Zellic

Aura Finance

Smart Contract Security Assessment

June 16, 2023

Prepared for:

Aura Finance

Prepared by:

Katerina Belotskaia and Ulrich Myhre

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Aura Finance from May 30th to June 8th, 2023. During this engagement, Zellic reviewed Aura Finance's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could a mint rate of AURA on the sidechains exceed available AURA?
- Could a malicious message trigger a lockup of funds?
- Could a malicious message lead to unallowed obtainment of funds?
- Do the changes implemented to Convex-ETH contracts disrupt the operation of the protocol in any way?
- Could an on-chain attacker drain the vaults?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- The codebase that has not been modified
- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.3 Results

During our assessment on the scoped Aura Finance contracts, we discovered no findings.

However, Zellic recorded its notes and observations from the assessment for Aura Finance's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	0
Informational	0

2 Introduction

2.1 About Aura Finance

Aura Finance is a protocol built on top of the Balancer system to provide maximum incentives to Balancer liquidity providers and BAL stakers (into veBAL) through social aggregation of BAL deposits and Aura's native token.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Aura Finance Contracts

Repositories https://github.com/aurafinance/aura-contracts/pull/202

https://github.com/aurafinance/convex-platform/pull/55

Versions aura-contracts: `843cc67b154c2b590fb62f6c95811bfb567ea8b3`

convex-platform: `49a83c027aeb34173678704c09af6242f762e787`

Programs

- BaseRewardPool.sol
- BaseRewardPool4626.sol
- BoosterLite.sol
- BoosterOwnerLite.sol
- PoolManagerLite.sol
- VirtualBalanceRewardPool.sol
- VoterProxyLite.sol
- GenericVault.sol
- SimpleStrategy.sol

- Strategy.sol
- LzLib.sol
- LzApp.sol
- NonblockingLzApp.sol
- LZEndpointMock.sol
- OFT.sol
- OFTCore.sol
- ProxyOFT.sol
- AuraBalOFT.sol
- AuraBalProxyOFT.sol
- AuraOFT.sol
- AuraProxyOFT.sol
- Create2Factory.sol
- CrossChainConfig.sol
- CrossChainMessages.sol
- L1Coordinator.sol
- L2Coordinator.sol
- PausableOFT.sol
- PausableProxyOFT.sol
- PauseGuardian.sol
- BridgeDelegateReceiver.sol
- BridgeDelegateSender.sol
- GnosisBridgeSender.sol
- SimpleBridgeDelegateSender.sol

Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of two and a half person-weeks. The assessment was conducted over the course of eight calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia, Engineer
kate@zellic.io

Ulrich Myhre, Engineer
unblvr@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

May 30, 2023	Kick-off call
May 30, 2023	Start of primary review period
June 8, 2023	End of primary review period

3 Detailed Findings

We discovered no significant security vulnerabilities during this assessment; however, please see the Discussion section (4) for our notes and observations.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 The `protectAddPool` is unsafe

PoolManagerLite.sol has ported over a feature from the initial pool manager that allows the `operator` role to remove authentication for the function `addPool`.

```
function addPool(address _gauge, uint256 _stashVersion)
    external returns (bool) {
        return _addPool(_gauge, _stashVersion);
    }

    function _addPool(address _gauge, uint256 _stashVersion)
    internal returns (bool) {
        require(!IPools(booster).gaugeMap(_gauge), "already registered
    gauge");
        require(!isShutdown, "shutdown");

        if (protectAddPool) {
            require(msg.sender == operator, "!auth");
        }

        address lptoken = ICurveGauge(_gauge).lp_token();
        require(!IPools(booster).gaugeMap(lptoken), "already registered
    lptoken");

        return IPools(booster).addPool(lptoken, _gauge, _stashVersion);
    }
```

By default this is on, but we believe this should never be disabled. Giving everyone the possibility to add pools leads to a few dangerous scenarios.

