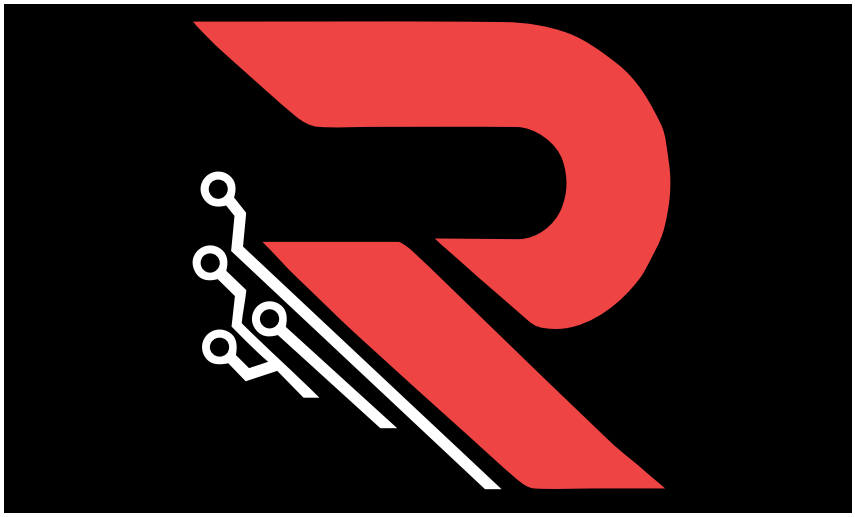
KaiAura Security Review



Version 2.0

21.12.2024

Conducted by:

MaslarovK, Lead Security Researcher

radev-eth, Lead Security Researcher

Table of Contents

1 About MaslarovK

MaslarovK is a security researcher from Bulgaria. Co-Founder of Rezolv Solutions.

2 About radev.eth

radev_eth is a security researcher from Bulgaria. Co-Founder of Rezolv Solutions.

3 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

4 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

4.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

4.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

5 Executive summary

Overview

Project Name	KaiAura
Repository	https://github.com/kai-aura/kai-Contracts/tree/main/contracts/core
Commit hash	47f903e381bab80c63824e7bd7bc110668149cc3
Resolution	N/A
Documentation	https://docs.kaiaura.finance/
Methods	Manual review

Scope

contracts/core/Aura.sol
contracts/core/AuraMinter.sol
contracts/core/KaiAura.sol
contracts/core/KaiLocker.sol
contracts/core/MetaRewardPool.sol
contracts/core/CommnalFarm.sol
contracts/core/MultiRewardPool.sol

Issues Found

Critical risk	0
High risk	0
Medium risk	1
Low risk	6
Informational	6

6 Findings

6.1 Medium risk

7 [M-01] Missing `MAX_SUPPLY` check in `Aura.sol#minterMint()` function

Summary

The `minterMint` function does not check against `MAX_SUPPLY`, allowing the minter to mint more tokens than the maximum supply or even an infinite amount.

Impact

This bypasses the supply cap defined by `MAX_SUPPLY`, potentially leading to inflation and loss of trust in the token's supply limits.

Tools Used

- Manual Review

Recommended Mitigation Steps

Add a MAX_SUPPLY check:

```
function minterMint(address _to, uint256 _amount) external {
    require(msg.sender == minter || allowedMinters[msg.sender] == true, "Only minter
        ");
    require(_amount + totalSupply() <= MAX_SUPPLY, "Would exceed max supply");
    minterMinted = minterMinted.add(_amount);
    _mint(_to, _amount);
}
```

Resolution: Acknowledged with plan to resolve issue during a transition during a subsequent phase.

7.1 Low risk

8 [L-01] No verification of AuraMinter as an authorized Minter in AuraToken

Summary

The `AuraMinter` contract calls `aura.minterMint()` in the `mint` function. However, the `AuraToken` contract does not ensure that `AuraMinter` is registered as the `minter` or is included in the `allowedMinters` mapping.

Issue:

- The `minterMint` function in `AuraToken` requires the caller (`msg.sender`) to either:
    1. Be the `minter`, or
    2. Be in the `allowedMinters` mapping.
- Without explicit registration, `AuraMinter` might not have permission to mint tokens.

