок.

Враховуючи різні особливості та виклики кожного з цих методів, їх комбінація може забезпечити більше охоплення та ефективність виявлення вразливостей:

Оцінка методів автоматизованого тестування безпеки програмного забезпечення є важливим етапом у забезпеченні ефективного виявлення потенційних вразливостей та загроз для безпеки системи. Нижче наведено аналіз декількох ключових аспектів оцінки таких методів:

- методи, які забезпечують велике покриття програми, мають високий потенціал виявлення вразливостей. Наявність невикритих частин програми може ослабити ефективність тестування;
- типи вразливостей: Методи, які виявляють різні типи вразливостей (наприклад, буферні переповнення, SQL-ін'єкції, XSS), є більш універсальними та корисними. Деякі методи можуть бути спеціалізованими і ефективними лише для певних категорій вразливостей;
- швидкість виконання: Швидкі тести дозволяють забезпечити швидкий зворотній зв'язок розробникам. Деякі методи можуть бути великими за ресурсами та часом, що може уповільнити розробку;
- автоматизація: Методи, які можна повністю автоматизувати, забезпечують ефективне використання ресурсів. Необхідність ручного втручання може призвести до збільшення трудовитрат та зменшення ефективності;

- надійність: Надійні тести мають низьку ймовірність генерації помилкових позитивів та негативів. Ненадійні тести можуть виводити у заблудження розробників та тестувальників;
- легкість інтеграції: Перевага: Методи, які легко інтегруються в чинний процес розробки, забезпечують зручність використання. Складний процес інтеграції може затримати впровадження;
- вартість використання: Ефективні тести з економічної точки зору забезпечують оптимальне використання ресурсів. Великі витрати на використання можуть бути обмежувальним чинником для широкого застосування;
- можливість виявлення нових загроз: Методи, які можуть ефективно працювати з новими видами загроз, мають довгострокову цінність. Деякі методи можуть бути менш адаптованими до швидких змін у ландшафті загроз.

2.3. Автоматизована генерація тестів

Автоматична генерація тестових вхідних даних є ключовою складовою процесу тестування програмного забезпечення, яка дозволяє виявляти помилки та покращувати його надійність. Різноманітні методи використовуються для вирішення цього завдання, і серед них виділяються випадкове тестування чорного ящика та символічне виконання білого ящика, представляючи два протилежні підходи до генерації тестових наборів. Давайте глибше розглянемо ці методи та розкриємо їхні особливості:

- випадкове тестування чорного ящика: Цей метод базується на генерації тестових вхідних даних, не враховуючи внутрішньої структури програми. Тобто тестування відбувається, ігноруючи деталі реалізації програми. Головна перевага полягає в простоті використання та здатності виявляти екстремальні випадки. Однак випадкові тести можуть бути менш

ефективними у виявленні складних помилок, таких як глибокі проблеми у внутрішній логіки програми;

- символічне виконання білого ящика: Цей метод, навпаки, дозволяє аналізувати програму, враховуючи її внутрішню структуру та логіку. Використовуючи символічні значення для представлення вхідних даних та виконання шляхів програми, цей метод забезпечує ефективне виявлення складних помилок та високий рівень покриття програмних шляхів. Однак вартість символічного виконання може бути високою через складність обчислень та обробку символьних значень.

Усі ці методи взаємодіють із завданням автоматичної генерації тестових вхідних даних та мають свої переваги та обмеження. Вибір конкретного методу залежить від конкретних вимог тестування та особливостей програмного продукту. Налаштування та оптимізація цих методів може значно покращити ефективність та результативність автоматичної генерації тестових наборів для підвищення якості програмного забезпечення.

2.4. Blackbox та фаззінг вхідних даних

Один із простих методів автоматизованої генерації тестових наборів полягає в випадковому виборі вхідних даних для програми [42]. Можна використовувати різноманітні розподіли ймовірностей вводу, такі як рівномірні або спрямовані на конкретні значення, рахуючи, що це призведе до цікавих випадкових ситуацій, наприклад, 0 −1 або MAXINT для цілих чисел. Більш вдосконалений підхід, відомий як тестування методом "фаззінг" (fuzz testing), передбачає використання добре структурованих вхідних даних, які ітеративно модифікуються випадковим чином для створення нових входів.

Це зберігає переваги тестування "чорної скриньки" (Blackbox), підвищуючи ймовірність того, що створені таким чином входи будуть "цікавими" (наприклад, здатними обійти перший рівень обробки вводу). Цей метод виявився ефективним у виявленні збоїв та вразливостей безпеки в

програмному забезпеченні, і саме таким чином було виявлено деякі з найвідоміших вразливостей безпеки [31]. Тестування методом "фаззінг" стало стандартною складовою більшості стратегій комерційного тестування програмного забезпечення [24].

Ефективність фаззінгу ґрунтується на двох ключових факторах: кількість тестів і їх якість. Кількість тестів визначає можливість генерації та випробування нових вхідних даних з великою швидкістю. У великомасштабних проектах фаззінгу, таких як ClusterFuzz, який тестує веб-браузер Chromium, програмне забезпечення тестується цілодобово, використовуючи всі доступні машини [26]. Кількість виявлених помилок обмежується ресурсами CPU, які надає фаззер — досвід вказує на те, що чим більше тестів вдається виконати фаззеру, тим вища ймовірність виявлення помилок.

Додатковий ключовий елемент — це якість тестів. По-перше, випробування великої кількості випадкових вхідних даних у "чорній скриньці" може виявити, в кращому випадку, поверхневі помилки, тоді як розумний вибір вхідних даних може проникнути глибше в програмний код і, можливо, зменшити кількість ініціалізацій, необхідних для виявлення помилки. Щоб покращити якість генерованих вхідних даних, сучасні фаззери використовують зворотний зв'язок від попередніх ініціалізацій, щоб керувати генерацією вхідних даних у напрямку тих входів, які ймовірніше виявлять помилки.

По-друге, автоматичне виявлення аномальної поведінки під час виконання тесту збільшує ймовірність виявлення прояву помилки. Таким чином, фаззери перевіряють широкий спектр "неправильних" поведінок, і найпопулярнішими є порушення безпеки пам'яті. Підставою є те, що тести вищої якості мають більшу ймовірність виявлення помилок.

Більшість сучасних фаззерів у "чорній скриньці", таких як AFL або LibFuzzer, відійшли від початкового підходу "випадкового тестування" і зараз працюють в межах зворотного зв'язку. Ці фаззери використовують інструментування для виявлення функцій програми, які активуються тестами, таких як виконання базового блоку чи переповнення буфера. Коли функція

виявляється вперше, фаззер реагує: він додає тест до свого корпусу цікавих тестових випадків або повідомляє про знайдену помилку.

Різні форми інструментування володіють здатністю виявляти різноманітні цікаві функції. Наприклад, біти покриття визначають виконання конкретного ребра в графі потоку управління програмою. Коли активується біт покриття, фаззер отримує повідомлення про виявлення нового коду. Так само, лічильник покриття слідкує за тим, як часто виконується ребро, сигналізуючи про прогрес фаззеру під час дослідження циклу. Перевірки безпеки виявляють аномальні умови, повідомляючи фаззер про виявлення помилки.

Розробники можуть додавати такі перевірки у вигляді тверджень, або вони можуть бути автоматично реалізовані за допомогою інструментів, таких як AppVerifier, Valgrind, UndefinedBehaviorSanitizer, ThreadSanitizer, AddressSanitizer, Fortify source, AFL’s libdislocator, stack-protector та інші [33, с. 485–499]. Впровадження перевірок безпеки підвищує якість тестів, отже, збільшує кількість помилок, які можуть виявляти фаззери.

Без використання цих перевірок розробники повинні сподіватися, що неправомірна поведінка призведе до виникнення помилки сегментації або іншого видимого виключення. Це не завжди відбувається, особливо в складних випадках, таких як використання пам'яті після звільнення або перевищення буфера. Ідеально було б, щоб фаззер одночасно мав велику продуктивність тестів і високу якість тестів, але, на жаль, ці два вимоги конфліктують; отримання якісного зворотного зв'язку відбувається за рахунок продуктивності. Виявлення неправильної поведінки програми та збір інформації про покриття коду здійснюється за допомогою інструментування програми, яке конкурує за тактові цикли з реальними інструкціями програми, яка аналізується. Нерідко фаззери вкладають менше половини своїх ресурсів у виконання коду цільової програми, а решта призначена для покращення якості тестів (рис. 2.1).

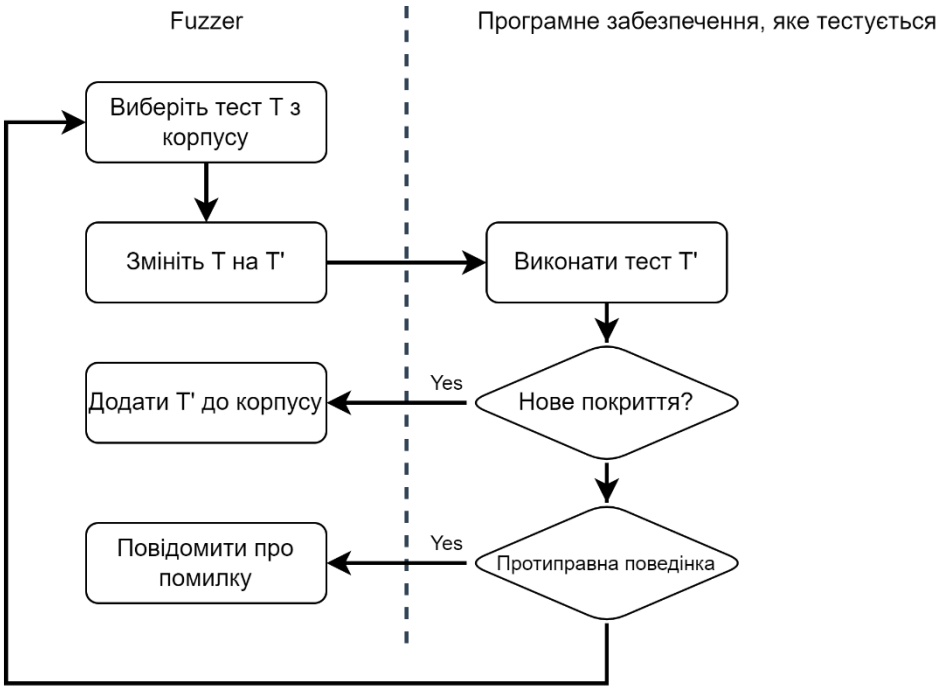


Рисунок 2.1 - Типовий робочий процес фаззера, що керується зворотним зв’язком покриття

2.5. Генерація тестів із символьним виконанням

Символьне виконання є одним з найефективніших способів перевірки безпеки програмного забезпечення, яке використовується в сучасній промисловості і наукових дослідженнях. Цей метод базується на ідеях, що код програми можна розглядати як математичну функцію, яка приймає вхідні дані і повертає вихідні дані. Замість того, щоб виконувати код з певними значеннями, як це робить звичайний інтерпретатор або компілятор, символьний виконавець використовує символьні змінні, які представляють будь-які можливі значення. Наприклад, якщо вхідним параметром функції є ціле число x, то символьний виконавець буде використовувати символ x замість конкретного числа, такого як 42 або -7.

Використовуючи символьні змінні, символьний виконавець може відстежувати, як вони впливають на стан програми й вихідні дані. Крім того, він може використовувати різні техніки, такі як розв’язування обмежень,

абстрактна інтерпретація або динамічне символьне виконання, щоб ефективно обходити різні шляхи виконання програми. Шлях виконання — це послідовність інструкцій, які виконуються при певному наборі вхідних даних. Різні шляхи виконання можуть призвести до різних результатів або поведінки програми.

Однією з основних мет цього методу є генерація тестів безпеки програмного забезпечення. Тест безпеки — це тест, який перевіряє, чи не містить програма вразливостей, які можуть бути використані зловмисниками для компрометації системи. Деякі з найпоширеніших вразливостей включають буферні переповнення, виконання довільного вихідного коду, витоки інформації, відмову в обслуговуванні, ін’єкцію коду або запитів, перехоплення сесії, підміну даних і т.д. Ці вразливості можуть призвести до серйозних наслідків, таких як втрата конфіденційності, цілісності або доступності даних, порушення конфіденційності, крадіжка ідентичності, шантаж, шкода або знищення обладнання тощо.

Символьне виконання генерує тестові випадки, які покривають різні шляхи виконання програми, і перевіряє, чи не порушуються умови безпеки. Умови безпеки - це логічні вирази, які визначають, які стани програми є безпечними або небезпечними. Наприклад, умова безпеки може стверджувати, що розмір буфера не повинен перевищувати певного ліміту, або що вихідні дані не повинні містити конфіденційної інформації. Якщо символьний виконавець знаходить шлях виконання, який порушує умову безпеки, він повертає відповідний тестовий випадок, який демонструє вразливість. Цей тестовий випадок може бути використаний для виправлення помилки в коді програми.

Символьне виконання є потужним інструментом для генерації тестів безпеки програмного забезпечення, оскільки воно дозволяє аналізувати код програми на більш високому рівні абстракції, ніж традиційні методи, такі як фаззінг або статичний аналіз. Фаззінг — це метод, який генерує випадкові або некоректні вхідні дані і спостерігає за реакцією програми. Статичний аналіз — це метод, який аналізує код програми без його виконання, використовуючи

різні евристики або правила. Обидва ці методи мають свої переваги й недоліки, але вони не можуть гарантувати повне покриття всіх можливих шляхів виконання програми або виявлення всіх потенційних вразливостей. Символьне виконання, навпаки, може бути більш точним і повним, оскільки воно враховує всі можливі значення вхідних даних і всі можливі наслідки їх використання.

Символьне виконання допомагає забезпечити надійність і захист програмного забезпечення від зловмисних атак, оскільки воно дозволяє виявляти й усувати вразливості на ранніх етапах розробки або тестування. Це може зменшити ризики, пов’язані з безпекою, і підвищити довіру до якості програмного забезпечення.

На іншому полюсі спектру найбільш точної форми автоматизованої генерації тестів, що використовується на сьогодні, - це динамічна генерація тестів з символьним виконанням. Символьне виконання - це метод аналізу програм, який виник у 1970-х роках. Символьне виконання передбачає виконання програми з використанням символьних значень, а не конкретних. Оператори присвоєння представлені як функції своїх (символьних) аргументів, а умовні оператори виражені як обмеження на символьні значення. Символьне виконання використовується для різних цілей, включаючи виявлення помилок, перевірку програм, налагодження, обслуговування та локалізацію помилок. Воно може служити для символьного дослідження структури всіх обчислень, які виконує програма під час розгляду всіх можливих присвоєнь значень вхідних параметрів.

Розгляньмо просту програму та її структуру обчислень. Ця програма приймає на вхід ціле значення «про/хв». Безліч можливих значень для програмної змінної «rpm» представлено символьним значенням λ, яке спочатку може приймати будь-яке ціле значення, позначається обмеженням $\lambda \in Z$. Під час символьного виконання цієї програми, щоразу, коли гілка, що залежить від λ, виникає нове обмеження, що визначає, як зробити так, щоб умова розгалуження, залежна від вхідних даних, оцінювалася як істинна (наприклад, > 1000) або як помилкова (наприклад, ≤ 1000). Ітеративно відстежуючи цей

процес по дереву, ми отримуємо структуру виконання, з зв'язками вхідних обмежень, які характеризують вхідні значення, необхідні для досягнення різних частин програми. Ці поєднання обмежень називаються обмеженнями шляху або умовами шляху та зображені сірим кольором у правій частині (рис. 2.2).