The `IPools(booster).addPool(lptoken, _gauge, _stashVersion)` ends up doing `poolInfo.push` of a `PoolInfo` object. There are no checks on the `_gauge` param except to check if it is already added and nonzero. The `PoolInfo` object can only ever be pushed to and individual pools can be shut down, but items are never popped off. In functions like `shutdownSystem`, this array is iterated over.

```
function shutdownSystem() external {
        require(msg.sender == owner, "!auth");
        isShutdown = true;

        for (uint256 i = 0; i < poolInfo.length; i++) {
            PoolInfo storage pool = poolInfo[i];
            if (pool.shutdown) continue;

            address token = pool.lptoken;
            address gauge = pool.gauge;

            //withdraw from gauge
            try IStaker(staker).withdrawAll(token, gauge) {
                pool.shutdown = true;
            } catch {}
        }
    }
```

If `poolInfo` is too large, the loop will run out of gas before it can finish. This blocks the possibility to run `shutdownSystem()` if someone has spammed the pool manager with random pools that implement the required view functions that are checked.

The same happens in BoosterOwnerLite.sol in its `shutdownSystem()` function, where it loops to `IOwner(booster).poolLength()`. The consequence is the same, and shutting down might be impossible or very costly in terms of gas.

When discussing the issue with Aura Finance, they mentioned that this functionality can be removed, as the protection is always enabled in the sidechain.

4.2 Withdrawal of funds from a shut down pool

During the execution of the `shutdownPool` function, tokens are withdrawn from the gauge contract and transferred to the address of the current contract. But since `try/catch` is used, the pool will be successfully shut down even if the funds have not been withdrawn. The withdrawn tokens can be received by users using the `withdraw` function. The function withdraws tokens from the staker contract if the pool is not shut down; otherwise, tokens are transferred from the current contract balance.

If the tokens were not withdrawn during the shutdown, there are two possible options. Firstly, a second attempt to withdraw funds will not be possible and users will not be able to receive tokens if the balance of the contract is empty. Secondly, even if the contract owns `lptoken` tokens, users can receive other users' tokens, for example, withdrawn from the previous pool that was shut down with the same `lptoken` but not yet withdrawn by depositors.

Therefore, shutting down the pool without guaranteed receipt of the `lptoken` tokens by the contract may lead to problems when withdrawing funds by users.

```
function shutdownPool(uint256 _pid) external nonReentrant returns(bool){
        require(msg.sender==poolManager, "!auth");
        PoolInfo storage pool = poolInfo[_pid];

        //withdraw from gauge
        try IStaker(staker).withdrawAll(pool.lptoken,pool.gauge){
        }catch{}

        pool.shutdown = true;
        gaugeMap[pool.gauge] = false;

        emit PoolShutdown(_pid);
        return true;
    }

    function withdraw(uint256 _pid, uint256 _amount)
    public returns(bool){
        _withdraw(_pid,_amount,msg.sender,msg.sender);
        return true;
    }

    function _withdraw(uint256 _pid, uint256 _amount, address _from,
    address _to) internal nonReentrant {
         ...
        if (!pool.shutdown) {
            IStaker(staker).withdraw(lptoken,gauge, _amount);
        }
         ...
        //return lp tokens
        IERC20(lptoken).safeTransfer(_to, _amount);
         ...
    }
```

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 Module: AuraBalProxyOFT.sol

Function: `processClaimable(address _token, uint16 _srcChainId)`

Allows accounts from `rewardReceiver` mapping to claim the reward. Only owner is able to add address to the `rewardReceiver` mapping.

Inputs

- `_token`
  - **Control**: Full control.
  - **Constraints**: `claimable[_token][_srcChainId]` > 0.
  - **Impact**: The address of reward tokens that will be claimed.
- `_srcChainId`
  - **Control**: Full control.
  - **Constraints**: `claimable[_token][_srcChainId]` > 0 .
  - **Impact**: The source chain ID. The address of receiver is taken from `rewardReceiver[_srcChainId]` set by owner.

Branches and code coverage (including function calls)

Intended branches

- `claimable[_token][_srcChainId]` is reset.
  ☑ Test coverage
- `totalClaimable[_token]` is increased.
  ☑ Test coverage

Negative behavior

- `rewardReceiver[_srcChainId]` does not contain `_srcChainId`.

