Code Assessment
of the StakeDAO-Frax-veSDT Smart Contracts

Apr 04, 2022

PRIVATE

Produced for

by

DRAFT

Contents

DRAFT

1 Executive Summary

Dear Julien,

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 StakeDAO-Frax-veSDT according to Scope to support you in forming an opinion on their security risks.

StakeDAO implements an alternative to staking into Curve, Angle or Frax and earn additional rewards. Similar to Curve the reward allocation can be voted on by Stake Dao token holders who locked their stake Dao in return for voting escrowed Stake Dao.

The first audit was limited to three contracts (see Version 1 and Version 2). The issues found are tagged accordingly in this report. As a result of the first audit the documentation and inline comments were refined and enhanced, however there is still room for improvement. In the second stage of the audit we reviewed most of the system as layed out in Scope.

We uncovered one high and one medium severity issue. In the high severity issue a wrong variable is used as index. The medium severity issue is already public. Angle tweeted about it and fixed it in their code base. The remaining issues are of low severity.

The communication with the team was always professional and quick. We are happy to help in the future and conduct the final review of the remaining system parts to assess the system's security as a whole. The current code base provides medium security. We highly recommend to improve the testing and review the test cases as they are occasionally incorrect and do not test correctly.

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 High -Severity Findings
• No Response
Medium -Severity Findings
Low -Severity Findings
0 1
1
3
26
• Code Corrected
1
• Specification Changed
1
• No Response
1
• Code Corrected
9
• Acknowledged
3
• No Response
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 StakeDAO/sd-frax-veSDT repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.
Version 3
The Solidity compiler version 0.8.7 and Vyper compiler version 0.2.16 were chosen for the smart contracts.

2.1.1 Excluded from scope

Version 3

The following files were excluded from the scope of this audit: • AngleAccumulatorV2.sol
• CrvDepositor.sol
• sdCRV.sol
• CrvAccumulator.sol
• contracts in external folder

The system is reviewed with the current configuration and tokens only and not for the general use with other tokens that might have different behavior or decimals. We only reviewed the functionality for gauges of type 0, for other types we did not review the respective contracts and, hence, we assume they work as expected and are secure.

V	Date	Commit Hash	Note
1	14 Jan 2022	7e702aba329d5780ef5841f44ad699385b8b428f	Initial Version
2	15 Feb 2022	86c5b856c17a8fe8c4b393eaeb967ec47830a499	Version 2
3	22 Mar 2022	3868862eb3fb3a360ddeb3a0e95fcf4b4acf2252	Version 3

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.

The contracts in the scope of this audit from StakeDAO-Frax-veSDT consist of two systems. One for Stake Dao's SDT owners to earn rewards on their STD and one for CRV, Angle and Frax token owners to earn rewards on their respective tokens. The systems are connected as SDT token owners can distribute newly minted SDT tokens to users invested in Stake Dao with CRV, Angle and/or Frax. We first describe the system for CRV, Angle and Frax token investors.

Curve, Frax or Angle tokens owners can stake their tokens into Stake Dao's depositor contracts. The users' funds are meant to be locked forever and can only be recovered by Stake Dao's governance account after the lock in the respective underlying protocol has expired. When depositing, the user receives sdTokens (e.g., sdFXS for Frax). The depositor contract forwards

the funds to a locker contract which locks up the CRV, FXS or ANGLE in the native staking contracts to receive native voting escrowed tokens (e.g., veCRV). The voting escrowed tokens are kept in Stake Dao's locker contract but can be taken out by Stake Dao's governance contract at anytime. Stake Dao's governance can also vote on the gauge weights with the full voting escrow token balance in the respective underlying protocol. The reward tokens earned from the native protocol (e.g., sanUSD_EUR for ANGLE or 3CRV for Curve) are collected in Stake Dao's accumulator contracts (e.g., Angle accumulator) and forwarded to Stake Dao's gauge contracts. Users can claim their rewards from the gauge contracts directly or through a ClaimRewards helper contract. Additionally to the rewards from Curve, Angle and Frax through the accumulator contracts, the gauge contracts get SDT from the SDT distributor contract.
The second part of the system is responsible for setting the SDT distribution weights between the gauges. Users can stake their SDT into the veSDT contract. As a reward for staking the users receive sdFrax3CRV token. Furthermore, users owning veSDT can vote on the SDT distribution in the GaugeController contract which will set the weights for the gauge allocation. The distribution weights can be always overwritten by Stake Dao's governance.

2.2.1 sdToken

In Version 1 this contract was named sdFXSToken and renamed to sdToken. The contract implements an ERC20-compliant token and in addition has a state variable that stores the address of the operator. Initially, the operator is set to the msg.sender in the constructor of the contract. The contract implements following functions:

- setOperator: sets a new operator for the contract. Only the current operator can call this function.
- mint: mints new tokens for an arbitrary address and can be called only by the operator.
- burn: burns tokens from an arbitrary address and reduces the total supply. Only operator can call this function.

