Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ДОПУСТИТИ ДО ЗАХИСТУ:
В.о. завідувача кафедри
кібербезпеки та захисту інформації
________ Іван ПАРХОМЕНКО
«___» червня 2025 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи

галузь знань 12 Інформаційні технології
(шифр і назва галузі знань)

спеціальність 125 Кібербезпека
(код і назва спеціальності)

освітній ступень бакалавр

освітня програма Кібербезпека
(назва освітньо-професійної програми)

на тему: «Метод тестування смарт-контрактів на вразливості»

Виконавець: студент IV курсу, групи КБ-44

________________ **Тарас БІЛОКІНЬ**
(підпис) (ім’я, прізвище)

	Підпис	Ім’я ПРІЗВИЩЕ
Керівник		Юрій ЩЕБЛАНІН
Нормоконтроль		Яніна ШЕСТАК

Київ 2025

Міністерство освіти і науки України

Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ЗАТВЕРДЖЕНО:
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
________ Іван ПАРХОМЕНКО
«29» листопада 2025 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності 125 Кібербезпека
(код і назва спеціальності)

освітньої програми Кібербезпека
(назва освітньо-професійної програми)

Студенту **КБ-44** **Білоконю Тарасу Ігоровичу**
(група) (прізвище ім'я по батькові)

Тема кваліфікаційної роботи Метод тестування смарт-контрактів на вразливості

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Тема кваліфікаційної роботи затверджена на засіданні кафедри кібербезпеки та захисту інформації протокол №6 від 28.11.2024 р.

2. ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Зменшення реалізації загроз з використанням автоматизованих інструментів

3. ЗМІСТ РОЗРАХУНКОВО-ПОЯСНЮВАЛЬНОЇ ЗАПИСКИ

Необхідно ознайомитись із сучасними вразливостями смарт-контрактів, провести огляд стандартів безпеки та розробити метод їх автоматизованого виявлення з оцінкою ризиків

4. ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Практична цінність Проектування методу гібридного тестування смарт-контрактів

4. ДАТА ВИДАЧІ ЗАВДАННЯ

Дата видачі завдання: 29 листопада 2024 року

Завдання видав ____________________ Юрій ЩЕБЛАНІН
(підпис) (ім'я, прізвище)

Завдання прийняв до виконання ____________________ Тарас БІЛОКІНЬ
(підпис) (ім'я, прізвище)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Найменування етапів робіт	Строки виконання робіт (початок-кінець)	Відмітка про виконання
1	Уточнення постановки задачі	25.11.2024 – 29.12.2024	*виконано*
2	Аналіз літератури	30.12.2024 – 19.01.2025	*виконано*
3	Обґрунтування вибору методів дослідження	20.01.2025 – 31.01.2025	*виконано*
4	Дослідження функціонування та вразливостей смарт-контрактів	03.02.2025 – 02.03.2025	*виконано*
5	Огляд сучасних методів та стандартів безпеки смарт-контрактів	03.03.2025 – 30.03.2025	*виконано*
6	Поглиблений аналіз інструментів тестування смарт-контрактів	31.03.2025 – 20.04.2025	*виконано*
7	Розробка методу тестування смарт-контрактів на вразливості	21.04.2025 – 18.05.2025	*виконано*
8	Оформлення пояснювальної записки	19.05.2024 – 25.05.2024	*виконано*
9	Підготовка до захисту кваліфікаційної роботи	26.05.2025 – 08.06.2025	*виконано*

Завдання видав ____________________ Юрій ЩЕБЛАНІН
(підпис) (ім'я, прізвище)

Завдання прийняв до виконання ____________________ Тарас БІЛОКІНЬ
(підпис) (ім'я, прізвище)

Термін подання кваліфікаційної роботи до ЕК 10 червня 2025 року

РЕФЕРАТ

Кваліфікаційна робота складається зі вступу, трьох розділів, загальних висновків, списку використаних джерел, додатків, має 73 сторінок основного тексту, 16 рисунків, 1 таблицю. Список використаних джерел містить 36 найменувань і займає 4 сторінки.

Метою роботи є розробка гібридного методу тестування смарт-контрактів на вразливості, що забезпечує автоматизований аналіз коду для виявлення та оцінки потенційних загроз.

Для досягнення зазначеної мети поставлено наступні завдання:

- дослідити теоретичні основу смарт-контракта, застосування, основні типи вразливостей та майбутні виклики
- проаналізувати сучасні сканери вразливостей та сформувати критерії оцінки їх ефективності
- проаналізувати фреймворки та стандарти безпеки смарт-контрактів для виявлення та оцінки вразливостей
- розробити метод тестування смарт-контрактів для виявлення та оцінки ризиків, використовуючи автоматизовані інструменти статичного та динамічного аналізу
- розробити рекомендації для розробників щодо зменшення ймовірності вразливостей та підвищення рівня безпеки смарт-контрактів

Об'єктом дослідження є процес автоматизації транзакцій в блокчейн технологіях за рахунок використання смарт-контрактів.

Предметом дослідження є методи виявлення, аналізу та оцінки вразливостей смарт-контрактів.

Методи дослідження кваліфікаційної роботи: теоретичний аналіз сучасних стандартів і гайдлайнів, порівняльний аналіз інструментів та емпіричне тестування смарт-контрактів.

Практичною цінністю роботи є розробка комплексного методу тестування та рекомендацій щодо, виявлення, аналізу та усунення вразливостей для розробників смарт-контрактів, аудиторів безпеки та організацій, знижуючи ризики компрометації та зменшуючи фінансові втрати.

Ключові слова: смарт-контракти, блокчейн, вразливості, безпека, OWASP SCSVS, ISO 22739, NIST IR8202, Ethereum Security Best Practices, Code4rena, Solidity Style Guide, OpenZeppelin, Slither, Mythril, Aderyn, DeFi, DAO, аудит, статичний аналіз, динамічний аналіз, фаззинг.

ЗМІСТ

ВСТУП.....9
РОЗДІЛ 1 СМАРТ-КОНТРАКТИ В БЛОКЧЕЙН-ТЕХНОЛОГІЯХ: ТЕХНОЛОГІЯ, РЕАЛІЗАЦІЯ, БЕЗПЕКА ТА МАЙБУТНІ ВИКЛИКИ.....11
1.1 Визначення, принципи функціонування, архітектура смарт-контрактів.....11
1.2 Технічна реалізація смарт-контрактів.....12
1.3 Життєвий цикл смарт-контрактів та управління безпекою.....14
1.4 Сфери застосування смарт-контрактів у фінансах, технологіях та традиційних галузях.....15
1.5 Класифікація вразливостей смарт-контрактів.....18
1.6 Наслідки та статистика компрометації смарт-контрактів.....20
1.7 Моделі загроз та сучасний підхід до безпеки смарт-контрактів.....22
Висновки за розділом 1.....23
РОЗДІЛ 2 СПЕЦІАЛІЗОВАНІ ФРЕЙМВОРКИ ДЛЯ ЗМЕНШЕННЯ РІВНЯ ВРАЗЛИВОСТЕЙ СМАРТ-КОНТРАКТІВ.....24
2.1 Огляд сучасних фреймворків безпеки смарт-контрактів і аудиту.....24
2.2 OWASP SCSVS як система верифікації безпеки смарт-контрактів.....27
2.3 Методології тестування вразливостей OWASP Smart Contract Testing Guide 34
2.4 Рекомендації та стандарти безпеки NIST IR8202, ISO 22739 та ISO 23455.....40
2.5 Ethereum Security Best Practices: патерни та принципи безпечного кодування для Ethereum.....41
2.6 Solidity Style Guide як інструмент забезпечення стилістичної та структурної безпеки коду.....42
2.7 OpenZeppelin: бібліотеки безпечного коду та процеси аудиту.....44
Висновки за розділом 2.....45

РОЗДІЛ 3 РОЗРОБКА МЕТОДУ ТЕСТУВАННЯ СМАРТ-КОНТРАКТІВ ДЛЯ ВИЯВЛЕННЯ ТА ОЦІНКИ РИЗИКІВ..46
3.1 Методологія тестування смарт-контрактів від розгортання до аналізу 46
3.2 Статичний аналіз із застосуванням інструментів лінтингу та детекції вразливостей...48
3.3 Динамічний аналіз, фаззинг, симуляція атак смарт-контрактів 49
3.4 Розробка методу тестування смарт-контрактів. Оцінка результатів...............50
3.5 Практичне тестування смарт-контрактів, оцінка і класифікація ризиків........52
Висновок за розділом 3 68
ВИСНОВКИ 69
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 70
Додаток А 74
Додаток Б...77

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

DAO	–	Decentralized Autonomous Organization
DEFI	–	Decentralized Finance
EVM	–	Ethereum Virtual Machine
OWASP	–	Open Web Application Security Project
SCSVS	–	Smart Contract Security Verification Standard
SCSTG	–	Smart Contract Security Testing Guide
SAST	–	Static Application Security Testing
DAST	–	Dynamic Application Security Testing
CI/CD	–	Continuous Integration/Continuous Deployment
SSDLC	–	Secure Software Development Life Cycle
DApps	–	Decentralized Applications
DLT	–	Distributed Ledger Technology
PKI	–	Public Key Infrastructure

ВСТУП

Стрімкий розвиток блокчейн-технологій та зростання популярності смарт-контрактів суттєво впливають на трансформацію фінансового сектору, логістики, страхування, управління активами та інших галузей економіки. За останні роки смарт-контракти стали ключовим інструментом для автоматизації транзакцій, підвищення прозорості бізнес-процесів і зниження операційних витрат. За даними аналітичних агентств, ринок смарт-контрактів демонструє одні з найвищих темпів зростання у сфері цифрових технологій, а до 2030 року його обсяг може перевищити 73 мільярди доларів. Впровадження смарт-контрактів у такі сфери, як DeFi, логістика, страхування, управління нерухомістю та медіаіндустрія, відкриває нові можливості для оптимізації бізнес-моделей, підвищення ефективності та зменшення залежності від посередників.

За даними галузевих звітів, лише у 2024 році втрати від експлуатації вразливостей смарт-контрактів перевищили 2,3 млрд доларів, а найбільші інциденти були пов’язані з помилками у коді, порушенням контролю доступу, атаками типу реентрантності та маніпуляціями оракулами. Будь-яка помилка у смарт-контракті може призвести до незворотних фінансових втрат, компрометації цифрових активів і втрати довіри користувачів до блокчейн-платформ.

Аналіз сучасної науково-технічної літератури та практичного досвіду провідних компаній і дослідників у сфері блокчейну свідчить, що, незважаючи на наявність низки міжнародних стандартів, рекомендацій і спеціалізованих інструментів для аналізу безпеки смарт-контрактів, питання ефективного, комплексного та автоматизованого тестування залишається відкритим. Зростання складності смарт-контрактів, поява нових типів атак, а також швидке впровадження блокчейн-рішень у критично важливі сфери економіки підвищують вимоги до якості тестування та аудиту. Особливої актуальності ця проблема набуває в умовах

децентралізації, коли відсутність централізованих механізмів контролю та повернення коштів робить безпеку смарт-контрактів ключовим фактором стабільності та розвитку галузі.

Метою роботи є розробка гібридного методу тестування смарт-контрактів на вразливості, що забезпечує автоматизований аналіз коду для виявлення та оцінки потенційних загроз.

Об’єктом дослідження є процес автоматизації транзакцій у блокчейн-технологіях за рахунок використання смарт-контрактів.

Предметом дослідження виступають методи виявлення, аналізу та оцінки вразливостей смарт-контрактів.

Практична цінність роботи полягає у створенні комплексного методу тестування та рекомендацій щодо виявлення, аналізу та усунення вразливостей для розробників смарт-контрактів, аудиторів безпеки та організацій, що сприятиме зниженню ризиків компрометації та фінансових втрат та сприятиме подальшому розвитку децентралізованих сервісів і залученню інвестицій у галузь.

Перший розділ присвячується теоретичним засадам смарт-контрактів у блокчейн-технологіях, розгляду їх архітектури, принципів функціонування, особливостей реалізації, а також аналізу основних типів вразливостей і сучасних викликів у сфері безпеки.

Другий розділ присвячується огляду і порівняння спеціалізованих фреймворків, стандартів та інструментів, спрямованих на зменшення рівня вразливостей смарт-контрактів, а також сформовано критерії оцінки їх ефективності.

Третій розділ присвячено розробці гібридного методу тестування смарт-контрактів для виявлення та оцінки ризиків, практичній реалізації запропонованого підходу з використанням автоматизованих інструментів статичного й динамічного аналізу, а також формулюванню рекомендацій щодо підвищення рівня безпеки смарт-контрактів.

РОЗДІЛ 1
СМАРТ-КОНТРАКТИ В БЛОКЧЕЙН-ТЕХНОЛОГІЯХ: ТЕХНОЛОГІЯ, РЕАЛІЗАЦІЯ, БЕЗПЕКА ТА МАЙБУТНІ ВИКЛИКИ

1.1 Визначення, принципи функціонування, архітектура смарт-контрактів

Смарт-контракт (розумний контракт) – це самовиконувана програма, яка автоматично виконує заздалегідь визначені умови, записані в коді, без участі посередників. Вони дотримуються простої логіки «якщо/тоді»: коли певні умови виконані, контракт автоматично виконує заздалегідь визначені дії.

Вони забезпечують незмінність та автентичність транзакцій у блокчейн-системах, але їхня безпека залежить від якості коду та платформи. Після розгортання контракт захищений від несанкціонованого втручання завдяки криптографічним механізмам [1].

Смарт-контракти застосовуються для управління цифровими активами, автоматизації платежів і роботи децентралізованих додатків (DApps), трансформуючи укладання угод у децентралізованих екосистемах, зменшуючи потребу в посередниках.

Принцип роботи смарт-контрактів ґрунтується на узгодження умов між сторонами та їх реалізації в блокченйі [2]:

– Угода: Сторони, що беруть участь у транзакції, повинні перш за все узгодити умови. Вони повинні визначити, як працюватиме смарт-контракт, включаючи такі речі, як необхідні критерії. Прикладами таких умов є автентифікація та авторизація платежу, щоб перевірити, чи платіж надходить з правильного джерела та чи достатньо балансу для здійснення такої транзакції.

– Створення контракту: Після досягнення згоди учасниками блокчейн-транзакції розробник може розпочати написання смарт-контракту. Розробник повинен бути повністю поінформований про бажану реакцію на певні події, наприклад, що станеться, якщо балансу недостатньо.

– Розгортання: Після створення смарт-контракт розгортається в мережі блокчейн. Це зробить його активним у мережі, і тепер звідси можна буде виконувати транзакції.

– Виконання: смарт-контракт розроблений для прослуховування подій. Ці події надходять із криптографічно захищених джерел, відомого як «Децентралізована мережа Oracle (DON)» або так званий оракул. Коли ці події спрацьовують, смарт-контракт активується та виконуються транзакції, такі як переказ коштів.

– Запис: Результат кожної транзакції смарт-контракту записується в мережі блокчейн. Мережа блокчейн перевіряє транзакцію та реєструє завершені дані. Ці дані доступні всім, хто має доступ до транзакції, і вони не є оборотними.

1.2 Технічна реалізація смарт-контрактів

Смарт-контракти не можуть функціонувати самостійно, вони потребують віртуальної машини – це спеціалізоване програмне середовище, призначене для виконання смарт-контрактів. Віртуальна машина забезпечує безпечне, детерміноване та ізольоване виконання.

Різні блокчейн-платформи використовують власні віртуальні машини, адаптовані до їхньої архітектури. Провідники прикладами є Ethereum Virtual Machine (EVM) і Solana Virtual Machine (SVM), які домінують завдяки своїм технічним характеристикам і підтримці розробників.

Віртуальна машина Ethereum (EVM) – це машина Тюрінга, що означає, що вона може виконувати будь-які обчислення за наявності достатньої кількості

ресурсів. Здатність EVM виконувати будь-які обчислення робить її гнучкою для різних застосувань, від простих передач токенів до складних децентралізованих додатків (dApps). Смарт-контракти, написані мовами високого рівня, компілюються в байт-код EVM, який потім виконує EVM. Цей байт-код є узгодженим на всіх вузлах Ethereum, що забезпечує рівномірне виконання смарт-контрактів по всій мережі. EVM використовує газову систему для вимірювання обчислювальної роботи, необхідної для виконання транзакцій та смарт-контрактів. Плата за газ запобігає нескінченним циклам та надмірному споживанню ресурсів, підтримуючи стабільність та безпеку мережі. EVM ізолює виконання смарт-контрактів від блокчейну Ethereum, гарантуючи, що шкідливі або помилкові контракти не вплинуть на всю мережу [3].

Віртуальна машина Solana (SVM) є частиною блокчейну Solana, відомого своєю високою пропускною здатністю та низькою затримкою. SVM була розроблена для максимізації продуктивності та ефективності, підтримуючи мету Solana щодо масштабування блокчейн-транзакцій до виняткових рівнів. SVM використовує формат байт-коду фільтра пакетів Berkeley (BPF). Спочатку BPF був розроблений для фільтрації мережевих пакетів в Unix-подібних операційних системах. Смарт-контракти та програми, написані для Solana, зазвичай пишуться мовами програмування високого рівня, які компілюються в байт-код BPF, а потім виконується віртуальною машиною Solana. Він може обробляти тисячі транзакцій за секунду, що робить його одним із найшвидших блокчейнів. Середовище виконання Solana, Sealevel , дозволяє паралельне виконання смарт-контрактів, на відміну від моделі послідовного виконання EVM. Такий вибір підвищує швидкість обробки смарт-контрактів.