- ☐ Negative test
- `claimable[_token][_srcChainId] == 0`.
  - ☑ Negative test

Function call analysis

- `_lzSend`
  - **External/internal?** Internal.
  - **Argument control?** `_srcChainId`.
  - **Impact** Send the `innerToken` reward to the receiver to the `_srcChainId` .
- `IProxyOFT(oft).sendFrom`
  - **External/internal?** External.
  - **Argument control?** `_srcChainId`.
  - **Impact** Send the `_token` reward to the receiver to the `_srcChainId`.

5.2 Module: AuraBalRewardPool.sol

Function: `stakeAll()`

The same as the `stake` function, but `amount` is full with `msg.sender`'s `stakingToken` balance.

Function: `stakeFor(address _for, uint256 _amount)`

The same function as `stake`, but `_balances`, `rewards`, and `userRewardPerTokenPaid` is updated for the `_for` address provided by caller. The `stakingTokens` are transferred from the caller address.

Function: `stake(uint256 _amount)`

The `msg.sender` provides `stakingToken` to the current contract. The `_totalSupply` and `_balances` of `msg.sender` is increased by `_amount`-provided tokens. Also, the `updateReward` modifier is triggered.

Inputs

- `_amount`
  - **Control**: The caller has full control but cannot use more tokens than owned.
  - **Constraints**: If `msg.sender` owns less amount of `stakingToken`, transaction will be reverted.
  - **Impact**: The amount of staking tokens.

Branches and code coverage (including function calls)

Intended branches

- After first `stake` call, the `rewards[msg.sender]` is zero.
  - ☑ Test coverage
- If the caller's balance is nonzero, the `rewards[msg.sender]` is calculated properly.
  - ☐ Test coverage

Negative behavior

- Revert if `msg.sender`'s `stakingToken` balance is less than `_amount`.
  - ☐ Negative test
- `_amount` is zero.
  - ☑ Negative test

Function call analysis

- `stakingToken.safeTransferFrom(msg.sender, address(this), _amount)`
  - **External/internal?** External.
  - **Argument control?** `_amount`.
  - **Impact** Transfer `stakingToken` from `msg.sender` to current contract.
- `updateReward(msg.sender)`
  - **External/internal?** Internal.
  - **Argument control?** N/A.
  - **Impact** Updates global `lastUpdateTime` and `rewardPerTokenStored`. Sets `rewards` and `userRewardPerTokenPaid` of `msg.sender`.

Function: `withdraw(uint256 amount, bool claim, bool lock)`

Allows the caller to withdraw staking funds and claim reward.

Inputs

- `amount`
  - **Control**: Controlled.
  - **Constraints**: Cannot be more than `_balances[msg.sender]`.
  - **Impact**: The amount of `stakingToken` will be withdrawn.
- `claim`
  - **Control**: Controlled.
  - **Constraints**: N/A.
  - **Impact**: If claim is true and lock is false, the reward will be transferred to the `msg.sender`.

- `lock`
  - **Control**: Controlled.
  - **Constraints**: N/A.
  - **Impact**: If claim is true and lock is true.

Branches and code coverage (including function calls)

Intended branches

- Withdraw amount without claim reward.
  - [x] Test coverage
- Withdraw amount with claim reward.
  - [x] Test coverage
- Withdraw amount with lock reward.
  - [x] Test coverage

Negative behavior

- Amount is zero.
  - [x] Negative test
- The amount is more than `_balances[msg.sender]`.
  - [ ] Negative test

Function call analysis

- `stakingToken.safeTransfer(msg.sender, amount)`
  - **External/internal?** External.
  - **Argument control?** Amount cannot be more than `_balances[msg.sender]`.
  - **Impact** Transfer `stakingToken` to `msg.sender`.

5.3 Module: AuraOFT.sol

Function: `lock(uint256 _cvxAmount)`

Lock the OFT tokens of the `_canonicalChainId`.

Inputs

- `_cvxAmount`
  - **Control**: Full control.
  - **Constraints**: `msg.sender` should have more or an equal amount of tokens.
  - **Impact**: The amount of tokens will be transferred to another chain.