A special version of the sdToken is the sdCRV which in the constructor handles the fact that Stake Dao already has an sdVeCrv token that is still supported but should be converted into the new token in the CrvDepositor.

2.2.2 Depositor

In Version 1 this contract was named FxsDepositor. Depositor contract enables end users to deposit, lock, and stake their tokens. All users locking tokens via this contract commit to the same unlock time. Whenever locking is triggered, the function _lockFXS postpones the unlock time to four years from the current timestamp. The governance can change this default behavior by setting relock to false. In this situation, users can still deposit, lock and stake tokens but the unlock time is not postponed.

- deposit: allows any user to deposit FXS tokens into the contract. The users should specify their preference if they want the tokens to be also locked and staked in veFXS. It is important to note that FXS tokens cannot be withdrawn by users once they are locked.
- depositFor Version 1 : similar to deposit but the tokens are minted for a specified address, while the FXS tokens are paid by the msg.sender. Differently from deposit, this function always locks and stakes the deposited tokens.

• depositAll: a wrapper function to call deposit with the whole balance of the caller.
• lockFXS renamed to lockToken: anyone can call this function to trigger the locking of FXS tokens that are held by this contract. The function rewards the caller with incentiveToken tokens that are minted on behalf of the caller.
• Setters: the governance can update the important state variables, such as: governance, gauge, relock, lockIncentive, and the operator of the sdToken.
A special version of the Depositor is the CrvDepositor that supports Stake Dao's legacy curve contract sdVeCrV. The token can be converted by locking it forever to the new sdCrv by calling lockSdveCrvToSdCrv.

2.2.3 Locker contracts

In Version 1 this contract was named FraxLocker, while in Version 2 it was renamed to FxsLocker. Version 2 supports FXS in the FxsLocker and Angle in the AngleLocker. They work very similar and we describe the logic with the FXS locker. The curve locker is already deployed. Existing sdveCRV holders can migrate to the new sdCRV token, by forever locking their sdveCRV tokens in the CrvDepositor contract. Locker contracts are responsible for locking the native tokens to voting escrow tokens like veFXS (voting escrow FXS). Therefore, FxsLocker should be whitelisted by the voting escrow contract like veFXS. End users do not interact directly with this contract as its functionalities are restricted to special addresses: governance, fxsDepositor and accumulator. The contract
implements the following functions:
• createLock: creates a lock in e.g., veFXS contract that locks FXS tokens for a given time. The function can be called only by the governance.
• increaseAmount: increases the amount of locked tokens while preserving the unlocking time. The function can be called only by the governance or the depositor.
• increaseUnlockTime: postpones the unlock time for the locked tokens. This increases the vote weight of tokens already locked. The function can be called only by the governance or the depositor.
• claimFXSRewards: transfers the yield for the Yield Distributor to an arbitrary address passed as parameter to the function. The function can be called only by the governance or the accumulator.
• release: withdraws the locked tokens after the lock time has expired. The tokens are transferred to an arbitrary address passed as parameter to the function. The function can be called only by the governance.
• voteGaugeWeight: forwards the call to Gauge Controller. The function can be called only by the governance.
• Setters: the contract implements multiple functions that are restricted to the governance and allow the update of important state variables, such as governance, fxsDepositor, yieldDistributor, gaugeController and accumulator.
Finally, the locker contracts implement a special function execute(to, value, data) which allows the governance to call any address to, with any msg.value and arbitrary data. The already deployed CRV locker is out of scope.
Lockers are communicating with the external protocols to get the rewards and forward the rewards to the corresponding accumulators.

DRAFT

2.2.4 Accumulators

Accumulator contracts can be triggered to call the locker which will get the rewards from the native third party protocol, send it to the locker and the locker will forward it to the respective gauge. In Angle's AngleAccumulatorV2 it will convert the reward token SAN_USDC_EUR to agEUR before sending it to the gauges.

The main functionality of the accumulator are the two functions:

• notify and notifyAll which trigger the locker to withdraw and send potential rewards to the accumulator which forwards the funds the gauge.

Besides this functions the accumulator has:

• notifyExtraReward and notifyAllExtraReward which are simply forwarding either a precise balance given as argument or all reward tokens that might be in the accumulator to the gauge.

• depositToken to deposit tokens to send any tokens into the contract (which could also be done with a normal transfer).

• rescueERC20 allows the governance account to withdraw any funds from the contract.

• setter function restricted to the governance to set gauge, governance, locker and the reward token.

There are two Angle accumulators (one converting the SAN_USDC_EUR one simply forwarding it), one CurveAccumulator and the FxsAccumulator.

2.2.5 Liquidity Gauges

After receiving the funds from the accumulators, users that locked and staked into the LiquidityGaugeV4 contract via the depositor or directly in the gauge contract, can claim and withdraw their rewards. The gauges calculate the amount a user is eligible to receive.

