HashEx
BLOCKCHAIN SECURITY

Onyx DAO

smart contracts
final audit report

April 2023

hashex.org

contact@hashex.org

Contents

1. Disclaimer

2. Overview

HashEx was commissioned by the Onyx DAO team to perform an audit of their smart contract. The audit was conducted between 22/03/2023 and 27/03/2023.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code is available at @OnyxDAOFinance/OnyxContracts GitHub repository and was audited after the commit 4060e66.

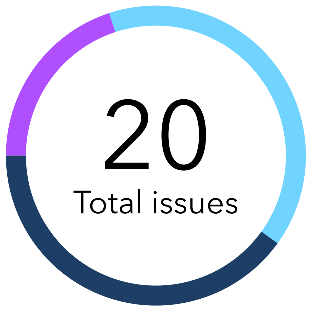
2.1 Summary

Project name	Onyx DAO
URL	https://onyxdao.finance
Platform	Arbitrum Network
Language	Solidity

2.2 Contracts

Name	Address
OnyxToken	0xb7cd6c8c4600aed9985d2c0eb174e0bee56e8854
MasterChefOnyx	0xf9c83ff6cf1a9bf2584aa2d00a7297ca8f845cce
OnyxVault	
OnyxLockedVesting	0x85aca003b2c6481ca2c4547420fa0e211b51df00
StakingPool	0x586C7D0ced41310C299AC47AdD5d0c3Df8651C0e
StakingPoolFactory	0x586C7D0ced41310C299AC47AdD5d0c3Df8651C0e

3. Found issues



●	Medium	4 (20%)
●	Low	8 (40%)
●	Info	8 (40%)

C1. OnyxToken

ID	Severity	Title	Status
C1-01	● Low	Gas optimizations	✓ Acknowledged
C1-02	● Info	Possible exceedance of supply cap	✓ Acknowledged

C2. MasterChefOnyx

ID	Severity	Title	Status
C2-01	● Medium	Emission rate update without updating pools	✓ Acknowledged
C2-02	● Medium	Wrong pending tokens calculation	✓ Acknowledged
C2-03	● Low	Gas optimizations	✓ Acknowledged
C2-04	● Low	Lack of checks for input token in the add pool function	✓ Acknowledged
C2-05	● Low	Ignored result of try-catch	✓ Acknowledged

C2-06	● Low	Lack of input checks	✓ Acknowledged
C2-07	● Info	Possible rounding error for tokens with large supply	✓ Acknowledged
C2-08	● Info	Possible exccedance of emission rate limit	✓ Acknowledged

C3. OnyxVault

ID	Severity	Title	Status
C3-01	● Medium	Hardcoded pid	✓ Acknowledged
C3-02	● Low	Gas optimizations	✓ Acknowledged
C3-03	● Info	Lack of events	✓ Acknowledged
C3-04	● Info	Typos	✓ Acknowledged
C3-05	● Info	One-time approve	✓ Acknowledged

C4. OnyxLockedVesting

ID	Severity	Title	Status
C4-01	● Low	Gas optimizations	✓ Acknowledged

C5. StakingPool

ID	Severity	Title	Status
C5-01	● Medium	Rewards source and locked withdrawals	✓ Acknowledged
C5-02	● Low	Gas optimizations	✓ Acknowledged

C5-03	● Info	Confusing error message	✓ Acknowledged

C6. StakingPoolFactory

ID	Severity	Title	Status
C6-01	● Info	Typos	✓ Acknowledged

4. Contracts

C1. OnyxToken

Overview

An ERC20 standard token made on OpenZeppelin's implementation with historical snapshots for use in governance voting forked from Compound Finance. OnyxToken is meant to be used with the MasterChefOnyx contract as it has mint functionality available to the owner. The total supply is limited by the immutable cap that can be checked with the `cap()` external viewer.

Issues

C1-01 Gas optimizations ● Low ✓ Acknowledged

1. The MasterChef contract, which is intended to be OnyxToken's owner, can't use the `mint(uint256)` function. It can be removed to reduce the deployed bytecode.

C1-02 Possible exceedance of supply cap ● Info ✓ Acknowledged

Initial mint is not constrained by cap limit, as opposed to external `mint()` functions.

```
    constructor(uint256 initialSupply, uint256 cap_) ERC20("Onyx Token", "ONYX") {
        require(cap_ > 0, "ERC20Capped: cap is 0");
        _mint(msg.sender, initialSupply);
        _cap = cap_;
    }

    function mint(address _to, uint256 _amount) public onlyOwner {
        require(ERC20.totalSupply() + _amount <= cap(), "ERC20Capped: cap exceeded");
        _mint(_to, _amount);
    }

    function mint(uint256 _amount) public onlyOwner {
        require(ERC20.totalSupply() + _amount <= cap(), "ERC20Capped: cap exceeded");
        _mint(_msgSender(), _amount);
    }
```

Onyx team response

Will set emissions to 0 at 1,570,000 or burn any that reach over 1.57M.