Branches and code coverage (including function calls)

Intended branches

- The balance of `msg.sender` was decreased by `_cvxAmount`.
  - ☐ Test coverage
- The totalSupply was increased by _cvxAmount
  - ☑ Test coverage

Negative behavior

- `msg.sender` does not have enough tokens.
  - ☑ Negative test
- `_cvxAmount == 0`.
  - ☑ Negative test
- without fee
  - ☑ Negative test

Function call analysis

- `_debitFrom(msg.sender, canonicalChainId, bytes(“”), _cvxAmount) → _debitFrom → _burn(_from, _amount)`
  - **External/internal?** Internal.
  - **Argument control?** `_cvxAmount`
  - **Impact** Burn the `_cvxAmount` amount of tokens from the `msg.sender` balance before transfer to the `canonicalChainId`.

5.4 Module: AuraVestedEscrow.sol

Function: `claim(bool _lock)`

Allows to claim reward by recipient or lock it inside auraLocker contract.

Inputs

- `_lock`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: If false reward, it will be transferred to the caller; otherwise, it will be locked inside the auraLocker.

Branches and code coverage (including function calls)

Intended branches

- If `_lock` is true, funds were locked.
  - ☐ Test coverage
- If `_lock` is false, funds were transferred to the caller.
  - ☑ Test coverage

Negative behavior

- Claimable is zero.
  - ☐ Negative test
- AuraLocker is zero
  - ☐ Negative test

Function call analysis

- `_claim(msg.sender, _lock) → available(_recipient)`
  - **External/internal?** Internal.
  - **Argument control?** N/A.
  - **Impact** Return available amounts of funds for claim that does not include already claimed funds.
- `_claim(msg.sender, _lock) → auraLocker.lock(_recipient, claimable)`
  - **External/internal?** External.
  - **Argument control?** N/A.
  - **Impact** If `_lock` is true, then claimable funds will be locked inside auraLocker contract.
- `_claim(msg.sender, _lock) → rewardToken.safeTransfer(_recipient, claimable)`
  - **External/internal?** External.
  - **Argument control?** N/A.
  - **Impact** If `_lock` is true, then claimable funds will be transferred to the `_recipient`.

5.5 Module: BaseRewardPool4626.sol

Function: `transferFrom(address owner, address recipient, uint256 amount)`

Moves tokens from the sender to the given recipient, using the allowance mechanism. The given amount is deducted from the caller's allowance for the provided owner.

Inputs

- `owner`
  - **Control**: Arbitrary.
  - **Constraints**: Must be a valid entry in the `_allowances` 2D mapping; otherwise, setting the new allowance will fail. Cannot be 0.
  - **Impact**: Decides which allowance to use.
- `recipient`
  - **Control**: Arbitrary.
  - **Constraints**: Cannot be 0.
  - **Impact**: Decides where the amount should be transferred.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: Cannot be more than the actual allowance or the subtraction will underflow and revert.
  - **Impact**: Decides the amount to transfer and how much that will be left in the allowance.

Branches and code coverage (including function calls)

Intended branches

- Transfer when token owner has enough balance.
  - [x] Test coverage
- Transfer when the spender has enough approved balance.
  - [x] Test coverage

Negative behavior

- Transfer when the spender does not have enough approved balance.
  - [x] Negative test
- Transfer when token owner does not have enough balance.
  - [x] Negative test
- Transfer from `address(0)`.
  - [x] Negative test
- Transfer to `address(0)`.
  - [x] Negative test

Function call analysis

- `rootFunction → _transfer(args)`
  - **What is controllable?** Everything.

- **If return value controllable, how is it used and how can it go wrong?** Not checked.
- **What happens if it reverts, reenters, or does other unusual control flow?** New allowance is set before transfer, making reentrancy less useful. A reward manager could add an `extraReward` that hooks every transfer before the balances are updated.

