SMART CONTRACT AUDIT REPORT

for

Aura Finance

Prepared By: Patrick Lou

PeckShield

April 18, 2022

Document Properties

Client	Aura Finance
Title	Smart Contract Audit Report
Target	Aura Finance
Version	1.0-rc
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	April 18, 2022	Xiaotao Wu	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the `Aura Finance` design document and related smart contract source code, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Aura Finance

`Aura Finance` is a protocol built on top of the `Balancer` system to provide maximum incentives to `Balancer LPs` and `BAL` stakers through social aggregation of `BAL` deposits and `Aura`'s native token `AURA`. It is a fork of `Convex Finance` with additional adaptations to make the protocol more generic. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Aura Finance

Item	Description
Name	Aura Finance
Website	https://aura.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 18, 2022

In the following, we show the Git repositories of reviewed files and the commit hash value used in this audit. Note this audit only covers the `Aura.sol`, `AuraBalRewardPool.sol`, `AuraLocker.sol`, `AuraMath.sol`, `AuraMinter.sol`, `AuraStakingProxy.sol`, `AuraVestedEscrow.sol`, `BalInvestor.sol`, `CrvDepositorWrapper.sol` `Booster.sol`, `BaseRewardPool4626.sol`, `BaseRewardPool.sol`, and `VoterProxy.sol` contracts.

- https://github.com/aurafinance/aura-contracts.git (f5249fc)
- https://github.com/aurafinance/convex-platform.git (e1add5b)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/aurafinance/aura-contracts.git (456bd50)
- https://github.com/aurafinance/convex-platform.git (cc2c8fc)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [9]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Aura Finance` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	3	■ ■ ■
Informational	2	■ ■
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 3 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Aura Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incorrect delegatee Updating In AuraLocker::_processExpiredLocks()	Business Logic	Resolved
PVE-002	Informational	Type Mismatch In AuraLocker::updateReward()	Coding Practices	Resolved
PVE-003	Low	Improved Logic In AuraLocker::getReward()	Business Logic	Resolved
PVE-004	Informational	Meaningful Events For Important State Changes	Coding Practices	Confirmed
PVE-005	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Incorrect delegatee Updating In AuraLocker::_processExpiredLocks()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: AuraLocker
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `AuraLocker` contract provides an external `kickExpiredLocks()` function for users to unlock the staked tokens of a staker if the unlock time plus the grace period is met. A portion of the unlocked tokens will be sent to the function caller as a reward. This function internally calls the low-level helper routine, i.e., `_processExpiredLocks()`, to calculate the reward for the `msg.sender`, transfer the staked tokens to the staker, and transfer the reward to the `msg.sender`. Our analysis with this routine shows its current implementation for updating the checkpoint of the staker's delegatee is not correct.

To elaborate, we show below its code snippet. Specifically, the delegatee account used for updating the checkpoint should be `delegates(_account)`, instead of current `delegates(msg.sender)` (line 369).

```
346     function kickExpiredLocks(address _account) external nonReentrant {
347         //allow kick after grace period of 'kickRewardEpochDelay'
348         _processExpiredLocks(_account, false, msg.sender, rewardsDuration.mul(
                kickRewardEpochDelay));
349     }
```

Listing 3.1: `AuraLocker::kickExpiredLocks()`

```
346     // Withdraw all currently locked tokens where the unlock time has passed
347     function _processExpiredLocks(
348         address _account,
```

```
        bool _relock,
        address _rewardAddress,
        uint256 _checkDelay
    ) internal updateReward(_account) {
        LockedBalance[] storage locks = userLocks[_account];
        Balances storage userBalance = balances[_account];
        uint112 locked;
        uint256 length = locks.length;
        uint256 reward = 0;
        uint256 expiryTime = _checkDelay == 0 && _relock
            ? block.timestamp.add(rewardsDuration)
            : block.timestamp.sub(_checkDelay);
        require(length > 0, "no locks");
        ...

        //update user balances and total supplies
        userBalance.locked = userBalance.locked.sub(locked);
        lockedSupply = lockedSupply.sub(locked);

        //checkpoint the delegatee
        _checkpointDelegate(delegates(msg.sender), 0, 0);

        emit Withdrawn(_account, locked, _relock);
        ...
    }
```

Listing 3.2: `AuraLocker::_processExpiredLocks()`

Recommendation Use the correct delegatee account to update the checkpoint.

Status This issue has been fixed in this commit: `456bd50`.

