Security Audit Report for Resonate Oracle

Date: September 1, 2022

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.09.01 22:24:39 +08'00'

Contents

Report Manifest

Item	Description
Client	Revest
Target	Resonate Oracle

Version History

Version	Date	Description
1.0	September 1, 2022	First Release

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is Resonate Oracle, which is used to provide price oracles for the Resonate project. The audit scope is limited to the contracts under the `hardhat/contracts/oracles` folder in the repository [1].

The auditing process is iterative. Specifically, we audit the initial version and following commits that fix the discovered issues. If there are new issues, we will continue this process. The commit hash values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Resonate Oracle	`Version 1`	`db62bb068f1530191346c56c483a24c2f3b3dda9`
	`Version 2`	`95a10a569bb0955aae47d5fa0f918da9d0640e1b`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

[1] https://github.com/Revest-Finance/Resonate/

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization

∗ Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ **Likelihood**	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **four** potential issues. Besides, we have **three** recommendations and **four** notes.

- High Risk: 3
- Medium Risk: 1
- Recommendation: 3
- Note: 4

ID	Severity	Description	Category	Status
1	Medium	Potential price manipulation	DeFi Security	Fixed
2	High	Unhandled token decimals when calculating the oracle prices	DeFi Security	Fixed
3	High	Ineffective check of Chainlink oracle data	DeFi Security	Fixed
4	High	Incorrect assumption of the quote token	DeFi Security	Fixed
5	-	Remove the unused function	Recommendation	Fixed
6	-	Emit events when updating important state variables	Recommendation	Fixed
7	-	Avoid consecutive division	Recommendation	Fixed
8	-	Range check of ChainLink prices	Note	
9	-	Potential integer overflow of Uniswap cumulative prices	Note	
10	-	The maintenance of price updates of `UniswapV2TWAPOracle`	Note	
11	-	Different sources of Chainlink oracle	Note	

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potential price manipulation

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `_getLPPrice` function of the `BalancerV2WeightedPoolPriceOracle` contract, there exists a potential vulnerability which can be used to perform the price manipulation attack.

In this function, the price of the LP token is calculated based on division, i.e., dividing the total value of tokens in the pool by the total supply of the LP token, as follows:

$$P_{LP} = \frac{\sum p_i B_i}{L}$$

Here p_i is the canonical price of the i-th token in the pool fetched from other providers, while B_i is the balance of the i-th token in the pool. Besides, L is the total supply of the LP token.

```
function _getLPPrice(address _bpt, bool isSafePrice)
internal
view
returns (uint256 price)
```

```
102    {
103        bytes32 poolId = IBPoolV2(_bpt).getPoolId();
104        uint256[] memory weights = IBPoolV2(_bpt).getNormalizedWeights();
105        uint256 totalSupply = IBPoolV2(_bpt).totalSupply();
106        (IERC20[] memory tokens, uint256[] memory balances, ) = vault.getPoolTokens(
107            poolId
108        );
109
110        uint256 totalFTM;
111        uint256[] memory prices = new uint256[](tokens.length);
112        // update balances in 18 decimals
113        for (uint256 i = 0; i < tokens.length; i++) {
114            balances[i] =
115                (balances[i] * (10**18)) /
116                (10**ERC20(address(tokens[i])).decimals());
117            prices[i] = isSafePrice
118                ? _getTokenSafePrice(address(tokens[i]))
119                : _getTokenCurrentPrice(address(tokens[i]));
120
121            if (i >= 1) {
122                _checkRatio(
123                    (balances[i - 1] * 10**18) / weights[i - 1],
124                    (balances[i] * 10**18) / weights[i],
125                    prices[i - 1],
126                    prices[i]
127                );
128            }
129
130            totalFTM += balances[i] * prices[i];
131        }
132
133        price = totalFTM / totalSupply;
134    }
```

Listing 2.1: BalancerV2WeightedPoolPriceOracle.sol

To avoid fetching price from an unbalanced pool, there is a `_checkRatio` function that checks the spot price of a token pair in the pool and the ratio of the canonical price of this pair. In Balancer, the spot price of two tokens in the pool is determined by:

$$SP_i^o = \frac{B_i/W_i}{B_o/W_o}$$

Here (B_i, B_o) is the balance of the token pair in the pool, while (W_i, W_o) is the corresponding weights of the tokens in the pool.