• user_checkpoint takes care of the accounting of the claimed and claimable funds.

• set_rewards_receiver is used to set a default receiver for funds. It can be called by any user to set it for the same user.

• claim_rewards claims the user's rewards and triggers the accounting

• claim_rewards_for is called by the helper contract claimRewards and helps users to claim and re-stake their funds.

• kick user that boosted their voting power and did not update after boosting will have an incorrect voting power. By calling kick other users can correct the voting power back to the correct one.

• deposit allows sdToken owners to stake their tokens and be eligible for the gauge's rewards.

• withdraw allows users who staked sdTokens to withdraw their staked sdTokens.

The following functions are similar to the ERC20 functions but include the accounting for past rewards. • transfer

• transferFrom

• approve

• increaseAllowance • decreaseAllowance

The flowing functions are functions only the admin can call

• add_reward sets a new distributor (accumulator) contract for a new reward token.

•set_reward_distributor (also callable by the distributer contract itself) to change the distributor for a given token

• set_claimer sets the claimer contract's ClaimRewards address. •commit_transfer_ownership and accept_transfer_ownership are called by the old admin to nominate a new admin and by the new admin to accept the admin role.

The remaining function deposit_reward_token can be called by the accumulator contracts to deposit the reward into the gauge.

2.2.6 ClaimRewards

The ClaimRewards contract is a convenient helper contact for users who want to claim and re-strake their rewards. Instead of calling the gauge and veSDT contracts separately, users can save gas and use the ClaimRewards contract.

Users can call claimRewards and claim rewards from a specified array of gauges or call claimAndLock to additionally stake the received SDT token into the veSDT contract. The remaining functions are onlyGovernance to recover any token from the contract, allow or disallow certain gauges or depositor contracts.

2.2.7 Gauge Controller

The gauge controller contract calculates the weights to distribute the SDT rewards that are send to the different gauges. Each gauge has an individual weight and a type weight. The type weight is set for certain kinds/groups of gauges. Together, it determines the weight and, hence, the percentage a gauge will receive from the newly minted SDT tokens.

The admin restricted functionality is:

• commit_transfer_ownership and accept_transfer_ownership to transfer the ownership of the contract.

• add_gauge to add a gauge and optional a initial weight and assign it to a type.

• addType allows adding a new gauge type (grouping).

• change_type_weight is used to change the weight of this type/group.

• change_gauge_weight overrides the users' votes and sets a weight for a gauge.

The entry point for users that own veSDT is vote_for_gauge_weights. It allows users to vote on the weights of the gauges. The functions checkpoint, checkpoint_gauge and gauge_relative_weight_write can be called by anyone. They update/checkpoint the weights for all or specific gauges.

2.2.8 StdDistributor

The StdDistributor contract distributes the new SDT tokens to the gauges according to the weights the veSDT owners voted. It queries the gauge controller to get the weights and sends the SDT to the gauges. The SDT is withdrawn from the Masterchef contract.

The main function is distributeMulti which will distribute the SDT rewards to multiple gauges. The remaining functions are all restricted to a GOVERNOR_ROLE or GUARDIAN_ROLE. The functions setDistribution, setGaugeController and setDelegateGauge are simple setters for the corresponding variable.

• toggleGauge allows to deactivate/kill a gauge to not be considered for the distribution anymore. • approveGauge sets max approval to a specific gauge to transferFrom tokens.

• setTimePeriod sets the interval to pull from MasterChef.

DRAFT

• recoverERC20 will send any token from the contract to a specified address.
• toggleInterfaceKnown is restricted to the GUARDIAN_ROLE role. It toggles the fact that a gauge delegate can be used for automation or not and therefore supports.

2.2.9 Masterchef and Mastercheftoken

The Masterchef contract is the owner of the SDT token contract and mints the new SDT tokens. To receive tokens from the Masterchef, a deposit needs to be done for a specific pool. Each pool will get a predefined amount of new token minted. Each pool has a token that can be deposited into the pool. The Stake Dao governance account needs to create a token (MasterchefMasterToken), add a new pool with this token, define the SDT this pool receives per block and deposit the according pool token into the pool. Hence, the MasterchefMasterToken is just a workaround to have a token for a pool and some kind of access control.

2.2.10 veSDT

veSDT is the voting escrow contract for the Stake Dao token and works similar to Curve's and Angle's voting escrow contract. Allowing users to lock their SDT and receive veSDT. Depending on the time and amount locked the voting weight determined. The core functionalities for the users are:

• create_lock to lock a certain amount of SDT for a defined time.
• deposit_for, deposit_for_from and increase_amount which are functions to increase the amount tokens locked for a specific address.
• withdraw allows users to withdraw their staked STD after the unlock time has passed.