3.2 Type Mismatch In AuraLocker::updateReward()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `AuraLocker`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

The `AuraLocker` contract implements the `updateReward()` modifier to update reward for a specified account. While examining the `updateReward()` modifier, we notice there is a type mismatch when updating reward related informations for an account.

To elaborate, we show below its code snippet. Specifically, the member variables `rewards` and `rewardPerTokenPaid` in structure `UserData` is defined as `uint128`, but the current implementation assigns `uint112` values to these two variables (lines 181-182).

```
170     modifier updateReward(address _account) {
171         {
172             Balances storage userBalance = balances[_account];
173             uint256 rewardTokensLength = rewardTokens.length;
174             for (uint256 i = 0; i < rewardTokensLength; i++) {
175                 address token = rewardTokens[i];
176                 uint256 newRewardPerToken = _rewardPerToken(token);
177                 rewardData[token].rewardPerTokenStored = newRewardPerToken.to96();
178                 rewardData[token].lastUpdateTime = _lastTimeRewardApplicable(rewardData[
                        token].periodFinish).to32();
179                 if (_account != address(0)) {
180                     userData[_account][token] = UserData({
181                         rewardPerTokenPaid: newRewardPerToken.to112(),
182                         rewards: _earned(_account, token, userBalance.locked).to112()
183                     });
184                 }
185             }
186         }
187         _;
188     }
```

Listing 3.3: `AuraLocker::updateReward()`

Recommendation Assign `uint128` values for member variables `rewards` and `rewardPerTokenPaid`.

Status This issue has been fixed in this commit: `456bd50`.

3.3 Improved Logic In AuraLocker::getReward()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AuraLocker`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `getReward` contract provides a public `getReward()` function for users to claim all pending rewards for a specified account. While examining the routine, we notice the current implementation logic can be improved.

To elaborate, we show below its code snippet. It comes to our attention that the current implementation allows a user to stake the `cvxCrv` reward for the specified account, which may not be the `_account`'s real intention (lines 311-312).

```
303     // Claim all pending rewards
304     function getReward(address _account, bool _stake) public nonReentrant updateReward(
            _account) {
305         uint256 rewardTokensLength = rewardTokens.length;
306         for (uint256 i; i < rewardTokensLength; i++) {
307             address _rewardsToken = rewardTokens[i];
308             uint256 reward = userData[_account][_rewardsToken].rewards;
309             if (reward > 0) {
310                 userData[_account][_rewardsToken].rewards = 0;
311                 if (_rewardsToken == cvxCrv && _stake) {
312                     IRewardStaking(cvxcrvStaking).stakeFor(_account, reward);
313                 } else {
314                     IERC20(_rewardsToken).safeTransfer(_account, reward);
315                 }
316                 emit RewardPaid(_account, _rewardsToken, reward);
317             }
318         }
319     }
```

Listing 3.4: `AuraLocker::enterRaffle()`

Recommendation Only allow the function caller to stake the the `cvxCrv` reward if `msg.sender == _account`.

Status This issue has been fixed in this commit: `456bd50`.

3.4 Meaningful Events For Important State Changes

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `AuraLocker` contract as an example. While examining the events that reflect the `AuraLocker` dynamics, we notice there is a lack of emitting related events to reflect important state changes. Specifically, when the `addReward()/approveRewardDistributor()` are being called, there are no corresponding events being emitted to reflect the occurrence of `addReward()/approveRewardDistributor()`.

```
    // Add a new reward token to be distributed to stakers
    function addReward(address _rewardsToken, address _distributor) external onlyOwner {
        require(rewardData[_rewardsToken].lastUpdateTime == 0, "Reward already exists");
        require(_rewardsToken != address(stakingToken), "Cannot add StakingToken as reward");
        rewardTokens.push(_rewardsToken);
        rewardData[_rewardsToken].lastUpdateTime = uint32(block.timestamp);
        rewardData[_rewardsToken].periodFinish = uint32(block.timestamp);
        rewardDistributors[_rewardsToken][_distributor] = true;
    }

    // Modify approval for an address to call notifyRewardAmount
    function approveRewardDistributor(
        address _rewardsToken,
        address _distributor,
        bool _approved
    ) external onlyOwner {
        require(rewardData[_rewardsToken].lastUpdateTime > 0, "Reward does not exist");
        rewardDistributors[_rewardsToken][_distributor] = _approved;
    }
```

Listing 3.5: `AuraLocker::addReward()/approveRewardDistributor()`

Note a number of routines in the `Aura Finance` contracts can be similarly improved, including `AuraStakingProxy::setCrvDepositorWrapper()/setKeeper()/setPendingOwner()/applyPendingOwner()/setRewards()`, `AuraVestedEscrow::setAdmin()/setLocker()`, `BaseRewardPool::addExtraReward()/clearExtraRewards()`, `CrvDepositor::setFeeManager()/setFees()/setCooldown()`, and `CurveVoterProxy::setOwner()/setRewardDeposit()/setOperator()/setDepositor()/setStashAccess()/setVote()`.