5.6 Module: BoosterLite.sol

Function: `addPool(address _lptoken, address _gauge, uint256 _stashVersion)`

Creates all the contracts required for a new pool and adds them to the `poolInfo` list. Can only be called by the pool manager. The pool ID is a sequential number that corresponds to the index in the list. Note that the list can never remove items, so care should be taken to limit the number of pools added for loops that require going through every pool.

Inputs

- `_lptoken`
  - **Control**: Arbitrary.
  - **Constraints**: Cannot be 0.
  - **Impact**: Decides which token to use in the pool.
- `_gauge`
  - **Control**: Arbitrary.
  - **Constraints**: Must be some gauge that passes the version test in CreateStash.
  - **Impact**: Decides which gauge contracts to use in, for example, the stash.
- `_stashVersion`
  - **Control**: Arbitrary.
  - **Constraints**: Must be 1, 2 or 3.
  - **Impact**: Picks the expected stash version and does some checks to verify that the gauge matches that version later.

Branches and code coverage (including function calls)

Intended branches

- Add single pool.
  - ☑ Test coverage

- Add multiple pools.
  - ☑ Test coverage

Negative behavior

- Called by someone who is not the pool manager.
  - ☐ Negative test
- Called during shutdown.
  - ☐ Negative test
- Called with `gauge = address(0)`.
  - ☐ Negative test
- Called with `lptoken = address(0)`.
  - ☐ Negative test
- Called with `stash = address(0)`.
  - ☐ Negative test
- Called with bad or mismatching stash version.
  - ☐ Negative test

Function: `depositAll(uint256 _pid, bool _stake)`

Helper function for depositing the sender's full balance to a gauge (specified by `_pid`). Optionally stakes the minted `DepositToken` on BaseRewardPool.

Inputs

- `_pid`
  - **Control**: Arbitrary.
  - **Constraints**: Must be a valid entry in the `poolInfo` array, or `balanceOf` will revert.
  - **Impact**: Decides the pool to deposit to.
- `_stake`
  - **Control**: Arbitrary.
  - **Constraints**: Boolean.
  - **Impact**: Decides if the deposit should be staked after minting.

Branches and code coverage (including function calls)

Intended branches

- Deposit with stake.
  - ☐ Test coverage
- Deposit without stake.

☐ Test coverage

Negative behavior

- Deposit with invalid `_pid`.
  ☐ Negative test

Function: `deposit(uint256 _pid, uint256 _amount, bool _stake)`

Deposits `_amount` to a gauge specified by `_pid`, then mints a `DepositToken` and optionally stakes it if `_stake` is true.

Inputs

- `_pid`
  - **Control**: Arbitrary.
  - **Constraints**: Cannot specify a shut down pool. If `_pid` is not in the `poolInfo` array, the resulting empty struct will appear to be shut down.
  - **Impact**: Decides the pool to deposit to.
- `_amount`
  - **Control**: Arbitrary.
  - **Constraints**: Cannot be more tokens than the user owns.
  - **Impact**: Decides the amount of tokens to deposit/stake.
- `_stake`
  - **Control**: Arbitrary.
  - **Constraints**: Boolean.
  - **Impact**: Chooses if the amount should be sent to the rewards contract on behalf of the user.

Branches and code coverage (including function calls)

Intended branches

- Deposit with stake.
  ☑ Test coverage
- Deposit without stake.
  ☑ Test coverage

Negative behavior

- Deposit while pool is shut down.
  ☐ Negative test
- Deposit while full shutdown is in effect.

- ☐ Negative test
- Deposit to a gauge with incorrect settings (addr=0).
  - ☐ Negative test
- Deposit to a pool without a configured stash.
  - ☐ Negative test

Function: `earmarkRewards(uint256 _pid)`

Responsible for collecting the CRV from gauge and then redistributing to the correct place. Pays the caller a fee to process this.

The function is a thin wrapper for `_earmarkRewards(_pid)`, which claims CRV from the staker, transfers idle CRV in the Booster to the treasury, and finally transfers it to the LP provider reward contract. Incentives (fees) are paid to the caller and the lockers reward contract.

Inputs

- `_pid`
  - **Control**: Arbitrary.
  - **Constraints**: Must be a valid pool ID in `poolInfo`.
  - **Impact**: Decides which gauge to pull and redistribute CRV from.

