·Decurity·

SMART CONTRACT SECURITY AUDIT REPORT

ORACLE DAO

Decurity, 2022

CONTENTS

1 GENERAL INFORMATION

This report contains information about the results of the security audit of the Oracle DAO (hereafter referred as "Customer") smart contracts, conducted by DeFiSecurity.io in the period from 04/14/2022 to 04/25/2022.

1.1 Introduction

Tasks solved during the work are:

- Review the protocol design and the usage of 3rd party dependencies,
- Audit the contracts implementation,
- Develop the recommendations and suggestions to improve the security of the contracts.

1.2 Scope of Work

The audit scope included the contracts in the following repository: https://github.com/Oracle-DAO/core-contracts. Initial review was done for the commit d3d74462768d48ffa1a8a9d113922386fd784f3a and the re-testing was done for the commit 4ce70e3a61bda3d01850ce0ef3752a935e0048c1.

1.3 Threat Model

The assessment presumes actions of an intruder who might have capabilities of an external user. The centralization risks have not been considered upon the request of the Customer.

1.4 Weakness Scoring

An expert evaluation scores the findings in this report, an impact of each vulnerability is calculated based on its ease of exploitation (based on the industry practice and our experience) and severity (for the considered threats).

2 SUMMARY

As a result of this work, we have discovered a single critical exploitable security issue which has been fixed and re-tested in the course of the work.

The other suggestions included fixing the low-risk issues and some best practices (see 3.1).

The Oracle DAO team has given the feedback for the suggested changes and explanation for the underlying code.

2.1 Suggestions

The table below contains the discovered issues, their risk level, and their status as of 28 April 2022.

Table 1. Discovered weaknesses

Issue	Contract	Risk Level	Status
Improper Access Control in Staking	contracts/coreContracts/Staking.sol	High	Fixed
Incorrect total reserves calculation in Treasury	contracts/coreContracts/Treasury.sol	Medium	Fixed
Redundant checks during deposit in Bond	contracts/coreContracts/Bond.sol	Low	Fixed
Unused variable	contracts/coreContracts/Bond.sol	Low	Fixed

3 GENERAL RECOMMENDATIONS

This section contains general recommendations how to fix discovered during the testing weaknesses and vulnerabilities and how to improve overall security level.

Section 3.1 contains a list of general mitigations against the discovered weaknesses, technical recommendations for each finding can be found in section 4.

Section 3.2 describes a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level.

3.1 Current findings remediation

Follow the recommendations in the section 4.

3.2 Security process improvement

- Keep the whitepaper and documentation updated to make it consistent with the implementation and the intended use cases of the system,
- Perform regular audits for all the new contracts and updates,
- Ensure the secure off-chain storage and processing of the credentials (e.g. the privileged private keys),
- Launch a public bug bounty campaign for the contracts.

4 FINDINGS

4.1 Improper Access Control in Staking

Risk Level: High

Status:

The issue has been fixed in the commit https://github.com/Oracle-DAO/core-contracts/commit/4ce70e3a61bda3d01850ce0ef3752a935e0048c1 by adding the `onlyOwner` modifier:

```
56  56
57      -   function setRewardDistributor(address rewardDistributor_) external {
        57  +   function setRewardDistributor(address rewardDistributor_) external onlyOwner {
58  58          rewardDistributor = IRewardDistributor(rewardDistributor_);
59  59      }
```

Image 1. Fix for the setRewardDistributor function

Contracts:

contracts/coreContracts/Staking.sol

References: https://dasp.co/#item-2

Remediation:

Add the `onlyOwner` modifier to the `setRewardDistributor` function.

Description:

Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized

information disclosure, modification or destruction of all data, or performing a business function outside of the limits of the user.

Proofs:

The staking contract makes external calls to the reward distribution contract whose address is mutable and can be set using the function `setRewardDistributor`. However, the function is not protected and anyone can call it to change the address:

```
56
57    function setRewardDistributor(address rewardDistributor_) external {
58      rewardDistributor = IRewardDistributor(rewardDistributor_);
59    }
60
```

Image 2. The vulnerable function

As a result, an attacker can interfere with the contract execution (e.g. the `stake` function) and cause significant losses.

