**ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**
**НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ КІБЕРБЕЗПЕКИ ТА ЗАХИСТУ ІНФОРМАЦІЇ**

**КАФЕДРА ІНФОРМАЦІЙНОЇ ТА КІБЕРНЕТИЧНОЇ БЕЗПЕКИ**

# КВАЛІФІКАЦІЙНА РОБОТА

на тему:

**«Безпека смарт-контрактів у блокчейні на основі Solidity»**

зі спеціальності *125 Кібербезпека*
*(код, найменування спеціальності)*

освітньо-професійної програми *Інформаційна та кібернетична безпека*
*(назва програми)*

*Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело*

_____ Андрій СТЕПАНОВ
*(підпис)*

Виконав: здобувач вищої освіти групи БСД-41
СТЕПАНОВ Андрій
*(прізвище, ім'я)*

Керівник Завідувач кафедри ІКБ Гайдур Галина
*(науковий ступінь, вчене звання, прізвище, ім'я)*

Рецензент _____
*(науковий ступінь, вчене звання, прізвище, ім'я)*

Київ 2025

**ЗМІСТ**

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

SSDLC – Secure Software Development Lifecycle

EVM – Ethereum Virtual Machine

CWE – Common Weakness Enumeration

DoS – Denial of Service

DEX – Decentralized Exchange

DASP – Decentralized Application Security Project

POW – Proof or Work

POS – Proof of Stake

SWC – Smart Contract Weakness

# ВСТУП

*Актуальність дослідження.* Технологія блокчейн стрімко розвивається та активно використовується у фінансовому, державному та корпоративному секторах. Однією з ключових її складових є смарт-контракти. Вони дозволяють реалізовувати децентралізовані угоди у вигляді програмного коду, який виконується у блокчейн-мережі автоматично при виконанні визначених умов. Ці контракти не потребують посередників і гарантують незмінність та прозорість виконання.

Смарт-контракти зазвичай реалізуються мовою Solidity та запускаються у середовищі Ethereum Virtual Machine. Їх логіка і правила записуються у вигляді функцій, доступних для взаємодії будь-яким користувачам мережі. При цьому, один із головних принципів блокчейну – незмінність коду після розгортання, яка стає водночас і перевагою, і вразливістю. Якщо у коді смарт-контракту допущена помилка або не враховано критичні аспекти безпеки, виправити її після публікації буде неможливо.

У той самий час смарт-контракти є об’єктом великої фінансової привабливості. Через них проходять мільйони доларів у вигляді токенів, криптовалют, застав, позик або ставок. Це робить їх бажаною ціллю для зловмисників, які активно досліджують публічні контракти у пошуках вразливостей, щоб використати їх для власної вигоди.

У зв’язку з цим виникає необхідність у глибокому аналізі безпеки смарт-контрактів ще на етапі їх розробки.

*Об’єкт дослідження* – Механізми функціонування та забезпечення безпеки смарт-контрактів у блокчейн-мережах.

*Предмет дослідження* – Методи аналізу, виявлення та усунення вразливостей у смарт-контрактах, створених мовою програмування Solidity.

*Мета роботи* – розробити практичні рекомендації щодо їх захисту шляхом використання сучасних інструментів аудиту та підходів до безпечної розробки.

*Наукові завдання:*

- Дослідити принципи функціонування смарт-контрактів та мережі Ethereum;
- Визначити основні типи вразливостей у Solidity-контрактах та типові вектори атак;
- Проаналізувати та протестувати інструменти аудиту смарт-контрактів;
- Оцінити ефективність виявлення вразливостей;
- Розробити практичні рекомендації щодо їх усунення та попередження;

*Методи дослідження* – Опрацювання технічної документації мови програмування Solidity, аналітичний огляд експертних матеріалів з кібербезпеки, а також практичне застосування інструментів аудиту.

*Практичне значення одержаних результатів*: Отримані результати можуть бути використані розробниками децентралізованих додатків та фахівцями з кібербезпеки для підвищення стійкості смарт-контрактів до атак.

# 1 АНАЛІЗ АРХІТЕКТУРИ СМАРТ-КОНТРАКТІВ ТА ТИПОВИХ ВРАЗЛИВОСТЕЙ НА ОСНОВІ МОВИ ПРОГРАМУВАННЯ SOLIDITY

## 1.1. Технологія блокчейн: загальні принципи

Блокчейн є розподіленою базою даних, яка забезпечує надійне та незмінне зберігання інформації шляхом об’єднання транзакцій у послідовний ланцюг блоків. Основною метою блокчейну є створення середовища, в якому всі учасники мають доступ до єдиного достовірного джерела даних, яке не піддається несанкціонованим змінам чи фальсифікаціям. Всі блоки зв'язані між собою через криптографічні хеш-функції, що забезпечує цілісність і незмінність даних у мережі. Будь-яка зміна даних в одному з блоків призводить до зміни хешу, що одразу порушує всю структуру ланцюга і помічається іншими учасниками мережі.

Хеш – це результат криптографічної хеш-функції, яка перетворює довільну вхідну інформацію у вихідну строку фіксованої довжини. Для блокчейн-систем характерним є використання SHA-256 або аналогічних алгоритмів. Ключовою властивістю хешу є те, що навіть незначна зміна вхідних даних повністю змінює вихідний результат. Саме завдяки цій властивості хеш-функцій забезпечується незмінність і цілісність даних у блокчейні – будь-яка модифікація транзакції в одному з блоків змінює його хеш, що спричиняє розсинхронізацію всього ланцюга.

Щоб зберегти логіку ланцюгового зв’язку, кожен блок містить хеш попереднього. Таким чином, порушення будь-якого блоку призводить до автоматичного порушення цілісності всієї мережі, що одразу виявляється іншими учасниками.

Кожен блок у блокчейні містить службову інформацію та перелік підтверджених транзакцій. Він складається з двох основних частин: заголовка (header) і тіла (body). Заголовок блоку включає кілька ключових елементів: хеш попереднього блоку, мітку часу, nonce (спеціальне число, що використовується в алгоритмах консенсусу, зокрема Proof of Work), а також власний хеш поточного

блоку. Тіло блоку, своєю чергою, містить перелік транзакцій, які були включені до блоку після валідації.

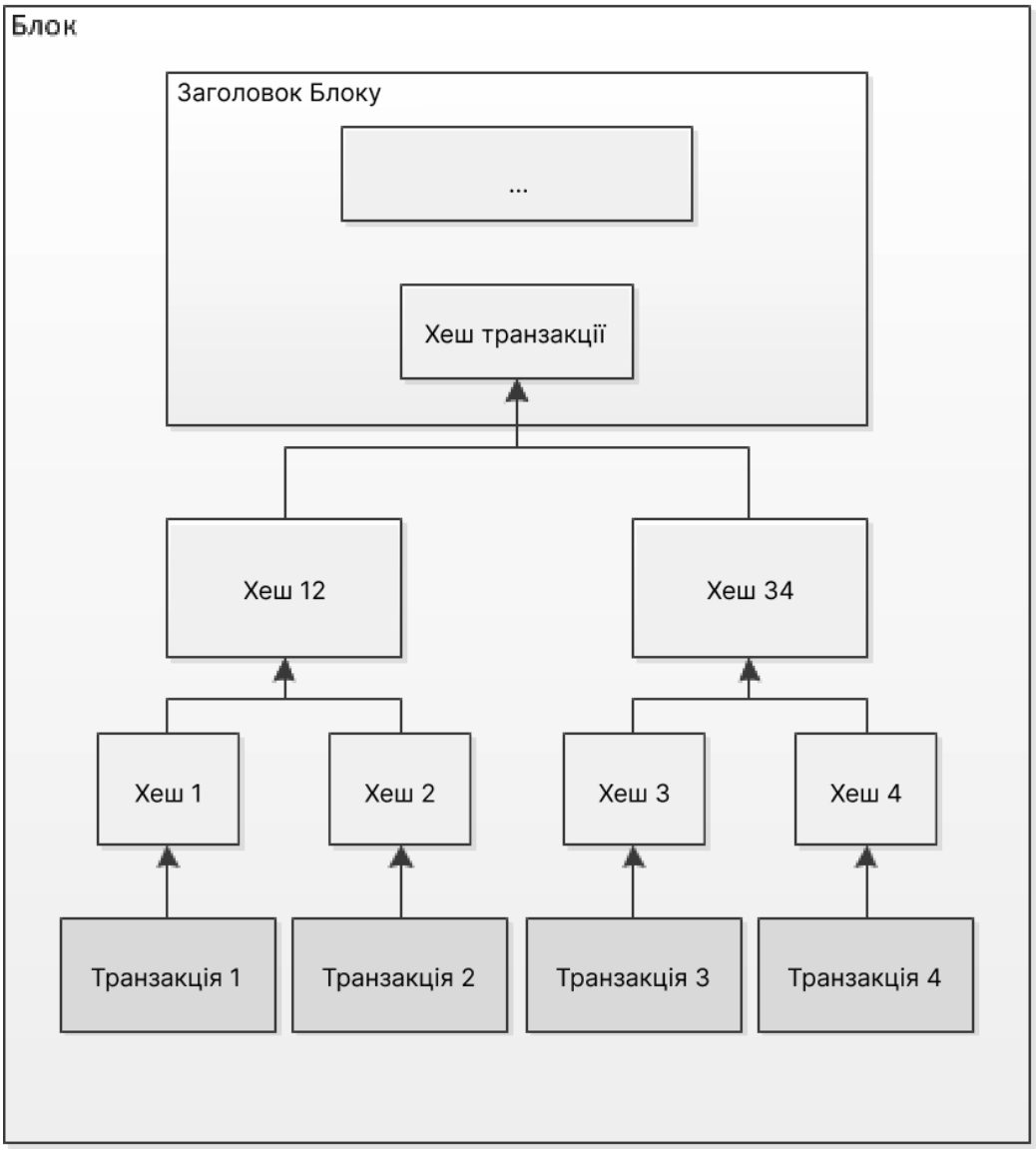


Рис.1.1. Графічне зображення структури блока

Іншою важливою особливістю блокчейн-систем є децентралізація. Усі блоки та транзакції розповсюджуються між вузлами (нодами) мережі, які зберігають повну копію реєстру та беруть участь у перевірці нових даних. Для того щоб транзакція була підтверджена, вона має бути схвалена більшістю вузлів згідно з обраним алгоритмом консенсусу. Найпоширенішими серед них є Proof of Work (PoW), механізм, у якому учасники виконують складні обчислення, та Proof of Stake (PoS), алгоритм, у якому право на додавання блоку визначається пропорційно до кількості заблокованих токенів.