The admin can call

• commit_transfer_ownership and accept_transfer_ownership to transfer the ownership of the contract.
• commit_smart_wallet_checker and apply_smart_wallet_checker to set a whitelisting contract that checks if a given contract has the permission to own veSDt.
• increase_unlock_time can be called to lock the already locked funds for a longer period and gain more voting power.

Anyone can call checkpoint to account for the global data needed to calculate the voting weights. 2.2.11 FeeDistributor

The FeeDistributor contract distributes the reward tokens (sdFrax3Crv) to veSDT holder. They can call claim or claim_many to receive their share of the reward or checkpoint_token to trigger the reward calculation and accounting. checkpoint_total_supply can be called to only trigger the total supply accounting. The last two functions can only be called if the admin allowed checkpointing for users. Additionally, users can send in all their tokens and donate them via calling burn to the contract.

The remaining functions are administrative functions.

• commit_admin and accept_admin transfer the admin power of the contract.
• toggle_allow_checkpoint_token toggles if anyone can checkpoint ot only the admin.
• kill_me deactivates the contract functions and transfers the token balance out to an emergency account.
• recover_balance allows to withdraw any token that supports the transfer method from the contract except for the reward token itself.

2.2.12 SmartWalletWhitelist

Is a simple whitelist that can be used to whitelist addresses that are allowed to hold veSDT. The whitelisting is done by the admin via approveWallet and revokeWallet. The whitelisting can be relayed/delegated to another checking contract. If the admin sets another checker contract via commitSetChecker and applySetChecker. Additionally the admin account can be changed by calling commitAdmin and applyAdmin.

2.2.13 Assumptions

The system has many roles as there is no centralized access control. Each contract has its own access control and admin role. We assume that all privileged roles are fully trusted. We additionally assume that all roles will be controlled by a properly setup governance account. We refer to all these roles as governance roles.

The audit was performed on a specific setup with a fixed set of tokens and contracts. Modifications and updates might break the system and need to be checked carefully before applied. We assume that critical functions like distributeMulti in the StdDistributor are always called in time and for all relevant contracts.

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 any open findings. Findings that have been resolved have been moved to the Resolved Findings section. 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 1

• Incorrect Index Used to Access depositorsIndex
Medium -Severity Findings 1
• Possible to Lock Users' Funds Into veSDT
Low -Severity Findings 17
• Broad Function Visibility: approveWallet
• Inconsistent Checks When Depositing in veSDT
• Inconsistent Procedure for Updating admin
• Inconsistent Specification: deposit_for_from
• Inconsistent Specification: initialize
• Mismatch of Specification With the Function Modifier in AngleLocker • Missing Documentation for Parameter
• Missing Events for Sensitive Operations
• Missing Sanity Checks
• Missing Sanity Checks: AngleLocker
• Non-indexed Events
• Possible Gas Optimization in Mappings
• Possible to Optimize the Check on Distributor of tokenReward
• Unused Events
• Missing Sanity Checks: FraxLocker Acknowledged
• Missing Sanity Checks: FxsDepositor Acknowledged
• Missing Sanity Checks: sdFXSToken Acknowledged
5.1 Incorrect Index Used to Access
depositorsIndex
Correctness High Version 3

In the function addDepositor of ClaimRewards , values of depositorsIndex are set using depositor addresses as indexes.
In claimAndLock this array is accessed twice using token addresses as indexes.
Given that there are no contract defining both a token and a depositor in the codebase, it would most likely lead depositorsIndex[token] to always evaluate to 0 and hence use the first element of lockStatus.staked and lockStatus.locked as decisions for each token.
5.2 Possible to Lock Users' Funds Into veSDT Design Medium Version 3
Users that lock their tokens into the voting escrow contract need to approve an allowance to veSDT and then call deposit_for or deposit_for_from to transfer the tokens. However, if a user approves to the veSDT an amount that is larger than the intended amount of tokens to be locked, or max uint for simplicity, the user's tokens are exposed to arbitrary locking. In such cases the function deposit_for allows anyone to lock more of user's tokens into the contract without their clear consent. This is possible because the function``deposit_for`` calls the internal function _deposit_for without passing the msg.sender as a parameter:
The internal function transfers the tokens from _addr if enough allowance exists, while the caller only pays the gas costs:
5.3 Broad Function Visibility: approveWallet Design Low Version 3

