Smart Contracts Exercise 06: Fool the Oracle

1 Introduction

Oracles are essential components in decentralized applications that require external data. In this exercise, you will become familiar with synchronous and asynchronous types of oracles, learn about the concept of Decentralized Exchanges (DEXs), and their use as on-chain oracles. Finally, you will practically implement a price oracle manipulation attack using a flash loan.

Prerequisites

Ensure that you have already installed the following on your system:

- **Node.js** - https://nodejs.org/en/ An open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser.
- **NPM**: Node Package Manager, which comes with Node.js.

Open your terminal and run the following commands to verify the installations:

```
$ node -v
$ npm -v
```

Both commands should return the installed version numbers of Node.js and NPM respectively. Node.js provides the runtime environment required to execute JavaScript-based tools like Hardhat, while NPM is used to manage the packages and dependencies needed for development. It is recommended that you use NPM 7 or higher.

Project Set Up

To get started, visit the following GitLab repository and clone it to your local machine. This repository contains a template in which you will complete this exercise. After you clone the repository, start with the following command within your project folder:

```
$ npm install
```

2 Oracles

Smart contracts cannot access data outside the blockchain on their own. They lack HTTP or similar network methods to access external sources. This is intentional to prevent non-deterministic behavior once a function is called. The blockchain does not have an internet connection! However, various Dapp applications often need external information, such as lending platforms, insurance contracts, betting contracts, wrapped cryptocurrencies, synthetics, and others. Whenever a smart contract relies on external data to compute future states, an oracle pattern is required. The disadvantages of oracles include the cost in terms of gas consumption and the risk of dependence on third parties in terms of data manipulation and data availability.

Synchronous Oracles

In our example, we illustrate a situation where Alice wants to borrow USDC tokens using ETH as collateral. The problem is that the ETH/USDC price is volatile, and we need to know the current price. In a synchronous oracle, external data is periodically pushed into the oracle contract from an oracle controller. The smart contract that needs this external data—in our example, the lending platform—simply trusts that the data in the oracle smart contract is genuine and updated regularly. Look closely at the ilustration in Figure 1 and the simplified code example below.

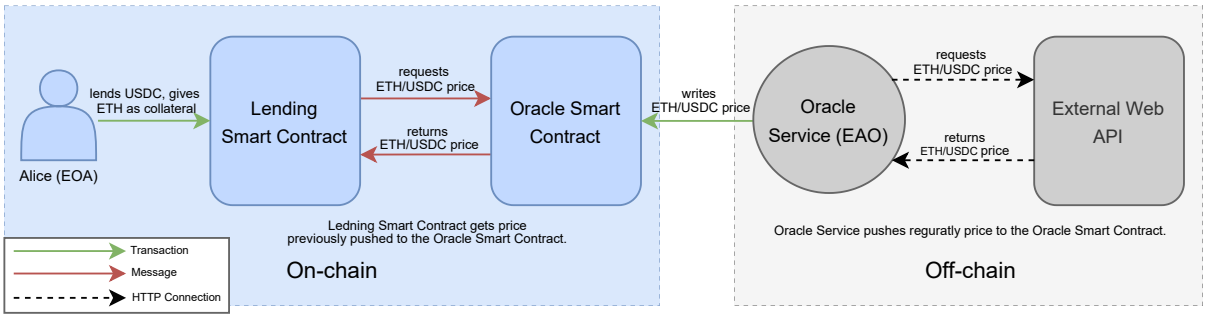

Figure 1: Synchronous Oracles

```
contract SimpleSynchronousOracle {
  // Stored ETH/USDC price
  int256 public price;
  // Oracle controller service address
  address public oracleService;

  modifier onlyController() {
      require(msg.sender == oracleService, "Not the oracle service");
      _;
  }
  constructor() {
      oracleService = msg.sender;
  }
  // Controller should periodically update the price
  function updatePrice(int256 _price) public onlyController {
      price = _price;
```

```
    }
    // Returns the latest price
    function getLatestPrice() public view returns (int256) {
        return price;
    }
}
```

Asynchronous Oracles

An asynchronous oracle provides data in a separate transaction after an initial request. A smart contract requests data from the oracle, which then emits an event. The oracle service fetches the data from an API or another off-chain source and subsequently sends the data back to the contract in a separate transaction.

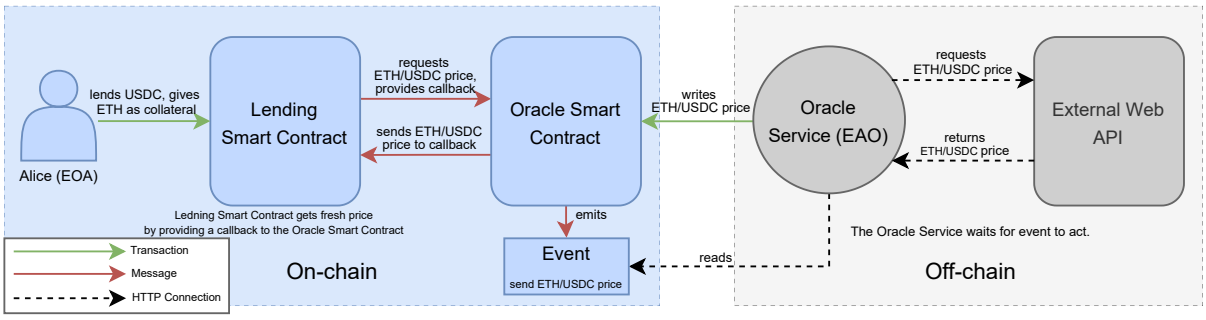

Figure 2: Asynchronous Oracles

To learn more about DEXes and oracles we recommend Berkeley RDI Center on Decentralization & AI course Decentralized Finance:

- DeFi MOOC Lecture 5: DEX
- DeFi MOOC Lecture 8: Oracles