Смарт-контракти створюються з використанням спеціалізованих мов програмування, вибір яких залежить від блокчейн-платформи. Нижче наведено найпопулярніші мови для розробки смарт-контрактів:

– Solidity – об’єктно-орієнтована мова програмування для EVM-сумісних мереж, зокрема Ethereum, що підтримує складні контракти, але через гнучкість схильна до вразливостей, як реентрантність.

– Vyper – альтернативна Solidity для Ethereum, орієнтована на простоту та безпеку, з обмеженим синтаксисом для зменшення помилок.

– Rust – високопродуктивна мова, яка використовується в Solana, Near Protocol та Polkadot для ефективних і безпечних контрактів.

1.3 Життєвий цикл смарт-контрактів та управління безпекою

Смарт-контракти в екосистемі блокчейну проходять чотири основні етапи життєвого циклу, кожен із яких впливає на їхню функціональність і безпеку [4]:

– Фаза створення включає ітеративні переговори щодо контракту та фазу впровадження. Спочатку сторони повинні домовитися про обсяг та цілі контракту. Безпека на цьому етапі критична, оскільки помилки в коді можуть призвести до вразливостей.

– Розгортання: Мережа комп'ютерів, відомих як вузли, виконує перевірку транзакцій у блокчейні. Майнери блокчейну знаходяться в цих місцях. Щоб запобігти переповненню екосистеми смарт-контрактами, майнерам необхідно компенсувати цю послугу невеликою платою. Мережа комп'ютерів, відомих як вузли, виконує перевірку транзакцій у блокчейні. Майнери блокчейну знаходяться в цих місцях. Щоб запобігти переповненню екосистеми смарт-контрактами, майнерам необхідно компенсувати цю послугу невеликою платою.

– Виконання: Контракти, записані в розподіленому реєстрі, зчитуються всіма вузлами, що беруть участь у мережі. Вузли автентифікації перевіряють цілісність смарт-контракту, тоді як код виконується механізмом інтерференції (або компілятором) смарт-контракту. Коли вхідні дані для виконання від однієї сторони надходять у вигляді монет (зобов'язання щодо продуктів через монети), механізм

інтерференції генерує транзакцію, що запускається за дотриманим критерієм. Виконання смарт-контракту генерує новий набір транзакцій та новий стан контракту. Відкриття та нові дані про стан додаються до розподіленого реєстру та перевіряються за допомогою процедури консенсусу.

– Фіналізувати: Після виконання смарт-контракту отримані транзакції та оновлена інформація про стан записуються в розподіленому реєстрі та перевіряються за допомогою процедури консенсусу. Заставлені цифрові активи передаються (активи розморожуються), і підписується контракт для підтвердження всіх транзакцій.

1.4 Сфери застосування смарт-контрактів у фінансах, технологіях та традиційних галузях

Розробка смарт-контрактів та їх безпека – це наступний крок у цифровізації бізнес-ландшафту. Оскільки повністю онлайн-партнерство вже стало стандартом, розумні контракти відіграють ключову роль у підвищенні прозорості та надійності транзакцій для всіх залучених сторін. Хоча організаціям може знадобитися деякий час, щоб реструктуризувати свої процеси навколо розумних контрактів, очікувана довгострокова віддача від використання цієї технології, здається, не має аналогів у жодній іншій технології, особливо у фінансах, страхуванні, нерухомості та охороні здоров'я.

Смарт-контракти забезпечують безпечний доступ до медичних записів, щоб будь-який медичний працівник з будь-якої уповноваженої організації охорони здоров'я міг безпечно отримати доступ до медичних записів будь-якого пацієнта. MediLedger використовує смарт-контракти для відстеження фармацевтичних ланцюгів поставок. Вони автоматизують управління клінічними випробуваннями, перевіряючи критерії учасників, зберігаючи дані в незмінному реєстрі та автоматизуючи виплати [5].

Смарт-контракти оптимізують відстеження ланцюга поставок зберігаючи автентичність продуктів. Проект IBM Food Trust використовує блокчейн для відстеження харчових продуктів, записуючи транзакції на кожному етапі. В наслідок чого смарт-контракти спрощують дотримання регуляторних вимог, безпеку фармацевтичних і харчових продуктів.

Смарт-контракти усувають неоднозначність традиційних контрактів, автоматизуючи платежі та перевірку завдань. Наприклад, використовуючи камери з підтримкою комп'ютерного зору, система може виявляти, коли постачальник постачає матеріали, та автоматично переказувати кошти цьому постачальнику. Це значно зменшує витрати на посередників, звільняє працівників від деяких ручних завдань і прискорює обробку рахунків-фактур.

Наразі реєстрація прав власності на землю часто є ризикованою, фрагментованою та громіздкою. Цей процес передбачає вивчення агентами з нерухомості купи документів щодо підтверджень іпотеки, права власності на нерухомість, сертифікатів, що стосуються будівництва, та угод між покупцем і продавцем. Ці ручні процеси часто є причиною зміни документів і, як наслідок, шахрайства. Смарт-контракти спрощують реєстрацію прав власності, замінюючи паперові документи токенами блокчейну. Платформа Upland токенізує нерухомість, зберігаючи дані про власників і угоди. Перевірка через відкриті ключі унеможливлює шахрайство, оскільки зловмисникам потрібен доступ до всіх копій реєстру.

Протягом тривалого часу медіаіндустрія боролася зі складнощами, пов'язаними з обробкою прав власності та розподілом роялті. Музика, фільми та інші активи можуть бути токенізовані в блокчейні, а це означає, що кожен, хто бере участь у мережі, може точно бачити, хто і коли заявив права інтелектуальної власності на певний актив. Крім того, наприклад, у музичній індустрії розрахунок та розподіл роялті можуть бути повністю автоматизовані, що гарантує, що артисти

завжди отримуватимуть справедливу та своєчасну оплату праці за допомогою смарт-контрактів [6].

Деякі компанії почали інтегрувати смарт-контракти в бізнес процес. Вони додають цінність традиційним системам, зменшуючи адміністративні витрати, прискорюючи транзакції та мінімізуючи ризики шахрайства.

Sonoco та IBM працюють над зменшенням проблем із транспортуванням життєво важливих ліків шляхом підвищення прозорості ланцюга поставок. Pharma Portal, відстежує фармацевтичні препарати з контрольованою температурою через ланцюг поставок, щоб надавати достовірні, надійні та точні дані від кількох сторін.

The Home Depot використовує смарт-контракти на блокчейні для швидкого вирішення суперечок з постачальниками. Завдяки комунікації в режимі реального часу та підвищеній прозорості ланцюга поставок, смарт-контракти автоматизують перевірку умов поставок, забезпечуючи доступність даних у реальному часі.

Мережа торговельного фінансування we.trade, від компанії IBM Blockchain, створює екосистему довіри для глобальної торгівлі. Смарт-контракти стандартизують правила торгівлі, зменшуючи ризики та спрощуючи процеси для компаній і банків. В наслідок чого проєкт скоротив транзакційні витрати на 20%.

DocuSign інтегрує смарт-контракти у свою платформу автоматизуючи складні робочі процеси, такі як умовне підписання, звільнення платежів або виконання певних дій за виконання певних умов.

Maersk у партнерстві з IBM створили TradeLens, платформу на основі блокчейну, яка використовує смарт-контракти та цифрові підписи для відстеження товарів по всьому ланцюжку поставок. TradeLens створює захищений від несанкціонованого доступу спільний реєстр усіх транзакцій у ланцюжку поставок. Це дозволяє Maersk та її партнерам відстежувати товари в режимі реального часу, зменшуючи затримки та помилки [7].

1.5 Класифікація вразливостей смарт-контрактів

Важливою складовою за 2024 рік 1,42 мільярда доларів США втрачено внаслідок 149 задокументованих інцидентів. Найбільшими векторами атак (за частотою, загальними втратами) [8]:

– вразливості контролю доступу: збитки у розмірі 953,2 млн доларів США.
– логічні помилки: збитки у розмірі 63,8 млн доларів.
– атаки з повторним входом: збитки у розмірі 35,7 млн доларів.
– атаки на миттєві позики: збитки у розмірі 33,8 млн доларів.
– відсутність перевірки вхідних даних: збитки у розмірі 14,6 млн доларів США.
– маніпуляція ціновим оракулом: збитки у розмірі 8,8 млн доларів.
– неперевірені зовнішні виклики: збитки у розмірі 550,7 тис. доларів США.

Вразливості контролю доступу – недоліки в контролі доступу дозволяють неавторизованим користувачам отримувати доступ до даних чи функцій контракту або змінювати їх. Ці вразливості виникають, коли код не забезпечує належної перевірки дозволів, що потенційно призводить до серйозних порушень безпеки.

Маніпуляція ціновим оракулом використовує вразливості в тому, як смарт-контракти отримують зовнішні дані. Змінюючи або контролюючи канали оракула, зловмисники можуть впливати на логіку контракту, що призводить до фінансових втрат або нестабільності системи.

Логічні помилки або вразливості бізнес-логіки виникають, коли поведінка контракту відхиляється від його передбачуваної функціональності. Прикладами є неправильний розподіл винагороди, проблеми з карбуванням токенів або недосконала логіка кредитування/позичання.

Відсутність перевірки вхідних даних може призвести до вразливостей, де зловмисник може маніпулювати контрактом, надаючи шкідливі або неочікувані вхідні дані, потенційно порушуючи логіку або спричиняючи неочікувану поведінку.

Атаки з повторним входом або реентрації використовують можливість повторного входу в вразливу функцію до завершення її виконання. Це може призвести до повторних змін стану, що часто призводить до виснаження коштів контракту або порушення логіки.

Неперевірені зовнішні виклики – це нездатність перевірити успішність викликів зовнішніх функцій може призвести до небажаних наслідків. Коли викликаний контракт завершується невдачею, викликаний контракт може продовжитися неправильно, що ризикує цілісністю та функціональністю.

Блискавичні атаки на позики або флеш-кредити, хоча й корисні, можуть бути використані для маніпулювання протоколами шляхом виконання кількох дій в одній транзакції. Ці атаки часто призводять до виснаження ліквідності, зміни цін або використання бізнес-логіки.

Переповнення та недоповнення цілих чисел – це арифметичні помилки, що виникають через перевищення лімітів цілих чисел фіксованого розміру, можуть призвести до серйозних вразливостей, таких як неправильні обчислення або крадіжка токенів. Беззнакові цілі числа переходять у режим недоповнення, тоді як знакові цілі числа перемикаються між крайнощами.

Незахищена випадковість виникає через детерміновану природу блокчейн-мереж, створення безпечної випадковості є складним завданням. Передбачувана або маніпульована випадковість може призвести до експлуатації в лотереях, розподілі токенів або інших функціональних можливостях, що залежать від випадковості.

DoS-атаки використовуються для виснаження ресурсів контракту, роблячи його нефункціональним. Прикладами є надмірне споживання газу в циклах або виклики функцій, призначені для порушення нормальної роботи контракту.

1.6 Наслідки та статистика компрометації смарт-контрактів

Наведено аналіз трьох значних атак, які демонструють необхідність аналізувати та тестувати смарт-контракти регулярно.

У 2016 році хакер здійснив атаку на The DAO, децентралізовану автономну організації на Ethereum, використавши вразливість реентрантності в смарт-контракті. Зловмисник викрав 3,6 мільйонів ETH [9].

Механізм атаки полягав у повторному виклику функції withdraw(). Хакер розгорнув смарт-контракт, який виступає в ролі «інвестора» та вносить певну кількість ETH на DAO. При виклику функції withdraw() контракт надсилав ETH на адресу хакера, але не оновлював баланс до завершення транзакції. Зловмисний контракт використовував функцію fallback, яка повторно викликала withdraw(), створюючи цикл на рис. 1.1.

Наслідком даної атаки призвело до хардфорку Ethereum, розділивши мережу Ethereum і Ethereum Classic.

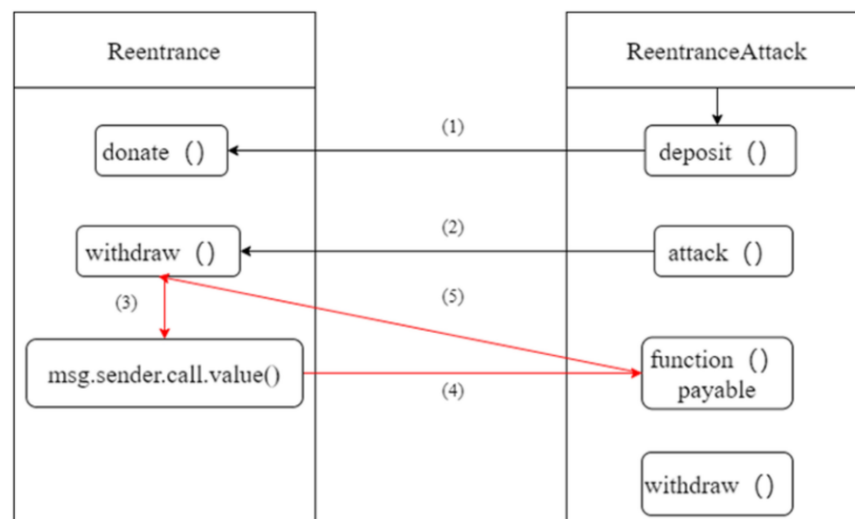

Рисунок 1.1 – Схема експлойта Reentrance

2 лютого 2022 року хакер розпочав атаку спрямовану на обхід процесу перевірки мосту Wormhole на Solana та видобувши 120000 Wormhole ETH (wETH). Вразливість полягала в застарілій функції load_current_index у процесі перевірки підписів (verify_signatures). Дана функція не перевіряла, чи введений "sysvar account" насправді є справжнім системним обліковим записом (system sysvar), дозволяючи зловмиснику підробити його та обійти перевірку (рис. 1.2) [10].

```
pub fn load_current_index(data: &[u8]) -> u16 {
    let mut instr_fixed_data = [0u8; 2];
    let len = data.len();
    instr_fixed_data.copy_from_slice(&data[len - 2..len]);
    u16::from_le_bytes(instr_fixed_data)
}

/// Load the current `Instruction`'s index in the currently executing
/// `Transaction`
pub fn load_current_index_checked(
    instruction_sysvar_account_info: &AccountInfo,
) -> Result<u16, ProgramError> {
    if !check_id(instruction_sysvar_account_info.key) {
        return Err(ProgramError::UnsupportedSysvar);
    }

    let instruction_sysvar = instruction_sysvar_account_info.try_borrow_data()?;
    let mut instr_fixed_data = [0u8; 2];
    let len = instruction_sysvar.len();
    instr_fixed_data.copy_from_slice(&instruction_sysvar[len - 2..len]);
    Ok(u16::from_le_bytes(instr_fixed_data))
}
```

Рисунок 1.2 – Схема вразливості в коді Wormhole

10 серпня 2021 року було здійснено злом мережі Poly Network, в результаті чого постраждало 58 активів у 11 блокчейнах. Збитки оцінюються понад 600 мільйонів доларів. Атака стала можливою через компроментацію приватних ключів релейного ланцюжка. Зловмисники виконували підроблені крос-чейн транзакції, маніпулюючи кількістю активів у цільовому ланцюзі. В наслідок чого Poly Network повернула частину коштів. Атака показала в необхідності захищати ключі та проводи аудит крос-чейн протоколів [11].

1.7 Моделі загроз та сучасний підхід до безпеки смарт-контрактів

Моделювання загроз у контексті децентралізованих додатків (DApps) включає детальний аналіз ризиків, пов'язаних із архітектурою додатка, потоками даних, взаємодією з користувачами та базовими технологіями, таких як блокчейн та смарт-контракти.

На відміну від централізованої інфраструктури Web2, Web3 побудований на децентралізованих мережах, що вимагає зосередження уваги на безпеці смарт-контрактів, механізмів консенсусу та однорангових мереж.

Після розгортання смарт-контракту неможливо змінити, що робить попереднє тестування, зокрема статичний аналіз та формальна верифікація критично важливим.

Криптографічна безпека відіграє важливу роль, оскільки втрата чи крадіжка приватних ключів може призвести до втрати активів. Хоча децентралізація зменшує ризик DDoS-атак, менш захищені вузли залишаються вразливими до цільових атак. Фронтенд DApps може становити загрозу пов'язану з фішинговими атаками, шкідливими розширеннями браузера чи недоліком інтерфейсу внаслідок чого відбуваєтся компроментація користувачів [12].

Крім того, прозорість блокчейну створює ризики для конфіденційності, адже транзакції можна відстежити, а аналіз мережевої активності може розкрити дані користувачів. Регуляторні зміни в сфері криптовалют додають ще один рівень складності, вимагаючи адаптації моделей загроз до нових вимог.

Оцінка потенційного впливу та ймовірність кожної загрози, визначають їх пріоритетність залежно від їхньої серйозності. Розробка та впровадження стратегій для зменшення виявлених ризиків, таких як аудит смарт-контрактів, безпечні методи кодування та навчання користувачів зможуть зробити ваш смарт-контракт більш безпечнішим.

