PUBLIC

Code Assessment

of the LiquidLockers Smart Contracts

25 October, 2022

Produced for

STAKE | DAO

by

CHAINSECURITY

Contents

1 Executive Summary

Dear StakeDAO team,

Thank you for trusting us to help StakeDAO with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of LiquidLockers according to Scope to support you in forming an opinion on their security risks.

StakeDAO implements so called liquid lockers. They allow users to earn yield on their locked tokens, make them transfer and still preserve voting power for the users.

We did not uncover critical issues. The most severe subjects covered in our audit are the following two medium rated issues: Admin Set Too Early in LiquidityGaugeV4Strat and Zero Address Reward Distributor. As the system is already deployed and the issues are not critical, StakeDAO decided to not change the code.

In summary, we find that the codebase provides a good level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	2
• Acknowledged	2
Low-Severity Findings	14
• Acknowledged	14

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the LiquidLockers repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	03 Oct 2022	f273980380f73d3d4ab965783e8060a20fa3daae	Initial Version

For the solidity smart contracts, the compiler version `0.8.7` was chosen. For the vyper smart contracts, the compiler version `0.2.16` was chosen.

The files in scope are the following:

- LiquidityGaugeV4Strat.vy
- BaseStrategy.sol
- AngleStrategy.sol
- AngleVault.sol
- AngleVaultGUni.sol
- BalancerStrategy.sol
- BalancerVault.sol
- CurveStrategy.sol

2.1.1 *Excluded from scope*

Third-party dependencies, tests and any files not listed above are not in scope of this review.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

StakeDAO's Liquid Lockers enables DeFi users to make the most of their lockable tokens, without having to sacrifice their yield, voting power, or liquidity. By depositing their lockable tokens in Liquid Lockers, users get the maximum voting power from their native protocol, receive bribes, and get a considerable yield boost.

The main entry point for a user is the vault contract. When depositing, the vault forwards its own minted token to the liquidity gauge and the native token to the strategy contract. The strategy contract forwards the funds to the locker which locks them in the native liquidity gauge contract of the protocol. Withdrawing reverses this flow.

2.2.1 Functionality overview for a user

The vault contracts allow users to call `deposit`, `withdraw` and `withdrawAll` (which has a known bug). The governance address can call various admin functions to set certain parameters.

The strategy contracts only have the `claim` function unrestricted. All other functions are restricted to the corresponding vault, governance or factory.

The LiquidityGaugeV4Strat contract has the following publicly accessible functions: `user_checkpoint` , `set_rewards_receiver`, `claim_rewards`, `kick`. All other functions have access restrictions.

The most important user execution flows are described below.

2.2.2 Deposit Execution Flow

Users can start by locking their Token and receiving sdToken in return (e.g., getting sdsanUSDC_EUR in return for locking sanUSDC_EUR token). To do this, they call `deposit` in the vault contract. It triggers a sequence of actions, namely transferring the native token from the user to the vault, minting sdToken in return, and giving the `LiquidityGaugeV4Strat` the approval to for this amount of sdToken. Finally, the vault calls `deposit` in `LiquidityGaugeV4Strat` to deposit the sdToken in the gauge. The only token the user really receives are the gauge tokens from the `LiquidityGaugeV4Strat`. When depositing, the gauge checkpoints the user's balance. If the caller calls this function with `earn == true` all native tokens are transferred to the `Strategy` which forwards the token to the locker. The locker locks the tokens in the corresponding native protocol gauge.

As mentioned above, while depositing into the vault, `deposit` is called in `LiquidityGaugeV4Strat`. This function first checks the caller to be a pre-configured vault. Depending on the pass parameters, one of the following scenarios might happen:

1. **_value != 0**: If the user has deposited some amount of token, the gauge internally calls `_checkpoint_rewards` to checkpoint possible rewards for the user, who has initiated depositing. Thereafter, it updates the balance of the user as well as the total supply. Then, it calls `_update_liquidity_limit` to update the user's adjusted balance of staked sdToken. Finally, it transfers the passed-in amount of `staking_token` to itself.
2. **_value == 0**: In this case, depositing serves as a checkpoint reward collector, by calling `_checkpoint_rewards`.