Branches and code coverage (including function calls)

Intended branches

- Earmark rewards.
  - ☑ Test coverage
- Caller earns CRV.
  - ☑ Test coverage
- Call when there are idle rewards to transfer to treasury.
  - ☐ Test coverage

Negative behavior

- Call when pool is closed.
  - ☐ Negative test
- Call when stash is not set.
  - ☐ Negative test
- Call when there is nothing to earmark.
  - ☐ Negative test

Function: `withdraw(uint256 _pid, uint256 _amount)`

Passthrough function for `_withdraw`, which sets `from` and `to` to `msg.sender`.

Inputs

- `_pid`
  - **Control**: Arbitrary.
  - **Constraints**: Must be a a valid pool ID in `poolInfo`.
  - **Impact**: Decides the token to withdraw and the gauge to withdraw it from. This removes LP balance by burning `amount` from `msg.sender` and transfers LP tokens back to `msg.sender`.
- `_amount`
  - **Control**: Arbitrary.
  - **Constraints**: Cannot exceed the amount of LP balance the sender is allowed to burn.
  - **Impact**: The amount of LP balance to exchange for LP tokens.

Branches and code coverage (including function calls)

Intended branches

- Withdraw from pool.
  - ☐ Test coverage
- Withdraw from this contract (in case of shutdown).
  - ☐ Test coverage
- Withdraw when a stash is defined.
  - ☐ Test coverage

Negative behavior

- Withdraw while pool is shut down.
  - ☐ Negative test
- Withdraw under a full shutdown.
  - ☐ Negative test
- Withdraw more than the sender has.
  - ☐ Negative test

5.7 Module: ExtraRewardsDistributor.sol

Function: `addRewardToEpoch(address _token, uint256 _amount, uint256 _epoch)`

The same function as `addReward`, but it allows to control the `_epoch` amount.

Function: `addReward(address _token, uint256 _amount)`

Added reward tokens from the caller to the last epoch. The caller should be whitelisted by owner.

Inputs

- `_token`
  - **Control**: Full control.
  - **Constraints**: No.
  - **Impact**: The reward token address.
- `_amount`
  - **Control**: Full control.
  - **Constraints**: Caller should have enough amount of tokens to transfer.
  - **Impact**: The amount of reward tokens will be transferred.

Branches and code coverage (including function calls)

Intended branches

- The balance of `msg.sender` was decreased by `_amount`.
  ☑ Test coverage
- The `rewardData` was updated properly.
  ☑ Test coverage

Negative behavior

- `msg.sender` is not whitelisted.
  ☑ Negative test
- `_token` is zero.
  ☐ Negative test
- `_amount` is zero.
  ☑ Negative test

Function call analysis

- `auraLocker.checkpointEpoch()`
  - **External/internal?**: External.

- **Argument control?**: N/A.
  - **Impact**: Added new checkpoint.
- `_addReward(_token, _amount, latestEpoch)`
  - **External/internal?**: External.
  - **Argument control?**: `_token` and `_amount`.
  - **Impact**: Add reward to the last epoch.

Function: `getReward(address _account, address _token)`

Allows `msg.sender` to claim reward calculated using the user balance locked inside the auraLocker contract.

Inputs

- `_account`
  - **Control**: Full control.
  - **Constraints**: Should have nonzero auraLocker balance.
  - **Impact**: The receiver of reward.
- `_token`
  - **Control**: Full control.
  - **Constraints**: If tokens are not added using the `addReward` function, the transaction will end without result.
  - **Impact**: The address of reward token. It will be called to transfer reward amount.

Branches and code coverage (including function calls)

Intended branches

- If `msg.sender` $\neq$ `_account`, `_account` received the reward.
  - ☐ Test coverage
- If `msg.sender == _account`, `_account` received the reward.
  - ☑ Test coverage

Negative behavior

- `_account` is zero.
  - ☐ Negative test
- `_token` is zero.
  - ☐ Negative test

Function call analysis

- `_getReward` → `_allClaimableRewards(_account, _token, _startIndex)`
  - **External/internal?**: Internal.
  - **Argument control?**: `_account`, `_token`, and `_startIndex`.
  - **Impact**: Returns the amount of tokens available for claim and index of last rewarded epoch.
- `_getReward` → `IERC20(_token).safeTransfer(_account, claimableTokens)`
  - **External/internal?**: External.
  - **Argument control?**: `_account`.
  - **Impact**: Transfer reward to the `_account`.