Висновки за розділом 1

У першому розділі дипломної роботи проведено аналіз сутності смарт-контрактів, їх функціонування, технічної реалізації, середовищ виконання та мов програмування. Визначено, що життєвий цикл смарт-контрактів, який включає етапи створення, розгортання, виконання та фіналізації, є важливим для розробки методів тестування, особливо з огляду на незмінність коду після розгортання. Приклади реальних випадків компрометації підтверджують актуальність розробки ефективного методу тестування, які будуть представлені у наступних розділах. Виявлено основні вектори атак, що формують основу для створення методологічного підходу до тестування.

Встановлено широкий спектр застосування смарт-контрактів у різних сферах, а досвід їх інтеграції у бізнес-моделі провідних компаній демонструє прозорість та автоматизацію процесів.

Отже, незважаючи на значний трансформаційний потенціал смарт-контрактів для різних секторів економіки, їх успішне впровадження вимагає надійного методу тестування та аналізу коду з урахуванням усіх потенційних ризиків. Належне моделювання загроз у контексті децентралізованих додатків, систематичний аудит та сучасні методи тестування є важливими складовими для мінімізації вразливостей та забезпечення надійного захисту цифрових активів.

РОЗДІЛ 2
СПЕЦІАЛІЗОВАНІ ФРЕЙМВОРКИ ДЛЯ ЗМЕНШЕННЯ РІВНЯ ВРАЗЛИВОСТЕЙ СМАРТ-КОНТРАКТІВ

2.1 Огляд сучасних фреймворків безпеки смарт-контрактів і аудиту

Розглянуті в попередньому розділі типові вразливості та сценарії атак демонструють, що ручний огляд коду не завжди є достатньо надійним інструментом для забезпечення комплексного захисту смарт-контрактів. За даними ChainSecurity, приблизно 64% критичних вразливостей у смарт-контрактах можна було б виявити за допомогою автоматизованих інструментів, тоді як ручний аудит дозволяє знайти лише близько 48% таких вразливостей.

Фреймворк безпеки смарт-контрактів можна визначити як набір інструментів, методологій та процедур, спрямованих на виявлення, оцінку та усунення вразливостей у коді смарт-контрактів. В даний момент часу еволюція фреймворків пройшла шлях від базових сканерів до комплексних аналітичних платформ, що поєднують різні методи перевірки.

Сучасні фреймворки безпеки смарт-контрактів можна класифікувати за методом аналізу та етапом застосування в життєвому циклі.

З точки зору методів аналізу, фреймворки поділяються на декілька категорій:

– Фреймворки статичного аналізу (SAST) – перевіряють код без його виконання.

– Фреймворки динамічного аналізу (DAST) – тестують контракти у середовищі виконання.

– Інструменти формальної верифікації – математично доводять коректність контрактів.

Наступним важливим аспектом класифікації є етап життєвого циклу смарт-контрактів, для якого призначений конкретний фреймворк. Як показано на рис. 2.1, кожен етап розробки має свої специфічні потреби в інструментах безпеки. Особливу увагу варто приділити повноцикловим рішенням, які забезпечують комплексний підхід до захисту протягом усього процесу створення та експлуатації смарт-контракту.

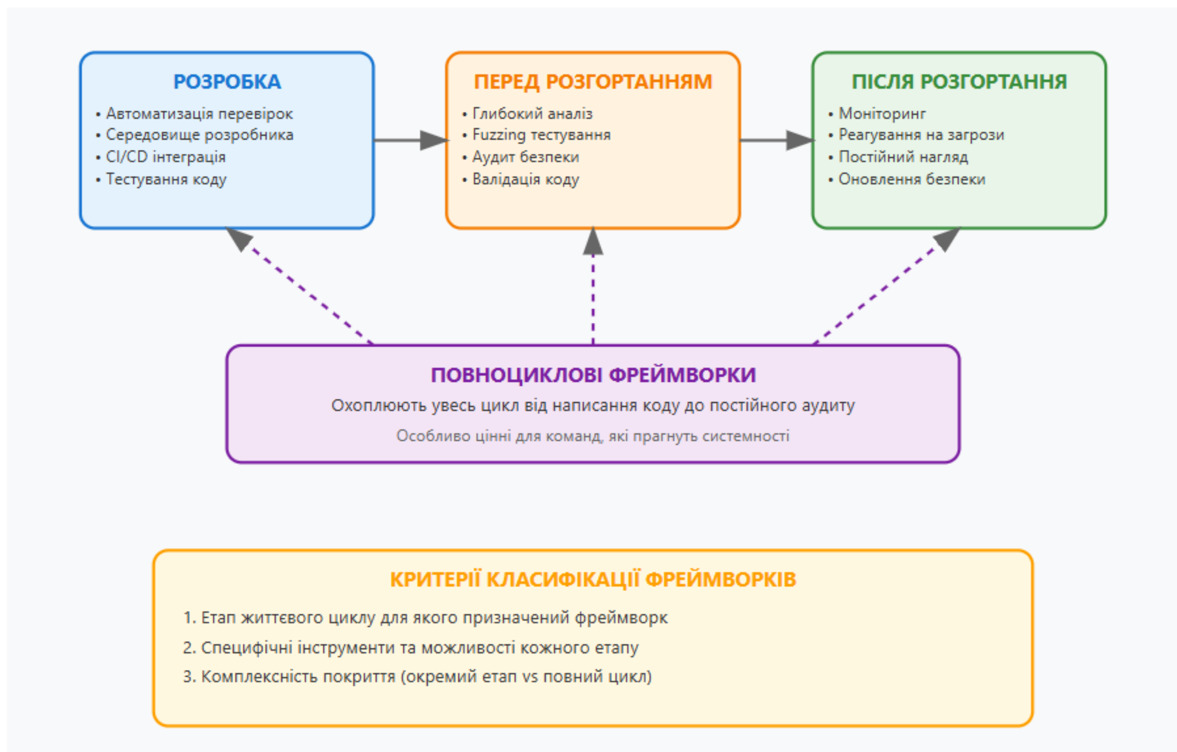

Рисунок 2.1 – Етапи життєвого циклу смарт-контракту для підвищення безпеки

Експерти в галузі blockchain-безпеки наголошують на тому, що для мінімізації ризиків та надійного захисту необхідно використовувати фреймворки, які скорочують час розробки через готові компоненти та шаблони, пропонують стандартизовані рішення для зниження ймовірності людських помилок, підтримуються активними спільнотами розробників, відповідають міжнародним стандартам та регуляторним вимогам та забезпечують комплексний підхід до тестування безпеки [13].

За даними дослідження ConsenSys, інтеграція автоматизованих інструментів безпеки у процес безперервної інтеграції знижує кількість вразливостей на 78% і скорочує час на їх виявлення на 65%. Це забезпечує виявлення помилок одразу після внесення змін у кодову базу, значно знижуючи ризик впровадження небезпечних оновлень у продуктивне середовище.

Більшість сучасних фреймворків підтримують написання модульних та інтеграційних тестів, емуляцію сценаріїв атак, фаззінг-тестування для виявлення непередбачуваних поведінок, аналіз витрат газу, символьне виконання для аналізу всіх можливих шляхів виконання.

Сучасні фреймворки дозволяють формалізувати вимоги до безпеки, впровадити політики перевірки коду та застосувати перевірені шаблони проєктування. Згідно з аналітикою OpenZeppelin, використання стандартизованих бібліотек і шаблонів знижує кількість критичних вразливостей на 82% порівняно з проєктами, що використовують власні рішення.

Окрему увагу слід приділити тому, що більшість фреймворків інтегруються з популярними мовами та інструментами розробки смарт-контрактів, зокрема Solidity, Vyper, Hardhat, Foundry, Truffle та OpenZeppelin. Завдяки цій екосистемній інтеграції вони не лише полегшують тестування та аудит, а й сприяють поширенню кращих практик серед розробників. Статистика GitHub демонструє, що проєкти, які використовують стандартні фреймворки безпеки, мають на 67% менше критичних помилок, ніж проєкти без таких інструментів.

Сучасні фреймворки також забезпечують інтеграцію з аудитними платформами, зокрема Code4rena або Sherlock, де код може бути перевірений не лише автоматичними інструментами, а й спільнотою незалежних експертів. Економічна ефективність такого підходу підтверджується даними Immunefi, згідно з якими середня вартість усунення вразливості на етапі аудиту становить приблизно 20000 доларів, тоді як після розгортання ця сума може зрости до кількох мільйонів.

Не менш важливою перевагою фреймворків є можливість налаштування симуляцій атак, що дозволяє виявляти складні логічні помилки, які неможливо зафіксувати класичними сканерами. Дослідження Trail of Bits показують, що такі методи здатні виявити до 91% складних логічних вразливостей, які залишаються непоміченими при статичному аналізу коду.

2.2 OWASP SCSVS як система верифікації безпеки смарт-контрактів

Стандарт перевірки безпеки смарт-контрактів (SCSVS) – це перелік конкретних вимог безпеки або тестів для смарт-контрактів, написаних переважно на Solidity та розгорнутих на блокчейнах на основі EVM. Ці вимоги призначені для використання архітекторами, розробниками, тестувальниками, фахівцями з безпеки, постачальниками інструментів та споживачами для визначення, створення, тестування та перевірки безпечних смарт-контрактів, децентралізованих додатків (dApps) та протоколів блокчейну. Стандарт сприяє найкращим практикам забезпечення безпеки та цілісності смарт-контрактів та децентралізованих фінансових (DeFi) систем [14].

Стандарт поділено на різні групи елементів керування, позначені як SCSVS-XXXXX, які представляють найважливіші області поверхні атаки смарт-контрактів:

– SCSVS-ARCH : Безпечна архітектура, принципи проектування та методи моделювання загроз для блокчейн-систем.

– SCSVS-CODE : Політики та процедури безпечної розробки, тестування та розгортання смарт-контрактів.

– SCSVS-GOV : Безпека бізнес-логіки та економічних механізмів для запобігання маніпуляціям або експлуатації.

– SCSVS-AUTH : Надійні заходи контролю доступу та автентифікації для блокчейн-застосунків.

– SCSVS-COMM : Безпечний зв'язок та взаємодія між контрактами, користувачами та зовнішніми системами.

– SCSVS-CRYPTO : Найкращі практики криптографічної реалізації в блокчейн-додатках.

– SCSVS-ORACLE : Забезпечує безпеку та стійкість до атак арифметичних та логічних операцій.

– SCSVS-BLOCK : Захист від атак типу «відмова в обслуговуванні» (DoS) та ризиків виснаження ресурсів.

– SCSVS-BRIDGE : Ефективні методи управління даними та станом для підтримки цілісності блокчейну.

– SCSVS-DEFI : Оптимізація та моніторинг використання газу для уникнення неефективності та обмежень.

– SCSVS-COMP : Рекомендації щодо безпеки для окремих компонентів, адаптовані до блокчейн-застосунків.

SCSVS класифікує перевірку безпеки на три окремі рівні, кожен з яких спрямований на різні рівні гарантування безпеки під час розробки та розгортання смарт-контрактів:

– SCSVS Рівень 1 – Базова безпека : Цей рівень розроблений для смарт-контрактів з нижчими ризиками безпеки. Він зосереджений на фундаментальних засобах контролю безпеки, забезпечуючи базовий захист для будь-якого децентралізованого застосунку.

– SCSVS Рівень 2 – Помірний рівень безпеки : Ідеально підходить для смарт-контрактів, які обробляють конфіденційні дані, фінансові транзакції або є частиною екосистеми DeFi. Рівень 2 забезпечує більш збалансований підхід до безпеки, усуваючи поширені вразливості, такі як атаки повторного входу, неефективність використання газу та слабкі місця в контролі доступу.

– SCSVS Рівень 3 – Висока гарантія безпеки : Цей рівень розроблений для критично важливих смарт-контрактів, де на кону значні фінансові активи, управління або транзакції з високою вартістю. Рівень 3 забезпечує розширені заходи безпеки та охоплює розширений захист, такий як формальна перевірка, гаманці з кількома підписами та децентралізоване управління.

Кожен рівень SCSVS надає детальний набір вимог безпеки, зіставляючи їх з основними функціями безпеки та практиками, необхідними для створення безпечних смарт-контрактів. Незалежно від того, чи йдеться про розробку, аудит чи розгортання смарт-контрактів, SCSVS пропонує чітку дорожню карту, яка допоможе командам на кожному етапі.

Хоча SCSVS пропонує надійну основу для покращення безпеки смарт-контрактів, він не може сам по собі гарантувати повну безпеку. Його слід вважати базовим стандартом безпеки, додаючи додаткові захисні заходи за необхідності для усунення конкретних вразливостей та загроз, що розвиваються, у децентралізованих середовищах. Під час використання SCSVS важливо враховувати такі припущення:

– SCSVS не замінює стандартні практики безпечної розробки, такі як безпечне кодування або дотримання життєвого циклу безпечної розробки програмного забезпечення (SSDLC) . Він має доповнювати ці практики, враховуючи конкретні потреби безпеки для смарт-контрактів на основі EVM та децентралізованих додатків.

– SCSVS не призначений для заміни комплексного моделювання загроз або оглядів безпеки . Він слугує спеціалізованим посібником, який допомагає виявляти та пом'якшувати вразливості, унікальні для смарт-контрактів. Використання SCSVS має покращити, а не замінити традиційні оцінки ризиків безпеки та тести на проникнення.

Стандарт OWASP Smart Contract Security Verification Standard (SCSVS) структуровано поділено на категорії, кожна з яких має унікальний ідентифікатор вимог і визначає стандарти безпеки смарт-контрактів.

Категорія SCSVS-ARCH-1, що має назву безпечні шаблони проектування, розподіляється на модульність та можливість модернізації, розподілення відповідальності. Метою контролю є переконатися, що смарт-контракти розроблені з урахуванням модульності, можливості оновлення та розділення відповідальностей, щоб забезпечити безпечну роботу, оновлення та обслуговування. Контракти повинні бути розроблені таким чином, щоб мінімізувати ризики безпеки, пов'язані зі складними оновленнями, передачею привілеїв та неправильним управлінням залежностями.

Метою контролю категорії SCSVS-ARCH-2, що має назву шаблони проксі-серверів є забезпечення безпечного впровадження та належне керування шаблонами проксі-серверів та механізмами оновлення, щоб зменшити ризики під час оновлення контрактів, їх припинення дії та переходів між версіями контрактів.

Метою контролю категорії SCSVS-ARCH-3 є виявлення, оцінювання та пом'якшення загрози безпеці для систем смарт-контрактів шляхом впровадження ретельного процесу моделювання загроз, забезпечуючи мінімізацію ризиків та наявність захисту для критично важливих функцій контракту.

Метою контролю категорії SCSVS-CODE-1 є встановлення та забезпечити дотримання безпечних стандартів кодування та процесів перевірки, щоб мінімізувати вразливості та забезпечити дотримання найкращих практик протягом усього життєвого циклу розробки.

Метою контролю категорії SCSVS-CODE-2 є забезпечення ясності та зручності підтримки коду завдяки ретельній документації, логічній структурі та дотриманню послідовних стандартів кодування, що полегшить розуміння та модифікацію розробниками.

Метою контролю категорії SCSVS-CODE-3 є забезпечення комплексного тестування смарт-контрактів, що охоплює модульні тести, інтеграційні тести та тести, пов'язані з безпекою, для виявлення вразливостей та підтримки якості коду протягом усієї розробки.

Метою контролю категорії SCSVS-GOV-1 є забезпечити розробку та впровадження економічних моделей, включаючи структури стимулювання та токеноміку, таким чином, щоб забезпечити цінність та стимулювати належну поведінку в екосистемі. Контракти повинні враховувати коливання вартості токенів та уникати створення можливостей для експлуатації.

Метою контролю категорії SCSVS-GOV-2 є забезпечити безпечну реалізацію токенів, що використовуються в екосистемі смарт-контрактів, включаючи такі аспекти, як управління цінністю, механізми перебазування та системи винагород. Контракти повинні запобігати вразливостям токенів, таким як подвійні витрати, неправильні винагороди та неналежне оброблення комісій.

Метою контролю категорії SCSVS-GOV-3 є забезпечити захист потоку транзакцій та логічної цілісності смарт-контракту від атак повторного входу та логічних недоліків. Контракти повинні реалізовувати надійні структури контролю та шаблони безпеки для запобігання повторному входу, обробки складних потоків та забезпечення безпечних та симетричних переходів станів.

Метою контролю категорії SCSVS-AUTH-1 є впровадження контролю доступу на основі ролей для керування дозволами та забезпечення доступу до певних функцій лише авторизованими користувачами. Це включає перевірку ідентифікаційних даних, застосування принципу найменших привілеїв та забезпечення належного контролю доступу.

Метою контролю категорії SCSVS-AUTH-2 є впровадження безпечних механізмів авторизації для захисту критично важливих функцій та конфіденційних операцій, гарантуючи, що лише уповноважені особи можуть виконувати ці дії.

Метою контролю категорії SCSVS-AUTH-3 є впровадження децентралізованих рішень для ідентифікації, щоб забезпечити безпечну та надійну перевірку та управління ідентифікацією, зберігаючи при цьому конфіденційність користувачів.

