**Лабораторна робота №5: Децентралізовані фінанси (DeFi) та безпека смартконтрактів**

**1. Мета роботи**

Вивчити архітектуру децентралізованих фінансових протоколів, механіку роботи автоматизованих маркет-мейкерів (AMM) та навчитися ідентифікувати й виправляти критичні вразливості у смартконтрактах Solidity.

**2. Теоретична частина**

**2.1. Екосистема DeFi та "Фінансові лего"**

Децентралізовані фінанси (DeFi) — це система фінансових інструментів у формі смартконтрактів. Головною особливістю є **композитність (composability)**, яку часто називають "фінансовими лего". Це можливість різних протоколів взаємодіяти один з одним без дозволу, створюючи складні ланцюжки операцій (наприклад, отримання позики в одному протоколі для забезпечення ліквідності в іншому).

**DEX (Decentralized Exchanges)** — це біржі, що працюють без централізованого посередника. На відміну від CEX (Binance, Kraken), де використовується книга ордерів (Order Book), більшість DEX базуються на пулах ліквідності.

**2.2. Механіка AMM (Automated Market Makers)**

Більшість DEX (як-от Uniswap v2) використовують формулу постійного продукту:

Де:

- — кількість токенів A в пулі.
- — кількість токенів B в пулі.
- — постійна величина (invariant), яка не повинна зменшуватися після виконання обміну.

**Постачальники ліквідності (LP)** вносять обидва активи в рівній пропорції, за що отримують LP-токени та частину комісій від торгів.

**2.3. Аналіз критичних вразливостей**

**Reentrancy (Повторний вхід)**

Це класична атака, яка стала причиною краху **The DAO**. Вона виникає, коли контракт надсилає кошти зовнішньому користувачу (або іншому контракту) до того, як оновити свій стан (наприклад, баланс користувача). Зловмисник може

створити контракт з функцією-колбеком, яка знову викликає функцію виведення коштів, поки перша транзакція ще не завершена.

**Overflow та Underflow**

Раніше (до Solidity 0.8.0) арифметичні операції могли призводити до переповнення. Наприклад, якщо до змінної uint8 зі значенням 255 додати 1, вона ставала рівною 0. Починаючи з версії 0.8.x, Solidity має вбудовану перевірку на ці помилки, що робить використання бібліотеки SafeMath необов'язковим, але логіку обчислень все одно потрібно ретельно перевіряти.

**Front-running та MEV**

**Front-running** — це маніпуляція порядком транзакцій. Оскільки транзакції в мемпулі публічні, зловмисник (або бот) може виставити вищу комісію (Gas Price), щоб його транзакція була оброблена раніше за вашу (наприклад, щоб купити актив перед вашим великим замовленням і перепродати його вам дорожче). Це частина концепції **MEV (Maximal Extractable Value)**.

**2.4. Стандарти безпеки**

Використання перевірених бібліотек, як-от **OpenZeppelin**, є золотим стандартом:

- Ownable: керування правами доступу.
- ReentrancyGuard: захист від атак повторного входу.
- ERC20 / ERC721: безпечні реалізації токенів.

**3. Практична частина**

**Завдання 1: Аналіз вразливого контракту**

Нижче наведено код контракту "Банк", який має критичну помилку. Студенту необхідно проаналізувати функцію withdraw та пояснити, чому вона вразлива до Reentrancy.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;


contract VulnerableBank {

    mapping(address => uint256) public balances;


    function deposit() public payable {
```

```
        balances[msg.sender] += msg.value;
    }

    function withdraw() public {
        uint256 amount = balances[msg.sender];
        require(amount > 0, "Insufficient balance");

        // ВРАЗЛИВІСТЬ: Надсилання коштів відбувається ДО оновлення балансу
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");

        balances[msg.sender] = 0;
    }
}
```

**Завдання 2: Виправлення вразливості**

Виправте контракт із Завдання 1 двома способами:

1. Використовуючи патерн **Checks-Effects-Interactions**.
2. Використовуючи модифікатор nonReentrant від OpenZeppelin.

*Приклад структури захищеного контракту:*

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract SecureBank is ReentrancyGuard {
    mapping(address => uint256) public balances;

    function withdraw() public nonReentrant {
        uint256 amount = balances[msg.sender];
        require(amount > 0, "Insufficient balance");
```

```
        balances[msg.sender] = 0; // Ефект перед взаємодією
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");
    }
}
```

**Завдання 3: Взаємодія з DeFi (Отримання ціни)**

Напишіть шаблон контракту, який може отримувати ціну активу. Для простоти можна використовувати інтерфейс для уявної пари токенів на AMM.

```
interface IPair {
    function getReserves() external view returns (uint112 reserve0, uint112 reserve1, uint32 blockTimestampLast);
}

contract PriceConsumer {
    address public pairAddress;

    constructor(address _pair) {
        pairAddress = _pair;
    }

    function getPrice() public view returns (uint256) {
        (uint112 res0, uint112 res1, ) = IPair(pairAddress).getReserves();
        // Розрахунок ціни токена 0 відносно токена 1: res1 / res0
        return uint256(res1) / uint256(res0);
    }
}
```

4. Інструментарій для безпеки

Для автоматичного пошуку багів рекомендується використовувати:

- **Slither**: Статичний аналізатор коду на Python, який швидко знаходить поширені вразливості.
- **Mythril**: Інструмент для символьного виконання, що дозволяє знаходити глибші логічні помилки.

**5. Контрольні запитання**

1. У чому принципова різниця між DEX на базі книги ордерів (Order Book) та AMM?
2. Опишіть покроково механіку атаки Reentrancy. Як зловмисник використовує функцію fallback?
3. Що таке патерн Check-Effects-Interactions і чому він важливий для безпеки?
4. Що таке Slippage (проковзування) у DeFi та як зміна параметрів і в пулі впливає на ціну для великих ордерів?
5. Чому розробники надають перевагу використанню бібліотек OpenZeppelin замість написання власних реалізацій стандартів?