Function: `getReward(address _token, uint256 _startIndex)`

The same function as `getReward(address _account, address _token)`, but the reward is calculated for the `msg.sender` address, and caller controls the index from which is started by the checking for rewards.

5.8 Module: L1Coordinator.sol

Function: `distributeAura(uint16 _srcChainId, address _sendFromZroPaymentAddress, byte[] _sendFromAdapterParams)`

Function allows to transfer AURA tokens to another chain. Before the transfer, tokens will be minted by calling `IBooster(booster).distributeL2Fees(_feeAmount)`. Function is available only for whitelisted distributor.

Inputs

- `_srcChainId`
  - **Control**: Full control.
  - **Constraints**: Should be whitelisted by owner as trusted chain.
  - **Impact**: ID of the recipient chain.
- `_sendFromZroPaymentAddress`
  - **Control**: Full control.
  - **Constraints**: No checks.
  - **Impact**: The address of the contract that will provide an LZ protocol fee using the LZ tokens.

Branches and code coverage (including function calls)

Intended branches

- AURA tokens were distributed properly.
  - ☑ Test coverage

Negative behavior

- `feeDebtOf[_srcChainId]` is zero.
  - ☑ Negative test
- Caller is not whitelisted distributor.
  - ☑ Negative test

Function call analysis

- `_distributeAura → IBooster(booster).distributeL2Fees(_feeAmount) → IERC20(crv).safeTransferFrom(bridgeDelegate, lockRewards, _lockIncentive)`
  - **External/internal?** External.
  - **Argument control?** No.
  - **Impact** Distribute fees.
- `_distributeAura → IBooster(booster).distributeL2Fees(_feeAmount) → IERC20(crv).safeTransferFrom(bridgeDelegate, stakerRewards, _stakerIncentive)`
  - **External/internal?** External.
  - **Argument control?** No.
  - **Impact** Distribute fees.
- `_distributeAura → IBooster(booster).distributeL2Fees(_feeAmount) → ITokenMinter(minter).mint(bridgeDelegate, eligibleForMint)`
  - **External/internal?** External.
  - **Argument control?** No.
  - **Impact** Mint CVX to current contract.
- `_lzSend → lzEndpoint.send`
  - **External/internal?** External.
  - **Argument control?** `_srcChainId`.
  - **Impact** On the side of L2Coordinator, there will be the triggered function `_nonblockingLzReceive` that will increase the `accAuraRewards`.
- `IOFT(auraOFT).sendFrom`
  - **External/internal?** External.
  - **Argument control?** `_sendFromZroPaymentAddress` and `_sendFromAdapterParams`.
  - **Impact** Transfer `auraAmount` of AURA tokens to the `_srcChainId`.

5.9 Module: PausableOFT.sol

Function: `sendFrom(address _from, uint16 _dstChainId, byte[] _toAddress, uint256 _amount, address payable _refundAddress, address _zroPaymentAddress, byte[] _adapterParams)`

Pausable wrapper over `OFT.sendFrom`.

5.10 Module: PausableProxyOFT.sol

Function: `processQueued(uint256 _epoch, uint16 _srcChainId, address _to, uint256 _amount, uint256 _timestamp)`

Initiates the queued transfer, which is possible if enough time has passed after creation. The queue is added during the receiving process inside the `_sendAck` function. After successful send, the `queue` is reset.

Function: `rescue(address _token, address _to, uint256 _amount)`

The `sudo` address can transfer any tokens from the current contract to an arbitrary recipient. The `sudo` is set during deploy and cannot be changed.