Залежно від цілей використання, блокчейн-мережі поділяються на кілька

типів:

1. Публічні блокчейни – відкриті для будь-якого користувача, який бажає брати участь у перевірці транзакцій або створенні нових блоків. Такі мережі забезпечують максимальну прозорість і децентралізацію. Найвідомішими прикладами є Bitcoin та Ethereum.

2. Приватні блокчейни – мають обмежений доступ і повністю контролюються однією організацією. Вони використовуються переважно в корпоративному середовищі, де важлива конфіденційність і керованість. Прикладом є Hyperledger Fabric.

3. Консорціумні блокчейни – керуються групою організацій, які спільно підтримують мережу. Це забезпечує баланс між децентралізацією та контрольованістю, і зазвичай використовується у сфері фінансів, логістики або державного управління.

4. Сайдчейни – це допоміжні ланцюги, які працюють паралельно з основним блокчейном. Вони дозволяють масштабувати систему або реалізовувати нові функціональні можливості без впливу на основну мережу. Прикладами сайдчейнів є Polygon та Liquid Network.

## 1.2. Платформа Ethereum і середовище виконання смарт-контрактів

Ethereum є однією з найпопулярніших і технологічно розвинених платформ для створення децентралізованих додатків (dApps) та реалізації смарт-контрактів. Це публічна блокчейн-мережа з відкритим вихідним кодом, яка була запущена у 2015 році з метою надати користувачам більше, ніж просто обмін криптовалютою – зокрема, можливість створювати автономні програми, що працюють без централізованого управління.

Ключовою інновацією Ethereum стало впровадження Ethereum Virtual Machine (EVM) – віртуального середовища, у якому виконується байт-код смарт-контрактів. EVM є ізольованим виконуваним середовищем, що гарантує однакову поведінку контрактів на всіх вузлах мережі. Завдяки цьому досягається повна

детермінованість результатів транзакцій і збереження цілісності системи. Контракти в Ethereum створюються мовою Solidity, яка компілюється у байт-код і виконується EVM.

Усі учасники мережі Ethereum представлені у вигляді акаунтів. Існує два типи облікових записів: зовнішні акаунти, якими керує приватний ключ користувача, та контрактні акаунти, що містять код смарт-контракту. Зовнішній акаунт може ініціювати транзакції та виклики функцій контрактів, тоді як контрактний акаунт самостійно транзакцій не створює, а лише реагує на ті, що отримує. Така модель дозволяє будувати складну логіку взаємодії між користувачами та децентралізованими додатками. Кожен обліковий запис в Ethereum має чотири ключові складові:

1. **Nonce** — лічильник транзакцій або створених контрактів, який запобігає повторному використанню транзакцій.
2. **Balance** — кількість токенів ETH на рахунку.
3. **StorageRoot** — це посилання на внутрішнє сховище даних облікового запису. Воно вказує на хеш дерева Merkle Patricia, де зберігаються змінні стану контракту, як-от лічильники, адреси користувачів тощо.
4. **CodeHash** — хеш EVM-коду, що асоціюється з контрактом або відсутній для звичайних акаунтів.

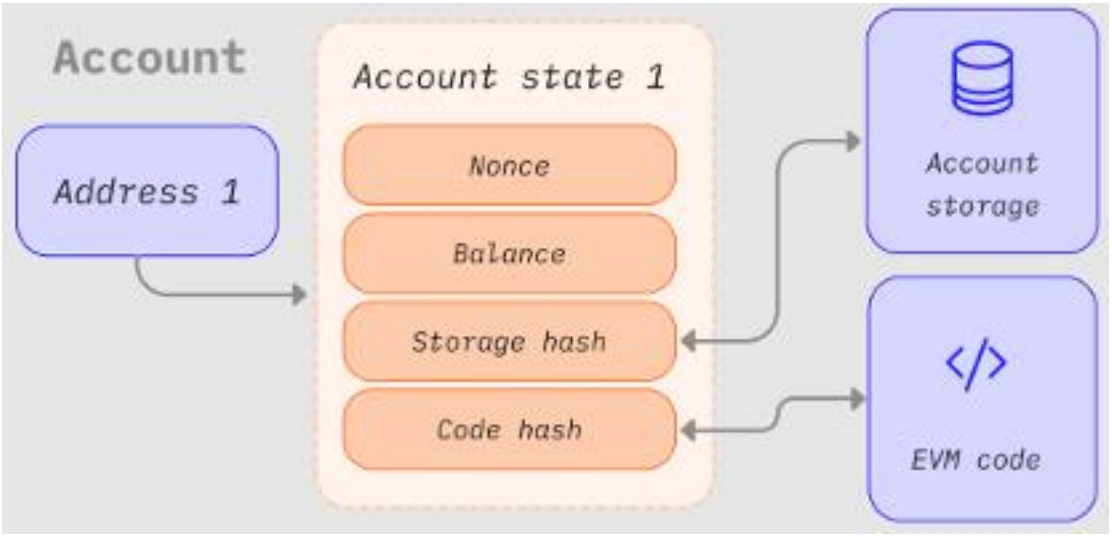


Рис.1.2. Схематичне зображення

## 1.3. Платформа Ethereum і середовище виконання смарт-контрактів

Смарт-контракт — це програма, яка автоматично виконується у блокчейн-мережі за заданими умовами. Його основна ідея полягає в тому, щоб виключити потребу в посередниках при укладенні угод, оскільки вся логіка договору записується у вигляді коду. Такий підхід забезпечує прозорість, незмінність і довіру до результату, оскільки після розгортання смарт-контракту в мережі Ethereum змінити його код стає неможливою без спеціального механізму

У технічному сенсі смарт-контракт — це обліковий запис Ethereum, який має не тільки баланс, але й власний програмний код. Цей код пишеться переважно мовою програмування Solidity, яка є спеціалізованою для розробки контрактів у середовищі Ethereum Virtual Machine. Код компілюється у байт-код і розміщується в мережі через транзакцію. Після цього контракт стає доступним для взаємодії будь-кому у мережі.

Ідея смарт-контрактів виникла ще у 1994 році, коли криптограф Нік Сабо запропонував концепцію "розумних угод" — цифрових механізмів, які автоматично виконують умови угоди без участі третіх сторін. Тоді ця ідея залишалася лише теорією, оскільки не існувало технологій, здатних забезпечити її безпечне і надійне виконання. Лише після появи блокчейн-технологій, а особливо — платформи Ethereum у 2015 році, смарт-контракти стали технічно можливими для масового використання.

Структура смарт-контракту у Solidity зазвичай включає кілька ключових компонентів: оголошення змінних, які зберігають стан; функції, які описують логіку поведінки; події для фіксації важливих дій у журналі блокчейну; та модифікатори для контролю доступу до функцій. Наприклад, у простому токен-контракті можна знайти змінні, що відображають баланс користувачів, функції переказу коштів та події, що сигналізують про ці операції.

Користувачі взаємодіють із контрактом через зовнішні облікові записи, використовуючи приватні ключі. Контракт сам ініціювати дії не може — він лише реагує на вхідні запити. Всередині контракту можуть зберігатися змінні стану,

функції, які обробляють логіку, модифікатори доступу та події, що фіксують важливі події у журналі блокчейну. Кожна взаємодія змінює стан мережі або залишає цифровий слід, що забезпечує прозорість і наглядовість.

Цікаво, що смарт-контракти можуть викликати інші контракти. Це дозволяє будувати складні системи, де кілька частин взаємодіють між собою. Наприклад, платформа DeFi може одночасно керувати ліквідністю, позиками, обміном активів — і все це через різні контракти.

На практиці смарт-контракти лежать в основі багатьох відомих проєктів:

- Aave — протокол для кредитування, у якому контракти керують депозитами, заставами, відсотками та ліквідацією.
- MakerDAO — система для створення стабільної криптовалюти DAI, в якій усі фінансові процеси — від генерації до контролю застав — закладено у смарт-контракти.
- Synthetix — платформа для створення та торгівлі активами (токенами, прив’язаними до реальних активів).

В той же час основою роботи смарт-контрактів є Ethereum Virtual Machine (EVM) — віртуальне середовище, що функціонує на кожному вузлі Ethereum. Саме вона відповідає за виконання коду смарт-контрактів. Розробники пишуть контракти на мові програмування високого рівня — переважно Solidity. Однак Solidity не може виконуватись напряму — її потрібно скомпілювати у байт-код, який розуміє EVM. Це пов’язано з кількома факторами: безпекою, уніфікацією, детермінізмом виконання та потребою ізоляції коду від операційної системи вузлів. Саме байт-код є тією формою, яку розуміє кожен вузол у мережі Ethereum.

Байт-код складається з послідовності так званих оп-кодів (operation codes) — інструкцій, які керують виконанням програми. Наприклад, ADD додає два значення на стеку, CALL викликає інший контракт, SSTORE зберігає дані у сховище. Усього в EVM понад 140 таких інструкцій. Під час виконання транзакції EVM зчитує байт-код, виконує оп-коди один за одним і змінює стан контракту чи блокує виконання, якщо виникає помилка. Усі ці операції оплачуються в одиницях газу, який визначає обчислювальну вартість виконання інструкції. Наприклад,

ADD коштує лише кілька одиниць газу, а SSTORE — до 20 000.

Ця модель дозволяє EVM бути детермінованою: тобто всі вузли мережі отримають однаковий результат при виконанні одного й того ж байт-коду. Крім того, виконання контракту відбувається в ізольованому середовищі, яке не має доступу до зовнішніх ресурсів чи операційної системи вузла. Це гарантує безпеку та передбачуваність поведінки контракту.

### 1.4 Структура смарт-контракту на прикладі

Для кращого розуміння механізмів функціонування смарт-контрактів доцільно розглянути приклад базової реалізації, що демонструє основні елементи структури контракту та порядок виконання операцій у середовищі блокчейну Ethereum.

```
pragma solidity ^0.8.0;

contract SimpleVault {
    address public owner;
    mapping(address => uint256) private deposits;

    constructor() {
        owner = msg.sender;
    }

    // Внесення коштів на рахунок користувача
    function deposit() external payable {
        require(msg.value > 0, "Must send ETH");
        deposits[msg.sender] += msg.value;
    }

    // Перевірка балансу користувача
    function getBalance() external view returns (uint256) {
        return deposits[msg.sender];
    }

    // Вивід усіх коштів користувача
    function withdraw() external {
        uint256 amount = deposits[msg.sender];
        require(amount > 0, "Nothing to withdraw");

        deposits[msg.sender] = 0;
        payable(msg.sender).transfer(amount);
    }
}
```

Рис.1.3. Приклад смарт-контракту

Розглянемо покроково функціональну логіку даного контракту:

1. Ініціалізація контракту: Під час розгортання контракту в мережі

Ethereum виконується конструктор constructor(), у якому фіксується адреса ініціатора як власника (owner).