2.2.3 Withdraw Execution Flow

When calling `withdraw` of a vault, it first queries the `liquidityGauge` to get the total shares of the user. It then checks that the amount of shares the user wants to withdraw doesn't exceed its total shares. Then it calls `withdraw` in `liquidityGauge` to withdraw an equivalent amount of tokens from it and transfer them to vault. It then burns the received `staking_token`. The transferrable number of tokens is calculated by deducting the `accumulatedFee` from the total token balance of the vault. If it doesn't suffice to pay the caller their share back, the vault calls into `withdraw` of the corresponding strategy, calculates a `withdrawalFee` for the `governance`, and deducts this fee from the shares of the user.

Withdrawing also calls into `liquidityGauge`. First, it checks the caller is the present vault. Then, similar to depositing, based on the `_shares` to withdraw, one of the following scenarios might take place:

1. **_value != 0**: If the user wants to withdraw some amount of token, it internally calls `_checkpoint_rewards` to get possible rewards for the withdrawing user, and updates their balance as well as their total supply. Then, it calls `_update_liquidity_limit` to update the user's adjusted balance of staked sdToken. Finally, it transfers the passed-in amount of `staking_token` from `liquidityGauge` to the vault.
2. **_value == 0**: In this case, calling `withdraw` solely collects the user's reward, by calling `_checkpoint_rewards`.

2.2.4 Claim Execution Flow

Users can directly call `claim` for a `_token` in the strategy contract. It first fetches the gauge for this token, calls `user_checkpoint` in this gauge, and initially claims rewards for itself by calling `claim_rewards` in the gauge. Then, for each reward tokens, it debits fees for different parties and distributes their associated fees. Finally, it deposits a `netRewards` of each reward token to the associated `multiGauges[gauge]`.

2.2.5 Roles and Trust Model

As vault contracts (`AngleVault`, `BalancerVault`, and `CurveVault`) are deployed through a proxy, the proxy admin should be fully trusted.

Vault Gauge Factory: This role is set upon deployment of any contracts inheriting from `BaseStrategy`. As it has permission to call `toggleVault`, `setGauge`, and `setMultiGauge`, we assume the party holding this role is fully trusted.

Governance: In addition to the functions listed for Gauge Factory, this role can change the configuration of the system with fewer restrictions (e.g., calling `setVeSDTProxy`, `setAccumulator`, etc.). Hence, we assume it to be fully trusted.

Approved Vaults: Only vaults whitelisted by Vault Gauge Factory or Governance are allowed to deposit into the strategy. We assume these vaults to be trustless.

LiquidityGaugeV4Strat: This smart contract provides our system with some fundamental functionalities. Some important ones are `deposit`, `withdraw`, `kick` (to re-adjust a user which has abused their voting boost), implementing ERC20-compliant interfaces, as well as some administrative functionalities to transfer ownership. As a backbone of the Liquid Lockers protocol, we assume this smart contract to be trustless.

Users: Their state-changing interaction with the system is limited to `deposit` and `withdraw`. Other functionalities exposed to users are either view-only or very restricted in terms of modifying the system. They are untrusted.

Strategy: Assumed to be trustless, as critical functions like `deposit`, `withdraw`, and `claim` are implemented through it.

Vault: As it provides ERC20-compliant interface to the untrusted users, while wrapping some access restricted functions of strategy and liquidity gauge, we assume it to be trustless.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe our findings. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	2

- Admin Set Too Early in LiquidityGaugeV4Strat Acknowledged
- Zero Address Reward Distributor Acknowledged

Low-Severity Findings	14

- Functions Marked public Should Be external Acknowledged
- Inconsistent Limits on Fees Acknowledged
- MANAGEFEE Enum Defined Multiple Times Acknowledged
- Misleading Governance Addresses Acknowledged
- Missing Check if withdrawFee Is Zero Acknowledged
- Missing Events Acknowledged
- Missing Sanity Checks Acknowledged
- NatSpec Missing Acknowledged
- Non-indexed Events Acknowledged
- State Variable Could Be Immutable Acknowledged
- Superfluous Variable Assignment Acknowledged
- Unnecessary External Call to approve Acknowledged
- Unused Imports Acknowledged
- Wrong Event Emitted Acknowledged

