PUBLIC

Code Assessment

of the FXS Locker and veSDT Smart Contracts

Feb 15, 2022

Produced for

by

Contents

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 FXS Locker and veSDT according to Scope to support you in forming an opinion on their security risks.

StakeDAO implements contracts to stake and allocate rewards over voting/gauges.

The audit was limited to three contracts. These contracts are a very small part of a bigger system. Their functional correctness and use in the overall system have not yet been part of the audit. Hence, at this very limited stage of this audit we cannot judge on the overall security of the system. Documentation and inline comments were refined and enhanced, however there is still room for improvement.

The issues we uncovered are of medium and low severity. Most severe ones are: i) an issue when updating the time where we think the units are incorrect, and ii) an inconsistent access control. All issues were acknowledged or resolved. The communication with the team improved significantly towards the end of the audit and we are happy to help in the future and conduct a review of the full system to assess the system's security as a whole.

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
• Code Corrected	1
• Specification Changed	1
Low-Severity Findings	12
• Code Corrected	9
• Acknowledged	3

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 FXS Locker and veSDT 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	14 Jan 2022	7e702aba329d5780ef5841f44ad699385b8b428f	Initial Version
2	15 Feb 2022	86c5b856c17a8fe8c4b393eaeb967ec47830a499	Version 2

Version 1

For the solidity smart contracts, the compiler version `0.6.12` was chosen.

The files in scope for this review were:

- sdFXSToken.sol
- FraxLocker.sol
- FxsDepositor.sol

Version 2

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

The files in scope for this review were:

- sdToken.sol
- FxsLocker.sol
- Depositor.sol

2.1.1 Excluded from scope

Excluded from scope are all other files not listed above and the functional correctness (especially when calling other contracts than the three listed above).

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 FXS Locker and veSDT project handle the locking and staking of FXS tokens into the system. The project allows end users to deposit FXS tokens and then lock or stake them depending on the user preference. We describe below the reviewed contracts and their main functionalities.

2.2.1 sdToken

In Version 1 this contract was named `sdFXSToken`. `sdToken` 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.

2.2.2 FxsLocker

In Version 1 this contract was named `FraxLocker`. `FxsLocker` is responsible for locking FXS tokens to `veFXS` (voting escrow FXS) contract, therefore FraxLocker should be whitelisted by the `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 `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, FraxLocker contract implements a special function `execute(to, value, data)` which allows the governance to call any address `to`, with any `msg.value` and arbitrary `data`.

2.2.3 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 4 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`: 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`.

The governance contract has full control and must be fully trusted. Contracts that have special permissions like the depositor or the accumulator contracts, are also trusted and need special care when being updated. We assume that contracts not in scope of this audit, such as `fxs`, `veFXS`, `yieldDistributor`, `gaugeController` behave correctly and do not act maliciously at any point.

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	0
Medium-Severity Findings	0
Low-Severity Findings	3

5.1 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.

5.2 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.

5.3 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:

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

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.

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 onlyGovernanceOrDepositor() {
        require(
                msg.sender == governance || msg.sender == fxsDepositor,
                "!(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.

6.8 Unused Imports: `FxsDepositor`

Design Low Version 1 Code Corrected

The file `FxsDepositor.sol` (Version 2 `Depositor` contract) has the following unused import:

```
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/utils/Address.sol";
import "@openzeppelin/contracts/utils/Context.sol";
```

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:

```
import "@openzeppelin/contracts/math/SafeMath.sol";
import "@openzeppelin/contracts/utils/Address.sol";
```

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:

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

Code corrected

The unused libraries listed above have been removed from the updated code.

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 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.

7.1 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.

```
uint256 private constant MAXTIME = 4 * 364 * 86400;
```

7.2 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 `0.8.11`.

7.3 `safeApprove` Usage

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.
 */
```