2. Внесення коштів: Функція deposit() дозволяє користувачу надіслати певну кількість криптовалюти (ETH) до контракту. Параметр msg.value вказує суму надходження, яка перевіряється умовою require(msg.value > 0).

3. Отримання балансу: Функція getBalance() є лише для читання (визначена як view) і повертає кількість коштів, які зберігаються на рахунку викликача функції.

4. Виведення коштів: Функція withdraw() надає можливість користувачу отримати усю суму, яку він раніше вніс. Спочатку перевіряється наявність коштів на балансі. У випадку успішної перевірки значення обнуляється (для запобігання повторному зняттю), після чого кошти надсилаються назад на адресу користувача за допомогою функції transfer().

## 1.5 Класифікація вразливостей смарт-контрактів

Попри визначеність логіки виконання та високий рівень децентралізації, смарт-контракти залишаються об'єктами значного ризику у сфері кібербезпеки. Через відкритість коду, незмінність після розгортання та обмеженість засобів відновлення контрольованого стану, помилки у контрактах можуть призводити до безповоротних фінансових втрат.

The DAO був децентралізованою інвестиційною платформою, яка працювала за допомогою складного смарт-контракту на Ethereum. Його контракт дозволяв учасникам вносити кошти та голосувати за підтримку певних проєктів. У 2016 році хакеру вдалося скористатися помилкою в логіці повернення коштів.

Проблема полягала в тому, що контракт спочатку виконував зовнішню операцію з переказу коштів користувачу, а лише після цього оновлював внутрішній баланс. Це дозволило зловмиснику багаторазово ініціювати одну й ту ж саму дію, перш ніж баланс був зменшений. Таким чином він отримував кошти неодноразово за один і той самий вклад. У результаті було виведено понад 3,6 мільйона ETH.

Атака мала настільки масштабні наслідки, що призвела до поділу мережі Ethereum на дві гілки: Ethereum (ETH) та Ethereum Classic (ETC).

Іншим відомим прикладом зламу є «Parity Multisig Wallet Hack». Parity Multisig Wallet був популярним рішенням для зберігання коштів за схемою кількох підписів, яке також реалізовувалось у вигляді смарт-контракту. Його структура передбачала винесення всієї логіки в окрему бібліотеку, яка використовувалась усіма гаманцями за допомогою механізму делегованого виклику.

У 2017 році користувач випадково активував функцію ініціалізації, яка перезаписала власника бібліотеки. Отримавши контроль, він викликав функцію видалення, яка знищила код бібліотеки. Після цього всі контракти-гаманці, що покладалися на цю бібліотеку, перестали працювати. Кошти користувачів залишились у блокчейні, але функціонал для доступу до них було повністю втрачено. Заморожено було понад 150 мільйонів доларів США.

У наведених випадках зловмисники скористалися відкритістю та прозорістю логіки смарт-контрактів. Проаналізувавши вихідний код, вони виявили вразливі ділянки та зуміли ефективно їх використати для досягнення власної мети.

З метою системного виявлення, опису та запобігання поширеним помилкам у реалізації смарт-контрактів були створені публічні класифікаційні системи. Вони надають розробникам структуровану інформацію про вразливості, приклади їх прояву, а також можливі способи усунення або уникнення.

Однією з найбільш визнаних у спільноті є SWC Registry (Smart Contract Weakness Classification Registry). Це відкритий реєстр, який підтримується компаніями, що спеціалізуються на безпеці смарт-контрактів. Він містить понад 40 типів уразливостей, кожна з яких має унікальний номер (наприклад, SWC-100 — Function Default Visibility, SWC-112 — Delegatecall to Untrusted Contract). Для кожної уразливості наведено опис, приклади коду, умови виникнення та посилання на інструменти виявлення.

Ще однією важливою системою є DASP Top 10 — список десяти найнебезпечніших категорій вразливостей, сформований на основі аналізу інцидентів у децентралізованих додатках. Цей перелік слугує аналогом відомого

OWASP Top 10 у класичній веб-безпеці, але адаптований до особливостей Solidity та децентралізованих платформ.

Також окремі категорії вразливостей були інтегровані у глобальний реєстр CWE (Common Weakness Enumeration), що дозволяє поєднувати аналіз смарт-контрактів з практиками безпеки традиційного програмного забезпечення.

У наступному підрозділі буде представлено узагальнену класифікацію найбільш поширених вразливостей у смарт-контрактах, яка базується на зазначених джерелах та результатах реальних інцидентів, що мали місце у блокчейн-індустрії.

Базуючись на узагальнену класифікацію реєстрів можна виділити найбільш популярні вразливості:

1. Reentrancy: Одна з найбільш відомих та небезпечних вразливостей у смарт-контрактах — повторний виклик. Вона виникає в тих випадках, коли смарт-контракт здійснює зовнішній виклик (наприклад, передає ефір) до того, як завершено зміну його внутрішнього стану. Якщо зовнішній контракт реалізує функцію зворотного виклику (fallback або receive), він може повторно викликати уразливу функцію до завершення її попереднього виконання. В результаті цього один користувач може неодноразово взаємодіяти з функцією, яка передбачалася як одноразова, наприклад, зняти кошти кілька разів до того, як його баланс буде обнулено.

Наведена нижче реалізація функції withdraw демонструє приклад коду, у якому присутня вразливість до повторного виклику:

```
1   pragma solidity ^0.6.0;
2
3   contract Vulnerable {
4       mapping(address => uint256) public balances;
5
6       function deposit() public payable {
7           balances[msg.sender] += msg.value;
8       }
9
10      function withdraw() public {
11          uint256 amount = balances[msg.sender];
12          require(amount > 0, "No balance");
13
14          (bool sent, ) = msg.sender.call{value: amount}("");
15          require(sent, "Failed to send");
16
17          balances[msg.sender] = 0;
18      }
19  }
```

Рис.1.4. Приклад вразливого коду

У цьому прикладі баланс користувача оновлюється лише після передачі ефіру. Якщо адреса msg.sender є контрактом, який реалізує функцію зворотного виклику, то під час виклику call{value: amount} цей контракт отримає керування і зможе повторно викликати функцію withdraw, оскільки його баланс ще не обнулено.

2. Access Control: Помилки контролю доступу виникають тоді, коли смарт-контракт не містить чітких перевірок прав користувача перед виконанням критичних функцій. У результаті зловмисник може виконати дії, які за задумом розробника мали бути доступні лише власнику, адміністратору чи довіреному обліковому запису.

Такі вразливості особливо небезпечні в контрактах, що дозволяють змінювати конфігурацію, передавати кошти або видаляти логіку контракту. Їх часто пов’язують із відсутністю перевірки змінної msg.sender або використанням ненадійних модифікаторів.

Наведений нижче приклад демонструє уразливу реалізацію, у якій будь-хто може викликати функцію destroy, що призводить до видалення контракту та передачі залишку коштів на зазначену адресу:

```
pragma solidity ^0.6.0;

contract Insecure {
    address public owner;

    constructor() public {
        owner = msg.sender;
    }

    function destroy() public {
        selfdestruct(payable(msg.sender));
    }
}
```

Рис.1.5. Приклад вразливого коду

У цьому прикладі функція destroy доступна будь-якому користувачу. Вона не перевіряє, чи msg.sender є дійсним власником контракту. Таким чином, будь-хто може викликати її і знищити контракт, отримавши залишок ефіру на свій баланс.

3. Arithmetic Issues: Арифметичні помилки у смарт-контрактах, зокрема переповнення (overflow) та заниження (underflow) чисел, можуть призвести до серйозних логічних збоїв і надати зловмисникам можливість змінювати баланс, обхід перевірок або створення некоректного стану. Такі помилки трапляються при використанні цілих чисел (uint, int), коли результат операції перевищує допустимі межі типу даних.

У прикладі нижче демонструється переповнення: при зменшенні значення до нуля, подальше віднімання призведе до циклічного переходу на максимальне значення типу uint256:

```
pragma solidity ^0.6.0;

contract OverflowExample {
    mapping(address => uint256) public balances;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw(uint256 amount) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;
    }

    function resetBalance() public {
        // Встановлюємо баланс у 0
        balances[msg.sender] = 0;

        // Потенційне заниження — 0 - 1 = 2^256 - 1
        balances[msg.sender] -= 1;
    }
}
```

Рис.1.5. Приклад вразливого коду

У даному прикладі, якщо користувач викликає resetBalance(), після встановлення балансу у 0, подальше зменшення на 1 призведе до арифметичного заниження і результатом стане максимальне можливе значення типу uint256. Це створює серйозну загрозу, оскільки контракт буде вважати, що користувач має величезний баланс, який не відповідає дійсності.

4. Unchecked External Call Return Value: У смарт-контрактах на Solidity часто використовуються зовнішні виклики, наприклад, за допомогою функцій call, send або delegatecall. Важливою умовою безпеки є обов'язкова перевірка результату таких викликів, оскільки вони можуть завершитися з помилкою або бути навмисно заблокованими контрагентом. Якщо результат зовнішнього виклику не перевіряється, контракт продовжить виконання, навіть якщо передача коштів або даних не відбулася, що може призвести до логічних помилок або втрати коштів.

Наведений нижче приклад демонструє потенційно небезпечну реалізацію, у якій не перевіряється результат надсилання ефіру:

```
1   pragma solidity ^0.8.0;
2
3   contract UncheckedCall {
4       function sendEther(address payable recipient) public payable {
5           recipient.send(msg.value);
6       }
7   }
```

Рис.1.6. Приклад вразливого коду

У цьому випадку, якщо адреса recipient є контрактом, який не реалізує функцію receive() або fallback() (або якщо її виконання перевищує ліміт газу), надсилання ефіру за допомогою send завершиться помилкою. Однак контракт не перевіряє повернене значення — отже, він вважає, що переказ відбувся успішно, навіть якщо кошти не були фактично передані.

5. Denial of Service: Вразливість типу *Denial of Service* (DoS) полягає в тому, що смарт-контракт може бути навмисно виведений з ладу або частково заблокований шляхом створення умов, за яких нормальне виконання його функцій стає неможливим або надмірно дорогим. Вразливість DoS у контрактах найчастіше виникає через:

- відсутність обмежень на кількість елементів у масивах або списках;
- надмірно складні цикли, що залежать від динамічних даних;
- використання require або зовнішніх викликів без захисту в критичних місцях.

Наведений нижче приклад демонструє, як один користувач може блокувати роботу функції розподілу винагород між учасниками:

```
1   pragma solidity ^0.6.0;
2
3   contract DoSExample {
4       address[] public recipients;
5
6       function addRecipient(address _addr) public {
7           recipients.push(_addr);
8       }
9
10      function distribute() public payable {
11          uint amount = msg.value / recipients.length;
12          for (uint i = 0; i < recipients.length; i++) {
13              (bool success, ) = recipients[i].call{value: amount}("");
14              require(success, "Transfer failed");
15          }
16      }
17  }
```

Рис.1.7. Приклад вразливого коду

У цьому прикладі, якщо будь-який із користувачів у списку recipients є контрактом, який навмисно викликає помилку при отриманні коштів, це призведе до того, що вся функція distribute завершиться з помилкою. Таким чином, один недобросовісний учасник здатен заблокувати виконання функції для всіх інших користувачів, створюючи ситуацію відмови в обслуговуванні.