The visibility of the function SmartWalletWhitelist.approveWallet is public, however it is not called internally. For functions that are expected to be called from other contracts only, the function visibility can be restricted to external instead of public. This allows to save gas costs, as public functions copy array function arguments to memory which can be expensive.

```
 depositorsIndex[_depositor] = depositorsCount;
  if (depositor != address(0) && lockStatus.locked[depositorsIndex[token]]) {
        IERC20(token).approve(depositor, balance);
        if (lockStatus.staked[depositorsIndex[token]]) {
    IDepositor(depositor).deposit(balance, false, true, msg.sender); } else {
IDepositor(depositor).deposit(balance, false, false, msg.sender);
     }
      def deposit_for(_addr: address, _value: uint256):
    ...
    self._deposit_for(_addr, _value, 0, self.locked[_addr], DEPOSIT_FOR_TYPE)
    def _deposit_for(_addr: address, _value: uint256, unlock_time: uint256, locked_balance:
LockedBalance, type: int128):
    ...
    if _value != 0:
    assert ERC20(self.token).transferFrom(_addr, self, _value)
```

5.4 Inconsistent Checks When Depositing in veSDT

Correctness Low Version 3

The function increase_amount requires that the msg.sender is either an externally owned contract or
a whitelisted contract:

However, the function deposit_for performs the same operation if addr is msg.sender and does not have the above restriction.

5.5 Inconsistent Procedure for Updating admin Design Low Version 3

Multiple contracts have an admin role that is privileged and can call sensitive functions. However, the procedure to update such privileged roles is not consistent among different contracts. Namely, SmartWalletWhiteList uses commit/apply approach, meaning the current admin initially calls commitAdmin and then should call applyAdmin to set the new admin. While, LiquidityGaugeV4, FeeDistributor, veBoostProxy use commit/accept approach. Differently from the previous contracts, veSDT provides both procedures commit/accept and commit/apply to update the admin.

5.6 Inconsistent Specification: deposit_for_from

Correctness Low Version 3

The functions deposit_for and deposit_for_from have a similar behavior, however their NatSpec specification is inconsistent. The comment for deposit_for:

while the respective description for deposit_for_from is:

5.7 Inconsistent Specification: initialize Correctness Low Version 3
The NatSpec description of veSDT's initialize function describe token_addr as being the address of the ERC20ANGLE contract while the contract is a voting escrow for the SDT token.

```
     self.assert_not_contract(msg.sender)
        @dev Anyone (even a smart contract) can deposit for someone else, but
     cannot extend their locktime and deposit for a brand new user
   @dev Anyone (even a smart contract) can deposit for someone else from their account
```

5.8 Mismatch of Specification With the Function
Modifier in AngleLocker Correctness Low Version 3
The specification of the AngleLocker's function createLock states that it can only be called by governance or proxy, however, the modifier onlyGovernance is used and the mentioned proxy is not declared anywhere.
5.9 Missing Documentation for Parameter
Design Low Version 3
The function GaugeController.__init__ has no NatSpec description for the parameter admin.
5.10 Missing Events for Sensitive Operations
Design Low Version 3
Multiple contracts do not emit events when sensitive operations are performed, e.g., the update of the
admin for a contract.
We provide below some examples:
• SmartWalletWhitelist.sol: applyAdmin and applySetChecker.
• ClaimRewards.sol: setGovernance.
• SdtDistributor.sol: initializeMasterchef, setDistribution and setTimePeriod. •
LiquidityGaugeV4.vy: add_reward, set_reward_distributor and set_claimer.
5.11 Missing Sanity Checks
Design Low Version 3
Several setter functions in multiple contracts do not perform sanity checks for the new values that are set.
We provide examples of such cases below:
•SdtDistributor.sol: _masterchef parameter in initialize and _delegateGauge in setDelegateGauge.
• LiquidityGaugeV4.vy: _distributor in add_reward.
• veSDT.vy: token_addr in initialize and addr in commit_smart_wallet_checker.
• FeeDistributor.vy: _start_time in the constructor.
5.12 Missing Sanity Checks: AngleLocker Design Low Version 3

The setter functions take an address as a parameter and assign it to a state variable. Given the sensitivity of such functions, basic sanity checks on the input parameter help to eliminate the risk of setting address(0) to the state variable of the contract by accident (e.g. UI bugs).

5.13 Non-indexed Events

Design Low Version 3

Events can be indexed to easily filter and search for the indexed arguments. This is used in most contracts. Without full specification about what needs indexing we simply highlight that the following occasions are not indexed and the need of indexing should be revised by StakeDAO.

- Completely no indexed events in BaseAccumulator
- Completely no indexed events in ClaimRewards
- Completely no indexed events in GaugeController
- Multiple not indexed events in LiquidityGaugeV4
- Multiple not indexed events in veBoostProxy
- No indexed events in CommitAdmin and ApplyAdmin in FeeDistributor

5.14 Possible Gas Optimization in Mappings

Design Low Version 3

Multiple contracts of the system use mappings in the format: mapping(key_type => bool). Solidity uses a word (256 bits) for each stored value and performs some additional operations when operating bool values (due to masking). Therefore, using uint256 instead of bool is slightly more efficient.

We provide below the list of mappings that can be optimized:

- SmartWalletChecker.sol: wallets.
- ClaimRewards.sol: gauges.
- SdtDistributor.sol: killedGauges, isInterfaceKnown and isGaugePaid.