Impact - If `AuraMinter` is not properly authorized in `AuraToken`: - The `mint` function in `AuraMinter` will revert when calling `aura.minterMint()`. - The system relying on `AuraMinter` for inflation will fail to operate as intended.

Tool used

- Manual Review

Recommendation

Ensure that `AuraMinter` is set as the `minter` or added to `allowedMinters` in `AuraToken` during deployment or initialization.

Example Initialization Process:

```
function initializeMinter() external onlyOwner {
    aura.setAllowedMinter(address(this), true); // Add ‘AuraMinter‘ to ‘
        allowedMinters‘
}
```

Alternatively, ensure `AuraMinter` is explicitly set as the `minter`:

```
function initializeMinter() external onlyOwner {
    aura.setMinter(address(this)); // Hypothetical method to set ‘minter‘
}
```

Resolution: Aknowledged

9 [L-02] No address(0) check in Aura.sol::setOperator and KaiAura.sol::setOperator

Summary

No address(0) check might lead to `_operator` being set to address(0) and could not be changed after.

```
function setOperator(address _operator) external {
        //@audit add the following check here:
        require(_operator != address(0));
        require(msg.sender == operator, "Only operator");
        operator = _operator;
    }
```

Tool used

- Manual Review

Resolution: Aknowledged

10 [L-03] Off-by-one issue in Aura.sol::Mint

Summary

This off-by-one issue prevents the last token of the `MAX_SUPPLY` to be minted.

```
function mint(address _to, uint256 _amount) external {
        require(totalSupply() != 0, "Not initialised");
        //@audit off-by-one, change it to <=
        require(_amount + totalSupply() < MAX_SUPPLY, "Would exceed max supply");

        if (msg.sender != operator) {
            return;
        }

        _mint(_to, _amount);
    }
```

Tool used

- Manual Review

Resolution: Aknowledged,the Tokenomic design makes it it impossible to mint the remaining tokens to reach Max Supply.

11 [L-04] Lack of address(0) check in CommunaFarm.sol::depositFor

Summary

Add `address(0)` check for _for, if wrongly set to `address(0)` it might lead to funds being transferred, but not associated with any user.

```
function depositFor(
        uint256 _amount,
        address _for
    ) public nonReentrant returns (bool) { user
        _processStake(_amount, _for);

        //take away from sender
        stakingToken.safeTransferFrom(msg.sender, address(this), _amount);
        emit Staked(_for, _amount);

        return true;
    }
```

Tool used

- Manual Review

Resolution: Aknowledged

12 [L-05] Use `safeTransfer` instead of `transfer` in MultiRewardPool.sol::_withdrawAndUnwrapTo

Summary

```
function _withdrawAndUnwrapTo(
      uint256 amount,
      address from,
      address receiver
  ) internal updateReward(from) returns (bool) {
      _totalSupply = _totalSupply.sub(amount);
      _balances[from] = _balances[from].sub(amount);
      //@audit Use ‘safeTransfer‘ instead of ‘transfer‘
      stakingToken.transfer(receiver, amount);
      emit Withdrawn(from, amount);

      return true;
  }
```

Tool used

- Manual Review

Resolution: Aknowledged

13 [L-06] Out-of-bounds access in `_earned()` function of `MetaRewardPool.sol`

Summary

The `_earned` function in the `MetaRewardPool` contract is prone to a **revert** due to an **out-of-bounds array access** when the `rewardCheckpointsMap[_rewardsToken]` array contains **fewer than two elements**. This leads to a systemic failure where users are unable to claim rewards via the `getReward()` function.

Description of the Bug

The `_earned` function is designed to calculate the rewards a user has earned for a given reward token (`_rewardsToken`). It iterates over the `rewardCheckpoints` array (fetched from `rewardCheckpointsMap[_rewardsToken]`) to compute the user's prorated share of rewards based on staking and withdrawal activity.

However, the function assumes that: 1. The `rewardCheckpoints` array always contains at least two elements. 2. The second element (`rewardCheckpoints[1]`) will always exist.

When the array has fewer than two elements, the following line in `_earned` leads to an **out-of-bounds access**, causing a revert:

```
uint256 checkpointDuration = rewardCheckpoints[i].date - rewardCheckpoints[i - 1].
    date;
```

Here, `rewardCheckpoints[i]` at `i = 1` does not exist if the array contains only one element (or is empty).

