# yAudit Ooga Booga Review

**Review Resources:**

- Original xGrail contract's audit report: link

**Auditors:**

- adriro
- watermelon

## Table of Contents

## Review Summary

**Ooga Booga**

Ooga Booga provides staking and reward distribution functionalities for their own **`OOGA`** token.

The contracts of the Ooga Booga repository were reviewed over 4 days. Two auditors performed the code review between the 31st of March and the 3rd of April, 2025. The repository was under active development during the review, but the review was limited to the latest commit b56a7291e17711b680d481bf5d12bb2e28a6d277.

## Scope

The scope of the review consisted of the following contracts at the specific commit:

```
contracts
├── OogaRewards.sol
├── OOGA.sol
└── sOOGA.sol
```

> *"During the review, the Ooga Booga team opted to rename the `sOOGA.sol` contract to `bOOGA.sol`. This change was executed on the 2nd of April at commit [8890ea5614e1ed8b57861036026b3d6432d2dfc7](). The team plans to use `bOOGA.sol` for the foreseeable future."*

After the findings were presented to the Ooga Booga team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, Ooga Booga and users of the contracts agree to use the code at their own risk.

## Code Evaluation Matrix

| Category | Mark | Description |
|---|---|---|
| Access Control | Good | The contracts correctly use access control checks on functions that need them. |
| Mathematics | Good | Correctly implemented mathematical relations are in the reviewed contracts. |
| Complexity | Average | A straightforward staking algorithm is implemented, which presents a linear unvesting schedule. However, the complexity of the reward distribution cycle mechanism led to several findings. |
| Libraries | Good | The protocol builds on the OpenZeppelin library. |
| Decentralization | Average | Authorized parties are provided with the power to execute delicate configuration changes to the reviewed contracts. |
| Code stability | Good | The protocol's code is forked from deployed contracts with minor naming and contract configuration modifications. |
| Documentation | Average | Contracts have adequate inline comments, but no high-level documentation was provided. |
| Monitoring | Good | The reviewed contracts correctly emit events for crucial actions and state changes. |
| Testing and verification | Low | The provided contracts lack any tests. Developing a strong test suite with high line, function, and branch coverage is strongly encouraged. |

## Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.
- Gas savings
  - Findings that can improve the gas efficiency of the contracts.
- Informational

- Findings including recommendations and best practices.

---

## Critical Findings

None.

## High Findings

None.

## Medium Findings

### 1. Medium - Skipping the checkpointing of at least one cycle breaks distribution

When the rewards checkpointing isn't executed for at least the duration of a cycle, unexpected behavior occurs.

#### Technical Details

The `_updateRewardsInfo()` function deals with the checkpointing of the rewards accumulator and the logic to jump between cycles.

The implementation first handles the case of a cycle change (`lastUpdateTime < currentCycleStartTime`, lines 426-450) to update the period `[lastUpdateTime; currentCycleStartTime]` and reset the token amount for the subsequent distribution (`currentDistributionAmount`, line 439). It then continues to linearly distribute any elapsed time during the current cycle (i.e., `[lastUpdateTime; currentBlockTimestamp], lines 452-460)`).

This mechanism assumes that after switching cycles, the next cycle **is** the effective current cycle, which may not be the case if an entire cycle elapses between two consecutive calls to the function. In other words, the logic assumes that the algorithm is called at least once during each cycle.

Suppose we have three cycles and imagine the following scenario:

- Stored cycle start time is the start of cycle 1 ( `currentCycleStartTime = start1` ).
- The last updated timestamp is before the end of cycle 1 ( `lastUpdated < end1` ).
- Current timestamp is after the end of cycle 2 ( `currentBlockTimestamp > end2` ).

If we now call `_updateRewardsInfo()` the distribution will flush any pending rewards for the first cycle, calculate the next batch of rewards for cycle 2, and distribute any pending amount up to the current block timestamp. However, this includes the whole second cycle and any elapsed part of the third cycle. Note that `toDistribute` will be wrongly calculated since it will take the period `[lastUpdateTime = end1; currentBlockTimestamp]` , spanning more than one cycle. However, the amount will be capped by the if in lines 454-456. Finally, `lastUpdateTime` will be updated to the current block timestamp (line 460). This means that a single batch of rewards is distributed for cycles 2 and 3, effectively missing rewards for one period.

#### Impact

Medium. Rewards are not distributed if the algorithm is never checkpointed during the cycle.

#### Recommendation

The algorithm would need to be modified so that it considers not only any pending period from the stored current cycle but also loops between any entire cycles that might be present between the last checkpoint and the current timestamp. Alternatively, implement an off-chain measure to ensure that `updateRewardsInfo()` is called at least once during each cycle.