5.1 Admin Set Too Early in `LiquidityGaugeV4Strat`

Correctness Medium Version 1 Acknowledged

The admin is transferred via a commit-accept scheme in `LiquidityGaugeV4Strat`. However, the scheme is bypassed by directly setting the new `admin` in `commit_transfer_ownership`:

```
@external
def commit_transfer_ownership(addr: address):
    """
    @notice Transfer ownership of Gauge to `addr`
    @param addr Address to have ownership transferred to
    """
    assert msg.sender == self.admin  # dev: admin only
    assert addr != ZERO_ADDRESS  # dev: future admin cannot be the 0 address

    self.future_admin = addr
    self.admin = addr
    log CommitOwnership(addr)
```

While this doesn't compromise the security of the contract, it causes the emitted event to be incorrect and is misleading to users who don't believe an immediate transfer of ownership is possible. Furthermore, the function `accept_transfer_ownership` is made redundant if this behavior is intended.

Acknowledged

StakeDAO acknowledges the issue and noted that they had to implement this function for the factory contract so that it can transfer the admin of the contract in one transaction.

5.2 Zero Address Reward Distributor

Correctness Medium Version 1 Acknowledged

The `add_reward` functions in all gauge contracts do not check that the `_distributor` address is not the zero address. This is problematic as `set_reward_distributor` asserts that the distributor address is not zero. Therefore, if `add_reward` is called with the zero address as the `_distributor` parameter, a reward distributor can never be set for this reward token entry.

Acknowledged

StakeDAO acknowledges the issue without changes as they rate the chances as very low that the described issue happens. They state the issue only occurs when they add a reward distributor manually to the `LiquidityGaugeV4`.

5.3 Functions Marked `public` Should Be `external`

Design Low Version 1 Acknowledged

The following functions are `public` but not called from within the corresponding contract. Hence, they should be marked `external`.

- `AngleStrategy.deposit`
- `AngleStrategy.withdraw`
- `AngleVault.init`
- `AngleVault.deposit`
- `AngleVaultGUni.deposit`
- `BalancerVault.init`

- `BalancerVault.deposit`
- `BalancerVault.provideLiquidityAndDeposit`
- `BalancerVault.withdraw`
- `CurveVault.init`
- `CurveVault.deposit`

Acknowledged

Client acknowledges the issue without changing the code.

5.4 Inconsistent Limits on Fees

Design | Low | Version 1 | Acknowledged

The constraints for which fees can be set is inconsistent between strategies. For example, the AngleStrategy contract allows each individual fee to be as high as `BASE_FEE`, whereas the BalancerStrategy contract only allows the sum of all fees for a gauge to be as high as `BASE_FEE`.

Acknowledged

StakeDAO is aware of the issue and acknowledges it.

5.5 MANAGEFEE Enum Defined Multiple Times

Design | Low | Version 1 | Acknowledged

The `MANAGEFEE` enum is defined once in each Strategy contract. It would be simpler and less error-prone to instead define it in the shared interface instead.

Acknowledged

StakeDAO is aware of the issues and acknowledges it. They might change it in future releases.

5.6 Misleading Governance Addresses

Design | Low | Version 1 | Acknowledged

A Strategy contracts calls its respective Locker contract in its `deposit`, `withdraw` and `claim` functions. It specifically calls the Locker's `execute` function, which has the `onlyGovernance` modifier. Any reasonable user would assume that `onlyGovernance` modifier means that the `execute` function could only be called by the governance contract. However, the Locker's `governance` address is instead set to the Strategy contract's address in order to allow this functionality.

Acknowledged:

According to StakeDAO, this is related to their flow. There are many layers of governance. The governance address of the Locker contract is the Strategy contract. The governance of the Strategy is the voter contract, which is a helper contract to facilitate on-chain voting and admin functionality. The voter contract is owned by a multisig.

5.7 Missing Check if `withdrawFee` Is Zero

Design | Low | Version 1 | Acknowledged

In the Angle vault contract's withdraw function a transfer to the governance contract is done to send potential fees. The transfer is also done if the fees are actually zero.