C2. MasterChefOnyx

Overview

A farming contract forked from MasterChef by Sushiswap. Implements optional deposit fee (up to 10%) and external rewarder individual for each pool.

Issues

C2-01 Emission rate update without updating pools ● Medium ✓ Acknowledged

An update of the pools should be performed prior to changing the emission rate in the `updateEmissionRate()` function.

```
    function updateEmissionRate(uint256 _onyxPerSecond) public onlyOwner {
        _updateEmissionRate(_onyxPerSecond);
        massUpdatePools();
    }
```

Recommendation

Consider updating pools before changing the emission rate to ensure fair reward distribution.

```
    function updateEmissionRate(uint256 _onyxPerSecond) public onlyOwner {
        massUpdatePools();
        _updateEmissionRate(_onyxPerSecond);
    }
```

Onyx team response

Will `massUpdatePools()` before future emissions changes.

C2-02 Wrong pending tokens calculation ● Medium ✓ Acknowledged

The function `pendingTokens()` calculates a user's not claimed rewards. The function calculates with the assumption that all minted tokens are used for the rewards, but 10% of them are minted to the `devaddr` address.

```
    function pendingTokens(uint256 _pid, address _user) external view returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 accOnyxPerShare = pool.accOnyxPerShare;
        uint256 stakeSupply = pool.totalStaked;
        if (block.timestamp > pool.lastRewardTime && stakeSupply != 0) {
            uint256 multiplier = getMultiplier(pool.lastRewardTime, block.timestamp);
            uint256 onyxReward =
multiplier.mul(onyxPerSecond).mul(pool.allocPoint).div(totalAllocPoint);
            accOnyxPerShare = accOnyxPerShare.add(onyxReward.mul(1e12).div(stakeSupply));
        }
        return user.amount.mul(accOnyxPerShare).div(1e12).sub(user.rewardDebt);
    }
```

The actual token distribution made in the function `updatePool()` uses 90% of the minted tokens for the rewards and 10% are minted to the developers' address.

```
    function updatePool(uint256 _pid) public {
        ...
        uint256 multiplier = getMultiplier(pool.lastRewardTime, block.timestamp);
        uint256 totalOnyx =
multiplier.mul(onyxPerSecond).mul(pool.allocPoint).div(totalAllocPoint);
        ...
        uint256 forDevs = totalOnyx.mul(devFee).div(1000);
        uint256 onyxReward = totalOnyx.sub(forDevs);
        onyx.mint(devaddr, forDevs);
        onyx.mint(address(this), onyxReward);
        pool.accOnyxPerShare =
pool.accOnyxPerShare.add(onyxReward.mul(1e12).div(stakeSupply));
        ...
    }
```

The issue results in a wrong value shown to a user and may implicate integration with other services like farm aggregators.

Recommendation

Fix the calculation for pending tokens using the same formulas as in the `updatePool()` function.

Onyx team response

We will ensure the Frontend is accurate.

C2-03 Gas optimizations

● Low ✓ Acknowledged

1. The `onyx` variable should be declared immutable.

2. No need to restrict duplicated pools since `PoolInfo.totalStaked` is stored and used during the pool updating.

3. Unnecessary reads of `pool.lastRewardTime` and `pool.accOnyxPerShare` from storage in the `updatePool()` function.

4. Multiple reads of `user.amount`, `pool.stakeToken`, `pool.depositFeeBP`, and `pool.accOnyxPerShare` from storage in the `deposit()` function.

5. Multiple reads of `user.amount` and `pool.accOnyxPerShare` from storage in the `withdraw()` function.

6. Multiple reads of `user.amount` from storage in the `emergencyWithdraw()` function.

C2-04 Lack of checks for input token in the add pool function ● Low Acknowledged

Adding an address to a pool that does not implement IERC20 will break the `massUpdate()` function. If a new pool is added without `massUpdate()` the unclaimed user's rewards will be recalculated and decreased.

Onyx team response

As we will not be adding any more pools. Sent mass updates before adding pool id.