Impact

1. **Users Cannot Claim Rewards**:
    - If `rewardCheckpointsMap[_rewardsToken]` contains fewer than two elements for a reward token, any user calling `getReward()` will encounter a revert because `_earned()` is invoked in the function flow.
2. **Broken Reward Distribution**:
    - Rewards cannot be calculated or claimed for tokens where checkpoints are improperly initialized or not updated.

Steps to Reproduce

1. **Initial Setup**:
    - Deploy the `MetaRewardPool` contract.
    - Deposit 100 tokens for a user.

    Example:

```
    // User deposits 100 tokens
    balances[user] = 100;
    deposits[user].push(TransactionData({ date: 100000, amount: 100 }));
```

2. **Create Initial Checkpoint**:
    - Harvest rewards for the first time, creating the first `RewardCheckpoint` with:

```
    rewardCheckpointsMap[_rewardsToken].push(
        RewardCheckpoint({ date: 110000, amount: 0, totalSupply: 200 })
    );
```

3. **User Calls `getReward()`**:
    - The user calls `getReward()` to claim rewards:

```
    metaRewardPool.getReward(user);
```

4. **Reversion Occurs**:
    - Inside the `_earned()` function, the loop starts:

```
    uint256 checkpointStart = rewardCheckpoints[i - 1].date; // i = 1,
        accessing index 0
    uint256 checkpointDuration = rewardCheckpoints[i].date - checkpointStart;
        // i = 1, accessing index 1
```

    - Since `rewardCheckpoints` contains only one element (index `0`), accessing `rewardCheckpoints[1]` results in an out-of-bounds revert.

- **Detailed Bug Flow**

- The `rewardCheckpointsMap[_rewardsToken]` array is empty or contains only one element:

```
rewardCheckpointsMap[_rewardsToken] = [
    RewardCheckpoint({ date: 110000, amount: 0, totalSupply: 200 })
];
```

Function Call Flow

1. **User Calls `getReward()`**:
   - Triggers `_claimableRewards()`:

```
EarnedData[] memory userRewards = _claimableRewards(user);
```

2. **Harvesting Rewards (`_harvestRewards()`)**:
   - Adds or updates checkpoints for the underlying reward tokens.
3. **Calculating Rewards (`_earned()`)**:
   - Loops through `rewardCheckpoints` for each token:

```
RewardCheckpoint[] memory rewardCheckpoints = rewardCheckpointsMap[
    _rewardsToken];
```

4. **Out-of-Bounds Access**:
   - The loop in `_earned` begins with `i = 1`, but `rewardCheckpoints[i]` (index `1`) does not exist if the array contains fewer than two elements:

```
uint256 checkpointStart = rewardCheckpoints[i - 1].date; // OK, index
    0
uint256 checkpointDuration = rewardCheckpoints[i].date -
    checkpointStart; // Reverts, index 1 out-of-bounds
```

Result

- The `_earned` function reverts, causing the `getReward()` call to fail. Users cannot claim rewards.

Examples

Example 1: Successful Case

- **Input**:
  * `rewardCheckpoints` contains two elements:

```
rewardCheckpoints = [
    RewardCheckpoint({ date: 100000, amount: 50, totalSupply: 100 }),
    RewardCheckpoint({ date: 110000, amount: 100, totalSupply: 200 })
];
```

- **Outcome**:
  * `_earned()` computes the rewards correctly by iterating over checkpoints.

Example 2: Failing Case

- **Input**:
  - `rewardCheckpoints` contains only one element:

```
rewardCheckpoints = [
    RewardCheckpoint({ date: 110000, amount: 0, totalSupply: 200 })
];
```

- **Outcome**:
  - `_earned()` reverts when attempting to access `rewardCheckpoints[1]`. **Tools Used**

- Manual Review

Recommended Mitigation Steps

Fix 1: Early Check in `_earned()`

```
Add a check to ensure the ‘rewardCheckpoints‘ array contains at least two elements:

‘‘‘solidity
if (rewardCheckpoints.length < 2) {
    return 0; // No rewards can be calculated
}

‘‘‘

**Fix 2: Initialize Checkpoints in ‘_harvestRewards()‘**

Ensure that ‘_harvestRewards()‘ initializes at least one valid checkpoint if none
    exists:

‘‘‘solidity
if (rewardCheckpointsMap[_rewardsToken].length == 0) {
    rewardCheckpointsMap[_rewardsToken].push(
        RewardCheckpoint({ date: block.timestamp, amount: 0, totalSupply:
            stakedSupply })
    );
}

‘‘‘

**Severity**

- **High**:
    - Users are completely blocked from claiming rewards in the affected scenario.
    - This issue impacts the core functionality of the contract, rendering it
        unusable for reward distribution.
```