#### Developer Response

We are aware of this issue. The nature of the algorithm, being time based, creates this kind of apparent malfunctions. Though, in order to combat this, we have already set an off-chain cron job that calls `massUpdateRewardsInfo/updateRewardsInfo` in order to keep the contract's state healthy.

## Low Findings

### 1. Low - `emergencyWithdrawAll()` reverts due to nested reentrancy guard

The `emergencyWithdrawAll()` function uses the **`nonReentrant`** modifier and calls internally **`emergencyWithdraw()`**, which also has the modifier.

### Technical Details

```
    function emergencyWithdraw(IERC20 token) public nonReentrant onlyOwner {
        uint256 balance = token.balanceOf(address(this));
        require(balance > 0, "emergencyWithdraw: token balance is null");
        _safeTokenTransfer(token, msg.sender, balance);
    }
```

```
    function emergencyWithdrawAll() external nonReentrant onlyOwner {
        for (uint256 index = 0; index < _distributedTokens.length(); ++index) {
            emergencyWithdraw(IERC20(_distributedTokens.at(index)));
        }
    }
```

### Impact

Low.

### Recommendation

Refactor the functionality into an internal function that can be shared between the two external functions.

### Developer Response

Amended as per recommendation from auditors.

## 2. Low - Reward token cannot be re-enabled if whitelist is full

The max tokens length check would prevent re-enabling a reward if the whitelist is full.

### Technical Details

The `enableDistributedToken()` can be used to re-enable a previously disabled reward token.

```
    function enableDistributedToken(address token) external onlyOwner {
        RewardsInfo storage rewardsInfo_ = rewardsInfo[token];
```

```
325:         require(
326:             rewardsInfo_.lastUpdateTime == 0 ||
rewardsInfo_.distributionDisabled,
327:             "enableDistributedToken: Already enabled rewards token"
328:         );
329:         require(
330:             _distributedTokens.length() < MAX_DISTRIBUTED_TOKENS,
"enableDistributedToken: too many distributedTokens"
331:         );
332:         // initialize lastUpdateTime if never set before
333:         if (rewardsInfo_.lastUpdateTime == 0) {
334:             rewardsInfo_.lastUpdateTime = _currentBlockTimestamp();
335:         }
336:         // initialize cycleRewardsPercent to the minimum if never set before
337:         if (rewardsInfo_.cycleRewardsPercent == 0) {
338:             rewardsInfo_.cycleRewardsPercent =
DEFAULT_CYCLE_REWARDS_PERCENT;
339:         }
340:         rewardsInfo_.distributionDisabled = false;
341:         _distributedTokens.add(token);
342:         emit DistributedTokenEnabled(token);
343:     }
```

However, the length check in line 329 would prevent this when the current list is already full. This happens because a disabled token is not removed from the `_distributedTokens` set.

#### Impact

Low.

#### Recommendation

Change the condition to check if the token is already a member of the set: `_distributedTokens.contains(token) || _distributedTokens.length() < MAX_DISTRIBUTED_TOKENS`.

#### Developer Response

Amended as per recommendation from auditors.

### 3. Low - `pendingRewardsAmount` returns incorrect value if `totalAllocation == 0` and users have pending rewards

`OogaRewards.pendingRewardsAmount` early returns `0` for every `(token, userAddress)` parameter combination if `totalAllocation == 0`. This value is incorrect if the user has pending rewards to harvest.

#### Technical Details

```
157:    function pendingRewardsAmount(address token, address userAddress) external view returns (uint256) {
158:        if (totalAllocation == 0) {
159:            return 0; // AUDIT no allocation => no pending rewards? - what about user.pendingRewards?
160:        }
           ...
187:    }
```

#### Impact

Low.

#### Recommendation

If `totalAllocation == 0`, the method should avoid executing the logic to account for any additional rewards that have been distributed and use the current `return` expression.

#### Developer Response

Amended as per recommendation from auditors.

## Gas Saving Findings

### 1. Gas - Cycle duration can be a constant

The `_cycleDurationSeconds` variable is allocated in storage and initialized at construction time but is never modified in the contract.

#### Technical Details

```
65:     uint256 internal _cycleDurationSeconds = 7 days;
```

#### Impact

Gas savings.

#### Recommendation

Use a constant to define the cycle duration.

#### Developer Response

Amended as per recommendation from auditors.

### 2. Gas - Reward debt is calculated twice in `_harvestRewards()`

The same calculation is performed twice in the implementation of [`_harvestRewards()`].