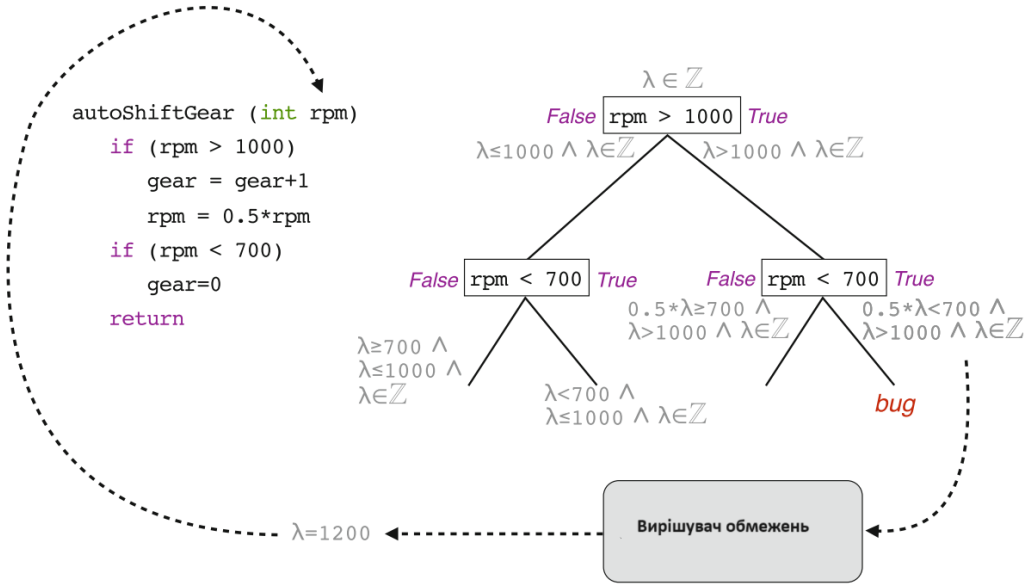


Рисунок 2.2 - Генерація тесту для простої програми з використанням символьного виконання

Проще кажучи, для кожного шляху керування (позначеного як p), який представляє послідовність місць керування в програмі, формують обмеження шляху (позначене як φp), щоб описати умови вхідних даних, за якими програма слідує шляху p. Алгоритм пошуку може перераховувати всі шляхи, досліджуючи різні гілки в умовних операторах. Шляхи (p), для яких φp задовільне, вважаються здійсненими і є єдиними, які може виконати фактична програма. Рішення для φp характеризують вхідні дані, які направляють програму через шлях p. Ця характеристика є точною, якщо символьне виконання досягає ідеальної точності. Якщо припустити, що перевірка теорем, яка використовується для перевірки виконання всіх формул φp, є обґрунтованою та повною, цей аналіз нагадує вичерпне символьне тестування всіх можливих шляхів керування в програмі.

Зусилля з автоматичної генерації тестів на основі коду з використанням символьного виконання можна загалом розділити на дві групи: генерація статичних і динамічних тестів. Генерація статичного тесту передбачає статичний аналіз програми P за допомогою методів символьного виконання для обчислення вхідних даних, які направляють P вздовж певних шляхів виконання або гілок без фактичного виконання програми. З іншого боку, генерація динамічного тесту передбачає виконання програми P із конкретними вхідними даними, динамічно використовуючи символьне виконання для збору символьних обмежень на вхідні дані з предикатів у операторах розгалуження під час виконання, а потім, використовуючи засіб розв'язання обмежень, вивести варіанти попередніх вхідних даних, щоб направити наступне виконання програми до альтернативної гілки програми [39].

Ключова практична перевага динамічної генерації тестів порівняно зі статичною генерацією полягає в тому, що для створення тесту не потрібно виконувати символьну програму. Неточності в динамічному символьному виконанні можна легко виправити, використовуючи конкретні значення: кожного разу, коли динамічне символьне виконання не може створити обмеження для оператора програми, залежно від вхідних даних, завжди можна спростити це обмеження, використовуючи поточні конкретні значення цих даних. Можна довести, що динамічна генерація тестів є точнішою, ніж статична генерація тестів, переважно через її здатність спостерігати конкретні значення та включати їх в обмеження шляху. На практиці, ключовою перевагою символьного виконання є те, що воно може створювати якісні тестові дані, які прокладають шляхи програми з великою точністю, що перевершує випадкове тестування або інші методи генерації тестів на основі евристики чорної скриньки. Однак символьне виконання не досягає ідеальної точності, інструменти розв'язання обмежень зазвичай не є абсолютно надійними та повними, а програми можуть мати (нескінченно) багато шляхів управління через цикли чи рекурсію. Крім того, правильне розроблення символьного виконання є вельми складним завданням.

2.6. Тенденції розвитку методів тестування безпеки

Методи автоматизованого тестування безпеки програмного забезпечення є важливим інструментом для забезпечення якості та надійності програмних продуктів, які використовуються в різних сферах діяльності, таких як фінанси, медицина, освіта, оборона тощо. Метод полягає в перевірці відповідності програмного забезпечення встановленим стандартам, нормам та вимогам безпеки, а також в виявленні та усуненні потенційних загроз, таких як вразливості, помилки, атаки, витоки даних тощо. Метод має багато переваг перед ручним тестуванням безпеки, таких як швидкість, точність, економія часу та ресурсів, покриття великої кількості тестових випадків, зменшення людського фактора тощо. Однак, Метод також зазнає змін і вдосконалення відповідно до сучасних вимог і технологій, які постійно розвиваються та ускладняються. Деякі з тенденцій розвитку цих методів є:

- застосування штучного інтелекту (ШІ) та машинного навчання (МН) для підвищення ефективності та точності методу. Це дозволяє адаптувати тестові сценарії до різних контекстів, виявляти складні вразливості та генерувати оптимальні рішення. Наприклад, за допомогою ШІ та МН можна створювати інтелектуальні тестові генератори, які аналізують програмний код, специфікації, історію змін, поведінку користувачів та інші джерела даних, і на основі цього формують тестові випадки, які максимально покривають можливі сценарії використання та атаки. Також, за допомогою ШІ та МН можна реалізовувати інтелектуальні тестові аналізатори, які оцінюють результати методу, визначають причини та наслідки виявлених проблем, пропонують рекомендації щодо їх вирішення та запобігання;
- використання хмарних сервісів та мікро сервісної архітектури для надання гнучкості та масштабованості методу. Це дозволяє проводити тестування в реальному часі, використовувати ресурси за запитом та інтегрувати різні компоненти програмного забезпечення. Наприклад, за

допомогою хмарних сервісів можна створювати віртуальні середовища для методу, які імітують реальні умови функціонування програмного забезпечення, включаючи різні платформи, пристрої, мережі, операційні системи тощо. Також, за допомогою хмарних сервісів можна забезпечувати безперервність та автоматизацію процесу методу, який включає такі етапи, як планування, виконання, аналіз, звітність, моніторинг тощо. Щодо мікро сервісної архітектури, вона дозволяє розбивати програмне забезпечення на незалежні модулі, які можна тестувати окремо та паралельно, що підвищує швидкість та якість методу;

- залучення користувачів та експертів до процесу методу за допомогою краудсорсингу та гейміфікації. Це дозволяє отримати більше даних, збагатити тестовий набір, покращити якість повернення зворотного зв’язку та підвищити мотивацію учасників. Наприклад, за допомогою краудсорсингу можна залучати велику кількість користувачів та експертів, які мають різний досвід, знання, навички, інтереси, очікування тощо, до проведення методу, що дозволяє враховувати різні сценарії використання та атаки, які можуть бути не передбачені автоматизованими методами. Також, за допомогою гейміфікації можна стимулювати учасників методу, використовуючи різні ігрові елементи, такі як рейтинги, нагороди, рівні, завдання, конкурси тощо, що дозволяє підтримувати їхню зацікавленість, відданість, задоволення та співпрацю.

Висновки до розділу 2

Вивчення сучасних методів розробки безпечного програмного забезпечення дозволяє визначити, що впровадження відповідних методів на етапі розробки сприяє створенню додатків, які володіють вбудованими заходами безпеки. Це є важливою складовою в підтримці високого рівня безпеки додатків.

Оцінка методів автоматизованого тестування безпеки програмного забезпечення вказує на їхню ефективність, але вимагає індивідуального підходу

для кожного проєкту. Застосування різних методів (SAST, DAST, IAST) дозволяє створити комплексну стратегію тестування.

Використання автоматизованої генерації тестів допомагає створювати різноманітні тестові сценарії, що важливо для виявлення вразливостей та підвищення загального рівня безпеки.

Аналіз Blackbox та фаззінгу підкреслює їхню ефективність у виявленні вразливостей через створення невідомих сценаріїв, що може бути критичним для ефективного тестування безпеки.

Використання генерації тестів із символьним виконанням підкреслює їхню здатність виявляти внутрішні вразливості та допомагає у покращенні загальної безпеки програмного забезпечення.

Аналіз тенденцій розвитку методів тестування безпеки вказує на важливість інтеграції новітніх технологій та підходів, таких як штучний інтелект, для підвищення ефективності процесу тестування.

Загальною метою розділу є виявлення оптимальних підходів до тестування безпеки програмного забезпечення, які враховують особливості кожного проєкту та забезпечують стійкість до потенційних загроз. Результати дослідження можуть бути використані для покращення стратегій тестування безпеки в практичних проєктах, забезпечуючи високий рівень захисту програмних систем.

РОЗДІЛ 3. РОЗРОБКА АЛГОРИТМІВ ТА ТЕХНОЛОГІЙ, ПРОЕКТУВАННЯ ПЗ ДЛЯ ВИРІШЕННЯ ПРОБЛЕМ БЕЗПЕКИ

3.1. Вивчення нових програмних шляхів

Динамічне символьне виконання є систематичним методом для навігації по структурі виконання програми. Ці шляхи поступово розкриваються, і їх можна досліджувати незалежно "паралельно". Кожен внутрішній вузол вказує на рішення розгалуження, тоді як кожен аркуш втілює стан програми з унікальним адресним простором, програмним лічильником і набором обмежень на програмні змінні. Щоб дослідити новий програмний шлях у цьому систематичному пошуку, обмеження шляху для цього нового шляху вирішується за допомогою засобу вирішення обмежень. Якщо обмеження виявляється достатнім, надається задовільне присвоєння для кожної символьної змінної в обмеженні, яке згодом переводиться в нові вхідні дані програми.

У сценаріях реального застосування символьного виконання для автоматичного створення тестів стикається з рядом помітних обмежень. У цьому розділі розглянуті ці проблеми та запропоновані різні рішення. На щастя, практичні випадки часто знаходять прийнятні альтернативи. Символьне виконання не обов'язково повинно бути ідеальним; воно просто має бути "достатньо добрим", щоб пройти по тестовій програмі через гілки, оператори та шляхи, які можуть бути важко пройти іншими, менш складними методами, такими як випадкове тестування. Хоча систематичний пошук може зустрічати труднощі у вивченні всіх можливих шляхів у великих програмах за обмежений час, він, зазвичай, забезпечує краще охоплення, ніж чисто випадкове тестування, що дозволяє виявляти нові помилки в програмному забезпеченні.

Існують два основні підходи до вивчення нових програмних шляхів. Один із методів передбачає запуск програми з певними фіксованими вхідними

даними, проведення динамічного символьного виконання під час виконання (з використанням інструментарію часу виконання), доки програма не завершиться або не досягне попередньо визначеного ліміту. У цьому підході для заглиблення в інший шлях виконання потрібно перезапустити цільову програму з самого початку з оновленими конкретними вхідними даними [39].

Другий метод передбачає буквальне "розгалуження" (за допомогою системного виклику fork) стану програми перед прийняттям рішень про розгалуження. Це створює новий адресний простір для дубліката програми, який потім досліджує альтернативний шлях через структуру. Методи копіювання під час запису ефективно усувають повторювані адресні простори.

Ці два підходи включають в себе різні компроміси. Перший метод, який використовує конколічне виконання у стилі DART, має перевагу в тому, що для визначення доцільності шляхів виконання не потрібно вирішувати обмеження під час виконання, що споживає менше пам'яті. Проте для кожного досліджуваного шляху потрібно перезапускати програму з самого початку. Другий підхід, який використовує символьне виконання у стилі KLEE, може ефективно досліджувати шляхи паралельно без перезапуску програми, використовуючи гнучкі стратегії пошуку для визначення, які шляхи програми вивчати далі, зокрема при наявності циклів, засновано на різних факторах. Однак цей підхід вимагає більше обчислювальних ресурсів, особливо для вирішення обмежень та визначення можливості здійснення шляху, а обмеження пам'яті обмежують масштабованість.

3.2. Взаємодія з оточенням

Наявні рішення, зазвичай, поділяються на дві категорії: конкретизація звернень до середовища, обхід символьного виконання середовища або абстрагування середовища за допомогою моделей з різним ступенем повноти.

Теоретично символьне виконання, позбавлене абстракції, зберігає повну точність стосовно семантики трансформатора предикату. Воно генерує "умови

перевірки для кожного шляху", виконання яких передбачає досяжність певного твердження, що мінімізує ймовірність помилкових спрацьовувань. Однак у практичних програмах необхідні механізми символьного виконання для взаємодії з середовищем виконання програми, що включає зовнішні бібліотеки, операційну систему, планувальники потоків і процесів, події переривань вводу/виводу і т.д. Таким чином, ці механізми повинні оптимізувати своє виконання в середовищі, забезпечуючи при цьому правильну поведінку програми.

Найперше механізми символьного виконання, такі як, застосовують підхід конкретного виконання [39]. Вони підтримують символьні стани виконання виключно під час виконання самого програмного коду. Коли символьне виконання стає недоцільним, наприклад, у викликах зовнішньої бібліотеки або при зустрічі з інструкціями з невідомою символьною семантикою, виконання програми може продовжуватися конкретно. Цей підхід змінює традиційну роль символьного виконання, перетворюючи його в додаток до конкретного виконання. Таким чином, конкретне виконання може слугувати автоматичним резервом для символьного виконання [39]. Це дозволяє поетапне впровадження: лише певні оператори програми потребують символьної інтерпретації, дозволяючи виконувати інші конкретно нативно. Розробник інструменту може підвищити точність символьного виконання з часом, модульно додаючи нові обробники інструкцій. Однак помітним недоліком є відсутність дослідження поведінки програми, що відповідає поведінці середовища.

Щоб подолати обмеження підходу до конкретного середовища, існує альтернативний метод, який передбачає моделювання середовища під час символьного виконання. Наприклад, KLEE перенаправляє виклики середовища до невеликих функцій, спеціально створених для розуміння семантики бажаної дії, генеруючи обґрунтовані відповіді. Всього 2500 рядків коду KLEE включають моделі для близько 40 системних викликів Linux, таких як

open, read і stat. Ці моделі виступають абстракціями фактичних реалізацій цих системних викликів.

Перевага цього підходу до моделювання полягає в тому, що цільові програми піддаються більш різноманітному діапазону поведінки навколишнього середовища, що дозволяє оцінити їхню реакцію. Наприклад, як програма реагує, коли операція запису у файл не вдається через повний диск? Добре розроблена модель запису може надати символьне значення, яке повертається, залежно від успіху чи невдачі операції. Крім того, моделі можуть бути написані для повернення різних кодів помилок для різних збоїв, що дозволяє механізму символьного виконання автоматично тестувати програму за цими сценаріями.

Однак цей підхід має два основні недоліки. По-перше, модель, яка є абстракцією реального коду, може не охоплювати всю можливу поведінку цього коду. Якби модель була повністю точною, вона, по суті, відображала б фактичну реалізацію. По-друге, ручне створення моделі є трудомістким і вразливим до помилок. Щоб усунути ці недоліки, вибіркове символьне виконання не використовує моделі, а замість цього автоматично абстрагує середовище. Цей процес керується моделями узгодженості, які визначають, коли потрібно перебільшувати, а коли недооцінювати.

3.3. Вибір шляху

Для висвітлення різних компромісів, ми подаємо більш детальний опис роботи механізму символьного аналізу за допомогою алгоритму 1 у форматі робочого списку. Функціональність алгоритму налаштована трьома функціями: pickNext для вибору наступного стану для додавання в робочий список, next, яка визначає, чи слід розглядати гілку, і відношення ~, яке керує злиттям або розділенням станів програми (деталі будуть розглянуті далі). Стан програми представлений у вигляді трійки (pc, s), яка включає розташування програми,

умову шляху pc і символьне сховище s, яке відображає кожну змінну або на конкретне значення, або на вираз над вхідними змінними.

Виконання символьного аналізу для всіх можливих програмних шляхів стикається з викликом масштабованості, особливо в разі великих програм. Кількість потенційних шляхів у програмі може зростати експоненційно залежно від її розміру. У випадках, наприклад, мережевих серверів з ітеративними циклами, залежно від необмежених вхідних даних (наприклад, потоку мережевих пакетів), кількість шляхів може бути навіть нескінченною. Наприклад, веббраузер Firefox з понад 500 000 операторів if може потенційно генерувати близько 2500 шляхів, навіть якщо лише невелика частина з них має можливі гілки then і else для певних вхідних даних. Це число все ще велике порівняно з кількістю атомів у спостережуваному Всесвіті. У наступних розділах ми розглянемо стратегії для вирішення цього виклику щодо вибуху шляху.