Метою контролю категорії SCSVS-COMM-1 є забезпечити безпеку всієї взаємодії між контрактами, мінімізувати ризики, пов'язані із зовнішніми викликами, підтримуйте мінімальну довірену поверхню та належним чином обробку помилок.

Метою контролю категорії SCSVS-COMM-2 є забезпечення безпечних, надійні та захищені від несанкціонованого доступу канали даних, зберігаючи при цьому цілісність даних та належним чином обробляючи збої.

Метою контролю категорії SCSVS-COMM-3 є забезпечення безпечну обробку зовнішніх викликів та атомарних обмінів під час міжланцюгових взаємодій для підтримки операційної надійності та запобігання шахрайству.

Метою контролю категорії SCSVS-COMM-4 є забезпечення безпеки крос-чейн транзакцій, впровадивши надійні механізми валідації та верифікації для запобігання шахрайству та збереження цілісності даних.

Метою контролю категорії SCSVS-CRYPTO-1 є забезпечення безпечного оброблення та зберігання закритих ключів і впровадьте надійні процеси перевірки підписів для запобігання несанкціонованому доступу та діям.

Метою контролю категорії SCSVS-CRYPTO-2 є впровадження криптографічних методи, що забезпечують безпечну перевірку підписів та дотримання стандартів для збереження цілісності автентифікованих транзакцій.

Метою контролю категорії SCSVS-CRYPTO-3 є впровадження найкращих практик безпечної генерації випадкових чисел, щоб забезпечити непередбачуваність та стійкість до маніпуляцій.

Метою контролю категорії SCSVS-ORACLE-1 є впровадження безпечних арифметичні методи, щоб запобігти вразливостям переповнення та недоповнення, які можуть поставити під загрозу функціональність та безпеку контракту.

Метою контролю категорії SCSVS-ORACLE-2 – переконатися, що всі обчислення та логічні операції в межах смарт-контракту виконуються правильно, щоб зберегти цілісність даних та запобігти маніпуляціям.

Метою контролю категорії SCSVS-BLOCK-1 є забезпечення ефективності проектування контрактів та реалізації функцій у використанні газу для зменшення ризиків, пов'язаних з помилками, пов'язаними з нестачею газу, та пов'язаними з ними вразливостями.

Метою контролю категорії SCSVS-BLOCK-2 є впровадження стратегії захисту контрактів від атак на вичерпання ресурсів, які можуть призвести до сценаріїв DoS.

Метою контролю категорії SCSVS-BRIDGE-1 є забезпечити ефективну та безпечну обробку стану в межах смарт-контрактів, щоб запобігти пошкодженню даних та неочікуваній поведінці.

Метою контролю категорії SCSVS-BRIDGE-2 є забезпечити захист конфіденційних даних у контрактах та ефективне впровадження заходів конфіденційності.

Метою контролю категорії SCSVS-BRIDGE-3 є впровадження прозорих та безпечних методів реєстрації, щоб забезпечити відстеження та виявлення несанкціонованих змін.

Метою контролю категорії SCSVS-BRIDGE-4 є забезпечення цілісності, безпеки та доступності даних, що зберігаються в децентралізованих рішеннях для зберігання даних.

Метою контролю категорії SCSVS-DEFI-1 є забезпечення мінімізації споживання газу для сприяння економічно ефективному виконанню смарт-контрактів.

Метою контролю категорії SCSVS-DEFI-2 є ефективна розробка контрактів для підвищення продуктивності та зниження витрат на газ завдяки оптимальній архітектурі.

Метою контролю категорії SCSVS-COMP-1 є забезпечення безпечного впровадження та управління стандартами токенів для запобігання вразливостям.

Метою контролю категорії SCSVS-COMP-2 є впровадження найкращих практик для незамінних токенів, щоб захиститися від вразливостей.

Метою контролю категорії SCSVS-COMP-3 є забезпечення безпечного зберігання та управління активами в системах сховищ.

Метою контролю категорії SCSVS-COMP-4 є забезпечення безпечних механізмів стейкінгу для захисту активів користувачів.

2.3 Методології тестування вразливостей OWASP Smart Contract Testing Guide

OWASP Smart Contract Security Guide (SCSTG) – це посібник, який є результатом зусиль спільноти, спрямованих на створення всеосяжного ресурсу для розуміння, тестування та покращення безпеки смарт-контрактів. З кожним оновленням протоколу, інновацією L2 або новим стандартом виникають нові виклики як для розробників, так і для фахівців з безпеки. Мета посібника заповнити програлини, озброївши практичними методами та ідеями, які допоможуть орієнтуватися постійно мінливому ландшафті безпеки смарт-контрактів. Таксономія смарт-контрактів формує структурований фреймворк для класифікації контрактів на основі функціональності, ризиків і кращих практик (з акцентом на EVM-блокчейни та Solidity). До основних категорій смарт-контрактів належать [15]:

– Токен-контракту. Передбачають реалізацію замінних (ERC20), незамінних (ERC-721) або напівзамінних (ERC-1155) токенів. Типовими ризиками

можна вважати некоректну реалізацію стандартів токенів, вразливості переповнення або заниження чисел, а також відсутність належного контролю доступу до функцій випуску чи спалювання токенів.

– Контракти залучення коштів (краудфандинг) реалізують функції збору капіталу в обмін на токени чи послуги. Основними ризиками можна пов’язати шахрайство, недоліками механізмів повернення внесків. Зловмисники можуть привласнити активи, а технічні помилки – зашкодити гарантування і можливості безпечного повернення (рефанду) інвесторам.

– Контракти децентралізованого управління (Governance) підтримують механізми голосування і прийняття рішень у DAO та іншим організаціях. Ризики пов’язані з маніпуляцією голосування, дефектами логіки кворуму чи виконання голосувань із недостатньою прозорістю рішень в ончейні.

– DeFi-протоколи. Прикладами є автоматизовані маркет-мейкери (AMM, як Uniswap), платформи кредитування та позик (Aave, Compound), агрегатори доходності тощо. Найвищими ризиками можуть стати експлоатовані механізми ліквідації в кредитних протоколах та атаки через флеш-кредити при арбітражних можливостях.

– Оракл-контракти (Oracle) забезпечують отримання даних із зовнішніх джерел у блокчейн. Наприклад, Chainlink чи Tellor передають ціни активів або події зі світу в ончейн-контракти. Загрозами стають маніпуляції значеннями даних, затримку чи некоректну доставку інформації, а також централізацію елементів ораклу.

– Ескроу та платіжні контракти, які керують переказом активів між сторонами за заздалегідь визначеними правилами, піддаються ризику пов’язані з логічними помилками у розподілі коштів, відсутністю мультипідписних механізмів при спорах та експлуатацією вразливостей повернення або заморозки коштів.

– Контракти ідентифікації та доступу забезпечують верифікацію користувачів та управління привілеями та пов'язані з ризиком несанкціонованого підвищення привілеїв та неправильним обробленням чутливих даних користувачів.

– Біржові мости (Bridge) дозволяють переміщувати активи чи дані між різними блокчейнами, можуть бути вразливими до атак повторного відтворення або систем валідаторів.

– Логістика та управління ланцюгами постачань дозволяють відстежувати виробництва або поставки товарів та пов'язані з довірою до зовнішнім даних (цілісність), а також некоректною реалізацією процедур аудиту та журналювання.

– Контракти для ігор та метаверсу підтримують ончейн-володіння предметами (NFT) та внутрішньоігрові економіки на токенах. Головною загрозою стають експлойти у механіках торгів або аукціонах.

– Засоби безпеки та аудиту (Security Tools) служать проведенням аудиту для покращення безпеки смарт-контрактів. Неправильне налаштування контролю доступу, логічні помилки у тимчасових блокуваннях операцій стають величезними загрозами.

– Арбітражні та MEV-боти використовуються для оптимізації комісій та пріоритизації транзакцій. Типовими атаками можна вважати фронт-раннінг на користувачів і експлуатацію мережевого навантаження.

Для кожного з наведених вище категорій OWASP SCSTG пропонує відповідні методи тестування. Нижче наведено коротко значення кожного підходу, а саме що потрібно перевірити, щоб виявити вразливості.

Тестування архітектури, дизайну та моделювання загроз враховує детальний етап перевірки правильності архітектури рішень та потенційних загроз на етапі проєктування. Недоліки дизайну і неякісне моделювання загроз призводять до критичних помилок, які включають незахищене зберігання даних, невідлагоджена логіка критичних функцій, відсутність механізмів захисту від відомих атак як

реентрансія чи флеш-кредити. Метою тестування є виявлення прогалин в архітектурі й застосування принципів безпечного дизайну, а саме модульність, мінімальні привілеї, захист від відкату стану тощо.

Тестування політик, процедур та управління кодом оцінюється в якості процесу розробки, яке включає дотримання кращих практик кодування, організація рецензування коду, система контролю версій та обробка інцидентів. Метою є забезпечення належного управління проєктом, щоб вихідний код проходив перевірку, а всі зміни документувалися й контролювалися, не менш важливо мати процедуру відповіді на інциденти та розголошення вразливостей. Таким чином, тестування гарантує, що організаційні процеси не спричинять появи помилок або вразливостей у коді.

До тестування бізнес-логіки та економічної безпеки входить аналіз механізмів стимулювання, стабільності вартості токенів, а також захист від логічних помилок, які призводять до фінансових втрат. Наприклад, неправильна структура винагород може викликати помилкову поведінку користувачів або експлуатацію системи, а погане розмежування потоків платежів можуть спричинити вразливість реентранс. Метою даного підходу є перевірка логіки контракту, який відповідає його функціональному призначенню й стійку до економічних атак.

До тестування контролю доступу та автентифікації потрібно оцінити чи може зловмисник отримати несанкціонований доступ до критичних функцій. Перевірка, що кожна чутлива операція, як наприклад, емісія токенів, зняття коштів, оновлення параметрів обмежена вхідними правами. Зловмисник зможе скомпрометувати систему або вивести чужі активи при відсутності прав доступу або неправильних налаштуваннях. Метою є забезпечення строгого управління ролями й автентифікацією, щоб лише уповноважені адреси виконували критичні функції смарт-контракту.

Тест на принцип мінімальних привілеїв не передбачає надмірних привілеїв. Наприклад, замість tx.origin потрібно використовувати msg.sender для перевірки

прав доступу, щоб уникнути атаки через посередників. Метою є переконатися, що жодна функція не відкриває зайвого доступу і всі перевірки відправника правильні. Данний підхід запобігає виконанню функцій через проксі-контракти чи скрипти, які не повинні мати прав, та унеможливлює підняття привілеїв користувачем.

До тестування необроблених зовнішніх викликів включається перевірка взаємодій контракту з іншими контрактами чи адресами для правильної обробки результатів виклику. Якщо зовнішній виклик, як наприклад call чи delegatecall, не перевіряє свій результат, контракт може продовжити роботу з некоректним станом, внаслідок чого, це призводить до втрати коштів або порушення бізнес-логіки. Модульні тести виявляють ситуації, коли після виклику не здійснюється перевірка значення success чи результату функції. Мета – гарантувати, що у випадку збою зовнішньої транзакції контракт належно реагує (відкидає операцію або обробляє помилку). Таким чином, попереджаються вразливості на кшталт невиправного підключення або можливі повторні зняття коштів.

До криптографічних практик входить перевірка коректності використання криптографічних механізмів і генерації випадкових чисел. Наприклад, в контрактах, що потребують підписання транзакцій, перевіряють валідацію через ecrecover і відповідність стандартам (EIP-712). Метою підходу є застосування безпечних криптографічних функцій (надійні хеші, захищені шифри, стійкі до атаки рандомайзера) та не використовують застарілі чи вразливі бібліотеки.

До тестування арифметики та логіки включають коректність математичних обчислень та умов у коді контракту. Наприклад, використання захищених математичних операцій (SafeMath) або інших перевірок, щоб запобігти переповненню чисел. Метою є гарантія, що всі розрахунки, особливо пов’язані зі змінами балансу та конвертацією токенів, виконуються точно і без помилок.

Під час тестування на відмову в обслуговування (DoS) перевіряється, чи не містить контракт функцій з нерелізними циклами чи надто витратними операціями, що призводять до перевитрату газу і вихід за ліміт блоку, тому обов’язково потрібно

проаналізувати обробку помилок та використання гарантій (фолбеків). Метою є забезпечення контракту виконуватися за умов обмеженого газу і не перехід у неконтрольований стан у разі помилок. Наприклад, перевіряють механізми обробки відкатів, використання fallback-функцій та інші засоби протидії DoS.

Під час тестування управління даними та станом блокчейну перевіряють поводження з ончейн-даними, яке включає верифікацію того, що стан контракту оновлюється коректно, а події (логування) записуються та зчитуються без зміни даних. Не мало важливо аналіз механізмів приватності даних, які ще називають шифруванням або ж zk-протоколи. Головною метою є запобігання втрачення чи пошкодження даних, уникнення неефективного управління пам’ятю, як наприклад оптимізовані масиви або надмірне використання гаса для операцій з великими даними. Дані поради допоможуть у безперебійній роботі смарт-контрактів з даними навіть за складних умов.

За тестуванням використання газу, ефективності та обмежень повинно бути перевірка оптимальності витрату газу на операції контракту, аналізуючи чи контракт може виконуватися за прийнятних витрат на газ, чи не містить зайвих циклів та записів, які підвищують комісію транзакції. Наслідками можуть стати надмірні платежі за транзакції, вихід за ліміт газу, затримки в мережі. Метою тестування є зменшення витрат газу, роблячи транзакції стабільними та доступними для користувачів. Наприклад, перевірка, що важливі операції розбиваються на батчі, використання адекватної оцінки газу та за можливості використання рішення другого рівня для зниження навантаження.

Останнім підходом залишається тестування специфічних компонентів, в якому оцінюється коректність реалізації окремих складових системи – токенів (ERC-20/721), гаманців, пулів ліквідності, складних механізмів як стейкінг тощо. Кожен компонент виділяється по своїму та має іншу структуру. Наприклад, ERC20-токен повинен правильно вести облік загальної пропозиції та балансів, NFT – забезпечуватись неперервністю метаданих та унікальністю, а AMM пул – коректно

розраховувати коефіцієнти обміну й захищатися від маніпуляції скороченням (slippage). Притаманними загрозами можуть стати помилки в імплементації стандартів, невірні розрахунки або збої у спеціалізованих бібліотеках. Головна мета полягає в тому, що кожен компонент повинен працювати згідно зі специфікаціями.

2.4 Рекомендації та стандарти безпеки NIST IR8202, ISO 22739 та ISO 23455

Посилаючись на документ NIST IR 8202, блокчейн гарантує цілісність даних через криптографічний ланцюжок блоків і цифрові підписи транзакцій. У документі вказано, що блокчейн не усуває звичайних кіберзагроз (мережевих атак, DoS, тощо), тому необхідно застосувати перевірені практики безпеки з урахуванням особливостей DLT.

Основні положення безпеки блокчейну та смарт-контракту NIST IR 8202 зазначають, що блокчейн описується як “append-only” журнал – додані блоки та транзакції неможливо змінити без виявлення. Кожна транзакція підписується приватним ключем відправника, що дозволяє верифікувати її походження і запобігти підробці. NIST IR 8202 чітко вказує про безпечне зберігання приватних ключів, адже втрата чи викрадення ключа означає втрачені активи. Стандарт приводить приклад найбільших атак, які ми розглянули в попередньому розділі, що призводить до фінансових втрат, тому смарт-контракти повинні проходити повний аудит та тестування. В permissionless-блокчейнах застосовуються атаки 51% – коли зловмисник із понад 50% майнінг-потужності формує власний ланцюг. NIST рекомендує контролювати розподіл ресурсів мережі та збільшувати кількість вузлів для посилення стійкості. Для захисту блокчейн та смарт-контрактів NIST наполягає на адаптування сучасних фреймворків з специфічними заходами захисту DLT [16].

ISO 22739 – словник термінів для блокчейн і DLT. Він містить визначення, безпекові властивості такі як автентичність, цілісність та незмінність, які ми вже роглянули в попередньому розділу [17].

ISO/TR 23455 розглядає та аналізує безпеку смарт-контрактів та їхню взаємодію на DLT.

Код і стан смарт-контракту публікуються у розподіленому реєстрі, тому всі учасники мають однакову версію. Оновлення контракту вимагає консенсусу мережі, що гарантує повну прозорість і незмінність його стану, що знижує ризик підміни смарт-контракту однією стороною.

Стандарт рекомендує ідентифікувати власника і користувача смарт-контракту через їхні DLT-адреси і на цій підставі обмежувати доступ до методів контракту. Транзакції зі смарт-контракту мають підписуватися відправником для підтвердження легітимності операції. Для зовнішніх даних звіт радить використовувати мультипідписи та перевіряти кілька незалежних джерел, а також пов'язувати реальні ідентичності з ключами, як наприклад через PKI.

ISO/TR 23455 попереджає, що публічне розкриття всіх умов смарт-контракту підвищує ймовірність реалізації загрози витоку чутливої інформації. Для захисту конфіденційних даних рекомендується мінімізувати обсяг відкритих даних, як наприклад виконувати частину логіки оффчейн чи застосовувати приватні канали. Стандарт вказує на обов'язкове впровадження механізму оплати за виконання коду. Якщо ресурс виконання вичерпано, контракт автоматично зупиняється, що унеможливлює нескінченні цикли або зависання через помилки [18].