5.15 Possible to Optimize the Check on Distributor of tokenReward

Design Low Version 3

The function BaseAccumulator._notifyReward checks if the distributor of _tokenReward is not address(0), then it performs the two external calls as shown below:

```
        if (ILiquidityGauge(gauge).reward_data(_tokenReward).distributor != address(0)) {
                IERC20(_tokenReward).approve(gauge, _amount);
 ILiquidityGauge(gauge).deposit_reward_token(_tokenReward, _amount);
 ... }
```

The function call deposit_reward_token succeeds only if the accumulator is the distributor for the _tokenReward, otherwise it reverts. Hence, the function could be optimized by directly checking if the distributor of the _tokenReward is the accumulator.

5.16 Unused Events

Design Low Version 3

Several contracts declare events that remain unused in the existing code base. The StakeDAO should
assess if such events should be removed or emit them accordingly. We provide a list of unused events:

- ClaimRewards.sol: DepositorDisabled, RewardClaimedAndLocked and

RewardClaimedAndSent.
• SdtDistributorEvents.sol: DistributionsToggled, RateUpdated, UpdateMiningParameters.
• GaugeController.vy: KilledGauge.

5.17 Missing Sanity Checks: FraxLocker Design Low Version 1 Acknowledged

The setter functions take an address as a parameter and assign it to a state variable. Given the sensitivity of such functions, basic sanity checks on the input parameter help to eliminate the risk of setting address(0) to the state variable of the contract by accident (e.g. UI bugs).

Acknowledged

Due to efficiency reasons, StakeDAO decided to keep the function as it is.

5.18 Missing Sanity Checks: FxsDepositor Design Low Version 1 Acknowledged

The setter functions that take an address as a parameter and assign it to a state variable lack basic sanity checks on the input parameter. Such checks would help to eliminate the risk of setting address(0) to the state variable of the contract by accident (e.g. UI bugs).

Acknowledged

Due to efficiency reasons, StakeDAO decided to keep the function as it is.

5.19 Missing Sanity Checks: sdFXSToken Design Low Version 1 Acknowledged

The function setOperator takes an address as a parameter and assigns it to the state variable operator. Given the sensitivity of this function, basic sanity checks on the parameter _operator help to eliminate the risk of setting address(0) as the operator of the contract by accident (e.g. UI bugs).

Acknowledged

Due to efficiency reasons, StakeDAO decided to keep the function as it is.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

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 • Inconsistent Access Control Code Corrected
• Update of unlockTime Specification Changed

Low -Severity Findings 9
• Broad Function Visibility Code Corrected
• Commented Code Code Corrected
• Mismatch of Specification With the Function Modifier Code Corrected • Revert Message on Modifier Code Corrected
• Unused Event Voted Code Corrected

• Unused Imports: FxsDepositor Code Corrected
• Unused Imports: FxsLocker Code Corrected
• Unused Imports: sdFXSToken Code Corrected
• createLock Access Control Code Corrected

6.1 Inconsistent Access Control

Design Medium Version 1 Code Corrected

The access control for FraxLocker.execute is onlyGovernanceOrDepositor. The function basically allows to call any arbitrary contract and function. The function FraxLocker.claimFXSRewards has the following access control onlyGovernanceOrAcc. As execute can replicate the behavior of claimFXSRewards the access control is inconsistent because claimFXSRewards can be replicated by execute. Ultimately, giving the Depositor the same power as Acc in this case.

This is only a theoretical problem in the current implementation due to another issue.

Code corrected

The updated code protects the function execute with the modifier onlyGovernance, which restricts the access to only the governance address.

6.2 Update of unlockTime Design Medium Version 1 Specification Changed

We do not have sufficient specification about the intended behavior, but the following seems to be an issue. The internal function _lockFXS updates the unlockTime if the following condition is satisfied:

Given that both unlockInWeeks and unlockTime store the number of seconds passed until a given week, the comparison with 2 (sec) seems incorrect.

Specification changed

The current code will always evaluate the if condition as true if the comparison is bigger than 1. StakeDAO changed the specification from two weeks to one week. Additionally, the 2 was changed to 1 (which has no effect but makes it more explicit). The code works but we need to highlight, that this only works for one week check.

6.3 Broad Function Visibility

Design Low Version 1 Code Corrected

The function depositFor in FxsDepositor is declared as public but it is never called internally. Following the best practices, functions expected to be called only externally should be declared as external.

Code corrected

The function depositFor has been removed from the contract.

6.4 Commented Code

Design Low Version 1 Code Corrected

The contract FraxLocker includes a function vote which is commented out. Please elaborate on the
cause and if this functionality should be implemented or the code removed completely.