Specifically, in the `_checkRatio` function, it is required that the ratio difference should fall into an acceptable range. However, the check is performed only between neighboring tokens in the `tokens` array, which means the differences can accumulate. The longer the `tokens` array is, the larger the accumulation could be. Specifically, if the allowed `diffLimit` (line 144 in the `_checkRatio` function) is 5% and the length of `tokens` is 5, the normalized value of the last token can be $1.05^4 = 1.2155$ times the normalized value of the first token. By carefully manipulating the token balances in the pool, an attacker can eventually manipulate the calculation of the LP token price.

```
function _checkRatio(
    uint256 reserve0,
    uint256 reserve1,
    uint256 price0,
    uint256 price1
) internal view {
    uint256 value0 = reserve0 * price0;
    uint256 value1 = reserve1 * price1;
    uint256 diffLimit = (value0 * ratioDiffLimitNumerator) /
        ratioDiffLimitDenominator;

    require(
        value1 < value0 + diffLimit && value0 < value1 + diffLimit,
        "INVALID RATIO"
    );
}
```

Listing 2.2: BalancerV2WeightedPoolPriceOracle.sol

Impact Price manipulation may enlarge the LP token's price and cause financial losses.

Suggestion Revise the code logic to make the calculation resistant to the price manipulation attack.

Feedback from the Project Revest Finance derived an equation to describe how to value BPT tokens securely and implemented it.

2.1.2 Unhandled token decimals when calculating the oracle prices

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Oracles are all registered in a `PriceProvider` contract, which can be invoked to fetch token prices. To support different tokens, it is reasonable to assume that these oracles should have a uniform output format. Specifically, all oracle prices should be consistent with the format of

$$\frac{1\ unit\ A\ token}{1\ unit\ B\ token} \times PRECISION$$

Note that `1 unit token` equals to 10^D wei tokens where D is the decimal of the token, while the B token should always be `WETH`, and $PRECISION$ is 10^{18}.

Based on the above assumption, there are two cases that token decimals are not properly handled:

1. In the `_getLPPrice` function of the `UniswapV2LPPriceOracle` contract, the LP token price is calculated by adopting the Fair LP Token Pricing Formula [1] for Uniswap. However, the meaning of the formula is that the price is the total value of the pair tokens divided by the total supply of the LP token, which represents the value of LP token *per wei*. The result is directly returned as the `price`, however, other prices from the oracles are the values of the tokens *per unit*, as described earlier.

```
function _getLPPrice(address pair, bool isSafePrice) internal view returns (uint price) {
    address token0 = IUniswapV2Pair(pair).token0();
```

[1] https://blog.alphaventuredao.io/fair-lp-token-pricing/

```
    address token1 = IUniswapV2Pair(pair).token1();
    uint totalSupply = IUniswapV2Pair(pair).totalSupply();
    (uint r0, uint r1, ) = IUniswapV2Pair(pair).getReserves();
    uint sqrtR = (r0*r1).sqrt();

    uint p0 = isSafePrice ? provider.getSafePrice(token0) : provider.getCurrentPrice(
        token0);
    uint p1 = isSafePrice ? provider.getSafePrice(token1) : provider.getCurrentPrice(
        token1);
    uint sqrtP = (p0*p1).sqrt();
    price =(2*sqrtR*sqrtP) / totalSupply; // in 1E18 precision
}
```

Listing 2.3: UniswapV2LPPriceOracle.sol

2. In the `_convertPrice` function of the `UniswapV2TWAPOracle` contract, the argument named `lastUpdatePrice` is the ratio of the reserves in the Uniswap pair, which represents the ratio of the amount of the tokens in *wei*, not in *unit*. The result does not handle different token decimals.

```
function getSafePrice(address asset) public view returns (uint256 amountOut) {
    require(block.timestamp - twaps[asset].timestampLatest <= MAX_UPDATE, 'ER037');
    TwapConfig memory twap = twaps[asset];
    amountOut = _convertPrice(asset, twap.lastUpdateTwapPrice);
}
```