6. Front-running: вразливість, яка виникає у публічних блокчейнах, де всі транзакції спочатку потрапляють у спільний пул очікування (*mempool*), перш ніж потрапити до блоку. Цей пул відкритий для спостереження, тому будь-хто (включно з майнерами або ботами) може побачити, які транзакції плануються до виконання, і подати власну з вищою платою за газ щоб вона була оброблена раніше.

Цей тип атаки особливо поширений у децентралізованих біржах (DEX), аукціонах, DeFi-протоколах або інших контрактах, де результат залежить від порядку виконання транзакцій.

Наведений нижче приклад демонструє ситуацію, в якій зловмисник може отримати перевагу, подавши власну транзакцію раніше користувача:

```
pragma solidity ^0.8.0;

contract Auction {
    address public highestBidder;
    uint public highestBid;

    function bid() public payable {
        require(msg.value > highestBid, "Bid too low");

        if (highestBidder != address(0)) {
            (bool sent, ) = highestBidder.call{value: highestBid}("");
            require(sent, "Refund failed");
        }

        highestBidder = msg.sender;
        highestBid = msg.value;
    }
}
```

Рис.1.8. Приклад вразливого коду

У цьому прикладі зловмисник, спостерігаючи за поданою в мережу транзакцією, що містить ставку, може оперативно відправити свою з дещо вищою сумою та збільшеною комісією за газ. Оскільки майнери обробляють транзакції з вищим gasPrice, транзакція зловмисника буде виконана раніше, і він стане переможцем аукціону навіть якщо зробив ставку пізніше за часом подання.

**Висновки до першого розділу**

У розділі було досліджено фундаментальні поняття, пов’язані з блокчейн-мережами, криптовалютами та смарт-контрактами. Визначено, що блокчейн-технології та цифрові активи стали невід’ємною частиною цифрового простору, викликаючи значний інтерес як з боку інвесторів, так і серед прихильників новітніх технологій.

Проаналізовано принципи функціонування блокчейн-мережі Ethereum, структуру та логіку роботи смарт-контрактів, а також роль користувачів у децентралізованій екосистемі. Окреслено загальні підходи до побудови контрактів мовою Solidity.

Наведено приклади резонансних інцидентів, пов’язаних із втручанням у логіку контрактів, що призвело до значних фінансових збитків як для розробників, так і для користувачів. Окрему увагу приділено найпоширенішим типам

вразливостей, що спостерігаються у практиці, із поясненням причин їх виникнення та можливих векторів атак, які можуть бути використані зловмисниками для компрометації смарт-контрактів.

# 2 ПРАКТИЧНЕ ДОСЛІДЖЕННЯ ІНТСРУМЕНТІВ ДЛЯ СТАТИЧНОГО ТА ДИНАМІЧНОГО АНАЛІЗУ СМАРТ-КОНТРАКТІВ

Для тестування було використано 10 смарт-контрактів, кожен з яких містить щонайменше одну відому вразливість. Контракти було відібрано з відкритої бази даних смарт-контрактів з відомими вразливостями. Цей набір дозволяє здійснити реалістичну оцінку ефективності обраних інструментів у виявленні вразливостей різних категорій.

Нижче наведено перелік контрактів, що використовувалися під час експерименту:

Таблиця 2.1

Список вразливих смарт-контрактів

| Назва | Тип Вразливості |
|---|---|
| Auction | Denial of service |
| EtherLotto | Time manipulation |
| EtherpotLotto | Unchecked low-level calls |
| OverflowSingleTx | Arithmetic |
| OddsandEvens | Front running |
| Rubixi | Access control |
| ShortAddressExample | Short addresses |
| SimpleDao | Reentrancy |
| SmartBillions | Bad randomness |
| Crypto_Roulette | Other |

Для проведення тестування на вразливості вибраних цільових контрактів, були обрані чотири інструменти, що дозволяють виконувати як статичний, так і динамічний аналіз безпеки смарт-контрактів. Серед них:

- Slither
- Mythril

- Echidna
- Manticore

Для забезпечення можливості проведення повноцінного аналізу та запуску обрані інструменти були встановленні на локальній машині з попередньо підготовленим відповідним середовищем. Оскільки всі обрані рішення є командно-рядковими утилітами з відкритим кодом, вони підтримуються у середовищі Linux (дослідження проводилося на базі Ubuntu 20.04).

Суть досліду полягає у послідовному запуску кожного з інструментів (Slither, Mythril, Echidna, Manticore) на кожному тестовому контракті з фіксацією виявлених вразливостей. Після цього результати будуть зіставлені з очікуваними — тобто відомими заздалегідь типами помилок, які дійсно містяться у відповідному смарт-контракті.

Для об’єктивної оцінки результативності кожного інструменту використовуються такі критерії:

- Кількість виявлених вразливостей
- Тип виявленої вразливості
- Опис результату
- Зрозумілість та повнота звіту
- Час виконання аудиту

## 2.1 Аналіз із використанням інструменту Slither

Slither — це статичний аналізатор смарт-контрактів на мові Solidity, розроблений компанією Trail of Bits. Інструмент призначений для швидкої перевірки вихідного коду на наявність відомих шаблонів вразливостей, недоліків стилю програмування, поганих практик або небезпечних конструкцій.

Slither підтримує понад 80 типів перевірок, серед яких: виявлення повторних викликів (reentrancy), невикористаних змінних, помилок видимості, залежностей від блокових змінних (timestamp, block.number), помилок ініціалізації тощо.

Інструмент працює шляхом послідовного аналізу вихідного коду смарт-

контракту, зчитуючи його структуру, логіку виконання функцій, зв'язки між змінними та залежності між елементами. Slither не обмежується поверхневим пошуком ключових слів або фрагментів — він будує цілісну внутрішню модель програми.

Slither формує внутрішню модель контракту на основі дерева синтаксичної структури коду, логіки переходів між функціями та зв'язків між змінними, що дозволяє аналізувати не лише окремі фрагменти, а й взаємодію всіх частин контракту в цілому.

На основі цієї моделі інструмент виконує перевірку коду на відповідність до типових шаблонів вразливостей, таких як повторні виклики, неправильна видимість функцій, залежність від блокових змінних, відсутність перевірки доступу, або ризикові зовнішні виклики.

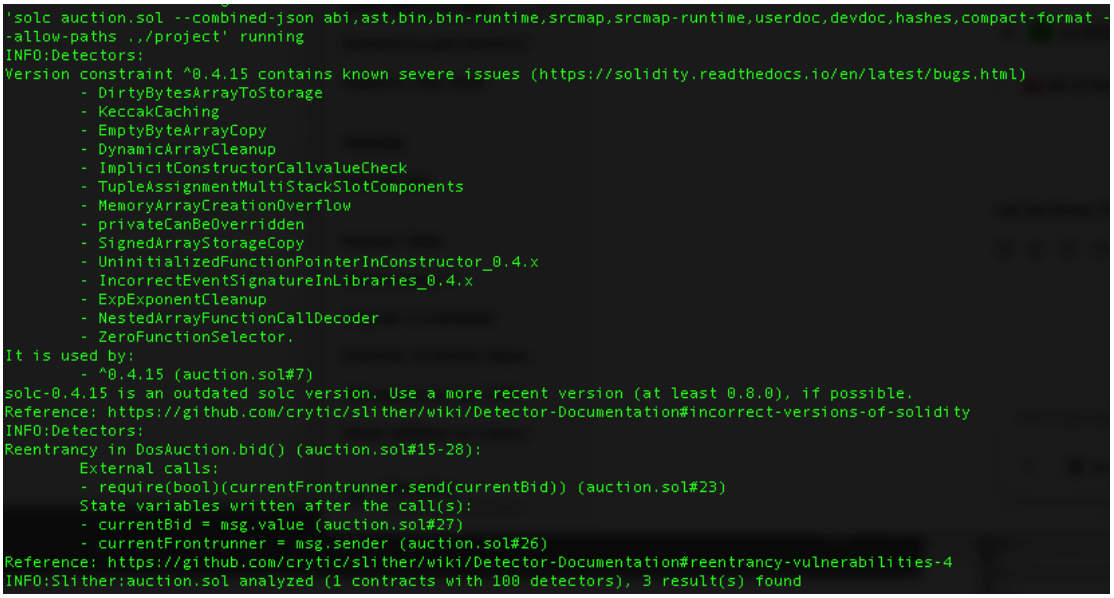


Рис.2.1. Приклад роботи Slither

За допомогою інструменту Slither було проведено аналіз 10 смарт-контрактів, кожен з яких містить відому вразливість. Нижче наведено узагальнені результати виявлених проблем:

Таблиця 2.2

Результати аналізу вразливих смарт-контрактів

| Назва | Кількість виявлених відомих вразливостей | Кількість виявлених не відомих вразливостей | Час виконання аудиту (секунди) |
|---|---|---|---|
| Auction | 1 | 3 | 2 |
| EtherLotto | 0 | 8 | 2 |
| EtherpotLotto | 1 | 10 | 1 |
| OverflowSingleTx | 0 | 2 | 2 |
| OddsandEvens | 0 | 6 | 1 |
| Rubixi | 4 | 15 | 1 |
| ShortAddressExample | 1 | 2 | 2 |
| SimpleDao | 2 | 4 | 2 |
| SmartBillions | 10 | 8 | 7 |
| Crypto_Roulette | 0 | 12 | 1 |

Після проведення аудиту смарт-контрактів було знайдено такі вразливості

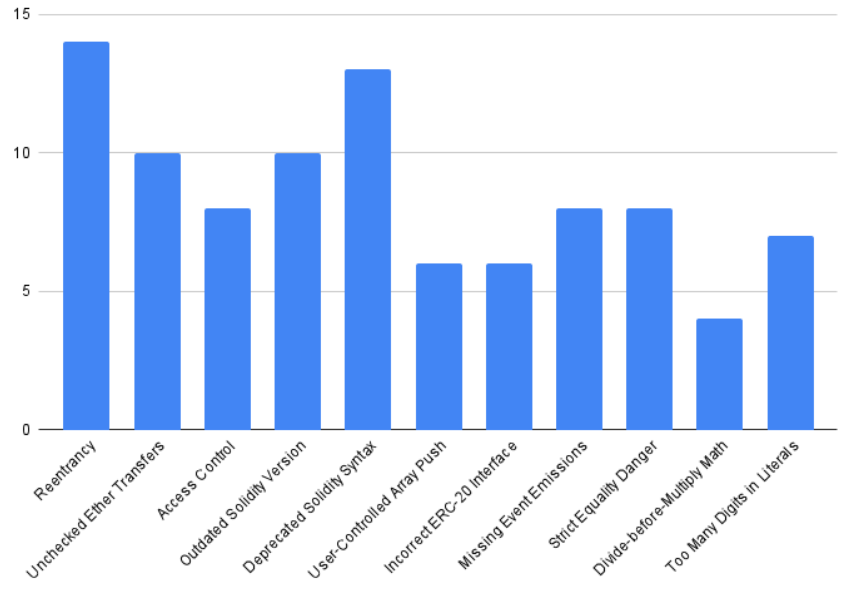


Рис.2.2. Знайдені вразливості інструментом Slither

Аналіз 10 смарт-контрактів за допомогою інструменту Slither показав, що цей

статичний аналізатор ефективно справляється з виявленням відомих вразливостей у коді. У випадках, коли уразливість була чітко вираженою на рівні структури або логіки функцій наприклад, reentrancy або відсутність перевірки прав доступу, інструмент коректно їх ідентифікував. Найбільше відомих проблем було зафіксовано у контрактах rubixi, simpledao та smartbillions.