#### Technical Details

The calculation of `userSOOGAAllocation.mul(accRewardsPerShare).div(1e18)` is repeated in lines 500 and 503.

```
499:         uint256 pending =
500:
user.pendingRewards.add(userSOOGAAllocation.mul(accRewardsPerShare).div(1e18).sub
(user.rewardDebt));
501:
502:         user.pendingRewards = 0;
503:         user.rewardDebt =
userSOOGAAllocation.mul(accRewardsPerShare).div(1e18);
```

#### Impact

Gas savings.

#### Recommendation

Move the repeated calculation to a local variable.

#### Developer Response

Amended as per recommendation from auditors.

### 3. Gas - Use `block.timestamp` instead of an internal getter

[`OogaRewards._currentBlockTimestamp`] and [`sOOGA._currentBlockTimestamp`] solely return `block.timestamp`, making them unnecessary internal methods.

#### Technical Details

Using an internal function adds gas overhead and extra bytecode related to jumping to the function's body and stack management operations, given that the highlighted method is solely used to fetch the current block's timestamp, using `block.timestamp` instead of `_currentBlockTimestamp()` reduces the contract's gas consumption and offers fully equivalent functionality.

#### Impact

Gas savings.

#### Recommendation

Replace every call to `_currentBlockTimestamp` with `block.timestamp` and remove the internal methods.

#### Developer Response

Amended as per recommendation from auditors.

### 4. Gas - Cache repeated calls to internal view methods

To reduce the contract's gas consumption, avoid calling methods that return the same value multiple times within the same transaction.

#### Technical Details

- `OogaRewards.pendingRewardsAmount` invokes `nextCycleStartTime` up to three times.
- `OogaRewards.pendingRewardsAmount` invokes `_currentBlockTimestamp` twice.

#### Impact

Gas savings.

#### Recommendation

Cache the methods' return values and avoid executing the same internal call more than once.

#### Developer Response

Amended as per recommendation from auditors.

## Informational Findings

## 1. Informational - Unnecessary usage of `SafeMath` library

`SafeMath` library isn't necessary in contracts built with Solidity `0.8.x`.

### Technical Details

By default, Solidity 0.8.x versions offer checked mathematical operations, making the use of libraries like SafeMath unnecessary.

### Impact

Informational.

### Recommendation

Refactor `OogaRewards.sol` and `sOOGA.sol` to use native Solidity arithmetical operations instead of `SafeMath`'s methods, taking special care to maintain the order in which such operations are executed.

### Developer Response

Amended as per recommendation from auditors.

## 2. Informational - `nextCycleStartTime()` can return stale information when queried from the outside

The function returns the previous cycle start time if the state hasn't been checkpointed.

### Technical Details

The implementation of `nextCycleStartTime()` simply returns `currentCycleStartTime + _cycleDurationSeconds` without considering if the cycle determined by `currentCycleStartTime` has already elapsed.

### Impact

Informational.

### Recommendation

Considering this function is used internally to determine if the cycle needs to be updated, it must be split into two different functions. Alternatively, document that `nextCycleStartTime()` could return stale information if the cycle hasn't been checkpointed yet during the current cycle.

### Developer Response

We are aware of this issue. The nature of the algorithm, being time based, creates this kind of apparent malfunctions. Though, in order to combat this, we have already set an off-chain cron job that calls `massUpdateRewardsInfo/updateRewardsInfo` in order to keep the contract's state healthy.

## 3. Informational - Reward dust amounts accumulation if `rewardsInfo[token] < 100%`

If for a given `token` it holds that `rewardsInfo[token].cycleRewardsPercent < MAX_CYCLE_REWARDS_PERCENT`, minimal amounts of reward tokens will never be distributed to users because of an implicit floor division.

### Technical Details

[`OogaRewards._updateRewardsInfo`] allocates part of the pending reward amount to be distributed in the new distribution cycle whenever a cycle change is detected. The method allocates such funds by calculating the fraction of pending rewards, based on `rewardInfo[token].cycleRewardsPercent`:

```
441:                    uint256 pendingAmount = rewardsInfo_.pendingAmount;
442:                    currentDistributionAmount =
pendingAmount.mul(rewardsInfo_.cycleRewardsPercent).div(10000);
443:                    rewardsInfo_.currentDistributionAmount =
currentDistributionAmount;
444:                    rewardsInfo_.pendingAmount =
pendingAmount.sub(currentDistributionAmount);
```

By rounding down the division at L442, the newly distributed token amount can be computed as 0 in the off-chance that `pendingAmount * rewardsInfo_.cycleRewardsPercent < 10000`. In this case, no tokens will be allocated to the next cycle, and the pending token amount will remain unchanged.

### Impact

Informational.

### Recommendation

Add a check to catch the edge case in which `pendingAmount > 0 && currentDistributionAmount == 0` and allocate all remaining pending rewards to be distributed on the next cycle.