Алгоритм розпочинається ініціалізацією робочого списку, де символьний магазин відображає кожну змінну на себе (ігноруючи іменовані константи для спрощення) (рядок 1). Тут λx.e позначає функцію, яка відображає параметр x у вираз e, а λ(x1,...xn).e відображає кілька параметрів. На кожній ітерації алгоритм вибирає новий стан з робочого списку (рядок 3). Після зустрічі з призначенням v := e (рядки 5–6) алгоритм генерує наступний стан у проміжному місці наступного succ() поточного стану, оновлюючи символьне сховище s з використанням відображення з v на новий символьний вираз, отриманий під час обчислення e в контексті s. Потім новий стан додається до набору S. У кожній гілці (рядки 7–11) алгоритм оцінює, чи слід слідувати будь-якому шляху, додаючи відповідну умову до наступного стану, який потім додається до S. Механізм символьного виконання може вибрати не слідувати гілці, якщо він вважає, що розгалуження неможливе або якщо воно перевищить обмеження на розгортання циклу.

Для тверджень (рядки 12–13), умова шляху, символьний магазин і заперечення твердження об'єднуються та перевіряються на виконуваність.

Інструкції "Halt" позначають завершення аналізу програми, і алгоритм виводить умову шляху. Задовільне призначення цієї умови може бути використано для створення тестового прикладу для виконання, що призводить до зупинки.

У рядках 16–21 нові стани в S об'єднуються з будь-якими відповідними станами в робочому списку перед додаванням до самого робочого списку. Два стани вважаються відповідними, якщо вони знаходяться в одному місці та схожі на основі відношення ∼. Процес об'єднання створює диз'юнкцію двох умов шляху та створює об'єднане символьне сховище з виразів, які стверджують будь-яке з вихідних значень, залежно від обраного шляху (рядок 19).

Вхідні дані (рис. 3.1): функція вибору "pickNext", відношення подібності ∼, засіб перевірки розгалужень і початкове розташування 0. Дані: робочий список "w" і набір наступних станів "S".

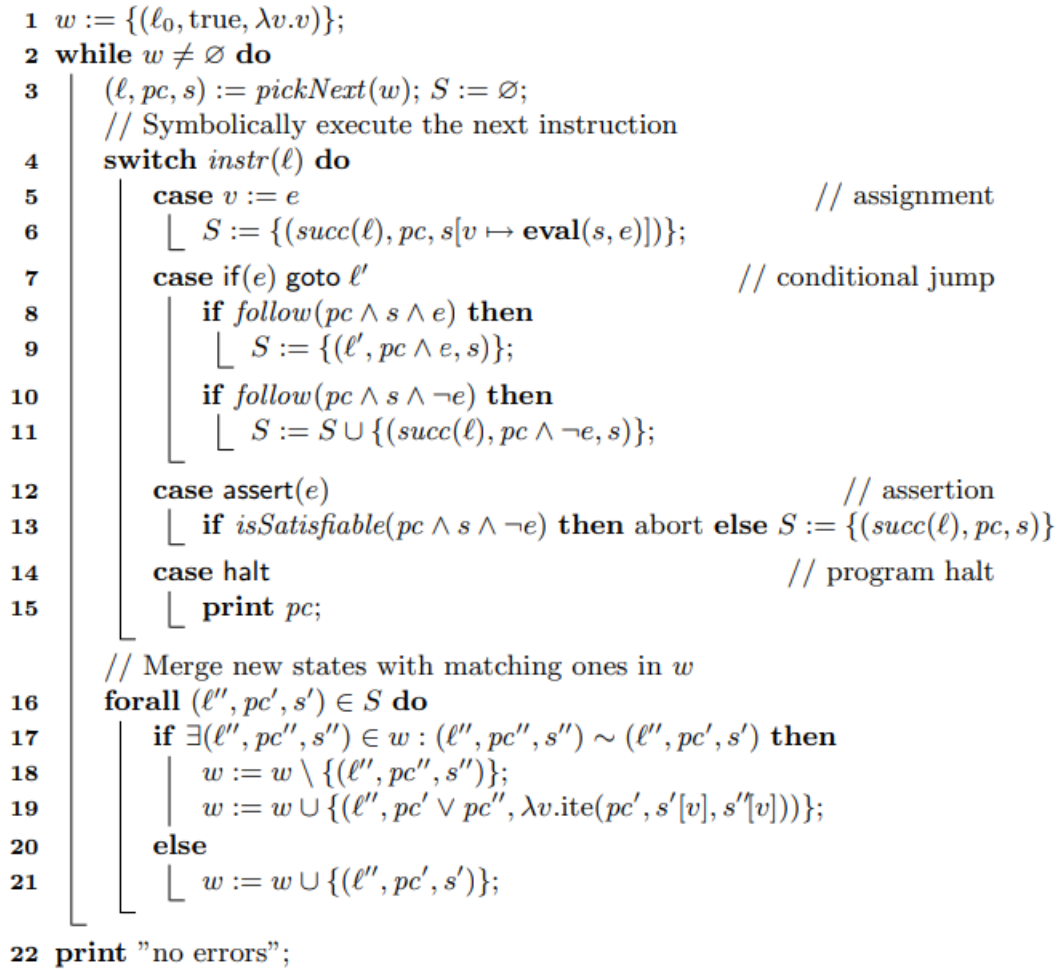


Рисунок 3.1 - Загальне символьне виконання

Перевірки обмеженої моделі й розширені статичні перевірки використовують розгортання циклу до заданої межі з можливістю ітеративного збільшення межі, якщо твердження розкручування не вдається [38]. Цей процес розгортання зазвичай досягається статичним переписуванням графа потоку керування. Його можна інтегрувати в Алгоритм 1, налаштувавши функцію "follow" на повернення "false" для розгалужень, які перевищують вказану межу розгортання циклу.

Цей алгоритм являє собою загальне символьне виконання програм, написаних простою мовою введення. Він обробляє призначення, умовні оператори, твердження та оператори зупинки. Алгоритм є внутрішньо процедурним, а виклики функцій мають бути вбудованими. Точні підсумкові символьні функції можуть бути згенеровані, якщо алгоритм викликається для процедури з відношенням подібності, яке об’єднує всі стани, коли функція завершується.

До тої пори, поки воно не може довести нездійсненність умови циклу, що потенційно може призвести до нескінченної кількості розгортань для необмежених циклів (рис. 3.2)

```
1: procedure SymbolicExecution(input_program)
2:     w ← {initial state with symbolic store mapping each variable to itself}
3:     S ← Ø   // Set of new states
4:     while w ≠ Ø do
5:       (pc, s) ← pickNext(w)  // Pick a state to expand
6:         if pc is an assignment v: = e then
7:             s' ← evaluate (e, s) // Update symbolic store with new expression
8:             S ← S ∪ {(succ(), s')}  // Add new state to set
9:         else if pc is a branch statement then
10:            if follow (pc, s) then
11:                S ← S ∪ {(true_branch(), addCondition(pc, s))}
12:            if follow(not(pc), s) then
13:                S ← S ∪ {(false_branch(), addCondition(not(pc), s))}
14:        else if pc is an assertion then
15:            if not isSatisfiable(conjunction(pc, s, not(evaluate(pc, s)))) then
16:                reportError("Assertion failed:", pc)
17:        else if pc is a halt statement then
18:            output(pc)  // Output path condition
19:        for each state' in S do
20:            for each state" in w do
21:                if state' ~ state" then
22:                    w ← w \ {state"}  // Remove matching state from worklist
23:                    w ← w ∪ {merge(state', state")}  // Add merged state to worklist
24:    end while
25: end procedure
```

Рисунок 3.2 - Нескінченний цикл

Перевага цього підходу до моделювання полягає в тому, що цільові програми піддаються більш різноманітному спектру поведінки навколишнього середовища, що дозволяє оцінити їх реакцію. Наприклад, якщо операція запису у файл не вдається через повний диск, добре розроблена модель може надати символьне значення, що повертається залежно від успіху чи невдачі операції. Моделі також можуть повертати різні коди помилок для різних випадків відмов, що дозволяє символьному виконанню автоматично тестувати програму за такими сценаріями.

Щоб подолати обмеження підходу до конкретного середовища, альтернативний метод передбачає моделювання середовища під час символьного виконання. Наприклад, у KLEE виклики середовища перенаправляються до невеликих функцій, які розуміють семантику бажаної дії, генеруючи обґрунтовані відповіді. Приблизно 2500 рядків коду KLEE включають моделі для близько 40 системних викликів Linux, таких як open, read і stat. Ці моделі служать абстракціями фактичних реалізацій цих системних викликів. Крім того, розширив моделі для KLEE, щоб охопити повне середовище POSIX [41].

Окрім евристики пошуку, інший спосіб зменшення кількості шляхів для дослідження під час символьного виконання передбачає запам’ятовування та об’єднання станів програми, досягнутих різними шляхами. Цей підхід до композиційного аналізу програм, вже широко використовуваний у статичному аналізі, є важливим для масштабованості великих програм . У статичному аналізі об’єднаний стан, як правило, надто наближено представляє окремі стани, що об’єднуються, що може призводити до потенційних неточностей та помилкових спрацьовувань (тобто невірних виконань) [41]. Однак у символьному виконанні для генерації тестів об’єднаний стан повинен точно представляти інформацію з усіх шляхів виконання, включених до цього стану, без надто наближеного представлення. Взагалі кажучи, композиційний статичний аналіз обчислює та запам’ятовує "можливі" максимально схожі

підсумки функцій, тоді як генерація композиційного тесту обчислює "повинні" занижені підсумки.

Однак цей підхід має два основних недоліки. По-перше, модель, яка є абстракцією реального коду, може не охоплювати всю можливу поведінку цього коду. Якби модель була повністю точною, вона, по суті, відображала б фактичну реалізацію. По-друге, створення моделі вручну є ресурс затратним та вразливим до помилок. Щоб усунути ці недоліки, вибіркове символьне виконання не використовує моделі, а замість цього автоматично абстрагує середовище. Цей процес керується моделями узгодженості, які вказують, коли потрібно перебільшувати або недооцінювати.

Результати цього символьного виконання запам’ятовуються за допомогою локальних вхідних передумов. Тоді символьне резюме для процедури визначається як диз'юнкція підсумків її внутрішньо процедурних шляхів. Коли високорівнева процедура (наприклад, foo) викликає вже підсумовану процедуру (наприклад, bar), підсумок для bar повторно використовується та включається в умову поточного шляху foo. Повторне використання символьного підсумку ефективно об’єднує всі стани, доступні після повернення підсумкової процедури. Ці зведені дані можна обчислювати різними способами, наприклад, у внутрішньому найперше порядку або ліниво на вимогу. Їх також можна використовувати в поєднанні з "можливими" підсумками, створеними за допомогою статичного аналізу.

Символьні підсумки в генерації тестів можуть бути обчислені на різних рівнях, таких як рівень блоку, методу, функції або процедури, або навіть в довільних точках програми. Ці підсумки можна обчислювати поступово, один між процедурний шлях за разом, а потім об’єднувати за допомогою диз'юнкції, як показано в рядках 17–22 Алгоритму 1. Цей підхід має перевагу в тому, що кожен внутрішньо процедурний шлях символьно виконується лише один раз.

На жаль, використання зведень і злиття станів ускладнює і збільшує вартість обчислень як для символьного виконання, так і для вирішення обмежень. Підсумки, представлені як диз’юнкції шляхів підпрограми,

необхідно обчислювати та запам’ятовувати, що призводить до більш складних умов шляху з додатковими диз’юнкціями. Це ускладнює вирішення цих умов. Отже, в літературі були запропоновані різні компроміси. Два надзвичайні компроміси включають (i) повне розділення шляхів без злиття станів взагалі, як це видно в деяких механізмах символьного виконання на основі пошуку, і (ii) повне статичне злиття станів, реалізоване перевіркою генераторів умов [39]. Повне статичне об’єднання станів об’єднує стани в точках з’єднання після кодування всіх шляхів, досліджуючи всі під шляхи, що ведуть до точки з’єднання, перед тим як обрати будь-які стани в точці з’єднання. У цьому підході відношення ~ містить усі пари станів. Деякі підходи використовують проміжні стратегії злиття. Наприклад, у перевірці обмеженої моделі (BMC) Ганай і Гупта досліджують розбиття умови перевірки вздовж підмножин шляхів, переміщаючи BMC до символьного виконання. Хансен та інші описують реалізацію статичного злиття станів, яка модифікує стратегію дослідження для обходу графа потоку керування в топологічному порядку та злиття станів з тим же розташуванням програми. Однак ця стратегія може збільшити час вирішення через складність умов об’єднаного шляху.Альтернативний підхід до реалізації злиття станів у механізмі символьного виконання полягає в тому, щоб представити механізму еквівалентний варіант цільової програми, яку легше символьно виконати. Overify, наприклад, компілює програми, щоб мати найпростіший можливий потік керування, використовуючи методи, такі як перемикання потоків, відключення циклу та інші оптимізації. Ця компіляція скорочує час повного символьного виконання майже на два порядки.

Проблема в об’єднанні станів полягає в балансуванні зменшення кількості шляхів виконання, яке може бути експоненціальним, із збільшенням часу, витраченого механізмом символьного виконання на вирішення більш складних обмежень. У цьому підході використовується алгоритм оцінки кількості запитів, щоб обчислити під час символьного виконання, чи переважує перевага продуктивності, що виникає внаслідок зменшення кількості шляхів

виконання, збільшення часу вирішення обмежень. Експерименти показали послідовне прискорення порівняно із сучасним рівнем техніки.

3.4. Ефективне вирішення обмежень

Ці інструменти відрізняються тим, як вони виконують динамічне символьне виконання (для таких мов, як C, Java, x86, .NET), за типом обмежень, які вони створюють (для таких теорій, як лінійна арифметика, бітові вектори, масиви, не інтерпретовані функції тощо), а також типом програм вирішення обмежень, які вони використовують (наприклад, lp solve, CVClite, STP, Disolver, Yikes, Z3). Вибір цих параметрів зазвичай визначається конкретним типом програми, що підлягає тестуванню, взаємодією програми з її середовищем і властивостями, які потрібно перевірити, подібно до традиційного статичного аналізу програми та абстрактної інтерпретації.

Іншим важливим компонентом у процесі генерації динамічного тесту є засіб вирішення обмежень, який використовується для вирішення обмежень шляху. За останнє десятиліття було розроблено кілька інструментів для реалізації генерації динамічних тестів для різних мов програмування, властивостей і областей застосування. Прикладами таких інструментів є DART, EGT, PathCrawler, CUTE, EXE, SAGE, CatchConv, PEX, KLEE, CREST, BitBlaze, Splat, Apollo, YOGI, Kudzu, S2E і JDart [39; 25, с. 474–494].

Крім того, існують різні компроміси між вартістю та точністю під час створення та вирішення обмежень. На щастя, завдяки програмам для створення тестів, які обговорюються в цій роботі, автоматизоване доведення теорем досягло значного прогресу за останнє десятиліття. Зокрема, ключову роль відіграла поява модульної теорії виконуваності (SMT) [36]. З-за допомогою подібних підходів ефективно перевіряється виконання комплексних обмежень, виражених у насичених областях. Крім того, за останні роки вони стали доступнішими з точки зору обчислень завдяки збільшенню обчислювальної потужності сучасних комп’ютерів.

3.5. Розпаралелювання та тестування як хмарної служби

Ці методи дозволяють ефективно виявляти і усувати помилки та вразливості в програмному коді, використовуючи символьне виконання, яке моделює всі можливі шляхи виконання програми та перевіряє їх на коректність. Ці методи можуть бути використані як для тестування окремих модулів програми, так і для тестування цілих систем, які складаються з багатьох компонентів, що взаємодіють між собою.

Однак, ці методи також мають свої виклики та обмеження. По-перше, вони потребують значних обчислювальних ресурсів, що може ускладнити їх застосування до великих та складних програм. По-друге, вони можуть порушувати конфіденційність та безпеку розроблених методів, якщо вони будуть несанкціоновано використані або неправильно.