Code corrected

The commented function was removed from the code base.

```
    if (unlockInWeeks.sub(unlockTime) > 1) {
      ILocker(locker).increaseUnlockTime(unlockAt);
      unlockTime = unlockInWeeks;
}
```

6.5 Mismatch of Specification With the Function Modifier

Correctness Low Version 1 Code Corrected

The specification of the function createLock states that it can only be called by governance or proxy, however, the modifier onlyGovernanceOrDepositor is used, which checks if the msg.sender is either the governance or fxsDepositor address. Additionally, the fxsDepositor contract does not implement any functionality which calls createLock currently.

Code corrected*

The updated spec state that createLock can be called only by the governance. The respective modifier onlyGovernance is now used.

6.6 Revert Message on Modifier

Correctness Low Version 1 Code Corrected

The modifier onlyGovernanceOrDepositor checks if the msg.sender is either governance or fxsDepositor address as shown:

```
        modifier
 }
require
  ); _;
onlyGovernanceOrDepositor() {
     (
msg.sender
== governance ||
== fxsDepositor,
msg.sender
 "!(gov||proxy||fxsDepositor)"
```

The error message claims that msg.sender is not proxy address, which is not declared in the contract.

Code corrected

The error message has been updated accordingly.

6.7 Unused Event Voted Design Low Version 1 Code Corrected

The contract FraxLocker declares the event Voted, however, it is not used in the current codebase.

Code corrected

The unused event Voted has been removed from the code.

DRAFT
6.8 Unused Imports: FxsDepositor Design Low Version 1 Code Corrected
The file FxsDepositor.sol (Version 2 Depositor contract) has the following unused import:
Code corrected
The unused libraries listed above have been removed from the updated code.
6.9 Unused Imports: FxsLocker Design Low Version 1 Code Corrected
The contract FxsLocker has the following unused imports:
Code corrected
The unused libraries have been removed from the updated code.
6.10 Unused Imports: sdFXSToken Design Low Version 1 Code Corrected
The file sdFXSToken.sol (Version 2 sdToken) has the following unused imports:
Code corrected
The unused libraries listed above have been removed from the updated code.

```
        import
import
import
"@openzeppelin/contracts/token/ERC20/ERC20.sol"
; ;
;
 "@openzeppelin/contracts/utils/Address.sol"
"@openzeppelin/contracts/utils/Context.sol"
        import
import
"@openzeppelin/contracts/math/SafeMath.sol"
"@openzeppelin/contracts/utils/Address.sol"
; ;
         import
import
import
import
"@openzeppelin/contracts/token/ERC20/IERC20.sol"
; ;
;
;
 "@openzeppelin/contracts/utils/Address.sol"
 "@openzeppelin/contracts/token/ERC20/SafeERC20.sol"
 "@openzeppelin/contracts/utils/Context.sol"
```

DRAFT
6.11 createLock Access Control Design Low Version 1 Code Corrected
The functions createLock, release and execute have the modifier onlyGovernanceOrDepositor, but the contract FxsDepositor never calls these functions. Specifications covering use cases when these functions are called by the depositor are missing.

Code corrected
The modifier for the functions listed above has been updated to onlyGovernance.

7 Open Questions

Here, we list open questions that came up during the assessment and that we would like to clarify to ensure that no important information is missing.

7.1 Motivation of BaseAccumulator.depositToken

Open Question Version 1

The function BaseAccumulator.depositToken allows users to transfer any ERC20 token to the accumulator contract. What is the motivation for providing this functionality for accumulators?

8 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.

8.1 Admin's Weight on a Gauge Can Be Overwritten

Note Version 3

The function change_gauge_weight in GaugeController allows the admin to set the weight of any gauge to an arbitrary value. This value can be altered by voters of the gauge. If users vote for the gauge, its weight is increased to a higher value than set by the admin. Furthermore, if users that previously voted the gauge (before the admin called change_gauge_weight) remove their votes, the weight of the gauge is decreased to a lower value than set by the admin.

8.2 All Gauges Should Be Trusted

Note Version 3

The gauges are added into the system by the admin of the GaugeController and they are considered to be non-malicious. If an untrusted gauge is added, then it can exploit an reentrancy vulnerability in the function SdtDistributor._distributeReward which can drain all rewards:

8.3 Calculation of Seconds in Years

Note Version 1

The constant MAXTIME assumes a year with 364 days which is fully divisible by 7 -- days in a week.

8.4 Dust Amounts Not Accounted in veSDT Note Version 3

The function veSDT._checkpoint ignores locked tokens with an amount smaller than MAXTIME = 4 * 365 * 86400:

```
ILiquidityGauge(gaugeAddr).deposit_reward_token(address(rewardToken), sdtDistributed);
uint256 private constant MAXTIME = 4 * 364 * 86400;
```

```
def _checkpoint(addr: address, old_locked: LockedBalance, new_locked: LockedBalance):
...
u_old.slope = old_locked.amount / MAXTIME
...
```

```
u_new.slope = new_locked.amount / MAXTIME
...
```

If old_locked.amount or new_locked.amount is less than MAXTIME, the respective slope is set to 0.