ISO/TR 23455 описує DLT-оракули – інтерфейси до зовнішніх джерел, впровадженням консенсусних механізмів, як наприклад кілька незалежних оракулів і мультипідписів гарантують, що дані достовірні.

2.5 Ethereum Security Best Practices: патерни та принципи безпечного кодування для Ethereum

Найкращі практики безпеки смарт-контрактів Ethereum, підтримувані ConsenSys Diligence, формують базову культуру безпеки для розробників Solidity.

Вони підкреслюють важливість інженерного підходу до кодування через незмінність смарт-контрактів і високу вартість помилок, закликаючи впроваджувати механізми реагування на вразливості, наприклад функцію паузи контракту.

Рекомендації містять приклади безпечних патернів для Solidity: використання require для перевірки умов, уникнення залежності від блокових міток часу, застосування бібліотек на кшталт SafeMath для запобігання переповненням, а також модифікаторів для контролю доступу. Документ детально класифікує основні типи атак (реентрантність, переповнення, DoS, залежність від часу, фронт-раннінг) і надає практичні поради щодо їх запобігання, що є основою для розробки тестових сценаріїв [19].

Для автоматизації аналізу коду та виявлення вразливостей рекомендується використовувати інструменти Slither, Mythril, Aderyn, які інтегруються у сучасні методи тестування. Важливу роль у підвищенні безпеки відіграють і програми баг-баунті, як-от Immunefi, що залучають спільноту до пошуку помилок у смарт-контрактах.

2.6 Solidity Style Guide як інструмент забезпечення стилістичної та структурної безпеки коду

У сфері розробки смарт-контрактів якість коду безпосередньо впливає на безпеку та функціональність системи. Solidity, як основна мова програмування для Ethereum Virtual Machine (EVM), вимагає зосередити увагу на стиль та структуру коду.

Solidity Style Guide представляє собою набір рекомендацій та правил, спрямованих на покращення читабельності, підтримуваності та безпеки коду. Дані правила не є обов’язковими у технічному плані, але їх дотримання значно знижує ймовірність введення потенційних вразливостей у код смарт-контрактів [20].

До запобігання помилок програмування входить форматування, яке допомагає виявляти синтаксичні помилки на ранніх стадіях, консистентне найменування змінних, функцій та структур даних, яке знижує ризик логічних помилок, стандартизація патернів коду, яке полегшує його рев’ю та тестування.

Підвищення читабельності та аудиту коду включає покроковий підхід до структурування контрактів, нормативи документування, які полегшують розуміння функціональності коду, стандартизація розміщення елементів коду (порядок функцій, змінних), яке спрощує навігацію для аудиторів.

Solidity Style Guide містить рекомендації щодо недопущення типових помилок у смарт-контрактах, заохочує використання перевірених практик та конструкцій.

Згідно з офіційною документацією, елементи в контракті повинні розташовуватися в наступному порядку:

– Pragma директиви
– Імпорти
– Інтерфейси
– Бібліотеки
– Контракти

Всередині контракту елементи рекомендується розміщувати в зазначеному порядку:

– Оголошення типів
– Змінні стану
– Події
– Модифікатори
– Конструктор
– Функції (згруповані за видимістю та мутабельністю).

Дане впорядкування не є випадковим – воно відображає залежності між елементами коду та послідовне знайомство з контрактом під час аудиту. Наприклад, розміщення модифікаторів перед функціями дозволяє аудитору спочатку ознайомитися з обмеженнями та умовами, які потім застосовуються до функцій.

Solidity Style Guide є одним із елементів комплексного підходу до забезпечення безпеки смарт-контрактів. Його ефективність значно підвищується при інтеграції з іншими інструментами та методами.

Style Guide може бути інтегрований у конвеєри безперервної інтеграції та розгортання (CI/CD) через автоматизовані лінтери та інструменти аналізу, який постійно контролює якість коду.

Посібник надає структуровану базу для проведення код-рев'ю, довзоляючи рецензентам фокусуватися на логіці та безпеці, а не стилістичних аспектах. Чек-лист для код-рев'ю включає:

– Відповідність найменування конвенціям Style Guide.
– Правильне упорядкування елементів контракту
– Належне документування функцій та змінних.
– Явне вказування видимості функцій та змінних.
– Дотримання принципу "checks-effects-interactions"

2.7 OpenZeppelin: бібліотеки безпечного коду та процеси аудиту

OpenZeppelin Contracts – це найпопулярніша бібліотека безпечних смарт-контрактів для Ethereum та інших блокчейнів. Вона містить перевірені часом шаблони та стандарти, які пройшли численні аудити та широко використовуються в екосистемі.

До основних переваг бібліотек OpenZeppelin відноситься перевірка коду, відповідність стандартам (ERC-20, ERC-721, ERC-1155), використання необхідних компонентів, легке успадкування та модифікація функціональності.

До основних компонентів бібліотеки можна віднести контракти токенів (ERC-20, ERC-721, ERC-777, ERC-1155), контролю доступу (Ownable, AccessControl, Roles), захист від відомих атак (ReentrancyGuard, Pausable), математичні операції (SafeMath, SignedSafeMath), криптографічні утиліти (ECDSA, MerkleProof), проксі та оновлення (Transparent, UUPS) [21].

Висновки за розділом 2

Отже, в розділу 2 проаналізовані спеціалізовані фреймворки, стандарти та методології, що сприяють у зменшенні вразливостей смарт-контрактів.

Проведений аналіз, демонструє, що сучасні підходи до безпеки охоплюють увесь життєвий цикл розробки – від написання коду до моніторингу, а використання автоматизованих інструментів дозволяє виявляти до 64% критичних вразливостей, що значно перевищує ефективність ручного аудиту. Фреймворки, такі як OWASP SCSVS та SCSTG, а також міжнародні стандарти NIST і ISO, забезпечують комплексний багаторівневий захист, а застосування бібліотек OpenZeppelin та дотримання стилю Solidity підвищують якість і безпеку коду. Інтеграція цих рішень у процеси CI/CD дозволяє суттєво зменшити кількість вразливостей і час їх виявлення. Отримані результати стали теоретичною основою для розробки власного методу тестування, яка буде розглянута у наступному розділі.

РОЗДІЛ 3
РОЗРОБКА МЕТОДУ ТЕСТУВАННЯ СМАРТ-КОНТРАКТІВ ДЛЯ ВИЯВЛЕННЯ ТА ОЦІНКИ РИЗИКІВ

3.1 Методологія тестування смарт-контрактів від розгортання до аналізу

Для початку необхідно налаштувати середовище для розробки, тестування та аналізу смарт-контрактів. У цьому випадку використовується Foundry, сучасний фреймворк для роботи зі смарт-контрактами на Ethereum [22].

Команди для встановлення Foundry:

```
curl -L https://foundry.paradigm.xyz | bash
foundryup
```

Створення нового проєкту:

```
forge init test
```

Це створює структуру проєкту:

– src/: каталог для смарт-контрактів (наприклад, BlockLottery.sol).

– test/: каталог для тестів, де функції, що починаються з test, автоматично виконуються як тести.

– foundry.toml: Файл конфігурації для налаштування компіляції, тестування та розгортання.

– lib/: Залежності, наприклад, бібліотеки OpenZeppelin.

– script/: Скрипти Solidity для розгортання та взаємодії з контрактами.

Написання коду смарт-контракту BlockLottery в файлі src/BlockLottery.sol, який зображений в додатку А.

Данна команда компілює контракт і створює артефакти в папці out/:

```
forge build
```

Контракт BlockLottery реалізує лотерею, де користувачі купують квитки за 0.1 ETH, а кожен раунд (6800 блоків) завершується розподілом поту на підпоти по 5 ETH, з можливим меншим останнім підпотом. Переможець кожного підпоту визначається хешем блоку, обчисленого як:

*((roundIndex + 1) * blocksPerRound) + subpotIndex*

Пот (pot): Загальна сума ETH, зібрана від учасників за один раунд, збережена в Round.pot.

Підпот (subpot): Частина поту, розподілена як виграш для одного переможця.

pot / subpotsCount

Пояснення кожної функції більш детально:

– getBlocksPerRound(): Повертає константу blocksPerRound (6800), що визначає тривалість раунду.

– getTicketPrice(): Повертає константу ticketPrice (0.1 ETH), ціну квитка.

– getRoundIndex(): Обчислює поточний раунд (block.number / blocksPerRound).

– getIsCashed(): Перевіряє, чи виплачено виграш для підпоту.

– calculateWinner(): Визначає переможця підпоту за хешем блоку (blockhash), що обчислюється як індекс квитка.

– getDecisionBlockNumber(): Обчислює номер блоку для визначення переможця.

– getSubpotsCount(): Визначає кількість підпотів на основі поту та blockReward (5 ETH).

– getSubpot(): Обчислює розмір підпоту, що може призводити до залишку.

– cash(): Виплачує виграш переможцю, використовуючи transfer для обмеження газу.

– getHashOfBlock(): Повертає хеш блоку за номером.

– getBuyers(): Повертає список покупців квитків.

– getTicketsCountByBuyer(): Повертає кількість квитків для покупця.

– getPot(): Повертає загальний пот раунду.

– receive(): Обробляє купівлю квитків, повертаючи надлишок ETH.

3.2 Статичний аналіз із застосуванням інструментів лінтингу та детекції вразливостей

Статичний аналіз – це метод перевірки вихідного коду або байткоду смарт-контракту без його виконання. Статичні аналізатори шукають патерни в коді, які можуть вказувати на вразливості. Даний підхід вважається швидким для раннього виявлення проблем, але може генерувати хибні спрацьовування через обмежену здатність враховувати контекст виконання.

Aderyn – це статичний аналізатор смарт-контрактів Solidity, створений на базі Rust компанією Cyfrin. Завдяки високій швидкості роботи та підтримці кастомних детекторів, він легко інтегрується в процес розробки, зокрема з фреймворками Hardhat і Foundry. Aderyn пропонує швидкий інтерфейс командного рядка (CLI) для зручної взаємодії та виводить результати аналізу у форматах Markdown і JSON [23].

Slither – це фреймворк для статичного аналізу Solidity та Vyper розроблений Trail of Bits на Python3. Він запускає набір із понад 90 детекторів для виявлення

вразливостей, виводить візуальну інформацію про деталі контракту та надає API для створення користувацьких аналізів. Slither дозволяє розробникам знаходити вразливості, покращувати розуміння коду та швидко створювати прототипи аналізів, автоматизувати перевірки через CI/CD і генерувати детальні звіти про знайдені вразливості [24].

Solhint – це інструмент для лінтингу коду Solidity, який підтримує підтримку стилю коду та базову безпеку смарт-контрактів шляхом інтеграції з існуючими робочими процесами. Він пропонує налаштування правил перевірки відповідно до вимог проекту, підтримує спільні конфігурації, дозволяє розширювати функціонал за допомогою плагінів для додавання нових правил та інтеграцій. Solhint інтегрується з популярними середовищами розробки, такими як Sublime Text, Atom, Vim, JetBrains IDEA та Visual Studio Code, що робить його зручним для розробників [25].

3.3 Динамічний аналіз, фаззинг, симуляція атак смарт-контрактів

Динамічний аналіз – це метод тестування смарт-контрактів, який передбачає виконання або моделювання коду в реальних чи симульованих умовах для виявлення вразливостей, які можуть бути пропущені статичним аналізом. На відміну від статичного аналізу, що перевіряє код без запуску, динамічний аналіз досліджує поведінку контракту, використовуючи фаззинг, символьне виконання та симуляцію атак.

Символьне виконання досліджує всі можливі шляхи виконання програми, представляючи вхідні дані символами. Наприклад, інструменти типу Mythril можуть знайти баги, пов’язані з некоректною обробкою даних, але потребують значних обчислювальних ресурсів.

Фаззинг генерує випадкові вхідні дані для тестування контрактів. Echidna швидко виявляє аномалії в Solidity-контрактах, тоді як Medusa прискорює тестування завдяки паралельним процесам.

Багатофункціональні інструменти, як-от Foundry, підтримують юніт-тестування, фаззинг і симуляцію атак, наприклад, моделювання флеш-позик для перевірки стійкості.

Симуляція атак дозволяє тестувати контракти на реальні загрози, такі як повторні входи чи маніпуляції. Комбінація цих методів забезпечує комплексний підхід до виявлення вразливостей.

Mythril аналізує байт-код EVM за допомогою символьного виконання, SMT-розв'язування та taint-аналізу. Виявляє вразливості в контрактах для Ethereum та сумісних блокчейнів [26].

Echidna – це фреймворк для фазинг-тестування Solidity-контрактів. Генерує різноманітні вхідні дані, забезпечуючи широке покриття коду та виявлення прихованих вразливостей [27].

Medusa – інструмент фазинг-тестування з підтримкою паралельних процесів. Оптимальний для великих проєктів завдяки швидкості та масштабованності [28].

Foundry – фреймворк для розробки й тестування. Включає юніт-тестування (Forge), фаззинг і взаємодію з контрактами (Cast).

3.4 Розробка методу тестування смарт-контрактів. Оцінка результатів

Методологія тестування смарт-контрактів, розроблена в рамках цієї роботи, є актуальною станом на 2025 рік. Вона базується на детальному аналізі відкритих джерел, включаючи офіційну документацію від провідних компаній, таких як ConsenSys Diligence та Crytic. Метод спрямований на досягнення високого рівня безпеки на всіх етапах життєвого циклу смарт-контрактів шляхом комбінування статичного аналізу, динамічного тестування та фазингу [29].

Перший етап методу передбачає аналіз коду смарт-контракту. Для оцінки безпеки застосовується методологія OWASP Smart Contract Security Weakness Enumeration (SCSWE) дозволяючи проводити всебічну оцінку потенційних ризиків і слабких місць. Код має бути написаний із використанням сучасних функцій і параметрів, уникаючи застарілих синтаксичних конструкцій, які можуть призводити до вразливостей. Далі проводиться перевірка відповідно до OWASP Smart Contract Security Verification Standard (SCSVS), яка включає набір критеріїв такі як архітектура, якість коду, управління, автентифікація, комунікації, криптографія, оракули, блокчейн, мости, DeFi та складність. Завершується цей етап аналізом тестів безпеки за допомогою OWASP Smart Contract Security Testing Guide (SCSTG) пропонуючи практичні рекомендації спрямованих на виявлення, перевірку та усунення уразливостей за різними категоріями ризику.

Наступним етапом вважається використання статичних інструментів. Гарантія стилістичним стандартам і виявлення потенційних проблем в коді досягається інструментом Solhint. Написання тестових сценаріїв, які можуть виявити логічні помилки або недоліки у функціонуванні смарт-контракту реалізується за допомогою інструменту Forge у фреймворку Foundry. Застосовування інструменту Cyfrin Aderyn проводить глибокий статичний аналіз без виконання коду, виявляючи уразливості на основі попередньої настройки параметрів, вивченої з офіційної документації.

Наступним кроком для оцінки реальних ризиків застосовується інструмент Mythril, який симулює виконання коду та виявляє уразливості за допомогою аналізу байткоду. Цей інструмент дозволяє детально описати можливі атаки та їхній вплив на контракт для подальшого аналізу.

Не менш важливим етапом є виявлення небажаної поведінки або збоїв у роботі смарт-контракту за допомогою інструментів Echidna та Medusa, які реалізують фазинг-тестування. У більшості випадків це передбачає написання спеціальних функцій або виділення ключових ділянок коду для перевірки

коректності обробки параметрів із використанням випадково згенерованих вхідних даних.

Використання безкоштовного онлайн-інструменту SolidityScan доповнює метод, надаючи детальну інформацію про уразливості, які могли б залишитися непоміченими, включаючи конкретні рядки коду, де вони можуть виникнути, а також оцінку ймовірності реалізації ризиків за допомогою моделі загроз.

Останнім пунктом у методології є аналіз результатів тестування, який здійснюється шляхом порівняння звітів від усіх використаних інструментів. Цей процес дозволяє визначити пріоритети для усунення виявлених уразливостей, базуючись на їхній критичності та ймовірності реалізації. Окрім того, проводиться оцінка покриття тестів, щоб переконатися, що всі критичні частини смарт-контракту, включаючи ключові функції та логічні блоки, були ретельно перевірені. Такий аналіз також включає ідентифікацію потенційних прогалин у тестуванні, які можуть потребувати додаткових сценаріїв або інструментів для забезпечення повного охоплення. Крім того, результати порівнюються з попередніми тестами, якщо такі проводилися, для відстеження прогресу в усуненні вразливостей і поліпшення загальної безпеки контракту. Це сприяє формуванню обґрунтованих рекомендацій для подальшого вдосконалення смарт-контракту та його адаптації до сучасних стандартів безпеки станом на 2025 рік.

3.5 Практичне тестування смарт-контрактів, оцінка і класифікація ризиків

Проведено комплексний аналіз смарт-контракту BlockLottery на основі стандартів OWASP Smart Contract Top 10 (2025), які охоплюють найкритичніші вразливості у смарт-контрактах. Проаналізувавши я виявив серйозні проблеми безпеки, які потребують негайного усунення для забезпечення захищеності децентралізованої системи лотереї.