Тому, для ефективного та безпечного використання цих методів, можна застосувати розпаралелювання та тестування як хмарної служби. Це означає, що використання цих методів буде здійснюватися у хмарному середовищі, яке надає необхідні ресурси, інструменти та сервіси для тестування програмного забезпечення. Це має багато переваг, таких як:

- масштабованість: хмарне середовище дозволяє динамічно збільшувати або зменшувати кількість ресурсів, які використовуються для тестування, залежно від потреб та навантаження;
- еластичність: хмарне середовище дозволяє гнучко налаштовувати параметри тестування, такі як типи, рівні, частота, тривалість тощо;
- доступність: хмарне середовище дозволяє отримувати доступ до тестування з будь-якого місця та пристрою, що має підключення до інтернету;
- економічність: хмарне середовище дозволяє знизити витрати на тестування, оскільки не потрібно купувати, налаштовувати, обслуговувати та оновлювати власне обладнання та програмне забезпечення.

Однак, для забезпечення конфіденційності та безпеки розроблених методів, слід дотримуватися наступних заходів:

- застосувати шифрування даних, які передаються між хмарним середовищем та локальним комп’ютером. Шифрування дозволяє запобігти перехопленню або зміні даних третіми сторонами;
- використовувати аутентифікацію та авторизацію користувачів, які мають доступ до хмарної служби, щоб переконатися, що вони мають належні права та ролі;
- застосовувати механізми аудиту та реєстрації подій, які дозволяють відстежувати дії користувачів та виявляти будь-які аномалії або порушення безпеки;
- обирати надійного та довіреного постачальника хмарних сервісів, який гарантує високий рівень захисту даних та ресурсів, а також дотримується відповідних стандартів та нормативів.

Альтернативна стратегія для прискорення символьного виконання включає розпаралелювання дослідження шляху в кластері машин, використовуючи їх комбіновані можливості ЦП і пам’яті. Це можна реалізувати шляхом статичного розподілу завдань символьного виконання між вузлами та надання їм незалежної роботи. Однак під час роботи з великими програмами цей метод часто призводить до значного дисбалансу робочого навантаження між вузлами, через що весь кластер розвивається зі швидкістю найповільнішого вузла. Якщо вузол постає перед труднощами, наприклад, застрягає під час символьного виконання циклу, це може перешкоджати загальному процесу тестування. У деяких працях представлено метод розпаралелювання символьного виконання на кластерах, які не мають спільного доступу, що забезпечує масштабованість [41]. Хоча паралельне символьне виконання не може змінити експоненціальний характер проблеми, воно ефективно використовує ресурси кластера, роблячи автоматизоване тестування можливим для більших систем, ніж було б практично.

Еволюція символьного виконання природно узгоджується з концепцією «тестування як послуги» (TaaS). Це бачення передбачає надання тестування програмного забезпечення як конкурентоспроможної та легкодоступної онлайн-послуги, а також ідею повністю автоматизованого тестування в хмарі. Мета полягає в тому, щоб використовувати величезні та еластичні хмарні ресурси, щоб автоматизоване тестування стало практичною реальністю для реального програмного забезпечення. Служба тестування програмного забезпечення дозволяє користувачам і розробникам завантажувати своє програмне забезпечення, вказувати необхідний тип тестування, ініціювати процес і отримувати повний звіт із результатами. У професійних умовах TaaS може легко інтегруватися в процес розробки, проводячи постійне тестування під час написання коду. Крім того, TaaS може служити загальнодоступним сервісом сертифікації, що полегшує порівняння надійності та безпеки різних програмних продуктів.

Висновки до розділу 3

Динамічне символьне виконання, конколічне виконання, розгалуження стану, зведення та злиття станів, розпаралелювання та концепція "тестування як послуги" є ключовими підходами у забезпеченні безпеки програм. Динамічне символьне виконання виявляє програмні шляхи, забезпечуючи компроміси у використанні ресурсів. Вибір стратегії – конкретизації чи абстрагування – залежить від умов використання та цілей аналізу. Вивчення взаємодії з оточенням підкреслює важливість адаптації програмного забезпечення до змінних умов та зовнішніх факторів для підвищення його безпеки. озгляд вибору шляху вказує на важливість стратегічного планування та врахування специфіки проєкту для забезпечення його безпеки. Аналіз ефективного вирішення обмежень вказує на значущість розробки програм, які ефективно працюють в умовах обмежень, забезпечуючи стійкість та безпеку. Розгляд розпаралелювання та тестування як хмарної служби вказує на перспективи

автоматизації тестування та використання розподілених ресурсів для забезпечення ефективності та доступності тестових сервісів. Цей розділ покликаний зробити внесок у покращення стратегій та підходів до забезпечення безпеки програмного забезпечення в умовах сучасного технологічного середовища. Результати дослідження використані для оптимізації практики розробки програм та підвищення їхньої безпеки в умовах постійних змін технологічного ландшафту.

РОЗДІЛ 4. ПРОГРАМНА РЕАЛІЗАЦІЯ ДЛЯ ВИРІШЕННЯ ПРОБЛЕМИ БЕЗПЕКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1. Фаззінг Whitebox із SAGE

Фаззінг Whitebox є методом тестування програмного забезпечення, який використовує статичний аналіз коду для генерації вхідних даних, які можуть викликати помилки або вразливості. SAGE є одним із найвідоміших прикладів фаззінгу Whitebox, який розроблений компанією Microsoft. SAGE використовує символьне виконання для побудови обмежень на вхідні дані, які досягають різних шляхів виконання коду. Потім SAGE застосовує модератор обмежень для генерації нових вхідних даних, які покривають більше шляхів.

Порівняння фаззінгу Whitebox із SAGE може бути зроблене за декількома критеріями, такими як ефективність, швидкість, масштабованість та універсальність. За ефективністю, SAGE перевершує багато інших методів фаззінгу Whitebox, оскільки він може знаходити складні помилки та вразливості, які потребують глибокого аналізу коду. За швидкістю, SAGE має деякі недоліки, оскільки символьне виконання та розв'язання обмежень є ресурсозатратними операціями. За масштабованістю, SAGE може бути застосований до великих та складних програм, але потребує значних ресурсів обчислення та пам'яті. За універсальністю, SAGE може працювати з різними мовами програмування та платформами, але потребує наявності джерельного коду програми.

Фаззінг аналізаторів файлів виділяється як потужний інструмент для автоматичного створення тестів з використанням динамічного символьного виконання та вирішення обмежень. Загроза безпеки, яка виникає внаслідок програмних помилок у коді, призначеному для аналізу файлів і пакетів, що передаються через Інтернет, робить цей підхід надзвичайно важливим. Основні операційні системи, такі як Microsoft Windows, включають аналізатори для

багатьох форматів файлів, що робить будь-яку вразливість у цих аналізаторах потенційною загрозою для великої кількості користувачів. Для усунення таких вразливостей можуть знадобитися дорогі та видимі заходи безпеки, що потребують значних фінансових інвестицій. З урахуванням значного впливу цієї проблеми, корпорація Microsoft виділяє значні ресурси для проактивного виявлення вразливостей безпеки у своїх продуктах, створюючи ідеальне середовище для розвитку фаззінгу білого ящика на безпрецедентний рівень.

Оскільки фаззінг білого ящика призначений для обробки великих додатків, йому необхідно ефективно опрацьовувати розширені виконання додатків, де символьне виконання може бути високо витратним з точки зору обчислень. Наприклад, одне символьне виконання Microsoft Excel із 45 000 вхідними байтами містить майже мільярд інструкцій x86. У відповідь на це фаззінг білого ящика використовує інноваційний алгоритм спрямованого пошуку, відомий як пошук поколінь. Цей алгоритм максимізує генерацію нових вхідних тестів від кожного символьного виконання. Систематично скасовуючи всі обмеження на заданому шляху по одному, він поєднує їх з обмеженнями шляху, що веде до нього, і намагається вирішити отримані обмеження за допомогою засобу вирішення обмежень. Цей підхід дозволяє одному символьному виконанню генерувати тисячі нових тестів. Навпаки, стандартний пошук у глибину або в ширину зазвичай вирішує лише останнє або перше обмеження в кожному шляху, що призводить до генерації щонайбільше одного нового тесту на символьне виконання.

Фаззінг білого ящика розширює можливості створення динамічних тестів за межами модульного тестування, зокрема на тестування безпеки всієї програми, використовуючи три ключові стратегії. По-перше, відзначається використанням фаззінгу чорного ящика, який ініціює генерацію динамічного тесту з одного або кількох добре сформованих входів. Ця евристика сприяє прискоренню розширення покриття коду та надає вигідну відправну точку для подальшого пошуку. По-друге, схоже на фаззінг чорного ящика, фаззінг білого ящика зосереджено на виявленні вразливостей безпеки, таких як переповнення

буфера, а не на перевірці функціональної коректності. Ідентифікація таких вразливостей безпеки може бути повністю автоматизована і не вимагає тестового оракула або функціональної специфікації для конкретної програми. По-третє, і особливо помітно, основна технічна інновація фаззінгу білого ящика полягає в його масштабованості. Він розширює обсяг генерації динамічних тестів від невеликих одиниць коду до цілих програм, роблячи його застосовним до аналізаторів великих файлів, вбудованих у програми з мільйонами рядків коду та трас виконання, що охоплюють сотні мільйонів машинних інструкцій.

SAGE включає в себе критичні оптимізації для обробки обширних трас виконання, які містять мільярди машинних інструкцій. Щоб ефективно керувати такими об'ємними трасами, SAGE використовує різні методи, спрямовані на підвищення швидкості та оптимізацію використання пам'яті для створення обмежень. Хешування символьних виразів гарантує, що структурно еквівалентні символьні терміни відображаються на той самий фізичний об'єкт. Вилучення непов'язаних обмежень зменшує розмір запитів модератора обмежень шляхом видалення обмежень, які не мають спільних символьних змінних із запереченим обмеженням. Хешування локальних обмежень пропускає обмеження, якщо воно вже додано до обмеження шляху. Обмеження кількості перевертань встановлює максимальну кількість разів, коли обмеження, створені з певної гілки програми, можуть бути перевернуті. Підсумування обмежень, використовуючи економну синтаксичну перевірку, усуває обмеження, логічно випливаючи з інших, введених у тій самій гілці програми, часто через послідовні ітерації залежного від вхідних даних циклу.

Первинна реалізація фаззінгу білого ящика відбулася в інструменті SAGE, що означає Scalable Automated Guided Execution. SAGE відзначився як перший інструмент для динамічного символьного виконання на рівні двійкового коду x86. Ця особливість дозволяє застосовувати його до будь-якої програми, незалежно від вихідної мови чи процесу компіляції. Крім того, вона гарантує, що "те, що ви передаєте, те й відправляєте", розв'язуючи проблеми,

пов'язані зі змінами вихідного коду, які можуть вносити компілятори та потенційно впливати на безпеку.

Під час фаззінгу SAGE виявив безліч нових вразливостей безпеки (наприклад, переповнення буфера) у багатьох синтаксичних аналізаторах Windows і програмах Office, включаючи процесори зображень, медіаплеєри, декодери файлів і аналізатори документів. Важливо зауважити, що SAGE виявив приблизно третину всіх помилок, виявлених під час розробки Microsoft Windows, що призвело до економії мільйонів доларів, уникаючи дорогих виправлень безпеки для майже мільярда комп'ютерів по всьому світу. Оскільки SAGE, як правило, запускався останнім, ці помилки уникнули всіх інших видів аналізу програми, включаючи статичний аналіз і фаззінг чорного ящика.

З початку 2008 року SAGE активно використовується в промисловості на сотнях машин, здійснюючи автоматичний фаззінг сотень додатків у лабораторіях безпеки Microsoft. Широке застосування охоплює понад 500 машино-років, роблячи його, за словами розробників модератора Z3 SMT, «найбільшим обчислювальним використанням будь-якого модератора Satisfiability-Modulo-Theories (SMT)». На сьогоднішній день було оброблено понад чотири мільярди обмежень [30].

У 2015 році SAGE та інші відомі фаззери blackbox, які широко використовуються всередині корпорації Майкрософт, були інтегровані в проєкт Springfield, першу комерційну хмарну службу фаззінгу (у 2017 році перейменовану на Microsoft Security Risk Detection). Клієнти, які замовляли цю послугу, можуть надсилати завдання фаззінгу, спрямовані на їхнє власне програмне забезпечення, і отримувати переваги від технології, яка описана в даній роботі. Для цього не потрібен вихідний код або символи. Завдання фаззінгу у Springfield легко налаштовувати навіть не відгалуженим користувачам, і вони обробляються за допомогою тих самих інструментів, які використовуються в корпорації Майкрософт протягом більше десяти років. Ці інструменти забезпечують автоматизовану перевірку завдань, мінімізацію вихідних даних, фаззінг на різних машинах (використовуючи хмарні ресурси),

кожна з яких працює з різними інструментами фаззінгу та конфігураціями. Після цього відбувається автоматичний аналіз, сортування та визначення пріоритетів знайдених помилок, результати яких надаються безпосередньо на вебсайт Springfield.

SAGE - це система автоматизованого генерування есе, яка використовує штучний інтелект для створення високоякісних текстів на будь-яку тему. SAGE використовує різні методи та алгоритми, такі як нейронні мережі, природні мови, логіка, статистика тощо, щоб аналізувати, структурувати, синтезувати та оцінювати текст. SAGE може генерувати есе на різні теми, такі як історія, література, філософія, наука, мистецтво, політика, економіка, право, психологія, освіта тощо. SAGE може також адаптуватися до різних вимог та критеріїв, таких як довжина, стиль, рівень складності, мета, аудиторія тощо. SAGE може допомогти покращити навички письма, розвитку критичного мислення, аргументації, дослідження тощо.

SAGE має можливості OWASP, тобто вона дотримується стандартів безпеки вебдодатків, що розроблені Організацією з аналізу безпеки вебдодатків. OWASP - це міжнародна неприбуткова організація, яка присвячується підвищенню безпеки вебдодатків. OWASP публікує список найбільш критичних вразливостей вебдодатків, які можуть бути використані зловмисниками для компрометації даних або системи. SAGE захищає свої дані від різних загроз, таких як вразливості SQL-ін’єкцій, міжсайтового скриптування, підробки запитів до сервера та інших. SAGE використовує різні технології та протоколи, такі як HTTPS, SSL, TLS, RSA, AES, SHA тощо, щоб шифрувати, перевіряти та захищати свої дані від несанкціонованого доступу, зміни або втрати. SAGE також дбає про конфіденційність та цілісність своїх користувачів, використовуючи сильне шифрування, аутентифікацію та авторизацію.

4.2. Вибіркове символьне виконання з S2E

Вибіркове символьне виконання з S2E - це техніка аналізу програм, яка дозволяє виконувати програму з частково символьними даними. Це означає, що деякі вхідні дані програми можуть бути представлені як символьні змінні, а не як конкретні значення. Наприклад, якщо програма приймає файл як вхідний параметр, то вибірковий символьний виконавець може використовувати символ f для представлення будь-якого можливого файлу, замість використання певного файлу, такого як test.txt. Таким чином, програма може розгалужуватися на різні шляхи виконання залежно від умов, яким задовольняють символьні змінні. Наприклад, якщо програма містить умовний оператор if (f.size > 100), то вибірковий символьний виконавець може виконати обидві гілки оператора, використовуючи символ f з різними обмеженнями на його розмір.

S2E - це платформа для вибіркового символьного виконання, яка інтегрується з емулятором QEMU та бібліотекою KLEE. QEMU - це відкритий емулятор або віртуальна машина, який дозволяє виконувати програми, написані для різноманітних архітектур процесорів, на одній архітектурі. KLEE - це відкрита бібліотека для символьного виконання, яка дозволяє аналізувати програми, написані на мові C або C++, використовуючи символьні змінні та розв’язування обмежень. S2E забезпечує взаємодію між QEMU та KLEE, дозволяючи виконувати програми з частково символьними даними на рівні емуляції. S2E також дозволяє користувачам налаштовувати обсяг символьного виконання за допомогою плагінів та конфігураційних файлів. Плагіни — це модулі, які розширюють функціональність S2E, додаючи нові можливості, такі як моніторинг, візуалізація, інструментування або моделювання апаратного забезпечення. Конфігураційні файли — це файли, які визначають параметри виконання S2E, такі як вхідні дані, вихідні дані, пам’ять, реєстри, стек, куча, шляхи виконання, умови безпеки та ін.