Крім цього, Slither виявив значну кількість додаткових, неочікуваних попереджень, що не були ціллю тестування, але можуть вказувати на небезпечні шаблони у коді. Наприклад, у контрактах crypto_roulette, etherpot_lotto та smart_billions інструмент зафіксував понад 10 унікальних повідомлень, які не мали прямого відношення до наперед заданої вразливості, проте вказували на потенційно ненадійні ділянки логіки.

Час виконання аналізу був мінімальним — для більшості контрактів він не перевищував 2 секунд. Це дозволяє використовувати Slither як ефективний інструмент для початкової перевірки великих обсягів коду в рамках процесу безпечної розробки.

Таким чином, інструмент Slither демонструє високу швидкість аналізу, точність виявлення шаблонних помилок та здатність знаходити неочевидні ризики, що робить його доцільним для регулярного використання у статичному аудиті смарт-контрактів.

### 2.2 Аналіз із використанням інструменту Mythril

Mythril — це потужний інструмент для статичного аналізу смарт-контрактів на мові Solidity, який використовує методи символічного виконання, аналізу потоку даних та модельного дослідження поведінки контракту. Основне його завдання — виявляти логічні та безпекові вразливості на рівні байткоду або вихідного коду, ще до розгортання контракту в мережі.

На відміну від інструментів, що базуються виключно на синтаксичному аналізі, Mythril моделює виконання контракту у різних умовах, перевіряючи можливі сценарії взаємодії з ним. Це дозволяє виявляти вразливості, які залежать від комбінацій дій користувачів, послідовності викликів функцій, умов виконання

тощо.

Mythril здійснює аналіз байткоду контракту, використовуючи алгоритми символічного виконання — тобто симулює виконання функцій з абстрактними значеннями, не прив’язаними до конкретних параметрів. Це дає змогу виявити всі можливі варіанти поведінки контракту, які можуть призвести до порушення цілісності або логіки.

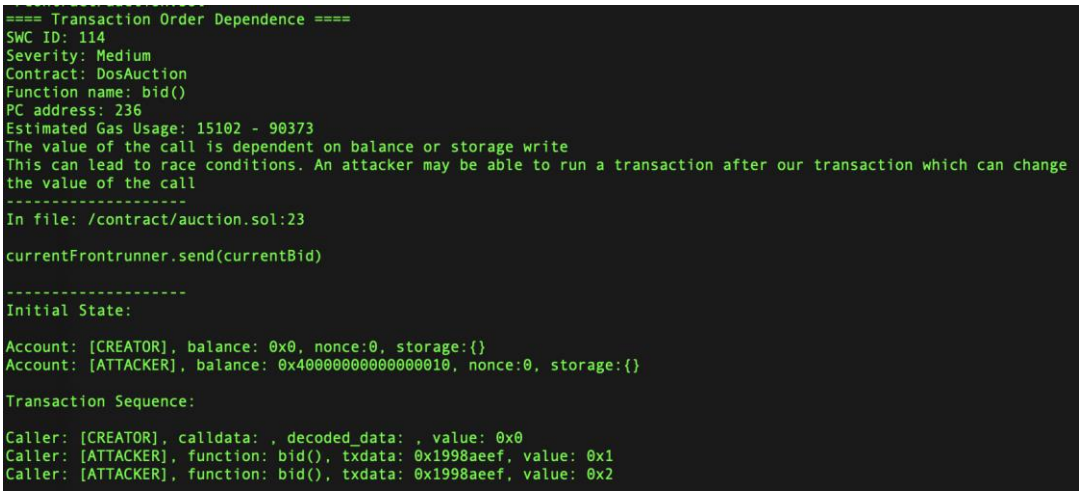


Рис.2.3. Приклад роботи Mythril

У процесі виконання аналізу інструмент Mythril формує структурований звіт у текстовому форматі, який відображається безпосередньо в командному рядку. Результати звіту містять:

- назву виявленої вразливості (наприклад, Reentrancy);
- функцію, в якій було виявлено проблему (Auction.bid()), із зазначенням відповідних рядків коду;
- послідовність операцій, які призводять до виникнення вразливого стану;
- опис змін, що відбуваються у змінних (currentBid, currentFrontrunner);
- посилання на документацію з описом вразливості.

За допомогою інструменту Slither було проведено аналіз 10 смарт-контрактів, кожен з яких містить відому вразливість. Нижче наведено узагальнені результати виявлених проблем:

Таблиця 2.3

Результати аналізу вразливих смарт-контрактів

| Назва | Кількість знайдених відомих вразливостей | Кількість знайдених не відомих вразливостей | Час виконання аудиту (секунди) |
|---|---|---|---|
| Auction | 1 | 2 | 5 |
| EtherLotto | 0 | 2 | 4 |
| EtherpotLotto | 1 | 1 | 4 |
| OverflowSingleTx | 1 | 1 | 3 |
| OddsandEvens | 0 | 2 | 1 |
| Rubixi | 1 | 2 | 5 |
| ShortAddressExample | 1 | 2 | 3 |
| SimpleDao | 1 | 2 | 4 |
| SmartBillions | 1 | 3 | 15 |
| Crypto_Roulette | 0 | 1 | 3 |

Після проведення аудиту смарт-контрактів було знайдено такі вразливості

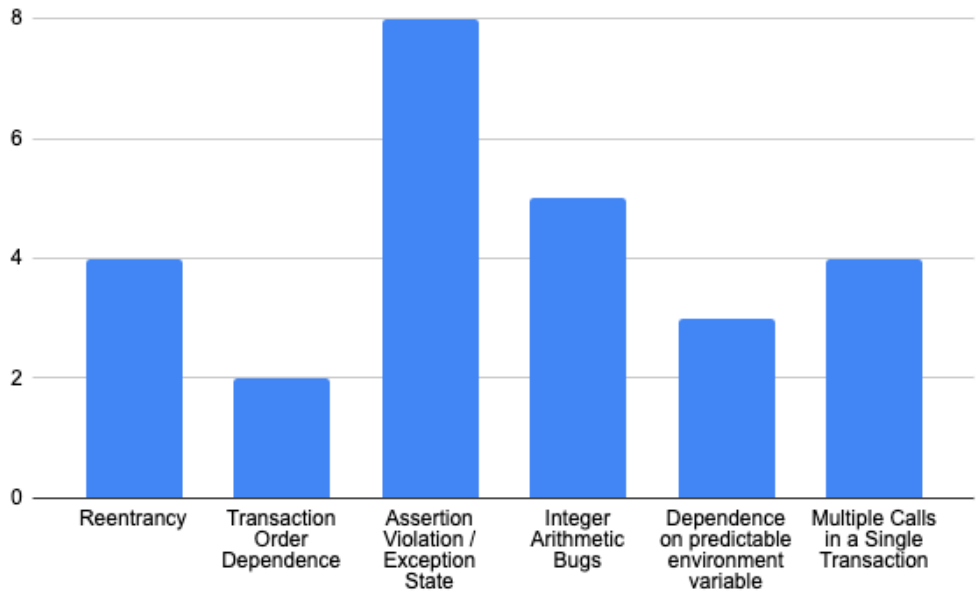


Рис.2.4. Знайдені вразливості інструментом Mythril

Аналіз 10 смарт-контрактів за допомогою інструменту Mythril показав, що цей засіб статичного аналізу ефективно виявляє як відомі, так і додаткові логічні вразливості у коді. У випадках, де вразливість була пов’язана із повторними викликами, некоректною арифметикою або порушенням логіки виконання, інструмент коректно ідентифікував проблему. Найбільшу кількість відомих вразливостей було зафіксовано у контрактах Auction, OverflowSingleTx, SimpleDao, SmartBillions і Rubixi.

Окрім очікуваних, Mythril також виявив низку **додаткових помилок**, які не входили до переліку відомих вразливих місць, але потенційно загрожують коректній роботі контракту.

Час виконання аудиту був дещо вищим порівняно з попереднім інструментом Slither, що пояснюється складністю методу символічного виконання. У середньому аналіз одного контракту займав 3–6 секунд, для складних кейсів — до 15 секунд.

Таким чином, інструмент Mythril продемонстрував свою ефективність як інструмент глибокого логічного аналізу. Його використання є доцільним у випадках, коли необхідно перевірити не лише структуру, але й повну модель поведінки контракту в умовах змінних сценаріїв виконання.

### 2.3 Аналіз із використанням інструменту Echidna

Echidna — це інструмент для динамічного аналізу смарт-контрактів, який реалізує техніку фузз-тестування. Його основне завдання — автоматично й багаторазово викликати функції контракту з випадковими або напіввипадковими параметрами, щоб виявити логічні помилки, неочікувану поведінку та ситуації, що можуть призвести до порушення інваріантів. На відміну від статичних аналізаторів, Echidna працює на основі фактичного виконання коду та спостерігає за його реакцією в умовах, максимально наближених до реального середовища.

Інструмент побудовано на мові Haskell і призначено для інтеграції з тестовими контрактами, у яких задано специфічні умови, що мають залишатися істинними при будь-яких запусках. Якщо умова порушується — це свідчить про

вразливість або логічну помилку. Таким чином, Echidna є ефективним способом виявлення вразливостей, які можуть залишитися непоміченими при звичайному ручному або навіть статичному аналізі.