SC01: Порушення контролю доступу. Контракт містить фундаментальну вразливість у системі контролю доступу. Функція cash() доступна для виклику будь-яким користувачем без будь-яких обмежень доступу. Це створює серйозний ризик безпеки, оскільки зловмисники можуть викликати цю функцію для отримання винагород, на які вони не мають права. Відсутність модифікаторів onlyOwner або системи ролей означає, що будь-хто може спробувати отримати виплати з контракту. Хоча функція містить певні перевірки, відсутність авторизації на рівні доступу залишає контракт вразливим до зловживань.

SC09: Небезпечна генерація випадковості. Контракт використовує вкрай небезпечний механізм генерації випадковості через функцію blockhash(). Функція getHashOfBlock() повертає хеш блоку, який може бути передбачуваним або маніпульованим майнерами. Детермінована природа блокчейну робить такий підхід до генерації випадковості критично небезпечним. Майнери можуть впливати на хеші блоків, особливо коли ставки високі, що дозволяє їм маніпулювати результатами лотереї. Крім того, blockhash() повертає 0 для блоків старше 256 блоків, що може призвести до передбачуваних результатів. Така вразливість може призвести до повної компрометації справедливості лотереї.

SC05: Атаки повторного входу. Функція cash() містить класичну вразливість reentrancy. Код виконує зовнішній виклик payable(winner).transfer(subpot) перед оновленням стану rounds[roundIndex].isCashed[subpotIndex] = true. Хоча transfer() має обмеження по газу, що теоретично захищає від деяких атак reentrancy, така структура коду все одно представляє ризик. Правильний підхід передбачає оновлення стану перед виконанням зовнішніх викликів (паттерн Checks-Effects-Interactions).

SC06: Неперевірені зовнішні виклики. Контракт використовує transfer() без перевірки успішності виконання операції. У функціях cash() та receive() відсутня обробка можливих помилок при переказі коштів. Якщо transfer() не вдасться, контракт може продовжити виконання, залишивши систему в непослідовному

стані. Функція transfer() може не спрацювати через різні причини: недостатню кількість газу, відмову контракту-отримувача прийняти кошти, або інші технічні проблеми. Відсутність перевірки таких ситуацій може призвести до втрати коштів або порушення логіки контракту.

SC03: Помилки бізнес-логіки. Алгоритм визначення переможця у функції calculateWinner() містить потенційні логічні помилки. Цикл для знаходження переможця може працювати некоректно при певних умовах, особливо коли winningTicketIndex дорівнює нулю або коли масив buyers пустий. Логіка розподілу субпотів також викликає занепокоєння. Функція getSubpot() може призвести до втрати коштів через цілочисельне ділення, коли залишок від ділення ігнорується. Це означає, що частина коштів може назавжди залишитися заблокованою в контракті.

SC10: Атаки відмови в обслуговуванні. Функція calculateWinner() містить цикл, який ітерує через масив покупців без обмеження кількості ітерацій. Якщо кількість покупців стане достатньо великою, виконання функції може перевищити ліміт газу блоку, роблячи контракт нефункціональним. Масив rounds[roundIndex].buyers може зростати необмежено, оскільки кожний новий покупець додається до масиву. При великій кількості учасників виклик функції calculateWinner() або cash() може стати неможливим через вичерпання газу.

SC04: Недостатня валідація вхідних даних. Функція cash() не виконує належної валідації параметрів roundIndex та subpotIndex. Хоча є перевірка subpotIndex >= subpotsCount, відсутня валідація того, чи існує раунд з вказаним індексом, чи чи не є параметри надмірно великими. Відсутність валідації може призвести до неочікуваної поведінки контракту або спроб доступу до неіснуючих даних. Належна валідація вхідних параметрів є критично важливою для захисту від маніпуляцій та забезпечення стабільної роботи контракту.

SC08: Потенційні проблеми з цілочисельними операціями. Хоча контракт використовує Solidity версії 0.8.5, яка має вбудований захист від переповнення,

деякі арифметичні операції все ще можуть призвести до неочікуваних результатів. Зокрема, операції ділення в функціях getSubpotsCount() та getSubpot() можуть призвести до втрати точності. Цілочисельне ділення може призвести до ситуацій, коли частина коштів залишається нерозподіленою. Наприклад, якщо pot не ділиться націло на blockReward, залишок може бути втрачений.

Для оцінки безпеки смарт-контракту BlockLottery використано інструмент Aderyn. Процес тестування розпочався зі встановлення Aderyn за допомогою команди:

```
curl –proto '=https' --tlsv1.2 -LsSf
https://github.com/cyfrin/aderyn/releases/latest/download/aderyn-installer.sh | bash
```

Далі аналіз смарт-контракту BlockLottery.sol проведено командою:

```
aderyn --src src/BlockLottery.sol -o aderynReport.md
```

Ця команда сканує файл BlockLottery.sol у каталозі src та зберігає результати у файл aderynReport.md, які зображені на рис. 3.1.

```
## Issue Summary

| Category | No. of Issues |
| --- | --- |
| High | 2 |
| Low | 4 |


# High Issues

## H-1: ETH transferred without address checks

Consider introducing checks for `msg.sender` to ensure the recipient of the money is as intended.

<details><summary>1 Found Instances</summary>


- Found in src/BlockLottery.sol [Line: 108](src/BlockLottery.sol#L108)

    ```solidity
        receive() external payable {
    ```

</details>
```

Рисунок 3.1 – Звіт Aderyn з виявленими вразливостями

Аналіз виявив шість вразливостей: дві високого рівня серйозності та чотири низького, як показано на Рисунку 3. Високосерйозні вразливості включають:

– ETH transferred without address checks
– Reentrancy: State change after external call

Низькосерйозні вразливості охоплюють:

– L-1: Unsafe ERC20 Operation
– L-2: Unspecific Solidity Pragma
– L-3: Public Function Not Used Internally
– L-4: State Change Without Event

Використано інструмент Slither, який забезпечує виявлення вразливостей у коді. Попередньо налаштовано ізольоване віртуальне середовище Python для встановлення необхідних пакетів:

```
python3 -m venv /home/trsbln/diplom/venv
source /home/trsbln/venv/bin/activate
python3 -m pip install slither-analyzer
```

Після встановлення Slither виконано аналіз смарт-контракту з виведенням структурованих результатів та їх збереженням у файл slitherReport.md за допомогою команди:

```
slither test/src/BlockLottery.sol --checklist --json test/slitherReport.md
```

Де параметр --checklist активує перевірку за стандартизованим списком вразливостей, а --json вказує формат звіту (рис. 3.2)

```
Impact: High
Confidence: Medium
 - [ ] ID-0
[BlockLottery.calculateWinner(uint256,uint256)](test/src/BlockLottery.sol#L36-L56) uses a weak PRNG: "[winningTicketIndex = decisionBlockHash % rounds[roundIndex].ticketsCount](test/src/
BlockLottery.sol#L43)"

test/src/BlockLottery.sol#L36-L56


## solc-version
Impact: Informational
Confidence: High
 - [ ] ID-1
Version constraint ^0.8.5 contains known severe issues (https://solidity.readthedocs.io/en/latest/bugs.html)
        - VerbatimInvalidDeduplication
        - FullInlinerNonExpressionSplitArgumentEvaluationOrder
        - MissingSideEffectsOnSelectorAccess
        - AbiReencodingHeadOverflowWithStaticArrayCleanup
        - DirtyBytesArrayToStorage
        - DataLocationChangeInInternalOverride
        - NestedCalldataArrayAbiReencodingSizeValidation
        - SignedImmutables.
It is used by:
        - [^0.8.5](test/src/BlockLottery.sol#L2)

test/src/BlockLottery.sol#L2


## reentrancy-unlimited-gas
Impact: Informational
Confidence: Medium
 - [ ] ID-2
Reentrancy in [BlockLottery.receive()](test/src/BlockLottery.sol#L108-L129):
        External calls:
        - [address(msg.sender).transfer(msg.value - value)](test/src/BlockLottery.sol#L117)
        State variables written after the call(s):
        - [rounds[roundIndex].ticketsCount += ticketsCount](test/src/BlockLottery.sol#L121)
        - [rounds[roundIndex].buyers.push(msg.sender)](test/src/BlockLottery.sol#L124)
        - [rounds[roundIndex].ticketsCountByBuyer[msg.sender] += ticketsCount](test/src/BlockLottery.sol#L127)
        - [rounds[roundIndex].pot += value](test/src/BlockLottery.sol#L128)

test/src/BlockLottery.sol#L108-L129


 - [ ] ID-3
Reentrancy in [BlockLottery.cash(uint256,uint256)](test/src/BlockLottery.sol#L74-L90):
        External calls:
        - [address(winner).transfer(subpot)](test/src/BlockLottery.sol#L87)
        State variables written after the call(s):
        - [rounds[roundIndex].isCashed[subpotIndex] = true](test/src/BlockLottery.sol#L89)

test/src/BlockLottery.sol#L74-L90
```

Рисунок 3.2 – Звіт Slither з виявленими вразливостями

Аналіз виявив одну високу вразливість та три інформативні зауваження. Звіт деталізує виявлені проблеми, включаючи назви вразливостей, рівні впливу (Impact) та впевненості (Confidence), а також вказує функції й рядки коду, де вони розташовані. Зокрема, високосерйозна вразливість пов’язана з використанням слабкого псевдовипадкового генератора чисел (PRNG) на основі blockhash, що може бути передбачуваним і використано зловмисниками для маніпуляцій. Серед інформативних зауважень відзначено: проблему з версією Solidity 0.8.5, яка має відомі серйозні недоліки, а також дві потенційні вразливості реентрансу в функціях receive (рядок 108) та cash (рядки 74–109), що можуть спричинити несанкціоновані зміни стану контракту через зовнішні виклики.

За допомогою онлайн-ресурсу SolidityScan, ми завантажили смарт-контракт на платформу та виконали початкову перевірку зображену на рис. 3.3 [30].

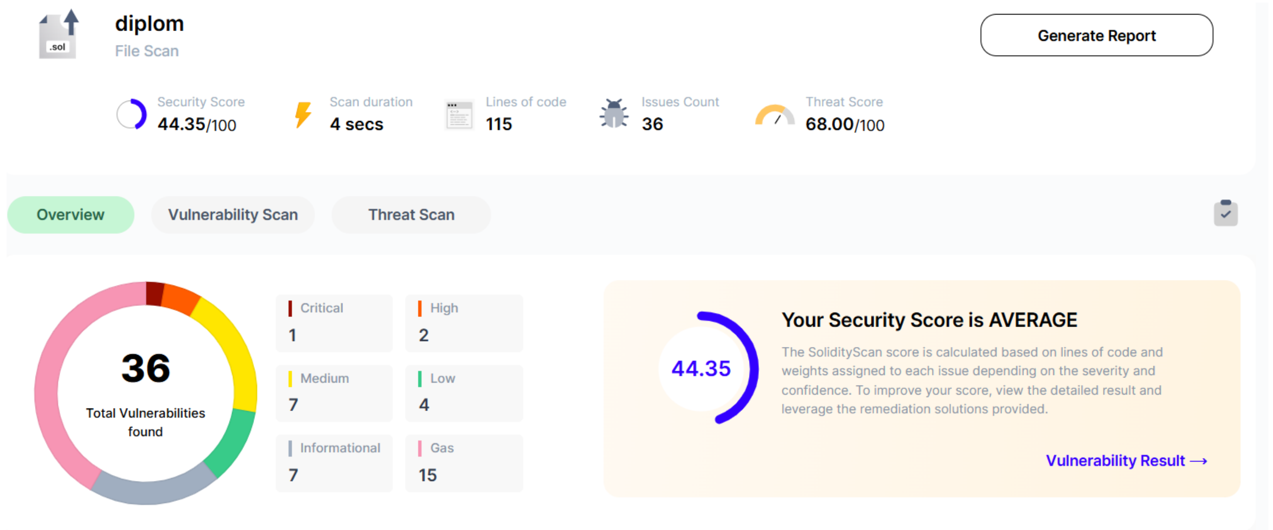

Рисунок 3.3 – Звіт SolidityScan з виявленими вразливостями

Скріншот інтерфейсу SolidityScan із завантаженим файлом смарт-контракту та результатами початкової перевірки включає загальний бал безпеки та кількість виявлених загроз.

Для поглибленого аналізу було обрано критичну вразливість, що потребує детального розгляду на рис. 3.4.

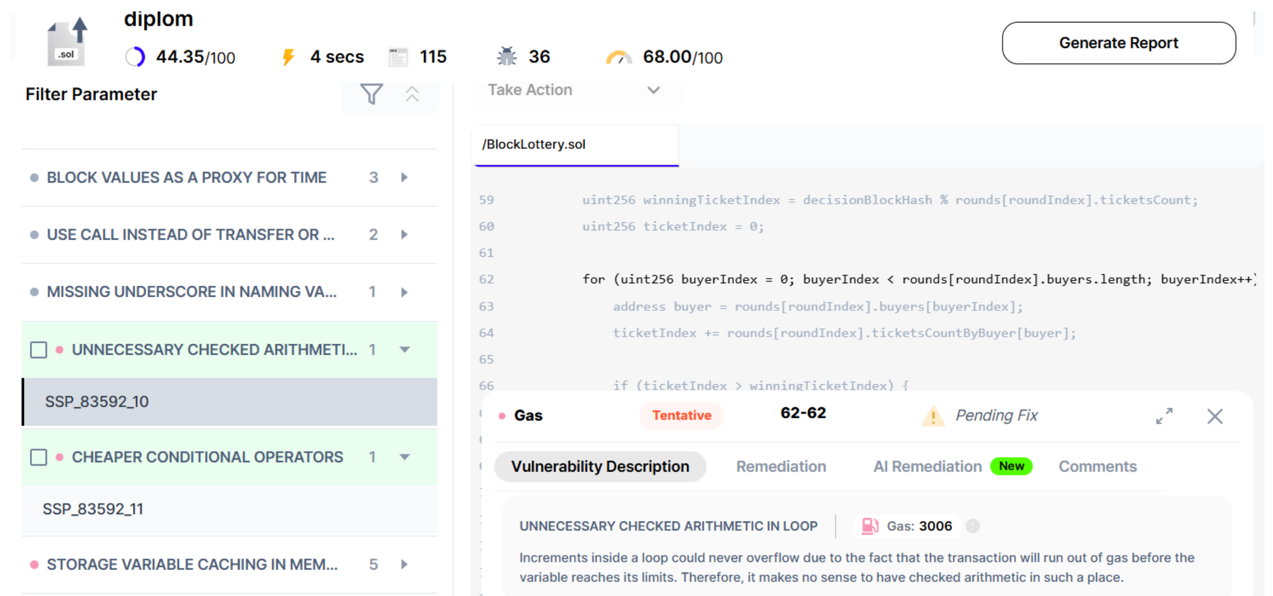

Рисунок 3.4 – Детальний опис критичної вразливості

Дана вразливість полягає у передачі ETH без перевірки адреси відправника (msg.sender) або у ризику повторного входу через зовнішні виклики, що загрожує небажаними транзакціями чи втратою коштів.

Додаткову інформацію про виявлені загрози можна переглянути у вкладці "Threat Summary", де наведено структурований аналіз ризиків (рис. 4).

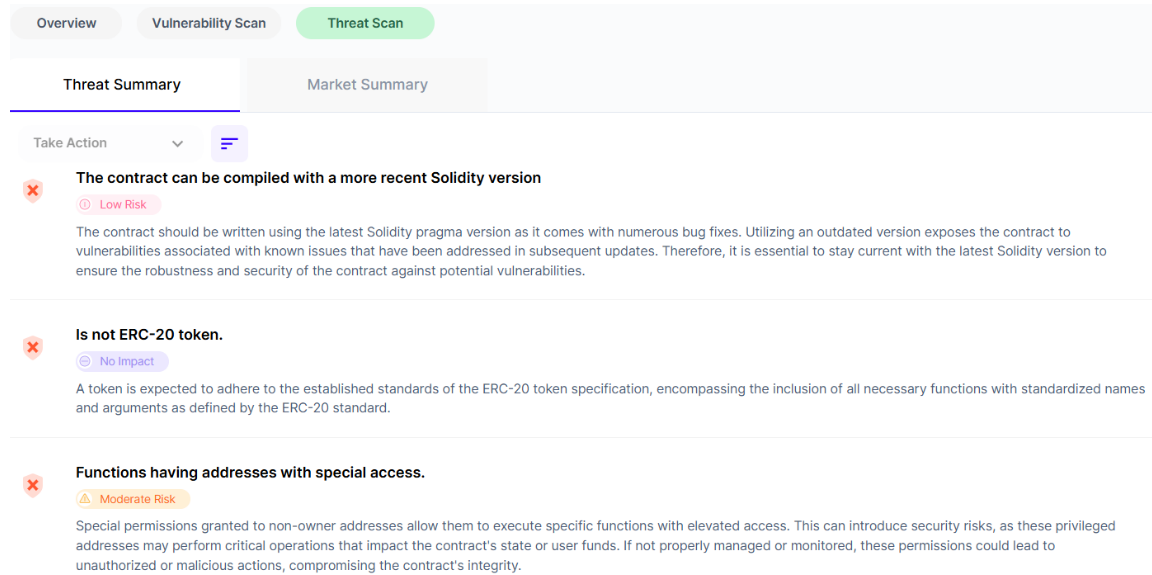

Рисунок 3.5 – Класифікація вразливостей та опис за рівнем ризику

Дана вкладка узагальнює всі виявлені проблеми, такі як використання застарілої версії Solidity чи недостатня оптимізація коду, дозволяючи визначити пріоритети для подальшого виправлення.