Вибіркове символьне виконання з S2E може застосовуватися для різних цілей, таких як пошук помилок, тестування безпеки, реверс-інжиніринг та оптимізація програм. Пошук помилок — це процес виявлення та виправлення дефектів або невідповідностей в коді програми.

Спочатку призначений для тестування повних системних стеків, які характеризуються складними середовищами, включаючи власні ядра операційної системи та численні взаємозалежні бібліотеки, S2E спрямований на забезпечення детальної оцінки поведінки програм. Визнавши складність попереднього визначення відповідних деталей навколишнього середовища, S2E використовує підхід символьного виконання "in vivo". Ця методологія передбачає автоматичне абстрагування середовища на льоту за потреби, на відміну від символьного виконання "in vitro", яке використовується іншими механізмами символьного виконання та перевірками моделей, які покладаються на заглушки або попередньо визначені моделі.

Удосконалення та розширення за межі окремих програм, особливо коли націлено на системний код, такий як драйвери пристроїв або цілі системні стеки (включаючи ядро операційної системи, драйвери, бібліотеки та програми), породжують важливі роздуми щодо взаємодії між тестовим програмним забезпеченням і його середовищем. У цьому контексті масштабування за межі однієї програми призводить до додаткових проблем. Для вирішення цих складнощів була розроблена система S2E. S2E виконує роль механізму символьного виконання і розроблений для двійкових файлів x86 і ARM. Його архітектура зосереджена навколо модифікованої версії віртуальної машини QEMU [6]. Система динамічно направляє інструкції гостьової машини або до центрального процесора для виконання нативно, або до вбудованого механізму символьного виконання (KLEE) у S2E. Більшість інструкцій, які не мають символьної участі, можуть виконуватися нативно, тоді як інші інструкції інтерпретуються символьно.

Дослідники мережі використовували S2E для аналізу площини даних програмних мережевих маршрутизаторів [35]. Це передбачало виконання вичерпного символьного виконання на конкретних компонентах маршрутизатора без необхідності моделювання всього маршрутизатора. Гнучкість S2E в плавному чергуванні між конкретним і символьним виконанням в межах одного прогону, зберігаючи уніфіковане глобальне

сховище станів, дозволяє досліджувати «цікаві» програмні шляхи в цільовому програмному забезпеченні, такому як драйвер. Це динамічне чергування між конкретним і символьним виконанням відбувається кілька разів під час виконання, що дозволяє детально вивчати цікаве програмне забезпечення без необхідності символьного виконання всього реального середовища.

Цей підхід до символьного виконання in vivo знайшов застосування як в промисловому, так і в академічному середовищах. Інженери компанії Intel успішно використовували S2E для виявлення вразливостей безпеки в реалізаціях UEFI BIOS [28]. DDT використовував S2E для тестування власних драйверів пристроїв Microsoft Windows із закритим кодом, навіть при відсутності доступу до вихідного коду драйвера або повного розуміння ядра Windows. Зазначимо, що DDT виявив витоки пам'яті, помилки сегментації, умови змагання та помилки пошкодження пам'яті в драйверах, які були частиною дистрибутивів Windows протягом тривалого періоду.

Уніфіковане сховище стану в S2E охоплює такі елементи стану машини, як пам’ять, регістри та прапорці процесора, системний годинник і пристрої, які спільно використовуються між механізмом символьного виконання та віртуальною машиною. S2E керує плавним перетворенням між конкретним і символьними станами, процесом, керованим моделлю узгодженості виконання. Наприклад, у моделі «локально узгодженого виконання», коли драйвер робить символьний аргумент λ у виклику kmalloc для виділення пам’яті ядра, S2E динамічно вибирає конкретне значення, яке задовольняє обмеження шляху, наприклад 64, і конкретно виконує kmalloc всередині ядра. Після того, як ядро повертає конкретний 64-байтовий буфер пам'яті, S2E надає драйверу символьний двозначний покажчик, позначений як p = 0 | &buffer, що представляє два можливі результати виклику kmalloc. Згодом S2E посилює обмеження шляху з λ = 64 і продовжує символьне виконання в драйвері. Залежно від моделі узгодженості, якщо λ пізніше буде задіяно в стані розгалуження, де одне з розгалужень стає неможливим через обмеження λ = 64, S2E може повторно переглянути оригінальний сайт виклику kmalloc, щоб

повторно конкретизувати λ до додаткового значення, яке дає змогу цьому розгалуженню та повторення виклику.

Моделі узгодженості виконання в S2E проводять аналогію з моделями узгодженості пам’яті. Загальноприйнятим припущенням у виконанні системи є те, що стан є узгодженим у будь-який момент часу, що означає існування можливого конкретного шляху виконання від початкового стану системи до її поточного стану. Це припущення відповідає тому, що S2E називає «суворо послідовним конкретним виконанням», що представляє найсильніше з усіх моделей узгодженості виконання. Послаблюючи це припущення, визначено п’ять додаткових моделей узгодженості, кожна з яких пропонує різні компроміси. Наприклад, «строго узгоджене виконання на рівні модуля» узгоджується з моделлю узгодженості, яка керує тим, як DART і EXE обробляють середовище, тоді як «локально узгоджене виконання», згадане раніше, є моделлю, яку використовує DDT для взаємодії між драйверами пристроїв і ядром ОС [39].

S2E включає оптимізацію, відому як "лінива конкретизація", де символьні значення конкретизуються на вимогу, саме тоді, коли конкретно запущений код збирається розгалужуватися за умовою, що залежить від цього значення. Цей підхід забезпечує безперебійне проходження значної кількості даних через рівні системного стека без негайного перетворення. Наприклад, коли програма записує буфер символьних даних у файлову систему, а в ядрі або драйвері дискового пристрою зазвичай немає гілок, залежних від самих даних, S2E може передавати буфер без конкретизації. Його можна записати на віртуальний диск у символьній формі, що дозволяє згодом прочитати його у символьній формі. Ця стратегія допомагає уникнути втрати точності, пов’язаної з передчасною конкретизацією.

Заслуговує на увагу концепція S2E — це символьне обладнання, що відповідає «приблизно узгодженому виконанню», що дозволяє віртуалізованому обладнанню надавати необмежені символьні значення. Цю модель узгодженості виконання використовували DDT і SymDrive для

апаратного інтерфейсу, дозволяючи тестувати драйвери на шляхи апаратних помилок, які складно використовувати, і компенсуючи повну відсутність фактичного апаратного забезпечення.

З часом S2E перетворилася на універсальну платформу для аналізу програмного забезпечення, продемонструвавши численні несподівані можливості. Наприклад, RevNIC використовував "приблизно узгоджене виконання" для автоматичного зворотного проєктування пропрієтарних драйверів пристроїв Windows і створення еквівалентних драйверів для різних платформ. У S2E відіграв вирішальну роль у розробці комплексного профайлера продуктивності, здатного вимірювати різні метрики, включаючи кількість інструкцій, промахи кешу, промахи TLB і помилки сторінки на всіх шляхах програми для довільних ієрархій пам’яті. Завдяки своєму дизайну як віртуальної машини, S2E не обмежується пропрієтарним програмним забезпеченням; він добре підходить для аналізу само модифікованих, JIT-файлів і/або обфускованих і упакованих/зашифрованих двійкових файлів. Ця характеристика зробила S2E особливо корисним для аналізу шкідливих програм у комерційних умовах.

Наразі S2E є проєктом з відкритим вихідним кодом і служить основною технологією для кількох комерційних продуктів кібербезпеки. Його подорож ілюструє, як автоматизована генерація тестів може трансформуватися в різні інші форми аналізу програм.

Крім того, S2E використовувався для створення Chef, інструменту, призначеного для перетворення ванільного інтерпретатора динамічно інтерпретованої мови (наприклад, Python) у надійний і повний механізм символьного виконання для цієї мови. В іншому важливому випадку дві із семи систем, які змагалися у фіналі Cyber Grand Challenge DARPA у 2016 році, базувалися на S2E. Це змагання являло собою турнір з комп’ютерної безпеки для всіх машин, де кожна машина-конкурент мала незалежно аналізувати комп’ютерні програми, визначати вразливі місця в безпеці, виправляти їх і запускати атаки на інших конкурентів. Як частина Galactica (один із

конкурентів DARPA), S2E здійснив 392 успішні атаки під час змагань, що вдвічі більше, ніж абсолютний переможець змагань.

Вибіркове символьне виконання (SSE) - це техніка аналізу програм, яка дозволяє виконувати частини програми з символьними вхідними даними, замість конкретних значень. Це означає, що програма може мати різну поведінку залежно від того, які умови виконуються або не виконуються для символьних змінних. Наприклад, якщо програма має рядок коду if (x > 10) {... } else {... }, де x є символьною змінною, то SSE може виконати обидва блоки коду, один за одним, і зберегти результати для кожного з них. Таким чином, SSE може дослідити різні сценарії, які можуть виникнути при різних значеннях x.

SSE може бути корисно для виявлення помилок, перевірки властивостей безпеки, генерації тестових випадків тощо. Помилки — це стани програми, які призводять до неправильних або небажаних результатів. Властивості безпеки — це умови, які гарантують, що програма не порушує конфіденційність, цілісність або доступність даних або системи. Тестові випадки — це набори вхідних даних, які викликають певну поведінку програми. SSE може допомогти знайти помилки, перевіряючи, чи не порушуються властивості безпеки, або генеруючи тестові випадки, які покривають різні шляхи виконання програми.

SSE може також використовуватися для розробки інструментів, які допомагають захистити програми від атак, пов’язаних з OWASP (Open Web Application Security Project). OWASP - це міжнародна неприбуткова організація, яка присвячується підвищенню безпеки вебдодатків. OWASP публікує список найбільш критичних вразливостей вебдодатків, які можуть бути використані зловмисниками для компрометації даних або системи. Деякі з найпоширеніших вразливостей включають SQL-ін’єкції, переповнення буфера, міжсайтовий скриптинг, виконання довільного коду, витоки інформації тощо. SSE може допомогти знайти потенційні вразливості, аналізуючи код програми з символьними даними, які можуть містити зловмисні вводи. SSE може також допомогти створити контрзаходи, які запобігають або зменшують наслідки

атак. Контрзаходи — це дії, які покращують безпеку програми, такі як санітарна обробка вхідних даних, перевірка цифрових підписів, застосування політик безпеки, шифрування даних тощо.

SSE є складною технікою, яка вимагає високого рівня знань та навичок в галузі програмування та безпеки. Одним з інструментів, який спрощує використання SSE, є S2E. S2E - це платформа для вибіркового символьного виконання, яка інтегрується з емулятором QEMU та бібліотекою KLEE. QEMU - це вільний і відкритий емулятор, який дозволяє виконувати програми для різноманітних архітектур процесорів на одній машині. KLEE - це вільна і відкрита бібліотека, яка дозволяє виконувати символьне виконання на програмах, написаних на мові C або C++. S2E дозволяє користувачам налаштовувати обсяг символьного виконання за допомогою плагінів та конфігураційних файлів. Плагіни — це модулі, які розширюють функціональність S2E, додаючи нові можливості, такі як виявлення помилок, аналіз коду, генерація тестів тощо. Конфігураційні файли — це файли, які визначають параметри виконання S2E, такі як вхідні дані, властивості безпеки, критерії зупинки тощо. Вибіркове символьне виконання з S2E може застосовуватися для різних цілей, таких як пошук помилок, тестування безпеки, реверс-інжиніринг та оптимізація програм.

4.3. Інші підходи до автоматизованої генерації тестів

Тестування на основі моделі: Цей підхід передбачає створення тестів за допомогою аналізу абстрактного представлення програми, відомого як модель. Головною метою є оцінка відповідності програми моделі. Ці моделі можуть бути специфікаціями програм, створеними вручну або автоматично згенерованими за допомогою методів машинного навчання [40]. На відміну від методів генерації тестів, що користуються керованим кодом, які розглядаються в даній роботі й не вимагають моделі програми, тестування на основі моделі

спрямоване на створення тестів, які вичерпно виконують різні оператори програми, включаючи твердження, вбудовані в код.

Як вказано в уводі, важливо відзначити, що ця робота не має на меті представити комплексне опитування щодо автоматичного створення тестів. Однак корисно коротко зазначити деякі важливі методи генерації тестів.

Фаззінг на основі граматики: Багато відомих випадкових фаззерів, розроблених для тестування безпеки, включають форму представлення граматики для визначення формату введення оцінюваної програми, таких як Peach і SPIKE. Зазвичай ці граматики створюються вручну, що є складним, трудомістким і піддається помилкам. Всупереч цьому, граматичний фаззінг став одним із найефективніших методів фаззінгу, особливо для додатків, які мають справу зі складними структурованими форматами введення, такими як веббраузер, які обробляють вебсторінка, містять складні документи HTML і код JavaScript. Генерація тестів з граматики може бути випадковою або вичерпною. Імперативна генерація, включає створення програми на замовлення, яка ефективно кодує граматику. Фаззінг на основі граматики легко інтегрується з Whitebox фаззінгом.

Генерація експлойта: Більш цілеспрямованим аспектом тестування безпеки є створення експлойта. У цьому сценарії метою є автоматичне виявлення вразливостей у програмі та створення відповідних експлойта. Системи, такі як Mayhem, використовують попередньо обумовлене символьне виконання для виявлення та використання помилок безпеки нульового дня [32; 27, с. 216–233]. Попередні роботи в цій області передбачали розширення генерації тестів за допомогою інформації, отриманої з патчів безпеки, для зворотного проєктування експлойта, які були об'єктом цих патчів [38].

Генерація тестів на основі пошуку: Процес генерації тесту можна уявити як проблему пошуку та оптимізації, спрямовану на досягнення максимальних цілей, таких як охоплення коду. У цьому контексті було визначено численні евристики та методи пошуку, як-то генетичні алгоритми та моделювання відпалу. Раніше згадана евристика фаззінгу, яка використовує зворотний зв'язок

щодо покриття коду, відповідає цим підходам. Крім генерації тестів, ці методи знайшли застосування в різних задачах інженерії програмного забезпечення, включаючи тестування, такі як мінімізація тестових випадків і пріоритезація тестових випадків.

Тестування паралелізму: Для паралельного програмного забезпечення були розроблені методи та алгоритми систематичного тестування [37]. Ці методи досліджують можливе чергування кількох процесів або потоків за допомогою планувальника виконання. Основна мета полягає в виявленні проблем, пов'язаних з паралелізмом, таких як взаємоблокування та умови змагання.

Комбінаторне тестування: Комбінаторне тестування спрямоване на ефективне створення набору тестових вхідних даних для програми та заданого набору вхідних параметрів з метою охоплення всіх пар вхідних параметрів. Також були запропоновані розширення від пар до довільних k-кортежів. У практичних застосуваннях ці методи проявляють себе ефективно, коли кількість вхідних параметрів є відносно невеликою, як у випадку параметрів конфігурації.

Перевірка виконання: Інструменти верифікації під час виконання активно відстежують поведінку програми під час виконання та порівнюють її зі специфікацією високого рівня, яка, як правило, подається у вигляді автомата кінцевого стану або формули часової логіки. Ці інструменти можна розглядати як розширення попередніх інструментів перевірки виконання, таких як Purify і AddressSanitizer, що забезпечують додатковий підхід до створення тестів.

Перевірка програми: Удосконалення в символьному виконанні, генерації обмежень і автоматизованому доведенні теорем протягом останніх десятиліть сприяли розвитку технік верифікації програм, включаючи генерацію умов верифікації, перевірку символьної моделі і перевірку обмеженої моделі [34]. Перевірка програми зосереджена на демонстрації відсутності програмних помилок, тоді як генерація тестів зосереджена на створенні конкретних тестових вхідних даних, здатних направити програму на виконання певних

операторів або шляхів. Попри прогрес, символьне виконання, генерація обмежень і розв'язання, як правило, не є абсолютно надійними та завершеними, що робить повну автоматичну перевірку програми складною для великих, складних програмних забезпечень в практичних сценаріях.