Acknowledged:

StakeDAO acknowledges this issue as it is only present in the `AngleVault` contract and has been fixed for other vault implementations.

5.8 Missing Events

Design | Low | Version 1 | Acknowledged

Multiple functions perform important state changes without emitting an event. For example, the setter functions in the vault and strategy contracts.

Acknowledged:

StakeDAO does not see the necessity to emit these events and acknowledges the issue.

5.9 Missing Sanity Checks

Design | Low | Version 1 | Acknowledged

There are multiple setter functions missing the address zero sanity check.

- `toggleVault` and all setter functions in `AngleStrategy` except for `setVaultGaugeFactory`
- Multiple setter functions in the different vault contracts

Additionally, `withdraw` and `deposit` in `LiquidityGaugeV4Strat` can be called with `_addr = 0`.

There are no limits imposed when setting the `keeperFee` and `withdrawalFee` for a vault. This allows setting an arbitrarily high fee, even above 1.

Acknowledged

StakeDAO states that they do the checks on their side.

5.10 NatSpec Missing

Design | Low | Version 1 | Acknowledged

Many contracts are missing NatSpec. Some only have partial or incomplete NatSpec.

Acknowledged:

StakeDAO is implementing NatSpec documentation in newer deployments.

5.11 Non-indexed Events

Design | Low | Version 1 | Acknowledged

The events in the following contracts signal important state updates which include addresses. The corresponding address is part of the event but not indexed.

In `liquidityGaugeV4Strat`:

- `UpdateLiquidityLimit`
- `CommitOwnership`
- `ApplyOwnership`

All Events in `AngleVault`, `AngleVaultGUni`, `BalancerVault`, `CurveVault` and `BaseStrategy`.

Acknowledged:

As indexed events cost more gas, StakeDAO decided not to add them but handle fetching and filtering events on their side.

5.12 State Variable Could Be Immutable

Design | Low | Version 1 | Acknowledged

The state variable `token` in the AngleVaultGUni contract could be made immutable to save gas.

Acknowledged:

StakeDAO will take this gas optimization into consideration in future implementations.

5.13 Superfluous Variable Assignment

Design | Low | Version 1 | Acknowledged

The BalancerVault contract writes `assets[i] = address(tokens[i])` in a loop. This variable assignment seems superfluous.

Acknowledged:

StakeDAO has decided to acknowledge this unnecessary gas consumption as it does not have any impacts on security.

5.14 Unnecessary External Call to `approve`

Design Low Version 1 Acknowledged

In `AngleVault.deposit`, `BalancerVault.setLiquidityGauge` and `CurveVault.deposit` the corresponding contract's approve function is called with an external call when an internal call could have been performed.

Acknowledged:

StakeDAO will consider changing this unnecessary external call to a function call in future implementations.

5.15 Unused Imports

Design Low Version 1 Acknowledged

Some contracts import code that is not used in the contract. E.g., `AddressUpgradeable` is imported in `AngleVault` and used in the `using` statement without ever using functionality from the library. Removing all unused imports and library contracts enhances the code quality and readability.

Acknowledged:

StakeDAO acknowledged the issue.

5.16 Wrong Event Emitted

Correctness Low Version 1 Acknowledged

The `Transfer` event emitted at the end of the `withdraw` function in the LiquidityGaugeV4Strat contract has incorrect parameters. It emits `msg.sender` as the `_from` address instead of `_addr`.

Acknowledged:

StakeDAO acknowledges the issue. To track the correct event, they would track the vault's `withdraw` event.

6 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

6.1 Inconsistent Withdraw Fees

Note **Version 1**

When withdrawing from a Vault it matters if the vault currently has accumulated a token balance due to deposits being made without using the `earn` option. If the withdrawn amount is smaller than the balance, the users end up in a race condition to save the withdraw fees.

6.2 Known Issues Before the Audit

Note **Version 1**

The following two issues were known before the audit:

In AngleVault:

> `withdrawAll()` wrongly defined. Since that every LP obtained will be staked directly into the related LGV4, for this reason the `msg.sender`'s balance would be always `0`.

In CurveStrategy:

> An edge case can happen within the `harvest()` for certain type of curve gauges, with more than one extra reward.