Для перевірки коду на стиль безпеки смарт-контракту BlockLottery.sol було використано інструмент Solhint.

Для встановлення Solhint використовується команда:

```
sudo npm install -g solhint@latest
```

Дана команда інсталює останню версію Solhint глобально, забезпечуючи доступ до нього для аналізу Solidity-файлів.

Наступним кроком є ініціалізація за допомогою команди:

solhint init

Ця команда генерує файл конфігурації .solhint.json, який визначає базові правила аналізу.

На рис. 3.6 у файлі .solhint.json вносяться зміни до параметрів, щоб адаптувати аналіз до потреб проєкту, наприклад, налаштування правил для перевірки вразливостей або стилю коду .

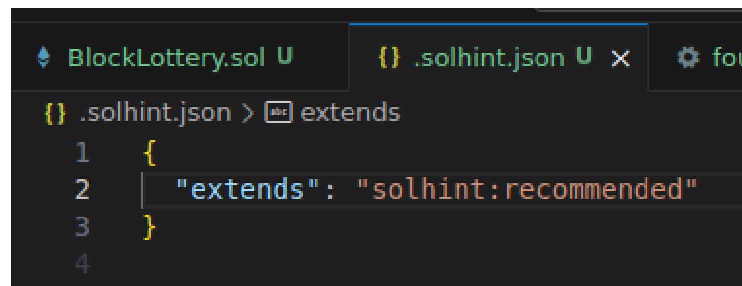

Рисунок 3.6 – Файл .solhint.json з внесеними змінами

Аналіз смарт-контракту BlockLottery.sol виконується командою:

solhint src/BlockLottery.sol –save

Ця команда запускає перевірку коду та зберігає результати у звіті, який проілюстровано на рис. 3.7.

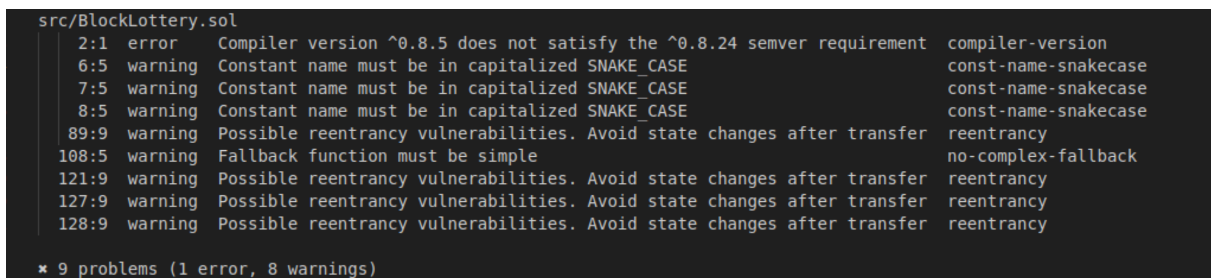

Рисунок 3.7 – Звіт Solhint із переліком виявлених проблем

Аналіз виявив 9 проблем у коді, які потребують негайного виправлення. Серед них: невідповідність версії компілятора, порушення стилю іменування констант та потенційні вразливості до реентрантних атак.

Встановлення Mythril здійснюється за допомогою команди:

```
pip3 install mythril
```

На рис. 3.8 зображено перелік детекторів, які Mythril використовує для виявлення вразливостей, застосовується команда:

```
myth list-detectors
```

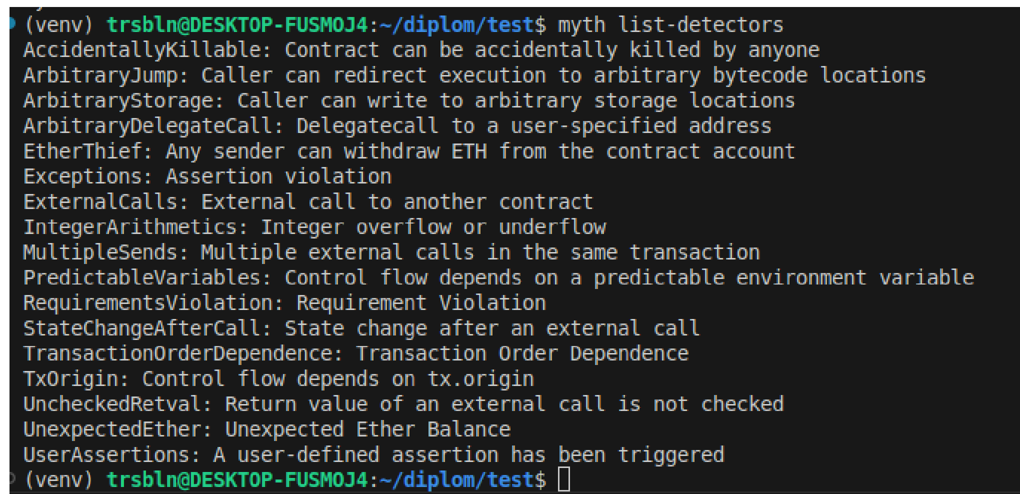

Рисунок 3.8 – Доступні детектори Mythril

Аналіз смарт-контракту BlockLottery.sol виконується командою:

```
myth analyze src/BlockLottery.sol -t 4 --execution-timeout 600
```

На рис. 3.9 зображено опцію -t 4, яка задає створення чотирьох транзакцій для тестування різних шляхів виконання, тоді як --execution-timeout 600 встановлює ліміт часу в 600 секунд, що сприяє глибшому виявленню вразливостей.

```
⊗ (venv) trsbln@DESKTOP-FUSMOJ4:~/diplom/test$ myth analyze src/BlockLottery.sol -t 4 --execution-timeout 600
==== Dependence on predictable environment variable ====
SWC ID: 120
Severity: Low
Contract: BlockLottery
Function name: calculateWinner(uint256,uint256)
PC address: 2065
Estimated Gas Usage: 1307 - 1402
A control flow decision is made based on The block.number environment variable.
The block.number environment variable is used to determine a control flow decision. Note that the values of va
and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. D
aware that use of these variables introduces a certain level of trust into miners.
--------------------
In file: src/BlockLottery.sol:38

if (decisionBlockNumber > block.number) {
            return address(0);
        }

--------------------
Initial State:

Account: [CREATOR], balance: 0x0, nonce:0, storage:{}
Account: [ATTACKER], balance: 0x0, nonce:0, storage:{}
```

Рисунок 3.9 – Результат сканування Mythril з описом вразливості

Mythril виявив одну вразливість, пов’язану з залежністю від передбачуваної змінної оточення. Зокрема, у смарт-контракті BlockLottery.sol використовується змінна block.number для прийняття рішень у логіці управління. Оскільки значення цієї змінної можуть бути частково передбачувані майнерами, це створює потенційний ризик для безпеки контракту.

Для підготовки середовища тестування необхідно встановити Echidna. Процес розпочинається із завантаження архіву з офіційного репозиторію за допомогою команди:

curl -fL https://github.com/crytic/echidna/releases/download/v2.2.6/echidna-2.2.6-x86_64-linux.tar.gz -o echidna.tar.gz

Далі архів розпаковується командою:

tar xvf echidna.tar.gz

Для фаззинг-тестування контракту BlockLottery до його коду додано дві функції-інваріанти: echidna_positive_pot та echidna_valid_tickets. На рис. 3.10

проілюстровані функції, що визначають ключові властивості контракту, а Echidna перевіряє на стійкість до порушень під час генерації випадкових транзакцій.

```
function echidna_positive_pot() public view returns (bool) {
    uint256 roundIndex = block.number / blocksPerRound;
    return rounds[roundIndex].pot >= 0;
}

function echidna_valid_tickets() public view returns (bool) {
    uint256 roundIndex = block.number / blocksPerRound;
    return rounds[roundIndex].ticketsCount * ticketPrice <= rounds[roundIndex].pot;
}
```

Рисунок 3.10 – Зображення коду з функціями echidna_positive_pot та echidna_valid_tickets

Функція echidna_positive_pot перевіряє, що пул раунду (pot) завжди невід’ємний.

$$(rounds[roundIndex].pot >= 0)$$

Ця умова є критично важливою для фінансової безпеки лотереї, оскільки від’ємний пул може призвести до некоректних виплат.

Функція echidna_valid_tickets забезпечує відповідність кількості виданих квитків внесеним коштам.

$$(rounds[roundIndex].ticketsCount * ticketPrice <= rounds[roundIndex].pot)$$

Це запобігає логічним помилкам у розподілі квитків, наприклад, видачі квитків без достатнього фінансування.

Ці інваріанти дозволяють Echidna автоматично тестувати контракт на наявність вразливостей, таких як від'ємний пул чи невідповідність квитків, шляхом генерації випадкових транзакцій.

На рис. 3.11 зображено результат фаззинг-тестування Echidna, яка згенерувала 50 139 викликів функцій, перевіряючи задані інваріанти.

```
(venv) trsbln@DESKTOP-FUSMOJ4:~/diplom/test$ ./echidna src/BlockLottery.sol --contract BlockLottery
[2025-05-27 21:42:57.07] Compiling src/BlockLottery.sol... Done! (0.224963476s)
Analyzing contract: /home/trsbln/diplom/test/src/BlockLottery.sol:BlockLottery
[2025-05-27 21:42:57.30] Running slither on src/BlockLottery.sol... Done! (0.454482158s)
echidna_positive_pot: passing
echidna_valid_tickets: passing


Unique instructions: 1306
Unique codehashes: 1
Corpus size: 16
Seed: 2995205554054515051
Total calls: 50139
```

Рисунок 3.11 – Вивід Echidna з результатами тестування

Інваріанти echidna_positive_pot та echidna_valid_tickets успішно пройшли перевірку. Пул раунду завжди залишається невід'ємним, забезпечуючи фінансову стабільність. Кількість виданих квитків відповідає внесеним коштам, виключаючи логічні помилки в розподілі.

Процес тестування розпочинається з встановлення Medusa за допомогою команди:

```
go install github.com/crytic/medusa@latest
```

Далі інструмент ініціалізується командою:

```
medusa init
```

Ця команда створює конфігураційний файл medusa.json, який містить базові параметри для фаззинг-тестування, зокрема обмеження кількості тестів, довжину послідовності викликів, час виконання та кількість воркерів.

На рис. 3.12 зображено результат фаззингу смарт-контракта BlockLottery.sol, який запускається командою:

medusa fuzz --compilation-target src/BlockLottery.sol

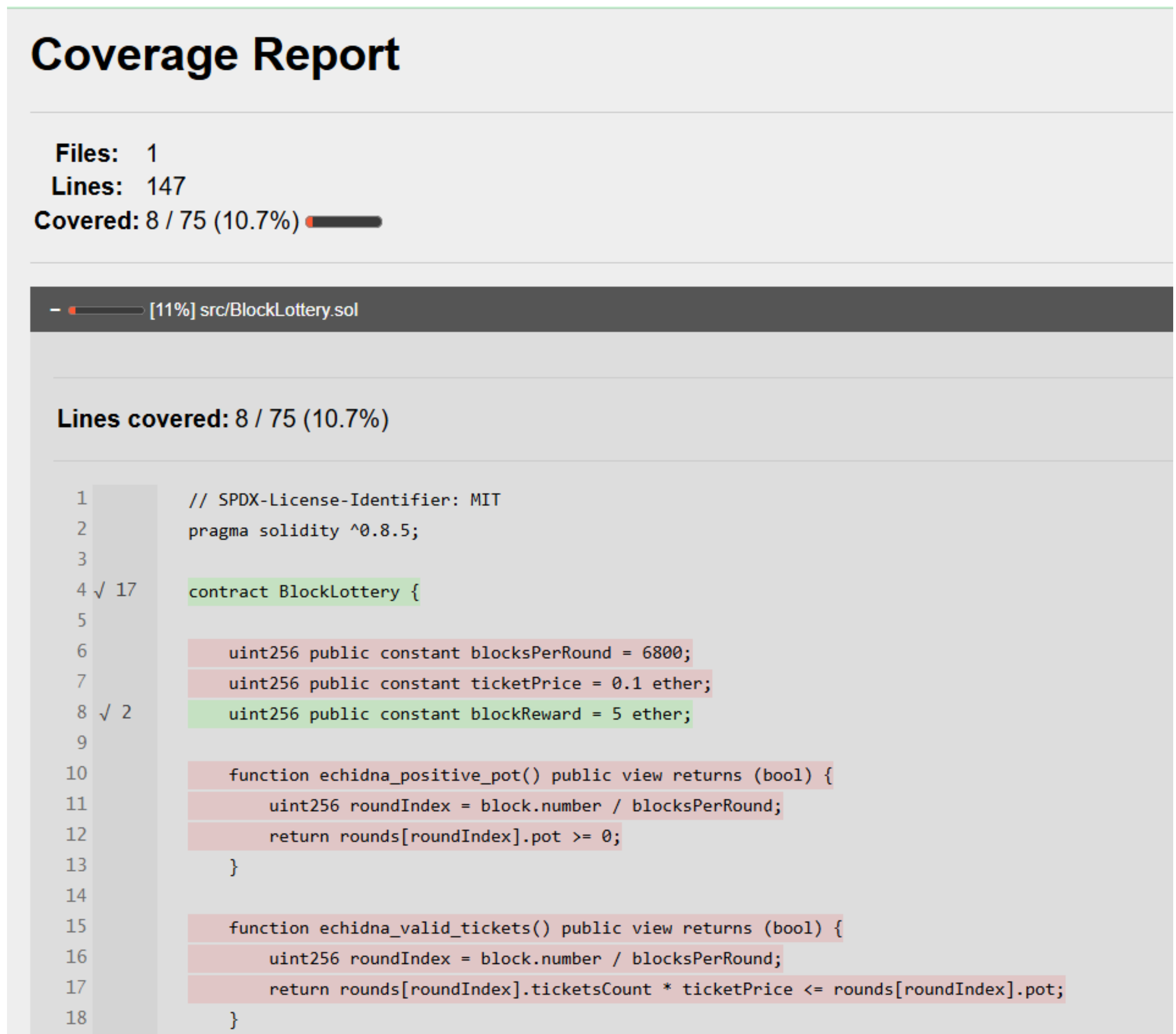

Рисунок 3.12 – Звіт з виявленими вразливостями

Під час фаззингу контракту BlockLottery за допомогою Medusa виникла проблема з тривалим часом виконання (понад 15 хвилин). Для прискорення тестування були зменшені параметри в medusa.json: testLimit із 100000 до 10000, callSequenceLength із 100 до 10, timeout із 3600 до 300 секунд, а також кількість воркерів із 10 до 4. Це дозволило скоротити час виконання, зберігаючи базову

ефективність тестування. Після 15 хвилин фаззингу Medusa завершила тестування асерцій у функції cash, не виявивши порушень (наприклад, ділення на нуль або арифметичних помилок). Звіт показав, що 1 тест пройшов, а звіт про покриття коду був збережений у файловій системі.

Для оцінки смарт-контракту на наявність недоліків чи вразливостей використано інструмент Forge у фреймворку Foundry. Тестування охоплювало фаззинг, інваріантне та диференціальне тестування. Тестовий файл BlockLotteryTest.t.sol, розташований у директорії test, детально описано у Додатку Б. Цей файл містить набір тестових функцій, спрямованих на перевірку різних аспектів функціональності контракту BlockLottery та зображено на рис. 3.13.

```
trsbln@DESKTOP-FUSMOJ4:~/diplom/test$ forge test
[⠊] Compiling...
No files changed, compilation skipped

Ran 3 tests for test/BlockLotteryTest.t.sol:BlockLotteryTest
[PASS] testGetDecisionBlockNumberDifferential(uint256,uint256) (runs: 256, μ: 15456, ~: 15456)
[PASS] testGetRoundIndexDifferential(uint256) (runs: 257, μ: 13742, ~: 13742)
[PASS] testReceiveFuzz(uint256) (runs: 257, μ: 36813, ~: 36844)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 79.97ms (114.92ms CPU time)

Ran 2 tests for test/BlockLotteryTest.t.sol:BlockLotteryInvariantTest
[PASS] invariant_positive_pot() (runs: 256, calls: 128000, reverts: 0)
[PASS] invariant_valid_tickets() (runs: 256, calls: 128000, reverts: 0)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 6.79s (12.10s CPU time)

Ran 2 test suites in 6.79s (6.87s CPU time): 5 tests passed, 0 failed, 0 skipped (5 total tests)
trsbln@DESKTOP-FUSMOJ4:~/diplom/test$
```

Рисунок 3.13. Успішне проходження диференційних та інваріантних тестів.

Тестування смарт-контракту BlockLottery за допомогою Foundry показало, що диференційні та інваріантні тести пройшли успішно. Це свідчить про коректність обчислень і стабільність стану контракту.