Принцип роботи Echidna базується на створенні спеціальних тестових функцій, що перевіряють виконання визначених умов (так званих інваріантів), які повинні залишатися істинними при будь-якому сценарії виконання. Після запуску інструмент здійснює випадкову генерацію вхідних параметрів до функцій смарт-контракту і багаторазово викликає ці функції з новими значеннями. Якщо в результаті тестування інваріант порушується — це вважається ознакою наявності вразливості або небажаної поведінки.

Пошук вразливих станів здійснюється через фузз-тестування: кожна сесія Echidna — це тисячі або навіть мільйони спроб взаємодії з контрактом у змодельованому середовищі. При виявленні помилки інструмент виводить точну послідовність дій, що призвела до збою, включно з функціями, параметрами виклику та змінними стану контракту. Це дозволяє розробникам відтворити ситуацію та усунути помилку ще до розгортання в блокчейн-мережу.

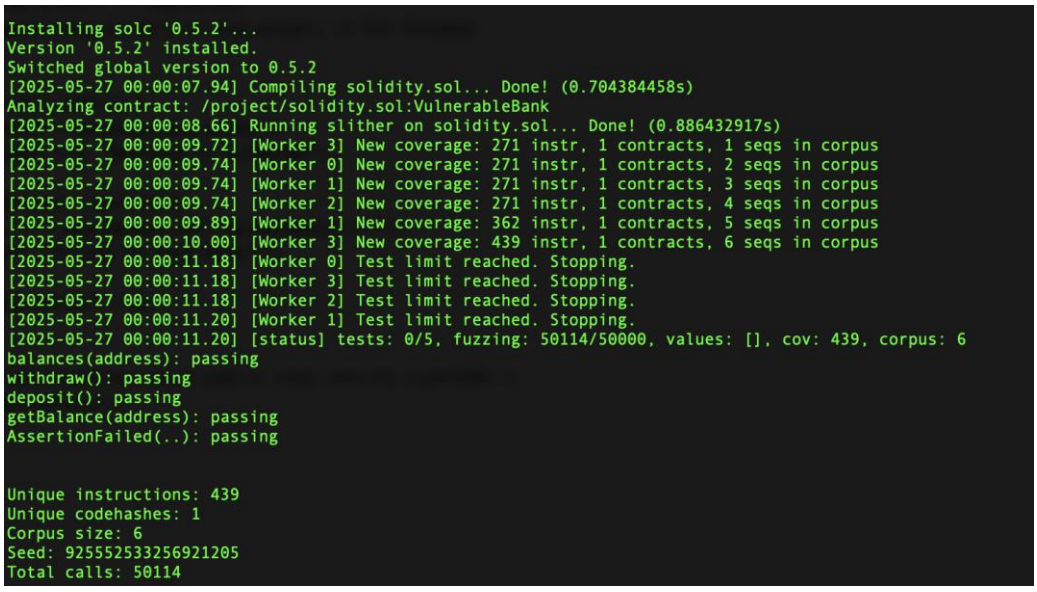


Рис.2.5. Приклад роботи Echinda

За допомогою інструменту Echinda було проведено аналіз 10 смарт-контрактів, кожен з яких містить відому вразливість. Нижче наведено узагальнені результати виявлених проблем:

Таблиця 2.4

Результати аналізу вразливих смарт-контрактів

| Назва | Кількість знайдених відомих вразливостей | Кількість знайдених не відомих вразливостей | Час виконання аудиту (секунди) |
|---|---|---|---|
| Auction | 1 | 1 | 3 |
| EtherLotto | 0 | 3 | 3 |
| EtherpotLotto | 0 | 1 | 2 |
| OverflowSingleTx | 1 | 2 | 3 |
| OddsandEvens | 0 | 1 | 4 |
| Rubixi | 0 | 3 | 5 |
| ShortAddressExample | 1 | 1 | 3 |
| SimpleDao | 1 | 2 | 2 |
| SmartBillions | 0 | 2 | 7 |
| Crypto_Roulette | 0 | 3 | 4 |

Після проведення аудиту смарт-контрактів було знайдено такі вразливості:

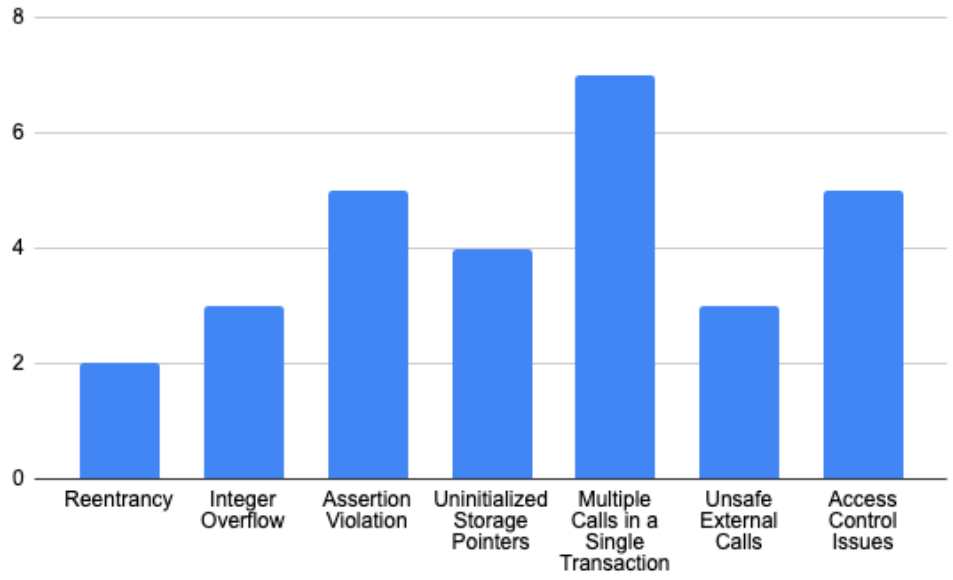


Рис.2.6. Знайдені вразливості інструментом Echinda

Аналіз 10 смарт-контрактів за допомогою інструменту Echidna підтвердив ефективність динамічного фузз-тестування для виявлення логічних помилок та нестійких сценаріїв виконання. Незважаючи на те, що інструмент не завжди виявляв заздалегідь відомі вразливості, він успішно виявив велику кількість додаткових проблем, які не були закладені у вихідні умови тестування.

Найбільше відомих вразливостей було зафіксовано у контрактах Auction, OverflowSingleTx, ShortAddressExample та SimpleDao. У той же час, значну кількість неочікуваних або непрямих вразливостей виявлено у Rubixi, CryptoRoulette та EtherLotto.

Середній час виконання аналізу для одного контракту становив від 2 до 5 секунд, залежно від складності логіки та кількості протестованих інваріантів. Найбільше часу вимагав контракт SmartBillions — до 7 секунд.

Таким чином, інструмент Echidna виявив себе як дієвий засіб для виявлення помилок, пов’язаних із поведінкою контракту в реальному середовищі. Завдяки фузз-тестуванню, цей інструмент дозволяє перевірити стійкість логіки виконання до великої кількості вхідних даних, що робить його незамінним на етапі тестування перед розгортанням смарт-контрактів у публічних мережах.

## 2.4 Аналіз із використанням інструменту Manticore

Manticore — це потужний інструмент динамічного аналізу, що застосовує метод символічного виконання для виявлення вразливостей у смарт-контрактах на рівні байткоду. Основною перевагою Manticore є здатність моделювати всі можливі шляхи виконання контракту, досліджуючи, чи можуть певні вхідні дані або комбінації дій призвести до небажаного стану.

Інструмент працює в умовах ізольованого середовища, симулюючи взаємодію користувачів із контрактом, створення нових транзакцій, зміну стану та ефекти зовнішніх викликів. Manticore виконує аналіз байткоду на рівні Ethereum Virtual Machine (EVM), що дозволяє йому працювати з будь-якими контрактами, незалежно від мови написання.

На відміну від Echidna, яка тестує контракти з великою кількістю випадкових параметрів, Manticore застосовує символічне виконання — тобто замість конкретних значень підставляє умовні змінні та логічно перевіряє всі можливі варіанти проходження коду. Якщо інструмент виявляє шлях, який може призвести до помилки, переповнення, відсутності перевірки або зміни стану без належного контролю — такий шлях заноситься до звіту як потенційна вразливість.

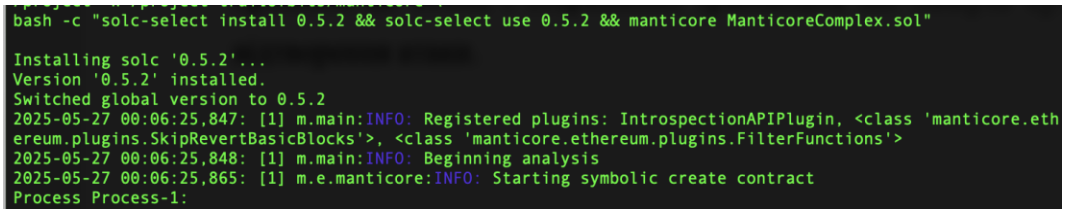


Рис.2.7. Знайдені вразливості інструментом Manticore

За допомогою інструменту Manticore було проведено аналіз 10 смарт-контрактів, кожен з яких містить відому вразливість. Нижче наведено узагальнені результати виявлених проблем

Таблиця 2.5

Результати аналізу вразливих смарт-контрактів

| Назва | Кількість знайдених відомих вразливостей | Кількість знайдених не відомих вразливостей | Час виконання аудиту (секунди) |
|---|---|---|---|
| Auction | 2 | 2 | 5 |
| EtherLotto | 0 | 2 | 6 |
| EtherpotLotto | 1 | 3 | 4 |
| OverflowSingleTx | 0 | 3 | 5 |
| OddsandEvens | 1 | 0 | 6 |
| Rubixi | 2 | 3 | 4 |
| ShortAddressExample | 1 | 2 | 4 |
| SimpleDao | 1 | 2 | 5 |
| SmartBillions | 1 | 1 | 10 |
| Crypto_Roulette | 0 | 2 | 3 |

Після проведення аудиту смарт-контрактів було знайдено такі вразливості:

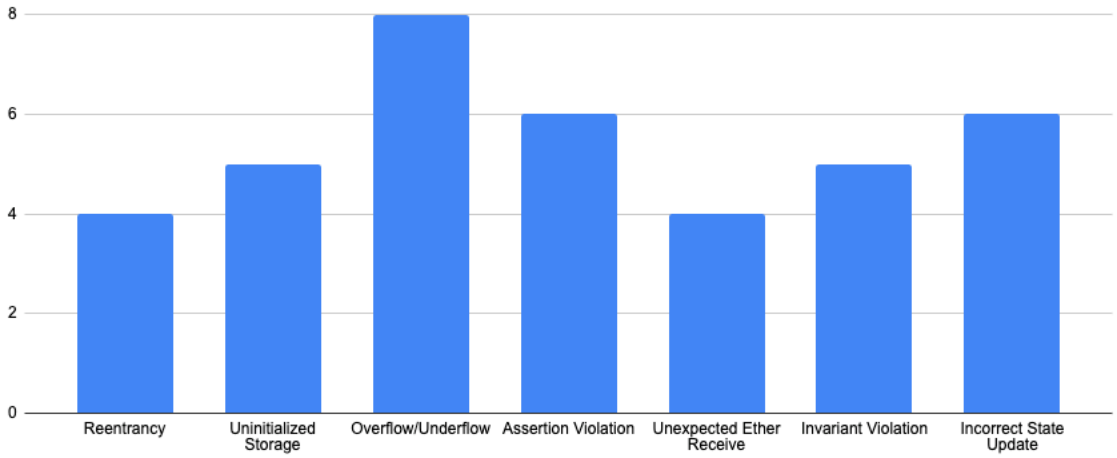


Рис.2.8. Знайдені вразливості інструментом Manticore

Аналіз 10 смарт-контрактів за допомогою інструменту Manticore засвідчив його високу ефективність у виявленні складних логічних вразливостей, які пов’язані зі змінами стану, глибокими викликами функцій та умовними переходами. Завдяки застосуванню символічного виконання, інструмент дозволяє дослідити всі можливі шляхи виконання коду, включно з рідкісними або важкодоступними логічними гілками.