Listing 2.4: UniswapV2TWAPOracle.sol

```
function _convertPrice(address asset, FixedPoint.uq112x112 memory lastUpdatePrice)
     private view returns (uint amountOut) {
    uint112 nativeDecimals = uint112(10**IERC20Metadata(asset).decimals());
    // calculate the value based upon the average cumulative prices
    // over the time period (TWAP)
    if (TOKEN == WETH) {
        // No need to convert the asset
        amountOut = lastUpdatePrice.mul(PRECISION).decode144();
    } else {
        // Need to convert the feed to be in terms of ETH
        uint conversion = provider.getSafePrice(TOKEN);
        amountOut = lastUpdatePrice.mul(conversion).decode144();
    }
}
```

Listing 2.5: UniswapV2TWAPOracle.sol

Impact Unhandled token decimals may lead to severe price deviation and financial losses.

Suggestion Revise the code logic accordingly.

2.1.3 Ineffective check of Chainlink oracle data

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `_feedPrice` function of the `ChainlinkPriceOracle` contract, the token price is fetched by calling the `latestRoundData` function of the Chainlink Aggregator contract through the interface named `AggregatorV3Interface`. To ensure the price is updated within an acceptable delay, this function checks whether `timestamp - startedAt` is less than `MIN_TIME` or not.

```
71 function _feedPrice(address _feed) internal view returns (uint256 latestUSD) {
72
73     /// To allow for TOKEN-ETH feeds on one oracle, TOKEN-USD feeds on another
74     if(_feed == address(0)) {
75         return PRECISION;
76     }
77
78     (uint80 roundID, int256 answer, uint256 startedAt, uint256 timestamp, uint80
           answeredInRound) = AggregatorV3Interface(_feed).latestRoundData();
79
80     require(answer > 0, "E112");
81     require(answeredInRound >= roundID, "E113a");
82     require(timestamp != 0, "E113b");
83
84     // difference between when started and returned needs to be less than 60-minutes
85     // require(block.timestamp - timestamp < MIN_TIME, "E113c");
86     require(timestamp - startedAt < MIN_TIME, "E113d");
87
88     return uint256(answer);
89 }
```

Listing 2.6: ChainlinkPriceOracle.sol

However, the implementation of the Chainlink Aggregator V3 contracts do not behave as expected. Specifically, only some of the these contracts would return the meaningful `startedAt` values, as stated by Chainlink [2]. Things get worse when it comes to the Chainlink Aggregator V4 contracts, which are backward compatible with the V3 interface. As the code snippet shown in the below, the returned `startedAt` and `updatedAt` (i.e., `startedAt` and `timestamp` in the `_feedPrice` function) are the same with each other in the V4 contracts. As a result, the check in line 86 of the `ChainlinkPriceOracle` contract is ineffective.

```
  function latestRoundData()
  public
  override
  view
  virtual
  returns (
    uint80 roundId,
    int256 answer,
    uint256 startedAt,
    uint256 updatedAt,
    uint80 answeredInRound
  )
{
  roundId = s_hotVars.latestAggregatorRoundId;

```

[2] https://github.com/smartcontractkit/chainlink/blob/e1e78865d4f3e609e7977777d7fb0604913b63ed/contracts/src/v0.6/EACAggregatorProxy.sol#L192

```
    // Skipped for compatability with existing FluxAggregator in which latestRoundData never reverts.
    // require(roundId != 0, V3_NO_DATA_ERROR);

    Transmission memory transmission = s_transmissions[uint32(roundId)];
    return (
      roundId,
      transmission.answer,
      transmission.timestamp,
      transmission.timestamp,
      roundId
    );
  }
```

Listing 2.7: OffchainAggregator.sol

Impact Potential incorrect price calculation due to the stale prices fed to the contracts.

Suggestion Revise the code (e.g., using the difference between `block.timestamp` and `updatedAt`).