Function: `sendFrom(address _from, uint16 _dstChainId, byte[] _toAddress, uint256 _amount, address payable _refundAddress, address _zroPaymentAddress, byte[] _adapterParams)`

Wrapper under `ProxyOFT.sendFrom()` function. Added `whenNotPaused` modifier and `outflow` of `currentEpoch` is increased by the `_amount` value. Allows any caller to send to another chain.

Inputs

- `_from`
  - **Control**: Full control.
  - **Constraints**: If `_from` $\neq$ `msg.sender`, transaction will be reverted inside `ProxyOFT._debitFrom`.
  - **Impact**: The receiver of `innerToken`.
- `_dstChainId`
  - **Control**: Full control.
  - **Constraints**: If `_lzSend.trustedRemoteLookup` mapping does not contain `_dstChainId`, transaction will be reverted.

- **Impact**: ID of the destination chain to which the tokens will be transferred.

- `_toAddress`
  - **Control**: Full control.
  - **Constraints**: No checks.
  - **Impact**: The address of the receiver of tokens in the `_dstChainId` network.
- `_amount`
  - **Control**: Full control.
  - **Constraints**: The `_from` account should have more or an equal amount of tokens.
  - **Impact**: The amount of `innerToken` that will be locked inside this contract and transferred to another chain.

Branches and code coverage (including function calls)

Intended branches

- The balance of the contract increased by amount value.
  - ☐ Test coverage
- The balance of the `from` address decreased by amount value.
  - ☐ Test coverage

Negative behavior

- `from` $\neq$ `msg.sender`.
  - ☐ Negative test
- `from` does not have enough `innerToken`.
  - ☐ Negative test
- The unknown `_dstChainId`.
  - ☐ Negative test

Function call analysis

- `OFTCore._send()` → `ProxyOFT._debitFrom(address _from,uint16,bytes memory, uint256 _amount)` → `innerToken.safeTransferFrom(_from, address(this), _amount);`
  - **External/internal?** External.
  - **Argument control?** `_from` and `_amount`.
  - **Impact** Will block the sent tokens inside this contract.

5.11 Module: PoolManagerLite.sol

Function: `addPool(address _gauge, uint256 _stashVersion)`

Adds a gauge to the pool using the provided stash version.

Inputs

- `_gauge`
  - **Control**: Arbitrary.
  - **Constraints**: The gauge address, nor its associated LP token, cannot already be registered in the booster. If `protectAddPool` is enabled, the function can only be called by the operator. The default is for this protection to be enabled.
  - **Impact**: Decides the address of the gauge and picks the LP token to add.
- `_stashVersion`
  - **Control**: Arbitrary.
  - **Constraints**: Is supposed to be 1, 2, or 3. Otherwise, `StashFactoryV2→createStash` will return `address(0)` or revert. The given version must have a valid implementation registered in the stash factory.
  - **Impact**: Decides which stash implementation to use.

Branches and code coverage (including function calls)

Negative behavior

- Try to add pool as normal user when `protectAddPool` is disabled.
  - ☑ Negative test
- Add pool as normal user when `protectAddPool` is disabled.
  - ☐ Negative test
- Add pool when the everything is shut down.
  - ☐ Negative test

Function call analysis

- `rootFunction → IPools(booster).addPool(lptoken, _gauge, _stashVersion)`
  - **What is controllable?** Everything, provided gauge is controlled by caller. There is no gauge whitelisting.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Return value is not checked.

5.12 Module: VirtualBalanceRewardPool.sol

Function: `processIdleRewards()`

Starts to process queued rewards, given that `periodFinish` is reached and there are rewards queued.

Branches and code coverage (including function calls)

Intended branches

- Call when there are queued rewards.
  - ☐ Test coverage

Negative behavior

- Called when there are no queued rewards.
  - ☐ Negative test
- Called before `periodFinish` has passed.
  - ☐ Negative test

Function call analysis

- `rootFunction → notifyRewardAmount(queuedRewards)`
  - **What is controllable?** Nothing, `queuedRewards` is an internal variable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

6 Audit Results

At the time of our audit, the audited code was not deployed to mainnet EVM.

During our assessment on the scoped Aura Finance contracts, we discovered no findings.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.