Найбільше відомих вразливостей було виявлено у контрактах Auction, Rubixi, ShortAddressExample та OddsandEvens. У контрактах EtherpotLotto, OverflowSingleTx, Rubixi, SimpleDao і CryptoRoulette також зафіксовано значну кількість неочікуваних проблем — зокрема помилки в оновленні стану, порушення інваріантів або неконтрольовані передачі ефіру.

Час виконання аналізу коливався в межах від 3 до 6 секунд у більшості випадків. Найскладніший контракт вимагав до 10 секунд перевірки через велику кількість логічних гілок.

Таким чином, Manticore продемонстрував високу точність і глибину аналізу, що дозволяє йому виявляти як прості шаблонні, так і складні умовно-залежні вразливості. Інструмент є доцільним для використання на етапах поглибленого тестування, зокрема при аудиті критично важливих контрактів, де безпека має пріоритетне значення.

## Висновки до другого розділу

У другому розділі було проведено практичне дослідження чотирьох автоматизованих інструментів для аналізу безпеки смарт-контрактів: Slither, Mythril, Echidna та Manticore.

Для перевірки ефективності обраних рішень було використано 10 тестових смарт-контрактів, кожен із яких містить щонайменше одну відому вразливість, а також потенційні логічні помилки. Це дозволило оцінити як здатність інструментів виявляти конкретні відомі уразливості, так і їхню ефективність у виявленні нових, неочевидних ризиків.

# 3 РОЗРОБКА РЕКОМЕНДАЦІЙ ЩОДО ПІДВИЩЕННЯ БЕЗПЕКИ СМАРТ-КОНТРАКТІВ ТА ЗАПРОВАДЖЕННЯ SSDLC

У ході проведеного практичного дослідження було здійснено аналіз чотирьох популярних інструментів безпеки для смарт-контрактів: Slither, Mythril, Echidna та Manticore. Для тестування було обрано 10 контрактів з відомими вразливостями, що дало змогу оцінити як точність, так і ефективність кожного з інструментів при виявленні потенційних ризиків.

Результати засвідчили, що жоден з інструментів не забезпечує повного охоплення всіх типів помилок, однак комбіноване використання статичного та динамічного аналізу дозволяє досягти найкращого охоплення ідентифікації вразливостей.

- Slither продемонстрував високу швидкість і ефективність при виявленні типових вразливостей, таких як reentrancy, невірна видимість функцій та недоліки стилю коду.
- Mythril забезпечив глибше логічне моделювання, дозволяючи виявити помилки, які пов’язані зі специфікою поведінки контракту при різних сценаріях.
- Echidna, що реалізує фузз-тестування, дозволив знайти численні помилки, які виникають під час реальної взаємодії з контрактами.
- Manticore, завдяки символічному виконанню, показав найвищу здатність до виявлення складних логічних залежностей і неочевидних шляхів виконання, які можуть бути використані зловмисниками.

Таблиця 3.1

| Критерій | Slither | Mythril | Echidna | Manticore |
|---|---|---|---|---|
| Тип аналізу | Статичний | Статичний | Динамічний | Динамічний |
| Рівень охоплення коду | Високий (структурний рівень) | Високий (структурний рівень) | Високий (структурний рівень) | Високий (структурний рівень) |
| Виявлення відомих вразливостей | Добре | Високо | Залежить від інваріантів | Високо |
| Виявлення неочевидних помилок | Обмежено | Частково | Добре | Відмінно |
| Швидкість роботи | Висока | Середня | Висока | Середньо-низька |
| Вивід результатів | Зручний CLI-звіт | Деталізовані треки виконання | Інваріант + вхідні дані | Повні сценарії з контекстом |
| Простота налаштування | Легка | Середня | Складна | Складна |
| Ціль використання | Початковий аудит | Глибокий логічний аналіз | Масове тестування логіки | Формальна перевірка безпеки |

Кожен із розглянутих інструментів має свої сильні сторони та найбільш придатний для певного типу аналізу. Slither забезпечує швидкий базовий огляд коду, Mythril дозволяє виявляти складні логічні помилки, Echidna ефективний для тестування інваріантів у поведінці, а Manticore демонструє найглибший рівень формального аналізу.

Комплексне використання декількох інструментів дозволяє підвищити повноту перевірки та досягти вищого рівня впевненості у безпечності смарт-контракту.

## 3.1 Формування рекомендацій щодо безпечної розробки смарт-контрактів

**Reentrancy:** Для усунення цієї вразливості необхідно дотримуватись шаблону "Checks-Effects-Interactions" — спочатку перевіряти умови, потім оновлювати стан, і лише після цього здійснювати зовнішні виклики.

Такий підхід запобігає повторному виклику функції до того, як завершиться її попереднє виконання. Крім того, рекомендується реалізувати механізм блокування повторного виклику за допомогою логічної змінної та спеціального модифікатора.

Це дозволяє заблокувати повторний доступ до критичної функції під час її виконання, що унеможливлює багаторазовий зловмисний виклик:

```
bool internal locked;
modifier nonReentrant() {
   require(!locked, "Reentrant call");
   locked = true;
   _;
   locked = false;
}
function withdraw() external nonReentrant {}
```

**Access Control**: Забезпечити захист від несанкціонованого доступу до функцій можна за допомогою впровадження чіткої системи контролю прав. Функції, які змінюють стан контракту або керують фінансами, не повинні бути доступними для будь-кого.

Для цього доцільно застосовувати модифікатори на кшталт onlyOwner, що дозволяють виклик функцій лише власником контракту, або використовувати бібліотеки типу OpenZeppelin AccessControl, які забезпечують гнучке управління роля:

```
import "@openzeppelin/contracts/access/Ownable.sol";

contract Example is Ownable {
   function changeAdminData() public onlyOwner {
      // лише власник може викликати
   }
}
```

**Insecure Randomness**: Для уникнення вразливостей, пов’язаних із передбачуваною ентропією, не слід використовувати блокові змінні на кшталт block.timestamp або block.number, оскільки вони можуть бути частково контрольовані майнерами.

Такий підхід створює ризики для лотерей, ігор або інших контрактів, де необхідна випадковість. Рекомендовано інтегрувати зовнішні сервіси генерації випадкових чисел, такі як Chainlink VRF, які забезпечують криптографічно безпечну ентропію, недоступну для маніпулювання:

```
requestRandomWords(...);
function fulfillRandomWords(...) internal override {
   uint random = randomWords[0] % maxValue;
}
requestRandomWords(...);
function fulfillRandomWords(...) internal override {
   uint random = randomWords[0] % maxValue;
```

}

**Integer Over/Underflow:** Для запобігання арифметичним переповненням необхідно використовувати компілятор Solidity версії 0.8.0 або вище, у якому реалізовано автоматичну перевірку переповнень.

Це дозволяє уникнути помилок, коли додавання або віднімання змінної призводить до некоректного переходу через межі типу. У старших версіях обов'язковим є підключення бібліотеки SafeMath, яка реалізує додаткову перевірку коректності арифметичних операцій.

```
uint256 public total;
function add(uint256 value) public {
   total += value; // автоматичні перевірки з 0.8.0
}
```

**Low-level Calls:** Щоб уникнути помилок при роботі з низькорівневими викликами call, слід завжди перевіряти значення success після виклику. Ігнорування результату виконання може призвести до того, що логіка контракту продовжить виконання після невдалого переказу коштів або виклику іншого контракту.

Використання delegatecall варто уникати повністю, якщо воно не є строго необхідним, оскільки воно виконує чужий код у контексті поточного контракту, що є надзвичайно ризикованим.

```
(bool success, ) = recipient.call{value: msg.value}("");
require(success, "Transfer failed");
```

**Uninitialized Storage:** Щоб уникнути неконтрольованої поведінки змінних сховища, слід чітко ініціалізувати їх при створенні або у конструкторі. Важливо коректно вказувати область пам’яті для змінних (storage, memory), особливо у функціях, які оперують структурами або масивами.

Невизначені покажчики можуть призводити до запису в неочікувані області сховища та компрометації даних контракту.

```
struct User {
    uint balance;
}
User public user = User(0);
struct User {
    uint balance;
}

User public user = User(0);
```

**Invariant Violations:** Для запобігання порушенню логіки виконання контракту необхідно реалізовувати перевірки за допомогою конструкції require, яка перевіряє вхідні параметри функцій та інші критичні умови. Також слід використовувати assert для перевірки внутрішніх інваріантів, які за логікою не повинні порушуватись.

Ці механізми дозволяють одразу припинити виконання функції у випадку, коли її логіка або стан змінюється у небажаний спосіб:

```
function withdraw(uint amount) public {
    require(amount <= balances[msg.sender], "Insufficient balance");
    balances[msg.sender] -= amount;
    payable(msg.sender).transfer(amount);
    assert(balances[msg.sender] >= 0);
}
```

Упровадження описаних практик дозволяє суттєво знизити ризики експлуатації вразливостей, забезпечити передбачувану логіку роботи контракту та підвищити довіру до проєкту з боку користувачів і сторонніх аудиторів.

### 3.2 Впровадження практик SSDLC

Забезпечення безпеки смарт-контрактів неможливе без впровадження системного підходу до розробки, відомого як Secure Software Development Lifecycle (SSDLC). Ця концепція передбачає включення заходів кібербезпеки на всіх етапах

життєвого циклу програмного забезпечення — від початкового проєктування до супроводу після розгортання. У контексті Web3-проєктів SSDLC набуває особливого значення, оскільки помилки в смарт-контрактах зазвичай призводять до незворотних фінансових втрат.