4.4. Модуль, який використовує функціонал OWASP ZAP для сканування програмного забезпечення

Для розробки модуля, який використовує функціонал OWASP ZAP для сканування програмного забезпечення на наявність різних вразливостей, використовується Python разом із ZAP API. Ось загальний план і приклад коду для такого модуля:

Встановлення необхідних бібліотек: Спершу необхідно впевнитися, що встановлені необхідні бібліотеки для роботи з ZAP API:

1. pip install python-owasp-zap-v2.4

Створення З'єднання з OWASP ZAP: Підключення до OWASP ZAP через його API, використовуючи Python:

1. from zapv2 import ZAPv2
2. target_url = "http://your-target-url.com"
3. zap = ZAPv2(apikey="your-api-key", proxies={"http": "http://127.0.0.1:8090", "https": "http://127.0.0.1:8090"})

Видаліть попередній скан (якщо потрібно)

4. zap.core.delete_all_sessions()

Конфігурація Завдання Сканування: Налаштування параметрів сканування, такі як типи вразливостей, які ви хочете перевірити:

1. scan_id = zap.spider.scan(target_url, apikey="your-api-key")

Очікування завершення сканування: Дочекатися завершення сканування перед перевіркою результатів:

1. while int(zap.spider.status(scan_id)) < 100:
2. print(f"Scan progress: {zap.spider.status(scan_id)}%")
3. time.sleep(5)

Аналіз Результатів: Необхідно переглянути та обробити результати сканування:

```
alerts = zap.core.alerts(baseurl=target_url)
for alert in alerts:
print(f"Alert: {alert['name']}, Risk: {alert['risk']}")
```

Приклад Загального Модуля: Ось простий приклад модуля для сканування вразливостей за допомогою OWASP ZAP API:

```
from zapv2 import ZAPv2
import time
def scan_with_zap(target_url):
zap = ZAPv2(apikey="your-api-key", proxies={"http": "http://127.0.0.1:8090", "https": "http://127.0.0.1:8090"})
# Видаліть попередній скан (якщо потрібно)
zap.core.delete_all_sessions()
# Запустіть павук для знаходження посилань
zap.spider.scan(target_url, apikey="your-api-key")
# Очікуємо завершення сканування
while int(zap.spider.status()) < 100:
print(f"Scan progress: {zap.spider.status()}%")
time.sleep(5)
# Запускаємо сканування вразливостей
zap.ascan.scan(target_url, apikey="your-api-key")
# Очікуємо завершення сканування вразливостей
while int(zap.ascan.status()) < 100:
print(f"Scan progress: {zap.ascan.status()}%")
time.sleep(5)
# Отримуємо та виводимо результати
alerts = zap.core.alerts(baseurl=target_url)
for alert in alerts:
print(f"Alert: {alert['name']}, Risk: {alert['risk']}")
```

15. target_url = "http://your-target-url.com"
16. scan_with_zap(target_url)

4.5. Модуль, який використовує функціонал BURP для сканування програмного забезпечення

Розробка модуля для використання функціоналу Burp Suite для сканування програмного забезпечення на наявність вразливостей включає наступні кроки. Зверніть увагу, що Burp Suite має свій API, який може варіюватися в залежності від версії, тому деякі функції можуть вимагати адаптації згідно із змінами в API. В цьому прикладі я використовую бібліотеку requests для HTTP-запитів.

Встановлення необхідних бібліотек: Спершу необхідно впевнитися, що встановлені бібліотеки, необхідні для роботи з Burp Suite API та HTTP-запитами:

1. pip install requests

Запуск Burp Suite та Налаштування API: Запуск Burp Suite, розділ "User options" > "Misc" > "REST API". Увімкнення опції "Enable REST API" та налаштуйте необхідні параметри (наприклад, порт).

Розробка Модуля для Сканування: Розробка Python-модуля для взаємодії з Burp Suite API та сканування вразливостей:

1. import requests
2. import time
3. class BurpScanner:
3.1. def __init__(self, base_url, api_port):
3.1.1. self.base_url = base_url
3.1.2. self.api_port = api_port
3.1.3. self.base_api_url = f"http://127.0.0.1:{api_port}/v0.1/"
3.1.4. self.scan_id = None
3.2. def start_scan(self):
3.2.1. start_scan_url = self.base_api_url + "scans"

```
3.2.2. scan_payload = {"urls": [self.base_url]}
3.2.3. response = requests.post(start_scan_url, json=scan_payload)
3.2.4. if response.status_code == 201:
3.2.4.1.     self.scan_id = response.json()["scan"]["id"]
3.2.4.2.     print(f"Scan started. Scan ID: {self.scan_id}")
3.2.4.3.     return True
3.2.5. else:
3.2.5.1.     print(f"Failed to start scan. Status code: {response.status_code}")
3.2.5.2.     return False
3.3.  def check_scan_status(self):
3.3.1. scan_status_url = self.base_api_url + f"scans/{self.scan_id}/status"
3.3.2. response = requests.get(scan_status_url)
3.3.3. return response.json()["status"]
3.4.  def get_scan_results(self):
3.4.1. scan_results_url = self.base_api_url + f"scans/{self.scan_id}/issues"
3.4.2. response = requests.get(scan_results_url)
3.4.3. return response.json()
4.    def scan_with_burp(base_url, api_port):
4.1.  burp_scanner = BurpScanner(base_url, api_port)
# Запуск сканування
4.2.  if burp_scanner.start_scan():
# Очікуємо завершення сканування
4.2.1. while burp_scanner.check_scan_status() != "finished":
4.2.1.1.     print("Scan in progress...")
4.2.1.2.     time.sleep(10)
# Отримуємо та виводимо результати
4.2.2. scan_results = burp_scanner.get_scan_results()
4.2.3. for issue in scan_results:
4.2.3.1.     print(f"Issue: {issue['name']}, Severity: {issue['severity']}")
5.    if __name__ == "__main__":
```

5.1. target_url = "http://your-target-url.com"

5.2. burp_api_port = 1337

Замініть на відповідний порт Burp Suite API

5.3. scan_with_burp(target_url, burp_api_port)

Необхідно замінити http://your-target-url.com на фактичний URL вашої програми та встановіть відповідний порт API Burp Suite. Цей модуль допоможе розпочати взаємодію з Burp Suite API та виконати основне сканування вразливостей.

4.6. Скрипт, який використовує функціонал Metasploit для тестування веб-додатку

Нижче наведено скрипт Metasploit для тестування вразливостей веб-додатку. Цей приклад використовує модуль для атаки SQL-ін'єкції:

Запускаємо msfconsole

1. msfconsole

Встановлюємо веб-сервер та базу даних PostgreSQL для Metasploit

2. db_status
3. db_connect -y /path/to/metasploit/config/database.yml

Знаходимо інформацію про SQL-ін'єкційну вразливість

4. search sql_injection

Вибираємо модуль для використання (наприклад, один з модулів для SQL-ін'єкції)

5. use auxiliary/scanner/http/sql_injection

Переглядаємо параметри модуля та встановлюємо необхідні значення

6. show options
7. set RHOSTS your_target_host
8. set RPORT 80
9. set TARGETURI /path/to/vulnerable/page

Запускаємо атаку

10. run

4.7. Забезпечення захищеності від "false positives" та "false negatives"

Одним із викликів, з якими стикаються методи символьного виконання, є велика кількість "false positives" та "false negatives" - ситуацій, коли символьний аналізатор неправильно ідентифікує помилки або пропускає їх. Це може призвести до ненадійності та неефективності аналізу. Існує кілька способів забезпечити захист від цих проблем в розрізі S2E, SAGE, Вибіркове символьне виконання.

S2E - це платформа для комбінованого конкретного та символьного виконання програм. Вона дозволяє виконувати програму в віртуальному середовищі та динамічно перемикатися між конкретним та символьним режимами. Це зменшує обсяг символьних даних, які потрібно обробляти, та покращує точність аналізу. S2E також використовує техніки оптимізації, такі як хешування, повторне використання та уникнення дублювання обчислень, для зменшення навантаження на модератор обмежень.

SAGE - це система для генерації тестових випадків на основі символьного виконання. Вона приймає на вхід програму та набір конкретних вхідних даних, які викликають певне покриття коду. Потім вона застосовує символьне виконання до цих даних та генерує новий набір даних, які покривають нові шляхи виконання. SAGE запобігає "false positives" та "false negatives" за допомогою двох механізмів: перевірки сумісності та повторне визначення символів. Перевірка сумісності полягає у тому, що SAGE перевіряє, чи не суперечать новостворені обмеження попередньо наявним. Якщо так, то SAGE викидає такий тестовий випадок. Перевизначення символів полягає у тому, що SAGE замінює символи, яким було присвоєно конкретне значення, новими символами, яким не було присвоєно жодного значення. Це дозволяє SAGE експлотувати більше можливих значень для цих символів.

Вибіркове символьне виконання — це метод символьного виконання, який обмежує область застосування символів до певних частин програми. Наприклад, можна обрати лише функції або модулі, яким потребується символьний аналіз, а решту програми виконувати конкретно. Це знижує складність символьного виконання та зменшує ймовірність "false positives" та "false negatives". Однак, вибіркове символьне виконання потребує додаткової інформації про програму, такої як анотації, специфікації або метадані, щоб визначити, які частини програми слід аналізувати символьно.

4.8. Оцінка вартості впровадження

Автоматизоване тестування програмного забезпечення є важливим інструментом для забезпечення якості, надійності та безпеки програмних продуктів. Одним із сучасних підходів до автоматизованого тестування є символьне виконання, яке дозволяє аналізувати всі можливі шляхи виконання програми та генерувати тестові вхідні дані, які покривають ці шляхи. Символьне виконання має багато переваг, таких як здатність виявляти складні помилки, покращувати покриття коду, зменшувати кількість необхідних тестів тощо. Однак символьне виконання також має свої недоліки, серед яких найбільшою є експоненційне зростання простору станів програми, що призводить до великої складності обчислень та великого обсягу пам'яті.

Для подолання цих недоліків було розроблено ряд методів, які дозволяють оптимізувати процес символьного виконання та зменшити його витрати. Одним із таких методів є вибіркове символьне виконання (selective symbolic execution), яке полягає у тому, що символьне виконання застосовується лише до обраних частин програми, які мають найбільший інтерес для тестування, а решта частин виконуються конкретно. Це дозволяє зосередити увагу на найбільш критичних або найбільш помилкових ділянках коду та уникнути непотрібного аналізу незначущих шляхів.

S2E і SAGE є двома прикладами систем автоматизованого тестування, які реалізують підхід вибіркового символьного виконання. S2E є платформою для аналізу властивостей програмного забезпечення на рівні системи. Вона дозволяє запускати програмне забезпечення у середовищі емуляції QEMU та застосовувати символьне виконання до обраних модулю програми. SAGE є системою генерації тестових даних для Windows-програм на основі байт-коду. Вона дозволяє аналогічно застосовувати символьне виконання до обраних функцій програми.

Оцінка вартості впровадження автоматизованих тестів програмного забезпечення S2E, SAGE, Вибіркове символьне виконання є складним завданням, яке потребує аналізу багатьох факторів, таких як характеристики програмного забезпечення, яке підлягає тестуванню, цілі та критерії тестування, наявність необхідних ресурсів, час та бюджет проєкту тощо. Одним із можливих способів оцінки вартості є порівняльний аналіз ефективності та витрат різних систем автоматизованого тестування на конкретних прикладах програмного забезпечення. Такий аналіз дозволить визначити переваги та недоліки кожної системи, а також оцінити їхню придатність для задоволення потреб проєкту.

4.9. Забезпечення конфіденційності та безпеки

Однією з актуальних проблем, яка виникає під час проведення автоматизованого тестування програмного забезпечення, є забезпечення конфіденційності, цілісності та доступності інформації, яка обробляється та передається між різними компонентами системи. Ця проблема стає особливо гострою, коли мова йде про тестування безпеки програмного забезпечення, яке вимагає застосування спеціальних методик та інструментів, таких як S2E, SAGE, Вибіркове символьне виконання. Ці методи дозволяють ефективно виявляти потенційні помилки та вразливості в програмному коді, але також

потребують доступу до великої кількості даних про структуру та поведінку програми.

Для вирішення цього питання необхідно використовувати різні підходи та технології, які допоможуть захистити програмне забезпечення від несанкціонованого доступу, модифікації, копіювання або видалення. Деякі з таких підходів та технологій описані нижче:

- шифрування — це процес перетворення зрозумілої інформації в незрозумілий формат, який може бути розшифрований тільки за допомогою спеціального ключа. Шифрування дозволяє захистити дані від проникнення або перехоплення, а також забезпечити їх цілісність. Існує багато алгоритмів шифрування, таких як AES, RSA, DES, Blowfish тощо;
- хешування — це процес генерації унікального ідентифікатора (хешу) для будь-якої інформації, який залежить від її вмісту. Хешування дозволяє перевірити, чи не була змінена інформація під час передачі або зберігання, а також використовувати хеші як паролі або цифрові підписи. Існує багато алгоритмів хешування, таких як MD5, SHA-1, SHA-256 тощо;
- цифровий підпис — це процес додавання до інформації спеціального коду, який підтверджує її автентичність та джерело. Цифровий підпис зазвичай створюється за допомогою асиметричного шифрування, коли використовуються два ключі: приватний та публічний. Приватний ключ використовується для генерації підпису, а публічний — для його перевірки. Цифровий підпис дозволяє забезпечити невід’ємність та непередбачуваність інформації;
- обфускація — це процес приховування або зміни логіки програмного коду, щоб ускладнити його аналіз або модифікацію. Обфускація дозволяє захистити програмне забезпечення від реверс-інжинірингу, крадіжки інтелектуальної власності, вставки шкідливого коду тощо. Існує багато методів обфускації, таких як зміна назв змінних, використання мета символів, додавання зайвого коду тощо.

Ці та інші підходи та технології допомагають підвищити рівень безпеки програмного забезпечення та його тестування, але також потребують додаткових зусиль та ресурсів від розробників та тестувальників. Тому важливо враховувати фактори, такі як вартість, продуктивність, сумісність, легкість використання та інші, при виборі та застосуванні відповідних рішень для автоматизованого тестування безпеки програмного забезпечення.

4.10. Легальність та етика використання

Легальність та етика використання із використанням S2E, SAGE, Вибіркове символьне виконання. Забезпечення конфіденційності та безпеки власних методів.

Ці методи дозволяють аналізувати програмний код на наявність помилок, вразливостей та інших проблем, що можуть загрожувати безпеці. Вони базуються на символьному виконанні, яке моделює всі можливі шляхи виконання програми та перевіряє їх на коректність. Це дуже потужний інструмент для забезпечення якості та надійності програмного забезпечення.

Однак, ці методи також мають свої обмеження та ризики. По-перше, вони потребують значних ресурсів для виконання, що може обмежувати їх застосування до деяких типів програм. По-друге, вони можуть бути використані не лише для покращення безпеки, але й для її порушення. Наприклад, зловмисники можуть використовувати ці методи для пошуку слабких місць у програмах своїх жертв або конкурентів.

Тому, легальність та етика використання цих методів залежать від того, хто, як і з якою метою їх застосовує. З одного боку, розробники програмного забезпечення мають право на захист своїх інтелектуальних прав та комерційних інтересів. З іншого боку, користувачі програмного забезпечення мають право на захист своєї конфіденційності та даних. Також, слід враховувати загальний інтерес суспільства до підвищення рівня безпеки та захисту від кіберзлочинності.

Забезпечення конфіденційності та безпеки розроблених методів полягає у дотриманні наступних принципів:

- використовувати ці методи лише для законних цілей, пов’язаних з покращенням якості та безпеки програмного забезпечення;
- не розголошувати деталей реалізації цих методів без належної авторизації або дозволу власників;
- не використовувати ці методи для атак на програмне забезпечення третіх осіб або для отримання доступу до їх даних;
- дотримуватися норм моралі та етики при роботі з цими методами та поважати права та інтереси інших сторін.

Окрім цього, можна додати наступні пункти:

- слідкувати за оновленнями цих методів, щоб використовувати найсучасніші та найбезпечніші версії;
- застосовувати ці методи у поєднанні з іншими методами тестування безпеки, такими як статичний аналіз, фаззінг, тестування на проникнення тощо, щоб отримати більш повний та точний результат;
- використовувати ці методи з урахуванням специфіки та вимог програмного забезпечення, яке тестується, та адаптувати їх до різних сценаріїв та ситуацій;
- забезпечувати належну документацію та звітність про процес та результати використання цих методів, а також про виявлені та усунені проблеми.