#### Developer Response

Amended as per recommendation from auditors.

### 4. Informational - Rewards are distributed from the second cycle if `_updateRewardsInfo` isn't triggered during the first cycle

[`OogaRewards`] allows to set the `startTime` during construction, which dictates the timestamp at which rewards are supposed to be distributed.

#### Technical Details

Once the first cycle has begun, if no account triggers a call to `_updateRewardsInfo`, the contract will fail to detect that the first cycle has already started. It will ultimately detect a cycle change once the cycle has begun and stream rewards starting from the beginning of the second cycle.

This occurs because cycle changes are detected by checking that `block.timestamp > currentCycleStartTime + _cycleDurationSeconds`. Because `currentCycleStartTime = startTime` is written within the contract's constructor, no cycle change will be computed until after `currentCycleStartTime + _cycleDurationSeconds`.

Note that this also applies to users who have created allocations **before** `startTime` and plan on simply collecting fees after the first cycle has passed.

#### Impact

Informational

#### Recommendation

The issue can be mitigated by invoking [`OogaRewards.updateRewardsInfo`] or modifying an account's allocation for every enabled token at some point in time within the first cycle.

#### Developer Response

We are aware of this issue. The nature of the algorithm, being time based, creates this kind of apparent malfunctions. Though, in order to combat this, we have already set an off-chain cron job that calls `massUpdateRewardsInfo/updateRewardsInfo` in order to keep the contract's state healthy.

## 5. Informational - `pendingRewardsAmount` returns incorrect amount if token was disabled

`OogaRewards.pendingRewardsAmount` fails to differentiate between enabled and disabled. When using a disabled token, the method incorrectly accounts for rewards distributed in the current cycle when it must account for distributions across the past and current cycles.

### Technical Details

`OogaRewards.disableDistributedToken` allows for the contract owner to schedule a pause on the emissions of rewards for a given token. Such pause becomes effective at the beginning of the next cycle, as can be seen within `OogaRewards.sol#L441-L446`.

### Impact

Informational.

### Recommendation

The method should avoid calculating the rewards for the new cycle if the given token is disabled.

### Developer Response

Amended as per recommendation from auditors.

## 6. Informational - Insufficient validation in `sOOGA.updateRedeemSettings`

`sOOGA.updateRedeemSeettings` allows the contract owner to modify contract parameters crucial for the `sOOGA` redemption process.

### Technical Details

The newly written values aren't required to adhere to strictly defined thresholds, specifically:

1. `minRedeemRatio` and `maxRedeemRatio` are allowed to be set to `0%`.

2. `minRedeemDuration` and `maxRedeemDuration` are allowed to be set to any arbitrary time duration.

### Impact

Informational.

### Recommendation

We recommend adding `require` statements to impose reasonable thresholds on such parameters. In particular:

1. `maxRedeemRatio_` should be above a given value to ensure the contract always allows a reasonable, partial redemption.
2. `maxRedeemDuration_` should be below a given value to ensure that a full redemption lasts no more than a reasonable amount of time and that the redemption's linear vesting does not occur within an excessive time period.
3. `minRedeemDuration_` should be below a given value to ensure users can reach the minimum redeem ratio within a reasonable time frame.

### Developer Response

From a business standpoint we would not like to set hard bounds to these variables. There might be situations where we would like to have full flexibility over the configuration of these parameters. Therefore, we will keep as is, without imposing any limits.

## 7. Informational - OOGA distributor address is an EOA

The recipient of the OOGA supply is an externally owned account.

### Technical Details

[OOGA.sol#L13]

```
13:     address public constant DISTRIBUTOR_ADDRESS =
0x4b741204257ED68A7E0a8542eC1eA1Ac1Db829d7;
```

### Impact

Informational.

### Recommendation

Consider using a multi-sig account.

**Developer Response**

All tokens are transferred to a multisig immediately upon deployment.

## Final Remarks

The Ooga Booga staking platform is forked from contracts originally developed for Camelot protocol. The security review focused on the rewards contract since it wasn't previously audited in the original codebase.

An assumption in how the distribution algorithm works could lead to a missing rewards period if no activity occurs during the cycle duration. Given the low likelihood of such an event, and considering that the cycle duration will be in the magnitude of days, the decision to control the checkpoint and trigger an automatic event was favored over reworking the algorithm. Other than this, the contracts were found to work correctly, with no high-severity issues detected.

Even though the codebase is mostly forked from contracts already used in production, auditors highlight the importance of having a strong test suite that could also serve as a basis for regression testing while implementing minor modifications.

Overall, the protocol's reward distribution functions correctly and as expected, provided the system is maintained and used minimally throughout its lifespan.