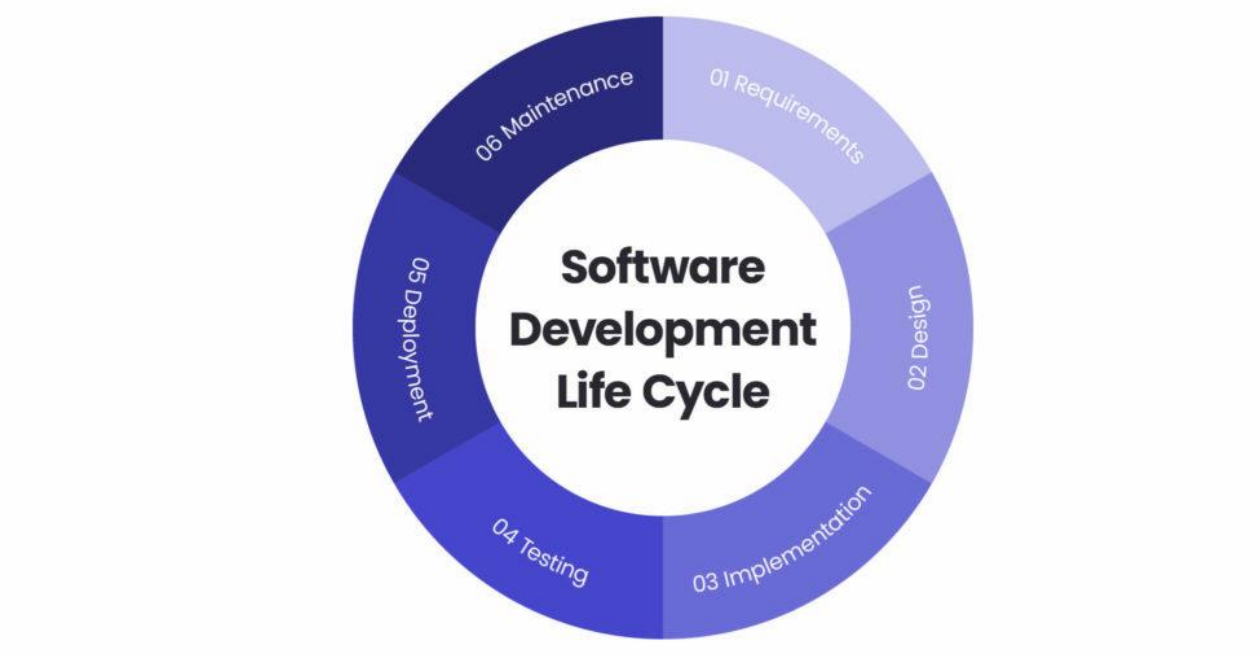


Рис.3.1. Схематичне зображення процесу SSDLC

1. **Планування**

На цьому етапі визначаються цілі контракту, логіка його функціонування, ролі користувачів і потенційні ризики. Важливо одразу врахувати можливі загрози, сценарії зловживання та критичні точки логіки. Рекомендується складати модель загроз і перелік функціональних вимог до безпеки.

2. **Проєктування**

На етапі проєктування визначається архітектура смарт-контракту, структури даних, логіка доступу та механізми перевірок. Варто уникати надмірної складності, передбачити обмеження на зовнішні виклики та перевірку вхідних даних. Бажано використовувати стандартизовані шаблони наприклад, OpenZeppelin Contracts та дотримуватись принципів мінімізації прав.

3. **Розробка**

У процесі реалізації коду потрібно дотримуватись принципів безпечного

програмування: оновлювати стан перед зовнішніми викликами, уникати delegatecall, обробляти помилки, перевіряти всі зовнішні вхідні дані та накладати обмеження на доступ до критичних функцій. Також важливо одразу реалізовувати інваріанти через конструкції require та assert, закладати граничні значення, таймлайни та логіку винятків.

**4. Тестування**

На цьому етапі слід застосовувати як юніт-тести, так і автоматизовані інструменти статичного й динамічного аналізу (наприклад, Slither, Mythril, Echidna, Manticore). Для виявлення логічних помилок рекомендується використовувати фреймворки типу Foundry або Hardhat, а також fuzz-тестування та тестування інваріантів.

**5. Розгортання**

Перед розгортанням контракту в основній мережі необхідно провести незалежний аудит безпеки або принаймні peer-review. Варто використовувати механізми захищеного розгортання, зокрема мультипідпис, валідацію адрес, обмеження доступу в перші хвилини після публікації. Також рекомендується додати можливість призупинення та поступове оновлення логіки через проксі-контракти, якщо це передбачено.

**6. Супровід**

Після запуску контракту важливо моніторити його активність, перевіряти події, контролювати баланси та реагувати на підозрілу активність. Слід впровадити регулярні ревізії логіки, реагування на звіти від спільноти та оновлення залежностей. У разі можливості — мати план міграції або заміни контракту.

Таким чином, SSDLC дозволяє системно підійти до питання безпеки та мінімізувати ймовірність критичних помилок ще до моменту експлуатації контракту. Його впровадження в процес розробки Web3-проєктів є важливим кроком до зрілої та відповідальної інженерної практики.

### 3.3 Загальні рекомендації щодо безпеки смарт-контрактів

На завершення аналітичної та експериментальної частин дослідження можна

сформулювати низку загальних практичних рекомендацій, які доцільно враховувати під час розробки смарт-контрактів на мові Solidity. Ці рекомендації базуються на виявлених проблемах, типових помилках програмування, а також на ефективності інструментів аналізу, розглянутих у попередніх розділах.

1. Дотримання принципів безпечного програмування є критично важливим на всіх етапах життєвого циклу контракту. Сюди входить завчасне оновлення стану перед зовнішніми викликами, перевірка вхідних даних, уникнення небезпечних конструкцій (наприклад, delegatecall), та чітке обмеження прав доступу до функцій.

2. Застосування інструментів автоматизованого аналізу коду повинно стати стандартною практикою в рамках розробки. Поєднання статичного та динамічного аналізу дозволяє виявити широкий спектр вразливостей, включно з тими, які неможливо виявити за допомогою одного типу перевірки.

3. Повноцінне тестування смарт-контрактів повинно включати не лише юніт-тести, але й fuzz-тестування, симуляції та перевірку інваріантів. Це дозволяє протестувати поведінку контракту при великій кількості вхідних сценаріїв, у тому числі граничних і неочікуваних.

4. Впровадження концепції Secure Software Development Lifecycle (SSDLC) має забезпечити безпеку контракту ще на етапі планування та проєктування. Інтеграція безпеки на кожному з етапів розробки дозволяє виявити та виправити помилки до моменту їх розгортання в основній мережі.

5. Незалежна перевірка коду та зовнішній аудит мають розглядатися не як додатковий крок, а як обов’язковий елемент перед публікацією. Peer-review, участь незалежних фахівців, а також відкритість до зворотного зв’язку від спільноти — усе це значно підвищує рівень безпеки.

Таким чином, дотримання цих рекомендацій дає змогу суттєво зменшити ризики, пов’язані з помилками в логіці, неправильним використанням механізмів Solidity або недооцінкою потенційних атак.

# ВИСНОВКИ

У бакалаврській роботі отримано наступні теоретичні та практичні результати:

1. Розглянуто принципи функціонування блокчейн-мереж, особливості роботи Ethereum Virtual Machine та структуру смарт-контрактів на мові Solidity. Проаналізовано логіку їх виконання, базові складові акаунтів і приклади застосування в реальних Web3-проєктах. Висвітлено відомі хакерські атаки The DAO, Wormhole, що призвели до значних фінансових втрат.
2. Досліджено найпоширеніші вразливості смарт-контрактів, класифіковані за DASP Top 10, та проаналізовано потенційні вектори атак. Описано механізми, що можуть призвести до несанкціонованого доступу, переповнення, повторних викликів, маніпуляцій із генерацією випадкових чисел та інші.
3. Проведено практичне дослідження чотирьох інструментів автоматичного аналізу безпеки Slither, Mythril, Echidna, Manticore на прикладі 10 вразливих контрактів. Здійснено порівняння їхньої ефективності, швидкості та точності виявлення помилок. Наведено приклади типових звітів та результати роботи кожного засобу.
4. Надано практичні рекомендації щодо усунення основних вразливостей: захист від повторних викликів, обмеження доступу, безпечна робота з арифметикою, обробка викликів, коректна ініціалізація змінних. Запропоновано кращі підходи до перевірки стану контракту, організації тестування та підвищення стійкості до типових помилок.
5. Розглянуто впровадження безпечного життєвого циклу розробки смарт-контрактів (SSDLC) як частину загального процесу проєктування Web3-систем. Визначено переваги інтеграції безпеки на всіх етапах — від планування до супроводу. Обґрунтовано необхідність автоматизованого тестування, peer-review та аудиту перед розгортанням контракту.

Таким чином, результати роботи свідчать про актуальність теми дослідження

та дозволяють сформувати системний підхід до створення безпечних смарт-контрактів, що сприяє підвищенню довіри до блокчейн-рішень і технологій у цілому.

# ПЕРЕЛІК ПОСИЛАНЬ


1. What is Blockchain | IBM URL: https://www.ibm.com/think/topics/blockchain
2. 3 Famous Smart Contract Hacks You Should Know URL: https://medium.com/firmonetwork/3-famous-smart-contract-hacks-you-should-know-dffa6b934750
3. Ethereum Virtual Machine (EVM) URL: https://ethereum.org/en/developers/docs/evm/
4. Slither: A Leading Static Analyzer for Smart Contracts URL: https://101blockchains.com/slither-tutorial/
5. A Practical Guide to Smart Contract Security Tools. Part 3: Mythril URL: https://mixbytes.io/blog/guide-smart-contract-security-tools-mythril
6. Fuzzing smart-contracts practical aspects: Echidna URL: https://mixbytes.io/blog/fuzzing-smart-contracts-practical-aspects-echidna
7. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts https://agroce.github.io/ase19.pdf
8. Solidity Best Practices: A Beginner’s Guide to Writing Secure Smart Contracts URL: https://coinsbench.com/solidity-best-practices-a-beginners-guide-to-writing-secure-smart-contracts-0d5f65887857
9. Solidity Security: Comprehensive list of known attack vectors and common anti-patterns URL: https://blog.sigmaprime.io/solidity-security.html
10. Smart Contract Weakness Classification URL: https://swcregistry.io/
11. DASP - TOP 10 URL: https://dasp.co/
12. Secure Software Development Lifecycle (SSDLC) – why is it important? URL: https://spyro-soft.com/blog/cybersecurity/secure-software-development-lifecycle-ssdlc-why-is-it-important

# ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

**(Презентація)**