C2-05 Ignored result of try-catch ● Low Acknowledged

The call result is ignored in the `tryCatchOnReward()` function: failed calls should be monitored in the catch section, e.g. by emitting an event.

```
    function tryCatchOnReward(uint256 _pid, uint256 _pending, uint256 _amount) internal
returns (bool) {
        try poolInfo[_pid].rewarder.onReward(_pid, msg.sender, msg.sender, _pending,
_amount) {
            return true;
        } catch {
            return false;
        }
    }

    function withdraw(uint256 _pid, uint256 _amount) public nonReentrant {
        ...
        tryCatchOnReward(_pid, pending, user.amount);
        ...
    }
```

C2-06 Lack of input checks ● Low Acknowledged

The functions `dev()` , `setFeeAddress()` lack zero address checks for the input values.

```
    function dev(address _devaddr) external onlyOwner {
        devaddr = _devaddr;
        emit SetDevAddress(msg.sender, _devaddr);
    }

    function setFeeAddress(address _feeAddress) public {
        require(msg.sender == feeAddress, "setFeeAddress: FORBIDDEN");
        feeAddress = _feeAddress;
        emit SetFeeAddress(msg.sender, _feeAddress);
    }
```

If the dev address is accidentally set to zero, it breaks the functionality of the contract as the crucial `updatePool()` function fails due to the minting of the reward token to a zero address.

Recommendation

Add zero checks for the input values to avoid accidentally setting a wrong value and blocking the contract function.

Onyx team response

Since we are limiting emissions over the month of April - there will be no need to change the dev address to 0 until emissions are complete.

C2-07 Possible rounding error for tokens with large supply ● Info ✓ Acknowledged

An `e12` multiplier is used to calculate rewards which may be insufficient for pools with big token supply. This may lead to zero rewards for tokens with a big supply.

```
    function updatePool(uint256 _pid) public {
        ...
                pool.accOnyxPerShare =
 pool.accOnyxPerShare.add(onyxReward.mul(1e12).div(stakeSupply));
    }
```

Recommendation

Use a 1e18 multiplier for calculations.

C2-08 Possible exccedance of emission rate limit ● Info ✓ Acknowledged

The initial set of the `onyxPerSecond` variable is not constrained by the `MAX_EMISSION_RATE` limit, as opposed to the external `updateEmissionRate()` function.

```
constructor(
    OnyxToken _onyx,
    address _devaddr,
    address _feeAddress,
    uint256 _onyxPerSecond,
    uint256 _startTime
) {
    onyx = _onyx;
    devaddr = _devaddr;
    feeAddress = _feeAddress;
    onyxPerSecond = _onyxPerSecond;
    startTime = _startTime;
}

function updateEmissionRate(uint256 _onyxPerSecond) public onlyOwner {
    _updateEmissionRate(_onyxPerSecond);
    massUpdatePools();
}
```

Onyx team response

Made proper updates on the front-end.

C3. OnyxVault

Overview

A vault contract to be used with the single pool of the MasterChefOnyx farm contract. Rewards harvesting is public and taxable for 2 kinds of fees: performance fee (up to 5% to the Treasury address) and call fee (up to 1% to the caller of the `harvest()` function).

Issues

C3-01 Hardcoded pid ● Medium ✓ Acknowledged

The `pid` value for the contract is set in the constructor.

```
constructor(
    IERC20 _token,
    IMasterChef _masterchef,
    uint256 _pid,
    address _admin,
    address _treasury
) {
    token = _token;
    masterchef = _masterchef;
    admin = _admin;
    treasury = _treasury;
    pid = _pid;

    // Infinite approve
    IERC20(_token).safeApprove(address(_masterchef), type(uint256).max);
}
```

But some functions like `emergencyWithdraw()` and `balanceOf()` use a hardcoded zero pid. If the vault contract is deployed with a pid different from zero it won't be fully functional.

Recommendation

Remove the hardcoded zero value and use the `pid` variable instead.

Onyx team response

We won't be deploying this contract again or have other uses for it in a few weeks, so hardcoded is fine.

C3-02 Gas optimizations ● Low ✓ Acknowledged

1. The `isContract()` function checks `extcodesize` and can't be a reliable method in terms of detecting EOA, thus the `!isContract` check should be removed as the second check is much stricter.

2. Multiple reads of `totalShares` and `user.shares` from storage in the `deposit()` function.

3. Multiple reads of `totalShares` and `user.shares` from storage in the `withdraw()` function.

4. The `lastHarvestedTime` could be checked against the current time first, to eliminate possible same-block calls for `harvest()`.

5. `Pause` and `Unpause` events duplicate the ones from the Pausable contract.

C3-03 Lack of events ● Info ✓ Acknowledged

Lack of events in governance functions `setAdmin()`, `setTreasury()`, `setPerformanceFee()`, `setCallFee()`, `setWithdrawFee()`, `setWithdrawFeePeriod()`, and `emergencyWithdraw()`, which complicates the tracking of important changes off-chain.