8.5 Event Can Be Emitted Multiple Times

Note Version 3

Several contracts follow the approach commit/accept to set a new admin for the contract. For such updates, an event CommitOwnership/ CommitAdmin is emitted on the commit operation, and ApplyOwnership / ApplyAdmin event is emitted when the new admin accepts the transfer. However, the accepting functions can be called multiple times, hence the respective events would be emitted for every call. We provide a list of such contracts here:

• GaugeController • veSDT
• FeeDistributor
• LiquidityGaugeV4 • veBoostProxy

8.6 Hardcoded Function parameter``unlock_time``

Note Version 3

The internal function _deposit_for_from has a parameter``unlock_time``, which is hardcoded to 0
whenever called.

8.7 Incorrect Checks in Tests

Note Version 3

Several tests were found to not perform the proper checks in the end of a task. Although it is out of the
scope of this assessment, we provide a non-exhaustive list of such tests here:

• step1_frax - Should lock FXS: The transfer's recipient is the locker and not the depositor, meaning that the assertion for the depositor to hold 0 FXS will trivially pass.
•step1_frax - Should deposit and stake in gauge for caller: expect(userSdFxsBalanceAfter).to.be.equal(userSdFxsBalanceAfter) will always pass.
• step1_frax - Should lock FXS: While the specification indicates "Lock FXS already deposited into the Depositor if there is any", each deposit call is followed by a call to lock. The functionality of locking already deposited FXS is not tested here.
•step1_angle - Should deposit and stake in gauge for caller: expect(userSdAngleBalanceAfter).to.be.equal(userSdAngleBalanceAfter) will always pass.

8.8 Outdated Compiler Version

Note Version 1
The compiler version: 0.8.7 is outdated (https://swcregistry.io/docs/SWC-102). The compiler version
has the following known bugs.
This is just a note as we do not see any severe issue using this compiler with the current code. At the
time of writing the most recent Solidity release is version 0.8.13.

8.9 Possible Reentrancy in lockToken for Special Tokens

Note Version 3
The function Depositor.lockToken performs a mint operation and afterwards emits an event and updates the state variable incentiveToken:
In the current code base, minter token is always the sdToken which extends the ERC20 standard and does not provide any callback functionality to the receiver, hence the code above is not vulnerable to reentrancy attacks. However, if in the future versions of the code the minter token is supposed to support callbacks, e.g., implement ERC777 standard and the mint operation provides an opportunity for reentrancy, the above function would be exploitable.

8.10 Proper Declaration of Constant Variables

Note Version 3
The variable MAX_REWARDS in ClaimRewards is declared as immutable and assigned to a literal
value. To improve code readability, the variable can be marked as constant.

8.11 Reward Distribution Should Be Called Periodically for All Gauges

Note Version 3
The function SdtDistributor.distributeMulti works correctly only if it is called periodically (at least once a day) for all the gauges, otherwise the following two issues arise:
1. Failing to call distributeMulti for a gauge on a given day means that the gauge does not receive its share of rewards for the respective day and the funds are locked in the contract. Only the governance can recover these funds via recoverERC20 function.

```
    if (incentiveToken > 0) {
ITokenMinter(minter).mint(   , incentiveToken); emit IncentiveReceived(   , incentiveToken);
incentiveToken = 0;
}
msg.sender
msg.sender
```

2. On the time period that overlaps with the weekly event of updating votes for gauges, there is a time window for a malicious user to manipulate the rewards distributed to gauges. For example, if a gauge receives a higher weight for the following week, it is profitable for a malicious user to call the function distributeMulti when the new weight is applied, and

vice-versa. This makes the accounting of rewards in SdtDistributor incorrect and potentially can prevent legit gauges from receives any reward.
As stated in the System Overview, StakeDAO should run a bot that guarantees the function is called periodically and correctly for all gauges to prevent the issues above.

8.12 Tautology in if Condition

Note Version 1

In the function setFees of the depositor, _lockIncentive is verified to be greater than or equal to zero, however, as it is a unsigned integer, this condition will always hold.

8.13 Typo in Mappings Specification

Note Version 3

The NatSpec comment for the SdtDistributor.pulls has a typo.

8.14 ClaimRewards Functions Should Be Called Only With Enabled Gauges

Note Version 3

The functions claimRewards and claimAndLock take a list of gauges as a parameter and check that each of them is enabled. If one of the gauges in _gauges is disabled by the governance, the functions reverts. Hence, the caller should always guarantee that that all gauges passed into the functions are enabled.

8.15 safeApproveUsage Note Version 1

The contract FxsDepositor (Depositor in Version 2) uses safeApprove to update the allowance give to the gauge. As explained in the specifications of the function, safeApprove is deprecated.

```
        /**
*
*/
 * @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.
```