4.11. Поєднання CCOF із існуючими методами автоматизованого ми тестування безпеки програмного забезпечення як стратегія покращення

CCOF (Common Control Objectives Framework) є стандартом, який визначає набір цілей та вимог для управління ризиками безпеки інформації в організаціях, що використовують інформаційні технології. CCOF базується на принципах розподілу відповідальності, мінімізації ризиків, забезпечення

відповідності та підвищення ефективності. CCOF допомагає розв'язати проблему, пов’язану з розробкою та оцінкою методів автоматизованого тестування безпеки програмного забезпечення, шляхом надання спільної мови та критеріїв для порівняння різних стандартів та регуляторних вимог, таких як:

- National Institute of Standards and Technology Special Publication 800-53 (NIST 800-53 Rev 4) - документ, що містить рекомендації щодо вибору та реалізації заходів безпеки для інформаційних систем федерального уряду США;
- FedRAMP - програма сертифікації хмарних послуг для федеральних установ США, яка вимагає від постачальників хмарних послуг дотримуватися певних стандартів безпеки, а також проходити незалежні аудити та моніторинг;
- Common Objectives for Information Technologies (COBIT) - міжнародний фреймворк для управління IT-процесами та ресурсами, який допомагає організаціям досягати своїх цілей за допомогою ефективного та контрольованого використання IT;
- Sarbanes-Oxley Act (SOX) Information Technology General Controls (ITGC) - набір заходів безпеки, які стосуються інформаційних систем, що впливають на фінансову звітність організацій, які підпадають під дію закону SOX, прийнятого у 2002 році для запобігання фінансовим шахрайствам4
- General Data Protection Regulation (GDPR) - регуляторний акт Європейського Союзу, який встановлює правила щодо захисту персональних даних громадян ЄС, а також відповідальність та санкції для організацій, які обробляють такі дані;
- PCI DSS - стандарт безпеки даних для платіжних карт, який визначає вимоги до захисту даних про картки та транзакції, які проводяться за допомогою них, а також процедури аудиту та сертифікації для організацій, які працюють з платіжними картами;
- ISO 27001 - міжнародний стандарт, який визначає вимоги до системи управління безпекою інформації (ISMS), яка дозволяє організаціям ідентифікувати, аналізувати та контролювати ризики безпеки інформації;

- FISMA - закон США, який встановлює вимоги до безпеки інформації для федеральних агентств та їх постачальників, а також процеси оцінки та звітування про стан безпеки інформаційних систем;
- HIPAA - закон США, який встановлює вимоги до захисту медичної інформації, яка стосується здоров’я, лікування та страхування пацієнтів, а також відповідальність та санкції для організацій, які обробляють таку інформацію.

CCOF дозволяє організаціям використовувати один і той же набір заходів безпеки для дотримання різних стандартів та регуляторних вимог, а також спрощує процес аудиту та перевірки ефективності цих заходів безпеки.

Можливі наукові й практичні результати CCOF пов’язані з розв'язанням проблеми, пов’язаної з автоматизованим тестуванням безпеки програмного забезпечення, можуть бути такими:

- розробка та удосконалення методів, інструментів та стандартів для автоматизованого тестування безпеки програмного забезпечення, які враховують специфіку різних доменів CCOF та вимоги різних регуляторних органів;
- підвищення якості та надійності програмного забезпечення, яке відповідає вимогам безпеки інформації, а також зменшення витрат на його розробку, впровадження та підтримку;
- підвищення рівня довіри та задоволення клієнтів, партнерів, регуляторів, аудиторів та інших зацікавлених сторін до програмного забезпечення, яке використовується в організаціях, що дотримуються CCOF;
- сприяння розвитку наукової спільноти та професійної діяльності в галузі безпеки інформації та програмного забезпечення, а також поширення знань та досвіду за допомогою наукових публікацій, конференцій, семінарів, тренінгів тощо.

Рекомендації щодо застосування CCOF у поєднанні з чинними автоматизованими програмними рішеннями для тестування безпеки програмного забезпечення:

- визначити домени CCOF, які стосуються програмного забезпечення, яке потребує тестування, та відповідні цілі та вимоги безпеки для кожного домену;
- обрати найбільш прикладні автоматизовані програмні рішення для тестування безпеки програмного забезпечення, які підтримують вимоги CCOF, та оцінити їхні переваги та недоліки;
- розробити та реалізувати план тестування безпеки програмного забезпечення, який враховує домени CCOF, вимоги безпеки, автоматизовані програмні рішення, тестові сценарії, тестові дані, тестове середовище, критерії успіху та процеси звітування та аналізу результатів;
- виконати тестування безпеки програмного забезпечення за допомогою обраних автоматизованих програмних рішень, відстежувати та документувати хід тестування, виявлені дефекти та відхилення від вимог CCOF;
- оцінити результати тестування безпеки програмного забезпечення, порівняти їх з очікуваними результатами, визначити ступінь відповідності програмного забезпечення вимогам CCOF, виявити проблемні та ризикові області, запропонувати рекомендації щодо покращення безпеки програмного забезпечення;
- повторити тестування безпеки програмного забезпечення після внесення змін або виправлення дефектів, переконатися, що програмне забезпечення відповідає вимогам CCOF, та підтвердити його готовність до використання.

Висновки до розділу 4

У розділі були використані інструменти автоматизованого тестування безпеки програмного забезпечення. Фаззінг Whitebox із SAGE виділяється як потужний інструмент, спроможний автоматично виявляти вразливості в програмах, зокрема у сфері аналізу файлів та пакетів. Застосування

динамічного символьного виконання та техніки "пошук поколінь" робить його ефективним для великих програм, а висока масштабованість підтверджує його перспективність у виявленні безпекових проблем.

Також, апробовано інший підхід до автоматизованої генерації тестів, такий як S2E. Його успішність у тестуванні системного коду та гнучкість в застосуванні підтверджують його значущість у науковому та промисловому середовищі. Всі розглянуті підходи відображають різноманітні техніки та стратегії для забезпечення якості та безпеки програмного забезпечення. Фаззінг Whitebox із SAGE має унікальні особливості, такі як генерація тестів, що покривають всі можливі шляхи виконання програми, та використання техніки "пошук поколінь". Його масштабованість та інтеграція з хмарними сервісами роблять його високоефективним інструментом.

Далі у роботі було досліджено стандарт CCOF, який визначає цілі та вимоги для управління ризиками безпеки інформації. Проаналізовані основні принципи, контролі та переваги CCOF, а також його відношення до різних стандартів та регуляторних вимог. Спостерігається, що CCOF дозволяє ефективно впроваджувати контролі безпеки в організаціях, що використовують інформаційні технології, та спрощує аудит та перевірку ефективності цих заходів безпеки.

Висновки щодо CCOF вказують на його значущість у забезпеченні єдиної мови та критеріїв для управління ризиками безпеки інформації в організаціях. Використання CCOF для порівняння різних методів автоматизованого тестування безпеки програмного забезпечення визначено як обґрунтований підхід, що може сприяти якості та надійності програмного забезпечення.

Загалом, результати підтверджують високий потенціал та ефективність розглянутих підходів та стандартів у забезпеченні безпеки програмного забезпечення та управлінні ризиками.

ВИСНОВКИ

Результати, які були представлені в даній магістерській роботі, доповнюються науковими та практичними висновками, враховуючи важливість та новаторство Common Control Objectives Framework (CCOF).

CCOF є ключовим стандартом, що визначає цілі та вимоги для управління ризиками безпеки інформації в організаціях, які використовують інформаційні технології. Вперше в рамках даної роботи висвітлено, як CCOF може бути використаний як основа для порівняння різних стандартів та регуляторних вимог у контексті автоматизованого тестування безпеки програмного забезпечення.

Результати наукового дослідження пов'язані з вирішенням проблем, що виникають при розробці та оцінці методів автоматизованого тестування безпеки програмного забезпечення. Застосування CCOF може сприяти розробці та удосконаленню методів, інструментів та стандартів для автоматизованого тестування, враховуючи різноманітні домени CCOF та вимоги різних регуляторних органів.

Висновки магістерської роботи вказують на можливі наукові та практичні результати, пов'язані із розробкою та удосконаленням методів тестування безпеки програмного забезпечення, зменшення витрат на розробку та підтримку, підвищення довіри зацікавлених сторін та сприяння розвитку галузі безпеки інформації та програмного забезпечення.

В цілому, матеріали магістерської роботи показують, що використання CCOF у контексті автоматизованого тестування безпеки програмного забезпечення є обґрунтованим та перспективним напрямком для подальших досліджень та практичного застосування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ДСТУ ISO/IEC 12207:2016 Інженерія систем і програмного забезпечення. Процеси життєвого циклу програмного забезпечення. Київ: Міністерство економічного розвитку України, 2016. с. 7-13.

2. Чумаченко І. В., Чумаченко О. В., Литвиненко О. В., Шевченко О. В. Технології розроблення програмного забезпечення. Київ: КНУБА, 2019. с. 1-10.

3. Шевченко О. В., Чумаченко І. В., Чумаченко О. В., Литвиненко О. В. Якість програмного забезпечення та тестування: базовий курс. Київ: КНУБА, 2020. С. 194.

4. Шевченко О. В., Чумаченко І. В., Чумаченко О. В., Литвиненко О. В. Тестування програмного забезпечення. Київ: КНУБА, 2021. с. 72-75.

5. Литвиненко О. В. Застосунок для аналізу результатів тестування програмного забезпечення: автореф. дис. на здобуття наук. ступеня канд. техн. наук: спец. 05.13.06 «Інформаційні технології». Київ: КНУБА, 2022. с. 8.

6. Нікс А., Стюарт Т. Ransomware: Defending Against Digital Extortion. Sebastopol: O'Reilly Media, 2016. с. 1-17.

7. Шевченко О. В., Чумаченко І. В., Чумаченко О. В., Литвиненко О. В. Якість програмного забезпечення та тестування: базовий курс. Київ: КНУБА, 2020. с. 194.

8. Шевченко О. В., Чумаченко І. В., Чумаченко О. В., Литвиненко О. В. Тестування програмного забезпечення. Київ: КНУБА, 2021. с. 72-75.

9. Литвиненко О. В. Застосунок для аналізу результатів тестування програмного забезпечення: автореф. дис. на здобуття наук. ступеня канд. техн. наук: спец. 05.13.06 «Інформаційні технології». Київ: КНУБА, 2022. с. 8.

10. Толюпа С.В., Наконечний В.С., Штаненко С.С., Кострица В.О. Алгоритм протидії кібератакам на основі стеганографічних методів технології

Soft Tempest. Вісник Інституту спеціальних засобів зв’язку та захисту інформації. 2020. № 10(1). Україна, Київ. с. 5-151.

11. Терейковський І.А. Методи розпізнавання кібератак. Робоча програма навчальної дисципліни (Силабус). 2021. Україна, Київ. с. 1-82.

12. Опорний конспект лекцій з дисципліни “Безпека програм та даних”. Міністерство освіти і науки України. 2019. Україна, Луцьк. с. 20-27.

13. Інформаційна безпека та захист даних в комп’ютерних технологіях і мережах. Національний технічний університет України «Київський політехнічний інститут». 2018. Україна, Київ. с. 5-6.

14. Авраменко А.С., Авраменко В.С., Косенюк Г.В. Тестування програмного забезпечення. Навчальний посібник. – Черкаси: ЧНУ імені Богдана Хмельницького, 2017 с. 72-75.

15. Семенов А.В., Левченко А.В., Кузьмін О.В. Методи та засоби автоматизованого тестування безпеки програмного забезпечення. Науково-технічний журнал “Інформаційні технології та комп’ютерна інженерія”. 2018. № 2(48). Україна, Харків. с. 5-141.

16. Шевченко О.О., Литвиненко О.В., Черній О.В. Автоматизоване тестування безпеки програмного забезпечення з використанням інструментів SAST, DAST та IAST. Вісник Національного університету “Львівська політехніка”. Серія: Комп’ютерні науки та інформаційні технології. 2019. № 947. Україна, Львів. с. 101-108.

17. Лісовський О.В., Лісовський В.В. Основи тестування програмного забезпечення. Навчальний посібник. 2017. Україна, Київ. с. 117-120.

18. Лісовський О.В., Лісовський В.В., Автоматизоване тестування безпеки програмного забезпечення: теорія і практика, 2020, Україна, Київ, с. 112-120.

19. Дмитро Шевченко, Володимир Ковальчук, Ігор Левченко, Тестування безпеки веб-додатків за допомогою інструментів OWASP, 2019, Україна, Львів, с. 45-53.

20. Марк Керсті, Джим ДелГроссо, Джон Стівенс, Application Security in the ISO/IEC 27000 Family, 2018, Великобританія, Лондон, с. 87-94.

21. Д-р Олександр Кнут, Девід Саха та Гаррі Полушкін, Прогрес та виклики цифрової трансформації бізнесу в Україні – результати репрезентативного опитування бізнесу, 2021, Німеччина, Берлін, с. 8-9.

22. Дергачова Г. М. Колешня Я. О., Цифрова трансформація бізнесу сутність, принципи та переваги, 2020, Україна, Київ, с. 3-4.

23. Павло Харламов, Цифрова трансформація: чому вона необхідна кожній компанії, 2021, Україна, Київ, с. 6-7.

24. Айзатський, М., Серебряний, К., Чанг, О., Ар'я, А., Уіттакер, М.: Представлення OSS-Fuzz: Постійне тестування на проникнення для відкритого програмного забезпечення (2016), посилання.

25. Арці С., Кізун А., Долбі Дж., Тіп Ф., Діг Д., Парадкар А.М., Ернст М.Д.: Пошук помилок у веб-застосунках за допомогою динамічного тестового генерування та перевірки моделі з явним станом. IEEE Trans. Softw. Eng., с. 474–494 (2010).

26. Ар'я А., Некар С.: Fuzzing для забезпечення (2012), посилання.

27. Авгерінос Т., Ча С.К., Хао Б.Л.Т., Брамлі Д.: AEG: автоматичне генерування експлойтів. У: Симпозіум з безпеки мережі та розподілених систем (2011), с. 216–233.

28. Бажанюк О., Лукаїдес Дж., Розенбаум Л., Таттл М.Р., Ціммер В.: Символьне виконання для безпеки BIOS. У: USENIX Workshop on Offensive Technologies (2015).

29. Беллар Ф.: QEMU, швидкий та портативний динамічний транслятор. У: Річна технічна конференція USENIX (2005).

30. Бунімова Е., Ґодфройд П., Мольнар Д.: Мільярди та мільярди обмежень: тестування на проникнення у виробництві за допомогою білого ящика. У: Міжнародна конференція з інженерії програмного забезпечення (2013).

31. CERT: База даних CERT з вразливостей безпеки (2024)

32. Ча С.К., Авгерінос Т., Реберт А., Брумлі Д.: Розпуск хаосу у бінарному коді. У: Симпозіум з безпеки та конфіденційності IEEE (2012).

33. Еделштейн О., Фархі Е., Голдин Е., Нір Й., Рацаб, Ґ., Ур С.: Фреймворк для тестування багатопотокових програм на Java. Concurrency Comput.: Practice Exp. с. 485–499.

34. EPFL та Cyberhaven Inc: Розподіл програмного забезпечення S2E.

35. Ґодфройд,П., Лахірі С.К.: Від програми до логіки: вступ. У: Літня школа LASER (2012).

36. Ґодфройд П., Левін М., Мольнар Д.: SAGE: біле фазз-тестування для тестування безпеки. Commun. ACM (2012).

37. Ґодфройд П., Мольнар Д.: Fuzzing у хмарі. Технічний звіт MSR-TR-201029, Дослідницька лабораторія Microsoft.

38. Халлем С., Челф Б., Сіє І., Енглер Д.: Система та мова для створення системно-специфічних статичних аналізів. У: Міжнародна конференція з проектування та реалізації мов програмування (2002).

39. Єлінек Я.: Fortify source: перевірка розміру об'єкта для запобігання (деяким) переповненням буфера.

40. МакМінн П.: Пошукове тестування програмного забезпечення: огляд. Int. J. Softw. Test. Verification Reliab. (2004).

41. Ренцельманн М.Дж., Кадав А., Свіфт М.М.: Symdrive: тестування драйверів без пристроїв. У: Симпозіум з проектування та впровадження операційних систем (2012).