Onyx team response

This pool will only be used for the month of April and is not essential for long-term governance

C3-04 Typos

● Info ✓ Acknowledged

Typos reduce the code's readability. Typo in 'targetted'.

C3-05 One-time approve

● Info ✓ Acknowledged

Even `type(uint256).max` allowance can be exhausted, especially if `token.decimals()` is high, limiting the lifetime of the contract.

```
constructor(
    IERC20 _token,
    IMasterChef _masterchef,
    uint256 _pid,
    address _admin,
    address _treasury
) {
    token = _token;
    masterchef = _masterchef;
    admin = _admin;
    treasury = _treasury;
    pid = _pid;

    // Infinite approve
    IERC20(_token).safeApprove(address(_masterchef), type(uint256).max);
}
```

C4. OnyxLockedVesting

Overview

A simple locking contract that holds any ERC20 tokens in favor of a single beneficiary for a fixed period of time.

Issues

C4-01 Gas optimizations ● Low ✓ Acknowledged

1. The `_beneficiary` and `_releaseTime` variables should be declared immutable.

2. No need in using the public functions `releaseTime()` and `beneficiary()` to access private variables for internal calculations.

C5. StakingPool

Overview

A single staking pool contract that allows depositing and withdrawing of predefined ERC20 token as well as receiving additional rewards (per staked amount and staking period) in the same or different ERC20 token or in the form of native currency.

Issues

C5-01 Rewards source and locked withdrawals ● Medium ✓ Acknowledged

The source of rewards is out of the scope, therefore rewards aren't guaranteed. At the same time the `withdraw()` function requires a successful transfer of pending rewards to complete a withdrawal.

```
    function withdraw(uint256 _amount) external nonReentrant {
        ...
        require(rewardBalance() >= pending, "insufficient reward balance");
        ...
    }
```

Recommendation

Consider allowing user withdrawals in case of insufficient reward balance by storing pending rewards to be claimed later.

Onyx team response

Mitigate with proper operational management of pools contracts & acknowledge use of emergencyWithdraw function means a user forfiets rewards.

C5-02 Gas optimizations

● Low ✓ Acknowledged

1. The `totalAllocPoint` and `poolInfo.allocPoint` are not used since the contract has only one pool.

2. Unnecessary reads from the storage of the `bonusEndTime` variable in the `setBonusEndTime()` function.

3. Multiple reads from the storage of `poolInfo.lastRewardTime` and `totalStaked` variables in the `pendingReward()` function.

4. Multiple reads from the storage of `user.amount`, `poolInfo.accRewardTokenPerShare` and `STAKE_TOKEN` variables in the `_depositTo()` function.

5. Multiple reads from the storage of `user.amount` and `poolInfo.accRewardTokenPerShare` variables in the `withdraw()` function.

6. Unnecessary reads from the storage of `REWARD_TOKEN` variable in the `rewardBalance()` function.

7. Unnecessary reads from the storage of `rewardPerSecond` variable in the `setRewardPerSecond()` function.

8. Multiple reads from the storage of `user.amount` variable in the `emergencyWithdraw()` function.

C5-03 Confusing error message ● Info ✓ Acknowledged

Confusing error message in the `setBonusEndTime()` function: it should say "new bonus end timestamp must be greater than current time" according to the code.

```
    function setBonusEndTime(uint256 _bonusEndTime) external onlyOwner {
        require(
            _bonusEndTime > block.timestamp,
            "new bonus end timestamp must be greater than current"
        bonusEndTime = _bonusEndTime;
        emit LogUpdatePool(bonusEndTime, rewardPerSecond);
    }
```

C6. StakingPoolFactory

Overview

A factory contract to deploy multiple StakingPool contracts, deployment is restricted to the owner. StakingPoolFactory doesn't store the addresses of deployed contracts.

Issues

C6-01 Typos ● Info ✓ Acknowledged

Typos reduce the code's readability. Typo in 'Deafult'.

5. Conclusion

4 medium, 8 low severity issues were found during the audit. No issues were resolved in the update.

The audited contracts are highly dependent on privileged accounts. Users using the project have to trust that the privileged accounts are properly secured.

Appendix A. Issues' severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Informational.** Issues that do not impact the contract operation. Usually, informational severity issues are related to code best practices, e.g. style guide.

Appendix B. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

contact@hashex.org

@hashex_manager

blog.hashex.org

linkedin

github

twitter

#HashEx

BLOCKCHAIN SECURITY