Проте фаззинг-тест testReceiveFuzz завершився невдачею через помилку "Transfer failed" під час обробки транзакції на 1.82 ETH. Причиною, ймовірно, став недостатній баланс контракту для повернення надлишкових коштів. Після коригування, яке полягало у додаванні необхідного балансу до контракту, повторне

тестування завершилося успішно. Цей випадок підкреслює критичну важливість тестування граничних сценаріїв у розробці смарт-контрактів.

Застосування розробленого методу тестування смарт-контракту BlockLottery дозволило ідентифікувати ряд вразливостей, що створюють суттєві ризики для безпеки децентралізованої системи. Виявлені вразливості класифіковано згідно з методологією OWASP Smart Contract Top 10 (2025) та систематизовано з урахуванням механізмів реалізації загроз, рівнів впливу та потенційних фінансових втрат на основі аналізу реальних інцидентів у сфері смарт-контрактів.

Таблиця 3.1

Результат виявлених вразливостей

Категорія	**Виявлені проблеми**	**Рівень впливу**	**Фінансовий ризик (OWASP 2025)**
SC01: Контроль доступу	Неавторизований виклик cash() (Aderyn L-2)	Критичний	До $953.2M
SC05: Реентранція	Небезпечний порядок операцій у cash() (Slither H-1)	Високий	До $35.7M
SC09: Випадковість	Генерація через blockhash() (Mythril)	Високий	До $8.8M
SC03: Логічні помилки	Помилки у calculateWinner() та розподілі субботів (Aderyn L-4)	Середній	До $63.8M

SC10: DoS	Необмежений цикл у calculateWinner() (SolidityScan)	Середній	Не кількісний

Висновок за розділом 3

Отже, я розробив метод тестування смарт-контрактів, яка поєднує статичний і динамічний аналіз для виявлення вразливостей. Застосування цього методу до смарт-контракту BlockLottery дозволило виявити низку критичних вразливостей, які класифіковано відповідно до стандартів OWASP Smart Contract Top 10 за 2025 рік. Я розглянув та застосував різноманітні інструменти, такі як Aderyn, Slither, Solhint, Mythril, Echidna, Medusa та Foundry, де кожен інструмент вніс унікальний внесок у виявлення специфічних проблем безпеки. Зокрема, статичні інструменти виявили проблеми зі стилем коду та потенційні уразливості, тоді як динамічні інструменти, такі як Echidna та Medusa, підтвердили стійкість контракту до граничних сценаріїв, хоча виявили недоліки в обробці певних вхідних даних. Розроблений метод може слугувати універсальним шаблоном для тестування інших смарт-контрактів, забезпечуючи їх відповідність найвищим стандартам безпеки перед розгортанням.

ВИСНОВКИ

У кваліфікаційній роботі розроблено гібридний метод тестування смарт-контрактів на вразливості із застосуванням автоматизованих інструментів для виявлення та оцінки потенційних загроз.

На початковому етапі дослідження було розглянуто поняття, архітектуру та принципи функціонування смарт-контрактів, досліджено їх технічну реалізацію та життєвий цикл. Систематизовано основні сфери застосування смарт-контрактів у фінансовій галузі, технологічному секторі та традиційних індустріях. Розроблено моделі загроз та визначено сучасні підходи до забезпечення безпеки смарт-контрактів.

Особлива увага приділялася спеціалізованим фреймворкам для підвищення рівня безпеки смарт-контрактів. Ретельно досліджені системи верифікації безпеки OWASP SCSVS та методології тестування OWASP Smart Contract Testing Guide. Розглянуто міжнародні стандарти безпеки NIST IR8202, ISO 22739 та ISO 23455, принципи безпечного кодування Ethereum Security Best Practices та структурні вимоги Solidity Style Guide.

Крім того, був написаний смарт-контракт BlockLottery, в якому користувачі купують квитки, відправляючи ETH на контракт, а переможці визначаються на основі хешів блоків. На практиці було проведено статичний аналіз за допомогою інструментів Solhint, Slither, Cyfrin Aderyn та SolidityScan та динамічний аналіз, фаззинг і симуляція атак з використанням інструментів Mythril, Echidna, Medusa та Forge.

Підсумовуючи, розроблена методологія тестування смарт-контрактів реалізує системний підхід до підвищення безпеки децентралізованих систем, сприяючи мінімізації ризиків і фінансових втрат, а також формуванню надійної основи для подальшого розвитку смарт-контрактів у фінансовій та інших галузях.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Що таке смарт-контракти в блокчейні та як вони працюють? [Електронний ресурс]. – Режим доступу до ресурсу: https://www.investopedia.com/terms/s/smart-contracts.asp
2. Опанування Solidity. Частина 3: Архітектура смарт-контрактів [Електронний ресурс]. – Режим доступу до ресурсу: https://coinsbench.com/mastering-solidity-part-3-smart-contract-architecture-%EF%B8%8F-ec70c22c8d93
3. Як EVM компілює смарт-контракти: детальне пояснення Електронний ресурс]. – Режим доступу до ресурсу: https://www.lcx.com/how-evm-compiles-smart-contracts-explained/
4. Пояснення життєвого циклу смарт-контракту [Електронний ресурс]. – Режим доступу до ресурсу: https://www.lcx.com/life-cycle-of-smart-contract-explained/
5. 10 реальних випадків використання смарт-контрактів [Електронний ресурс]. – Режим доступу до ресурсу: https://hedera.com/learning/smart-contracts/smart-contract-use-cases
6. Застосування смарт-контрактів, обмеження та перспективи на майбутнє [Електронний ресурс]. – Режим доступу до ресурсу: https://www.itransition.com/blockchain/smart-contract/applications
7. 5 реальних застосувань смарт-контрактів [Електронний ресурс]. – Режим доступу до ресурсу: https://www.coinsquare.com/en-ca/learn/5-real-life-applications-of-smart-contracts
8. OWASP Smart Contract Top 10 [Електронний ресурс]. – Режим доступу до ресурсу: https://owasp.org/www-project-smart-contract-top-10/

9. Найбільші хаки смарт-контрактів в історії, або як поставити під загрозу до 2,2 мільярда доларів США [Електронний ресурс]. – Режим доступу до ресурсу: https://medium.com/solidified/the-biggest-smart-contract-hacks-in-history-or-how-to-endanger-up-to-us-2-2-billion-d5a72961d15d

10. Wormhole Bridge Exploit Analysis [Електронний ресурс]. – Режим доступу до ресурсу: https://certik.medium.com/wormhole-bridge-exploit-analysis-5068d79cbb71

11. The Poly Network Exploit Analysis [Електронний ресурс]. – Режим доступу до ресурсу: https://polynetwork.medium.com/the-poly-network-exploit-analysis-b0a77aff6078

12. Моделювання загроз для смарт-контрактів: найкращий покроковий посібник [Електронний ресурс]. – Режим доступу до ресурсу: https://composable-security.com/blog/threat-modeling-for-smart-contracts-best-step-by-step-guide/

13. Що таке безпека блокчейну: приклади, проблеми та рішення [Електронний ресурс]. – Режим доступу до ресурсу: https://www.h-x.technology/ua/blog-ua/what-is-blockchain-security-examples-issues-and-solutions-ua

14. OWASP Smart Contract Security [Електронний ресурс]. – Режим доступу до ресурсу: https://scs.owasp.org/

15. OWASP Smart Contract Security Testing Guide (SCSTG) [Електронний ресурс]. – Режим доступу до ресурсу: https://scs.owasp.org/SCSTG/

16. NIST IR 8202 Огляд технології блокчейн [Електронний ресурс]. – Режим доступу до ресурсу: https://csrc.nist.gov/pubs/ir/8202/final

17. ISO 22739:2024 Блокчейн та технології розподіленого реєстру – словник [Електронний ресурс]. – Режим доступу до ресурсу: https://www.iso.org/standard/82208.html

18. ISO/TR 23455:2019 Блокчейн та технології розподіленого реєстру – Огляд та взаємодія між смарт-контрактами в системах блокчейну та технології

розподіленого реєстру [Електронний ресурс]. – Режим доступу до ресурсу: https://www.iso.org/standard/75624.html

19. Ethereum Smart Contract Security Best Practices [Електронний ресурс]. – Режим доступу до ресурсу: https://consensysdiligence.github.io/smart-contract-best-practices/

20. Style Guide – Solidity 0.8.31 документація [Електронний ресурс]. – Режим доступу до ресурсу: https://docs.soliditylang.org/en/latest/style-guide.html

21. Openzeppelin стандарт для безпечних ончейн-додатків будь-якого масштабу [Електронний ресурс]. – Режим доступу до ресурсу: https://www.openzeppelin.com/

22. Foundry Book [Електронний ресурс]. – Режим доступу до ресурсу: https://book.getfoundry.sh/

23. Cyfrin Aderyn [Електронний ресурс]. – Режим доступу до ресурсу: https://github.com/Cyfrin/aderyn

24. Slither, статичний аналізатор смарт-контрактів [Електронний ресурс]. – Режим доступу до ресурсу: https://github.com/crytic/slither

25. Лінтер Solidity з відкритим кодом [Електронний ресурс]. – Режим доступу до ресурсу: https://protofire.io/solhint

26. ConsenSysDiligence/mythril [Електронний ресурс]. – Режим доступу до ресурсу: https://github.com/ConsenSysDiligence/mythril

27. Echidna: швидкий фаззер смарт-контрактів [Електронний ресурс]. – Режим доступу до ресурсу: https://github.com/crytic/echidna

28. Crytic/medusa [Електронний ресурс]. – Режим доступу до ресурсу: https://github.com/crytic/medusa

29. Найкращі інструменти аналізу смарт-контрактів 2025 [Електронний ресурс]. – Режим доступу до ресурсу: https://www.h-x.technology/ua/blog-ua/the-best-smart-contract-analysis-tools-2025-ua

30. SolidityScan: Best Smart Contract Scanner & Auditing Tool [Електронний ресурс]. – Режим доступу до ресурсу: https://solidityscan.com/

ДОДАТКИ

ДОДАТОК А

Смарт-контракт BlockLottery

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.5;
contract BlockLottery {
   uint256 public constant blocksPerRound = 6800;
   uint256 public constant ticketPrice = 0.1 ether;
   uint256 public constant blockReward = 5 ether;
   function getBlocksPerRound() public pure returns (uint256) {
      return blocksPerRound;
   }
   function getTicketPrice() public pure returns (uint256) {
      return ticketPrice;
   }
   struct Round {
      address[] buyers;
      uint256 pot;
      uint256 ticketsCount;
      mapping(uint256 => bool) isCashed;
      mapping(address => uint256) ticketsCountByBuyer;
   }
   mapping(uint256 => Round) private rounds;

   function getRoundIndex() public view returns (uint256) {
      return block.number / blocksPerRound;
   }
   function getIsCashed(uint256 roundIndex, uint256 subpotIndex) public view returns
(bool)   {
      return rounds[roundIndex].isCashed[subpotIndex];
   }
   function calculateWinner(uint256 roundIndex, uint256 subpotIndex) public view
returns (address) {
```

```
        uint256 decisionBlockNumber = getDecisionBlockNumber(roundIndex,
subpotIndex);
        if (decisionBlockNumber > block.number) {
            return address(0);
        }
        uint256 decisionBlockHash = getHashOfBlock(decisionBlockNumber);
        uint256 winningTicketIndex = decisionBlockHash %
rounds[roundIndex].ticketsCount;
        uint256 ticketIndex = 0;
        for (uint256 buyerIndex = 0; buyerIndex < rounds[roundIndex].buyers.length;
buyerIndex++) {
            address buyer = rounds[roundIndex].buyers[buyerIndex];
            ticketIndex += rounds[roundIndex].ticketsCountByBuyer[buyer];
            if (ticketIndex > winningTicketIndex) {
                return buyer;
            }
        }
        return address(0);
    }
    function getDecisionBlockNumber(uint256 roundIndex, uint256 subpotIndex) public
pure returns (uint256) {
        return ((roundIndex + 1) * blocksPerRound) + subpotIndex;
    }
    function getSubpotsCount(uint256 roundIndex) public view returns (uint256) {
        uint256 subpotsCount = rounds[roundIndex].pot / blockReward;
        if (rounds[roundIndex].pot % blockReward > 0) {
            subpotsCount++;
        }
        return subpotsCount;
    }
    function getSubpot(uint256 roundIndex) public view returns (uint256) {
        return rounds[roundIndex].pot / getSubpotsCount(roundIndex);
    }
    function cash(uint256 roundIndex, uint256 subpotIndex) public {
        uint256 subpotsCount = getSubpotsCount(roundIndex);
        if (subpotIndex >= subpotsCount) return;
        uint256 decisionBlockNumber = getDecisionBlockNumber(roundIndex,
subpotIndex);
        if (decisionBlockNumber > block.number) return;
        if (rounds[roundIndex].isCashed[subpotIndex]) return;
        address winner = calculateWinner(roundIndex, subpotIndex);
```

```
      if (winner == address(0)) return;
      uint256 subpot = getSubpot(roundIndex);
      payable(winner).transfer(subpot);
      rounds[roundIndex].isCashed[subpotIndex] = true;
   }
   function getHashOfBlock(uint256 blockIndex) public view returns (uint256) {
      return uint256(blockhash(blockIndex));
   }
   function getBuyers(uint256 roundIndex) public view returns (address[] memory) {
      return rounds[roundIndex].buyers;
   }
   function getTicketsCountByBuyer(uint256 roundIndex, address buyer) public view
returns (uint256) {
      return rounds[roundIndex].ticketsCountByBuyer[buyer];
   }
   function getPot(uint256 roundIndex) public view returns (uint256) {
      return rounds[roundIndex].pot;
   }
   receive() external payable {
      uint256 roundIndex = getRoundIndex();
      uint256 value = msg.value - (msg.value % ticketPrice);
      if (value == 0) {
         return;
      }
      if (value < msg.value) {
         payable(msg.sender).transfer(msg.value - value);
      }
      uint256 ticketsCount = value / ticketPrice;
      rounds[roundIndex].ticketsCount += ticketsCount;
      if (rounds[roundIndex].ticketsCountByBuyer[msg.sender] == 0) {
         rounds[roundIndex].buyers.push(msg.sender);
      }
      rounds[roundIndex].ticketsCountByBuyer[msg.sender] += ticketsCount;
      rounds[roundIndex].pot += value;
   }
}
```

ДОДАТОК Б

Фаззинг тест для смарт-контракту

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.5;
import "forge-std/Test.sol";
import "forge-std/StdInvariant.sol";
import "../src/BlockLottery.sol";
contract BlockLotteryTest is Test {
    BlockLottery lottery;
    address user1 = address(0x1);
    address user2 = address(0x2);
    function setUp() public {
        lottery = new BlockLottery();
        vm.deal(user1, 100 ether);
        vm.deal(user2, 100 ether);
        vm.deal(address(lottery), 100 ether);
    }
    function testReceiveFuzz(uint256 amount) public {
        vm.assume(amount <= 100 ether);
        uint256 roundIndex = lottery.getRoundIndex();
        uint256 initialPot = lottery.getPot(roundIndex);
        uint256 initialTickets = lottery.getTicketsCount(roundIndex);
        vm.prank(user1);
        (bool success, ) = address(lottery).call{value: amount}("");
        if (!success) {
            emit log_named_uint("Transfer failed with amount: ", amount);
            return;
        }
        uint256 expectedValue = amount - (amount % lottery.getTicketPrice());
        uint256 expectedTickets = expectedValue / lottery.getTicketPrice();
        assertEq(lottery.getPot(roundIndex), initialPot + expectedValue, "Incorrect pot
update");
        assertEq(lottery.getTicketsCount(roundIndex), initialTickets + expectedTickets,
"Incorrect tickets update");
    }
    function testGetRoundIndexDifferential(uint256 blockNumber) public {
        vm.assume(blockNumber <= type(uint256).max / lottery.getBlocksPerRound());
        vm.roll(blockNumber);
        uint256 expected = blockNumber / lottery.getBlocksPerRound();
```

```
        assertEq(lottery.getRoundIndex(), expected, "Incorrect round index calculation");
    }
    function testGetDecisionBlockNumberDifferential(uint256 roundIndex, uint256
subpotIndex) public {
        vm.assume(roundIndex <= type(uint256).max / lottery.getBlocksPerRound());
        vm.assume(subpotIndex <= 1000);
        uint256 expected = ((roundIndex + 1) * lottery.getBlocksPerRound()) +
subpotIndex;
        assertEq(lottery.getDecisionBlockNumber(roundIndex, subpotIndex), expected,
"Incorrect decision block number");
    }
}
contract BlockLotteryInvariantTest is StdInvariant, Test {
    BlockLottery lottery;
    address user1 = address(0x1);
    address user2 = address(0x2);
    function setUp() public {
        lottery = new BlockLottery();
        vm.deal(user1, 100 ether);
        vm.deal(user2, 100 ether);
        targetContract(address(lottery));
    }
    function invariant_positive_pot() public view {
        uint256 roundIndex = lottery.getRoundIndex();
        assert(lottery.getPot(roundIndex) >= 0);
    }
    function invariant_valid_tickets() public view {
        uint256 roundIndex = lottery.getRoundIndex();
        assert(lottery.getTicketsCount(roundIndex) * lottery.getTicketPrice() <=
lottery.getPot(roundIndex));
    }
}
```