42. Серебряний К., Брюнінг Д., Потапенко А., В'юков Д.: AddressSanitizer: швидкий перевіряльник правильності адрес. У: Річна технічна конференція USENIX (2012).

ДОДАТКИ

Додаток А.

Типи автоматизованого тестування безпеки програмного забезпечення

Тип	Опис	Фокус	Переваги	Недоліки
Статичне тестування безпеки додатків (SAST)	Аналіз вихідного або бінарного коду без його виконання, виявлення вразливостей і слабких місць.	Раннє виявлення проблем безпеки на етапі розробки.	Раннє виявлення. Інтеграція з розробкою. Економічно ефективний	Помилкові спрацьовування та обмежений аналіз часу виконання
Динамічне тестування безпеки додатків (DAST)	Тестує програму під час виконання в розгорнутому середовищі, імітуючи реальні сценарії атак.	Виявлення вразливостей, які можуть виникнути під час виконання.	Симуляція реального середовища. Визначає вразливості під час виконання.	Обмежене покриття. Помилкові спрацьовування.
Інтерактивне тестування безпеки додатків (IAST)	Поєднує елементи SAST і DAST, забезпечуючи зворотній зв’язок у реальному часі щодо проблем безпеки під час виконання.	Реалізує аналіз як вихідного коду, так і поведінку під час виконання.	Симуляція реального середовища. Поєднує стандарти SAST і DAST. Менше хибних спрацьовувань	Ресурсовитратність. Обмежена доступність інструментів
Оцінка вразливостей	Систематичний процес виявлення, кількісного визначення та встановлення	Виявлення слабких сторін. Пріоритезація ризиків. Постійний	Проактивне управління ризиками. Всебічне розуміння. Вимоги до	Хибні позитивні та негативні результати. Обмежено відомими

Тип	Опис	Фокус	Переваги	Недоліки
	пріоритетів вразливостей.	моніторинг.	відповідності	вразливими місцями.
Тестування на проникнення	Моделює кібератаки для виявлення та використання вразливостей у контрольованому середовищі.	Комплексна оцінка стану безпеки програми.	Комплексна оцінка. Людський досвід. Визначає бізнес-ризики	Займе багато часу. Ресурсомісткий
Тестування Fuzz	Передбачає введення недійсних або неочікуваних даних для виявлення вразливостей, пов’язаних із перевіркою введення та обробкою помилок.	Виявлення проблем, які можуть бути неочевидними під час традиційного тестування.	Визначає вразливості, пов’язані з введенням даних. Автоматизація. Неупереджене тестування	Може пропустити комплексні вразливості. Ресурсомісткість

Додаток Б.

Популярні інструменти автоматизованого тестування безпеки програмного забезпечення

Тип	Інструмент	Опис	Фокус	Переваги	Недоліки	Країна
Статичне тестування безпеки додатків (SAST)	Checkmarx	Забезпечує статичний аналіз коду для виявлення вразливостей безпеки у вихідному коді.	Інтеграція з CI/CD та IDE для безперервного тестування.	Checkmarx ефективно аналізує код, забезпечуючи повне уявлення про потенційні вразливості системи безпеки. Пропонує повне охоплення, визначаючи широкий спектр проблем безпеки в програмах. Checkmarx легко інтегрується в конвеєри розробки, підтримуючи постійне тестування безпеки.	Checkmarx може бути відносно дорогим, що може бути міркуванням для невеликих організацій з обмеженим бюджетом. Як і будь-який автоматизований інструмент, Checkmarx може генерувати помилкові спрацьовування, що потребує ручної перевірки.	Ізраїль
	Fortify (Micro Focus)	Пропонує статичний аналіз коду для виявлення та	Інтеграція з різними середовищами розробки та системами збірки.	Fortify забезпечує надійний статичний аналіз коду, допомагаючи виявити та усунути вразливі	Складність Fortify може створити проблеми для користувачів, які не знайомі з	США

Тип	Інструмент	Опис	Фокус	Переваги	Недоліки	Країна
		усунення вразливостей у програмах.		місця безпеки. Можливості інтеграції Добре інтегрується з різними середовищами розробки та конвеєрами CI/CD. Fortify дозволяє налаштовувати політики безпеки, адаптуючи їх до конкретних вимог проєкту.	розширеними інструментами тестування безпеки. Виконання сканування Fortify може потребувати значних ресурсів, що вплине на продуктивність.	
	Veracode	Хмарне рішення SAST для виявлення та усунення вразливостей безпеки в коді.	Інтегрується з інструментами розробки та забезпечує централізовану платформу для тестування безпеки.	Veracode надає уніфіковану платформу для різних методологій тестування безпеки, що спрощує процес тестування. Хмарна модель усуває потребу в локальній інфраструкту	Модель ціноутворення Veracode може бути обговоренням для організацій з обмеженим бюджетом. Деякі користувачі можуть вважати, що	США

Тип	Інструмент	Опис	Фокус	Переваги	Недоліки	Країна
				рі, пропонуючи гнучкість. Veracode відомий своїми зусиллями зменшити помилкові спрацьовування, забезпечуючи більш точні результати	можливості налаштування платформи обмежені порівняно з іншими інструментами.	
Динамічне тестування безпеки додатків (DAST)	OWASP ZAP (Zed Attack Proxy)	Інструмент із відкритим кодом для пошуку вразливостей у веб-додатках під час виконання.	Автоматичні сканери, різні інструменти як для ручного, так і для автоматичного тестування.	OWASP ZAP, будучи відкритим і безкоштовним, доступний широкому колу користувачів і організацій. Розвиток, керований спільнотою, забезпечує постійне вдосконалення та адаптацію до нових тенденцій безпеки. OWASP ZAP відомий своїм зручним	Можливості автоматизації OWASP ZAP можуть бути менш просунутими порівняно з комерційними інструментами. Незважаючи на універсальність, OWASP ZAP може не забезпечувати такої ж	Глобально (відкритий вихідний код, керований спільнотою)

Тип	Інструмент	Опис	Фокус	Переваги	Недоліки	Країна
				інтерфейсом, що робить його доступним як для початківців, так і для досвідчених користувачів.	глибини аналізу певних вразливостей, як спеціалізовані інструменти.	
	Netsparker	Сканер безпеки веб-додатків для виявлення вразливостей під час виконання.	Точне сканування, сканування на основі доказів, інтеграція з конвеєрами CI/CD.	Netsparker має зручний інтерфейс, що полегшує використання різними користувачами. Відомий високою точністю виявлення вразливостей, що зменшує потребу в розширеній перевірці вручну. Netsparker включає сканування на основі доказів, надаючи докази виявлених вразливостей для підвищення довіри.	Модель ціноутворення Netsparker може бути доцільною для організацій з обмеженим бюджетом. Певні функції автоматизації не такі просунуті, як в інших комерційних інструментах.	Великобританія

Тип	Інструмент	Опис	Фокус	Переваги	Недоліки	Країна
	AppSpider (Rapid7)	Інструмент динамічного сканування веб-додатків для виявлення вразливостей безпеки.	Сканує веб-додатки на загальні вразливості, вичерпні звіти.	AppSpider пропонує комплексне сканування, виявляючи широкий спектр вразливостей у веб-додатках. Добре інтегрується з іншими рішеннями Rapid7 і часто є частиною ширших програм безпеки. AppSpider має зручний інтерфейс, що робить його доступним для різних користувачів.	Вартість AppSpider може бути міркуванням для невеликих організацій. Запуск сканування за допомогою AppSpider може потребувати значних ресурсів, що вплине на продуктивність.	США
Інтерактивне тестування безпеки додатків (IAST)	Contrast Security	Рішення IAST забезпечує зворотний зв’язок безпеки в режимі реальн	Постійний моніторинг, автоматичне виявлення вразливостей.	Contrast Security забезпечує безпеку додатків у реальному часі, пропонуючи миттєве розуміння вразливостей	Функції платформи можуть бути складними для користувачів, які не знайомі з розширеними	США

Тип	Інструмент	Опис	Фокус	Переваги	Недоліки	Країна
		ого часу під час виконання програми.		. Функціонує як комплексна платформа AppSec, що охоплює різні аспекти безпеки програм. Contrast Security відома високою точністю виявлення вразливостей, зменшенням помилкових спрацьовувань.	інструментами AppSec. Вартість Contrast Security може бути міркуванням для невеликих організацій.	
	Checkmarx SCA (Software Composition Analysis)	Інтегрується з SAST для аналізу в режимі реального часу компонентів із відкритим вихідним кодом та їх	Визначає ризики вихідного коду та керує ними.	Checkmarx ефективно аналізує код, забезпечуючи повне уявлення про потенційні вразливості системи безпеки. Пропонує повне охоплення, визначаючи широкий спектр проблем	Checkmarx може бути відносно дорогим, що може бути міркуванням для невеликих організацій з обмеженим бюджетом. Як і будь-який автоматиз	Ізраїль

Тип	Інструмент	Опис	Фокус	Переваги	Недоліки	Країна
		стану безпеки.		безпеки в програмах. Checkmarx легко інтегрується в конвеєри розробки, підтримуючи постійне тестування безпеки.	ований інструмент, Checkmarx може генерувати помилкові спрацьовування, що потребує ручної перевірки.	
Оцінка вразливостей	Nessus	Широко використовуваний сканер вразливостей для виявлення слабких місць.	Безперервний моніторинг, комплексне сканування, перевірки відповідності.	Nessus широко використовується в різних галузях і вважається стандартом у скануванні вразливостей. Він має велику базу даних відомих вразливостей, що сприяє точному скануванню. Nessus добре інтегрується з іншими рішеннями та інструментами безпеки, підвищуючи загальну безпеку.	Nessus може мати вартість, пов'язану з розширеними функціями, що може стати причиною розгляду для деяких користувачів. Як і будь-який інструмент сканування, Nessus може генерувати помилкові спрацьовування, що потребує ручної перевірки.	США
	OpenVAS	Сканер	Комплекс	Будучи	Порівняно	Глобал

Тип	Інструмент	Опис	Фокус	Переваги	Недоліки	Країна
	(Open Vulnerability Assessment System)	вразливостей з відкритим кодом для виявлення проблем безпеки.	не сканування, постійний моніторинг	відкритим і безкоштовним, OpenVAS доступний широкому колу користувачів і організацій Розвиток, керований спільнотою, забезпечує постійне вдосконалення та адаптацію до нових тенденцій безпеки. OpenVAS забезпечує гнучкість і параметри налаштування, що дозволяє користувачам адаптувати сканування до своїх потреб.	з комерційними рішеннями, OpenVAS може мати більш обмежені можливості підтримки. OpenVAS більш складним порівняно з деякими зручними комерційними інструментами.	ьно (відкритий вихідний код, керований спільнотою)
Тестування на проникнення	Metasploit	Платформа для тестування на проникнення з відкритим кодом	Модулі експлуатації вразливостей, інтеграція з іншими інструментами.	Metasploit — це комплексний інструмент для тестування на проникнення, що дозволяє симулювати	Metasploit може бути складним для користувачів, які не знайомі з інструментами тестування	Глобально (відкритий вихідний код, керований спільнотою)

Тип	Інструмент	Опис	Фокус	Переваги	Недоліки	Країна
		для пошуку та використання вразливостей		атаки для виявлення вразливостей. Metasploit, будучи відкритим кодом і керованим спільнотою, отримує переваги від постійних оновлень і внесків. Він має велику базу даних експлойтів, що забезпечує широкий спектр можливостей тестування	на проникнення. Використання Metasploit вимагає етичних міркувань і дотримання правових стандартів, щоб уникнути зловживання.	
	Burp Suite	Інструмент тестування безпеки веб-додатків для ручного та автоматичного тестування.	Проксі, сканер, інтрудер, репітер, секвенсор тощо.	Burp Suite пропонує широкий спектр функцій, включаючи сканування, сканування та різні інструменти для ручного тестування. Має зручний інтерфейс, що робить	Burp Suite Pro є безкоштовна версія, розширені функції Burp Suite Pro платні. Виконання певних сканувань може потребувати ресурсів, що вплине	Великобританія

Тип	Інструмент	Опис	Фокус	Переваги	Недоліки	Країна
				його доступним як для новачків, так і для досвідчених спеціалістів із безпеки. Burp Suite отримує переваги від спільноти активних користувачів, надає підтримку, розширення та постійне вдосконалення.	на продуктивність.	
Тестування Fuzz	Atheris	Фазер Python з відкритим кодом, розроблений Google	Підтримує тестування фаз для програм Python.	Atheris зосереджується на охопленні коду Python, ідентифікації вразливостей, характерних для програм Python. Будучи інструментом з відкритим кодом, Atheris доступний широкому колу	Atheris розроблено спеціально для Python, і його застосування може бути обмежено проектами на основі Python. Подібно до інших інструментів тестування нечіткості,	США

Тип	Інструмент	Опис	Фокус	Переваги	Недоліки	Країна
				користувачів і отримує переваги від внесків спільноти. Atheris легко інтегрується з кодовими базами Python, що робить його придатним для тестування програм Python.	Atheris спеціалізується на своєму фокусі та може не охоплювати інші аспекти тестування безпеки. Відносно новий інструмент, спільнота та документація можуть бути не такими великими порівняно з більш усталеними інструментами.	
	American Fuzzy Lop (AFL)	Популярний інструмент тестування fuzz для пошуку вразливостей безпеки.	Ефективне тестування fuzz, підтримує різні формати файлів.	AFL є високоефективним у фаз-тестуванні, допомагаючи виявити вразливі місця через мутацію вхідних даних. Будучи відкритим кодом, AFL	AFL спеціалізується на тестуванні нечіткості та може не охоплювати інші аспекти тестування безпеки. Може знадобитися	Глобально (відкритий вихідний код, керований спільнотою)

Тип	Інструмент	Опис	Фокус	Переваги	Недоліки	Країна
				доступний для широкого кола користувачів і активно розробляється спільнотою. AFL широко використовується в спільноті безпеки для пошуку вразливостей у програмному забезпеченні.	конфігурація та налаштування вручну для конкретних сценаріїв тестування.	
	Peach Fuzzer	Комплексна платформа для тестування fuzz для фаззингу протоколів.	Підтримка різних протоколів, простий у використанні інтерфейс.	Peach Fuzzer відомий своїми можливостями фаззингу незалежно від протоколу, підтримуючи широкий спектр протоколів. Розширюваний, що дозволяє користувачам створювати власні правила фаззингу та адаптувати їх	Користувачі можуть зіткнутися з кривою навчання, особливо під час налаштування Peach Fuzzer для певних програм. Проведення широких кампаній фаззингу може потребувати значних обчислювальних	США

Тип	Інструмент	Опис	Фокус	Переваги	Недоліки	Країна
				до різних програм. Peach Fuzzer надає комплексні функції звітування та аналізу, щоб зрозуміти та визначити пріоритети вразливостей	ресурсів.	

метадані

Заголовок

Розробка та оцінка методів автоматизованого тестування безпеки програм-ного забезпечення

Автор

Рекута В.В. Науковий керівник / Експерт

підрозділ

King Danylo University

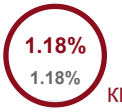
Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв	ß	11
Інтервали	A→	0
Мікропробіли	·	24
Білі знаки	ß	56
Парафрази (SmartMarks)	a	14

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.



КП 1

КП 2

25
Довжина фрази для коефіцієнта подібності 2

20841
Кількість слів

163162
Кількість символів

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз — Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)	
1	http://repository.ukd.edu.ua/bitstream/handle/123456789/397/%D0%94%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC%D0%BD%D0%B0_%D0%A2%D0%B0%D1%82%D0%B0%D1%80%D1%87%D1%83%D0%BA.pdf?sequence=1	23	0.11 %
2	http://repository.ukd.edu.ua/bitstream/handle/123456789/390/%D0%9C%D0%B0%D0%BD%D1%82%D1%83%D0%BB%D1%8F%D0%BA%20%D0%94.%D0%92.%20%D0%9A%D0%A0.pdf?sequence=1	23	0.11 %
3	http://repository.ukd.edu.ua/bitstream/handle/123456789/394/%D0%A0%D1%83%D0%B4%D0%B8%D0%B9%20%D0%90%D0%BD%D0%B4%D1%80%D1%96%D0%B9%20%D0%B4%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC%D0%BD%D0%B0.pdf?sequence=1	22	0.11 %