For the demonstration purposes, the following test code has been used:

```
it('[!] PWN', async function () {
  await staking.connect(hacker).setRewardDistributor(constants.zeroAddress);
  await orfi.approve(staking.address, stakingAmount);

  await expect(staking.stake(stakingAddress, stakingAmount)).to.be.reverted;
});
```

The screenshot below shows a demonstration of the exploitation of this vulnerability:

```
Staking Test
  ✔ [!] PWN
  Test user Stake and unstake
    ✔ Test user Stake and unstake (52ms)
  Test user Stake with warmup and claim and then unstake
    ✔ Test user Stake with warmup and claim and then unstake (1133ms)
  Test user Stake with warmup and forfeit
    ✔ DTest user Stake with warmup and forfeit (48ms)
```

Image 3. Exploit test successful

4.2 Incorrect total reserves calculation in Treasury

Risk Level: Medium

Status:

The issue has been fixed in the commit https://github.com/Oracle-DAO/core-contracts/commit/4ce70e3a61bda3d01850ce0ef3752a935e0048c1 by adding the call to `valueOfToken` function to calculate the required increment value:

```
98      -
99      -       _totalReserves = _totalReserves.add(_amount);
        98  +   uint256 value = valueOfToken(_token, _amount, isReserveToken, isLiquidityToken);
        99  +   _totalReserves = _totalReserves.add(value);
100     100     _totalORFIMinted = _totalORFIMinted.add(_orfiAmount);
```

Image 4. Fix for the total reserves calculation

Contracts:

contracts/coreContracts/Treasury.sol

References: -

Remediation:

Use the `valueOfToken` function to calculate the correct value to add to the `_totalReserves` variable.

Description:

Incorrect calculation of the total reserves can lead to further financial errors in case if the protocol uses more than one deposit token (e.g. the `principle` token in the `Bond` contract).

Proofs:

The `deposit` function of the `Treasury` contract updates the `_totalReserves` variable based on the passed `_amount` argument. However, it does not take into account the `_token` argument. As a result, the deposit amounts of distinct tokens will end up added to the same integer value. This could lead to incorrect accounting and `TAV` calculation. Currently, only a single deposit token (MIM stablecoin) is supposed to be used in the protocol but that could change.

```
77      /**
78      @notice allow approved address to deposit an asset for ORFI
79          @param _amount uint
80          @param _token address
81          @param _orfiAmount uint
82       */
83      function deposit(
84          uint256 _amount,
85          address _token,
86          uint256 _orfiAmount
87      ) external {
88          bool isReserveToken = ITreasuryHelper(treasuryHelper).isReserveToken(_token);
89          bool isLiquidityToken = ITreasuryHelper(treasuryHelper).isLiquidityToken(_token);
90
91          require(isReserveToken || isLiquidityToken, 'NA');
92
93          if (isReserveToken) {
94              require(ITreasuryHelper(treasuryHelper).isReserveDepositor(msg.sender), 'NAPPROVED');
95          } else {
96              require(ITreasuryHelper(treasuryHelper).isLiquidityDepositor(msg.sender), 'NAPPROVED');
97          }
98
99          _totalReserves = _totalReserves.add(_amount);
100         _totalORFIMinted = _totalORFIMinted.add(_orfiAmount);
101
102         IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
103         IORFI(ORFI).mint(msg.sender, _orfiAmount);
104
105         emit ReservesUpdated(_totalReserves);
106         emit Deposit(_token, _amount, _orfiAmount);
107     }
```

Image 5. The vulnerable Treasury deposit implementation

4.3 Redundant checks during deposit in Bond

Risk Level: Low

Status:

The issue has been fixed in the commit https://github.com/Oracle-DAO/core-contracts/commit/4ce70e3a61bda3d01850ce0ef3752a935e0048c1 by removing the `_depositor` argument and using `msg.sender` instead:

```
246  252          *  @notice deposit bond
247  253       *  @param _amount uint
248  254       *  @param _maxPrice uint
249       -     *  @param _depositor address
250  255       *  @return uint
251  256       */
252  257        function deposit(
253  258            uint256 _amount,
254       -        uint256 _maxPrice,
255       -        address _depositor
     259  +        uint256 _maxPrice
256  260       ) external returns (uint256) {
257       -        require(_depositor != address(0), 'Invalid address');
258       -        require(msg.sender == _depositor, 'LFNA');
259  261            decayDebt();
260  262
261  263            // convert stablecoin decimals to 18 equivalent
     @@ -280,8 +282,8 @@ contract Bond is Ownable {
280  282            totalDebt = totalDebt.add(_amount);
281  283
282  284            // depositor info is stored
283       -        bondInfo[_depositor] = BondInfo({
284       -            payout: bondInfo[_depositor].payout.add(payout),
     285  +        bondInfo[msg.sender] = BondInfo({
     286  +            payout: bondInfo[msg.sender].payout.add(payout),
285  287                vesting: terms.vestingTerm,
286  288                lastTime: uint32(block.timestamp),
287  289                pricePaid: priceInUSD
```

Image 6. Fix for the redundant checks in Bond

Contracts:

contracts/coreContracts/Bond.sol

References: -

Remediation:

Remove unnecessary checks and arguments.

Description:

Unnecessary checks make the code harder to read and cause extra gas costs.

Proofs:

The `deposit` function of the `Bond` contract accepts the depositor address as an argument and checks whether it matches the `msg.sender` value. Clearly, the argument could be removed in favour of the `msg.sender` value.

```
245      /**
246       *  @notice deposit bond
247      *  @param _amount uint
248      *  @param _maxPrice uint
249      *  @param _depositor address
250      *  @return uint
251      */
252       function deposit(
253           uint256 _amount,
254           uint256 _maxPrice,
255           address _depositor
256       ) external returns (uint256) {
257           require(_depositor != address(0), 'Invalid address');
258           require(msg.sender == _depositor, 'LFNA');
259           decayDebt();
```

Image 7. The suboptimal Bond deposit function

4.4 Unused variable

Risk Level: Low

Status:

Oracle DAO confirmed the variable is calculated correctly and will be used as a getter to show the total bonding rewards.

Contracts:

contracts/coreContracts/Bond.sol

References: -

Remediation:

Remove the variable and the calculation if not needed.

Description:

Unused variables make the code harder to read and cause extra gas costs.

Proofs:

The `deposit` function of the `Bond` contract initializes the `bondingReward` variable which never used anymore neither in the Bond contract nor other contracts.

```
245     /**
246      *  @notice deposit bond
247     *  @param _amount uint
248     *  @param _maxPrice uint
249     *  @param _depositor address
250     *  @return uint
251     */
252     function deposit(
253         uint256 _amount,
254         uint256 _maxPrice,
255         address _depositor
256     ) external returns (uint256) {
257         require(_depositor != address(0), 'Invalid address');
258         require(msg.sender == _depositor, 'LFNA');
259         decayDebt();
260 
261         // convert stablecoin decimals to 18 equivalent
262         uint256 amount = convertInto18DecimalsEquivalent(_amount);
263 
264         uint256 priceInUSD = bondPriceInUSD(); // Stored in bond info
265         bondingReward = bondingReward.add(calculateBondingReward()); // calculate bonding rewards
266 
267         require(_maxPrice >= priceInUSD, 'Slippage limit: more than max price'); // slippage protecti
268 
269         uint256 payout = payoutFor(amount); // payout to bonder is computed in 1e18
270 
271         require(totalDebt.add(amount) <= terms.maxDebt, 'Max capacity reached');
```

Image 8. The unused variable

5 APPENDIX

5.1 About us

The Decurity (former DeFiSecurity.io) team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we've gained an expertise in blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.