2.1.4 Incorrect assumption of the quote token

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description As stated in Issue 2.1.2, all the oracle price query interfaces must return prices with `WETH` as the quote token. On the other side, the `getCurrentPrice` function in the `UniswapV2TWAPOracle` contract returns the price with the state variable `TOKEN` as the quote token. Obviously, it assumes that `TOKEN` is just `WETH`. However, this assumption may not be true. For example, the `_convertPrice` function (see Listing 2.5) first checks the equality of `TOKEN` and `WETH`, and then takes different actions based on the result.

```
    function getCurrentPrice(address asset) public view returns (uint256 amountOut) {
        TwapConfig memory twap = twaps[asset];
        IUniswapV2Pair pair = IUniswapV2Pair(twap.pairAddress);

        (uint reserve0, uint reserve1, ) = pair.getReserves();
        if (twap.isToken0) {
            uint8 _token1MissingDecimals = 18 - (IERC20Detailed(TOKEN).decimals());
            amountOut = (reserve1 * (10**_token1MissingDecimals) * PRECISION) / reserve0;
        } else {
            uint8 _token0MissingDecimals = 18 - (IERC20Detailed(TOKEN).decimals());
            amountOut = (reserve0 * (10**_token0MissingDecimals) * PRECISION) / reserve1;
        }
    }
```

Listing 2.8: UniswapV2TWAPOracle.sol

Impact The incorrect assumption of the quote token may lead to unexpected results.

Suggestion Revise the code logic.

2.2 Additional Recommendation

2.2.1 Remove the unused function

Status Fixed in Version 2

Introduced by Version 1

Description The function named _divide in the UniswapV2TWAPOracle contract is declared but not used.

Impact N/A

Suggestion Remove the unused function.

2.2.2 Emit events when updating important state variables

Status Fixed in Version 2

Introduced by Version 1

Description In the SimpleOracle contract, there are two functions that modify important state variables without emitting events.

```
function updatePrice(address token, uint price) external onlyAdmin(token) {
    _currentPrices[token] = price;
}
```

Listing 2.9: SimpleOracle.sol

```
function setAdminStatus(address token, address admin, bool isApproved) external onlyOwner {
    tokenAdmins[token][admin] = isApproved;
}
```

Listing 2.10: SimpleOracle.sol

Impact N/A

Suggestion Emit events when updating important variables.

2.2.3 Avoid consecutive division

Status Fixed in Version 2

Introduced by Version 1

Description In the _tokenPriceFromWeights function of the BalancerV2PriceOracle contract, the final result is calculated from a consecutive division, which may cause precision loss. It is recommended to multiply the divisors before the division rather than consecutive division.

```
function _tokenPriceFromWeights(
    IERC20 token0,
    IERC20 token1,
    uint256 balance0,
    uint256 balance1,
    uint256 weight0,
    uint256 weight1
) internal view returns (uint256) {
```

```
        uint256 pairTokenPrice = _getTokenCurrentPrice(
            IPriceOracle(denominatedOracles[address(token0)]),
            token1
        );

        // price = balance1 / balance0 * weight0 / weight1 * usdPrice1

        // in denominated token price decimals
        uint256 assetValue = (balance1 * pairTokenPrice) /
            (10**ERC20(address(token1)).decimals());
        // in denominated token price decimals
        return
            (assetValue * weight0 * (10**ERC20(address(token0)).decimals())) /
            weight1 /
            balance0;
    }
```

Listing 2.11: BalancerV2PriceOracle.sol

Impact May lead to precision loss.

Suggestion Revise the code accordingly.

2.3 Note

2.3.1 Range check of ChainLink prices

Introduced by `Version 1`

Description As noted in the Chainlink document [3], the data feed aggregator contract has the `minAnswer` and `maxAnswer` variables, which prevent the aggregator from updating the `latestAnswer` outside the agreed range of acceptable values. To perform the best practice, the `ChainlinkPriceOracle` contract should check the return value of the Chainlink aggregator to guarantee the validity of the price, and take proper actions if necessary.