- **Conclusion**

  This bug is about the reward distribution mechanism. The proposed fixes ensure robustness by:

  1. Preventing reverts when checkpoints are insufficient.

2. Guaranteeing proper initialization of checkpoints.
3. Allowing users to claim rewards reliably.

Resolution: Aknowledged, when reward is added, it creates two checkpoints if none exist.

13.1 Informational risk

14 [I-01] `safeIncreaseAllowance` and `approve` used one after another in `_deposit`

Summary

In the `MetaRewardPool.sol` , the `safeIncreaseAllowance` and `approve` used one after another in `_deposit`. The use of `safeIncreaseAllowance` is not necessary, because only the amount deposited must be approved. Users will be paying extra gas, use `forceApprove` only.

Resolution: Aknowledged

15 [I-02] Move `require(msg.sender == operator, "Only operator");` in modifier

Summary

Note: The recommendation suggests moving the `require(msg.sender == operator, "Only operator");` check into a separate modifier.

Reasoning:

- This check is repeated across multiple functions (`init`, `setAllowedMinter`, `setOperator`).
- Using a modifier such as `onlyOperator` will improve code readability and maintainability.

Example Fix:

```
modifier onlyOperator() {
    require(msg.sender == operator, "Only operator");
    _;
}
```

Update relevant functions:

```
function init(address _to, address _minter) external onlyOperator { ... }
function setAllowedMinter(address _minter, bool _isAllowed) external onlyOperator {
    ... }
function setOperator(address _operator) external onlyOperator { ... }
```

Resolution: Aknowledged

16 [I-03] Ensure new operator is different in Aura.sol and KaiAura.sol contracts (and not zero address)

Summary

Add a check to ensure that the new operator is different from the current operator in the `setOperator` function.

Impact

- Without this check, setting the operator to the same address may result in unnecessary updates and confusion.

Example Fix

```
function setOperator(address _operator) external onlyOperator {
    require(_operator != operator, "Operator is already set to this address");
    operator = _operator;
}
```

Same in **`KaiAura.sol`** contract**.**

16.1 Resolution: Aknowledged

17 [I-04] Off-by-one issue in Aura.sol#mint() function

Summary

The current check **`require`**`(_amount + totalSupply()< MAX_SUPPLY, "Would exceed max supply ");` is incorrect due to an **off-by-one error**.

Impact:

- If `_amount + totalSupply()` equals `MAX_SUPPLY`, the condition fails to prevent minting, causing an overflow of the maximum supply.

Example Fix:

```
require(_amount + totalSupply() <= MAX_SUPPLY, "Would exceed max supply");
```

Resolution: Aknowledged

18 [I-05] Initialized variable set to immutable instead of constant

Summary

MetaRewardPool.sol

```
address public immutable auraAddress = 0xC0c293ce456fF0ED870ADd98a0828Dd4d2903DBF;
```

Resolution: Aknowledged

19 [I-06] SafeMath in CommunalFarm.sol is not needed after Solidity 0.8.0

Summary

```
using SafeMath for uint256;//@audit not needed
```

Resolution: Aknowledged

20 [I-06] There is no check if token already exists in the array

Summary

CommunalFarm.sol:

```
function addNewRewardToken(string memory _rewardSymbol, address _rewardToken,
    address _rewardManager, uint256 _rewardRate) external onlyOwner {
        //@audit validate the token is not present in the array
        _updateStoredRewardsAndTime();

        rewardTokens.push(_rewardToken);
        rewardRates.push(_rewardRate);
        rewardSymbols.push(_rewardSymbol);

        rewardTokenAddrToIdx[_rewardToken] = rewardTokens.length - 1;
        rewardsPerTokenStored.push(0);
        rewardManagers[_rewardToken] = _rewardManager;

        emi
```

Resolution: Aknowledged