Recommendation Properly emit the related event when the above-mentioned functions are being invoked.

Status This issue has been confirmed.

3.5 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf
    of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
}
```

Listing 3.6: USDT Token **Contract**

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20Upgradeable token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 3.7: `SafeERC20Upgradeable::safeApprove()`

In the following, we use the `CurveVoterProxy::withdraw()` routine as an example. This routine is designed to withdraw the distributed `ERC20` tokens from the vote contract to the `rewardDeposit` contract as extra rewards. To accommodate the specific idiosyncrasy, there is a need to `safeApprove()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

```
    /**
     * @notice  Withdraw ERC20 tokens that have been distributed as extra rewards
     * @dev     Tokens shouldn't end up here if they can help it. However, dao can
     *          set a withdrawer that can process these to some ExtraRewardDistribution.
     */
    function withdraw(IERC20 _asset) external returns (uint256 balance) {
        require(msg.sender == withdrawer, "!auth");
        require(protectedTokens[address(_asset)] == false, "protected");

        balance = _asset.balanceOf(address(this));
        _asset.approve(rewardDeposit, balance);
        IRewardDeposit(rewardDeposit).addReward(address(_asset), balance);
        return balance;
    }
```

Listing 3.8: `CurveVoterProxy::withdraw()`

Note a number of routines in the `Aura Finance` contracts can be similarly improved, including `CrvDepositorWrapper::setApprovals()` and including `AuraVestedEscrow::_claim()`.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`.

Status This issue has been fixed in this commit: `cc2c8fc`.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Aura Finance` protocol, there are two privileged account, i.e., `owner` and `admin`. These accounts play a critical role in governing and regulating the system-wide operations (e.g., change the locker contract address, cancel the vesting `rewardTokens`, and set the key parameters for the `Aura Finance` protocol, etc.). Our analysis shows that these privileged accounts need to be scrutinized. In the following, we use the `AuraVestedEscrow` contract as an example and show the representative functions potentially affected by the privileges of the `admin` account.

```
    /**
     * @notice Change contract admin
     * @param _admin New admin address
     */
    function setAdmin(address _admin) external {
        require(msg.sender == admin, "!auth");
        admin = _admin;
    }

    /**
     * @notice Change locker contract address
     * @param _auraLocker Aura Locker address
     */
    function setLocker(address _auraLocker) external {
        require(msg.sender == admin, "!auth");
        auraLocker = IAuraLocker(_auraLocker);
    }
```

Listing 3.9: `AuraVestedEscrow::setAdmin()/setLocker()`

```
    /**
     * @notice Cancel recipients vesting rewardTokens
     * @param _recipient Recipient address
     */
    function cancel(address _recipient) external nonReentrant {
        require(msg.sender == admin, "!auth");
```

```
        require(totalLocked[_recipient] > 0, "!funding");

        _claim(_recipient, false);

        uint256 delta = remaining(_recipient);
        rewardToken.safeTransfer(admin, delta);

        totalLocked[_recipient] = 0;

        emit Cancelled(_recipient);
    }
```

Listing 3.10: `AuraVestedEscrow::cancel()`

If the privileged `admin` account is a plain EOA account, this may be worrisome and pose counter-party risk to the protocol users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed `DAO`. In the meantime, a timelock-based mechanism can also be considered as mitigation. Moreover, it should be noted if current contracts are to be deployed behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed. The `Aura Finance` team confirms that admin functions will be operated by multi-sig initially and members of the `DeFi` community will operate these keys.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Aura Finance` protocol, which is built on top of the `Balancer` system to provide maximum incentives to `Balancer LPs` and `BAL` stakers through social aggregation of `BAL` deposits and `Aura`'s native token `AURA`. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, we need to emphasize that `Solidity-based` smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[10] PeckShield. PeckShield Inc. https://www.peckshield.com.