2.3.2 Potential integer overflow of Uniswap cumulative prices

Introduced by `Version 1`

Description In the `updateSafePrice` function of the `UniswapV2TWAPOracle` contract, the price update procedure fetches a variable named `cumulativeLast` and subtracts another variable named `lastCumPrice` to get the difference. However, `cumulativeLast` is the `price{0,1}CumulativeLast` variable in the Uniswap pair, which increases over time and can overflow by design. However, the `updateSafePrice` function doesn't consider the overflow case. Since the compiler version is over 0.8.0, once the `price{0,1}CumulativeLast` variable overflows, the `updateSafePrice` function will revert, which may lead to the DoS to the contract. It is not considered as an issue due to the low possibility, however, it still needs to be noted in the report.

```
    function updateSafePrice(address asset) public returns (uint256 amountOut) {
        // This method will fail if the TWAP has not been initialized on this contract
```

[3] https://docs.chain.link/docs/using-chainlink-reference-contracts/#monitoring-data-feeds

```
        // This action must be performed externally
        (uint cumulativeLast, uint lastCumPrice, uint32 lastTimeSync, uint32 lastTimeUpdate) =
            _fetchParameters(asset);
        TwapConfig storage twap = twaps[asset];
        FixedPoint.uq112x112 memory lastAverage;
        lastAverage = FixedPoint.uq112x112(uint224((cumulativeLast - lastCumPrice) / (lastTimeSync
            - lastTimeUpdate)));
        twap.lastUpdateTwapPrice = lastAverage;
        twap.lastUpdateCumulativePrice = cumulativeLast;
        twap.timestampLatest = lastTimeSync;

        // Call sub method HERE to same thing getSafePrice uses to avoid extra SLOAD
        amountOut = _convertPrice(asset, lastAverage);
    }
```

Listing 2.12: UniswapV2TWAPOracle.sol

```
    function _fetchParameters(
        address asset
    ) private view returns (
        uint cumulativeLast,
        uint lastCumPrice,
        uint32 lastTimeSync,
        uint32 lastTimeUpdate
    ) {
        TwapConfig memory twap = twaps[asset];
        require(twap.decimals > 0, 'ER035');
        // Enforce passage of a safe amount of time
        lastTimeUpdate = twap.timestampLatest;
        require(block.timestamp > lastTimeUpdate + MIN_UPDATE, 'ER036');
        IUniswapV2Pair pair = IUniswapV2Pair(twap.pairAddress);
        cumulativeLast = twap.isToken0 ? pair.price0CumulativeLast() : pair.price1CumulativeLast();
        lastCumPrice = twap.lastUpdateCumulativePrice;
        (, , lastTimeSync) = pair.getReserves();
    }
```

Listing 2.13: UniswapV2TWAPOracle.sol

2.3.3 The maintenance of price updates of `UniswapV2TWAPOracle`

Introduced by `Version 1`

Description The TWAP price in `UniswapV2TWAPOracle` contract should be constantly updated by invoking the `updateSafePrice` function. Otherwise, there's no other way to update the price. As such, the project should monitor the time interval between the calls to the `updateSafePrice` function and trigger the update procedure automatically.

2.3.4 Different sources of Chainlink oracle

Introduced by `Version 1`

Description In the `ChainlinkPriceOracle` contract, different tokens are linked to the corresponding Chainlink oracle contracts in the `setPriceFeed` function.

```
function setPriceFeed(address _token, address _feed) external onlyOwner {
    priceFeed[_token] = _feed;

    emit SetPriceFeed(_token, _feed);
}
```

Listing 2.14: ChainlinkPriceOracle.sol

Then the oracle price to be output is calculated by comparing the token price fetched from the Chainlink oracle and a `BASE_PRICE_FEED` after adjusting the decimal. Depending on the base token, Chainlink oracles may have different decimals, so a fixed `decimals` variable is not enough. To make this procedure reasonable, there is a requirement that all the Chainlink oracles set in the `priceFeed` mapping should return the prices with the same decimal, which means the prices should be all in USD or all in ETH. This requirement can be satisfied through the invocation to the `setPriceFeed` function with proper parameters set by the project.

```
function getSafePrice(address _token) public view returns (uint256 _amountOut) {
    return getCurrentPrice(_token);
}

function getCurrentPrice(address _token) public view returns (uint256 _amountOut) {
    require(priceFeed[_token] != address(0), "UNSUPPORTED");

    _amountOut = _divide(
        _feedPrice(priceFeed[_token]),
        _feedPrice(BASE_PRICE_FEED),
        decimals
    );
}
```

Listing 2.15: ChainlinkPriceOracle.sol