Spiral DAO Security Analysis

by Pessimistic

This report is public

January 30, 2023

Abstract

In this report, we consider the security of smart contracts of Spiral DAO project. Our task is to find and describe security issues in the smart contracts of the platform.

Disclaimer

The audit does not give any warranties on the security of the code. A single audit cannot be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts. Besides, a security audit is not investment advice.

Summary

In this report, we considered the security of Spiral DAO smart contracts. We described the audit process in the section below.

The audit showed one critical issue: Deleting a logic contract via delegatecall. The audit also revealed several issues of medium severity: Outdated address, Overpowered role, Tests issues, Incorrect setting of variable values, Operator skipped, Incorrect calculation of the reward, Incompatibility of pools, Incorrect reward calculation. Moreover, a lot of low-severity issues were found. At the moment we started the audit, the project was not ready for deployment. It was necessary to fix issues and write tests to check the entire logic of the project.

After the initial audit, the codebase was updated. In this update, the developers fixed most of the previously discovered issues and improved the code coverage of the tests. The issues that were not fixed do not affect the security of the codebase.

General recommendations

We recommend fixing the remaining issues and improving NatSpec coverage. We also recommend improving test cases and their code coverage. Also, consider analyzing code with linters and security tools.

Project overview

Project description

For the audit, we were provided with Spiral DAO project on a private GitHub repository, commit c77c22d3f1f727870dcaf7c2c96524ce2922d7e9.

The scope of the audit included everything.

The documentation for the project includes description and doc.

All 8 tests pass successfully. The overall code coverage is 31.49%.

The total LOC of audited sources is 2061.

Codebase update

After the initial audit, we were provided with a new commit eb9b3e526278dd51f3686ca3d6843dba6c4e07d8 on the private GitHub repository.

The scope included fixes of the issues, there was no additional functionality.

In this update, the developers fixed most of the initially discovered issues. The code coverage increased to 52.12%. All 8 tests pass.

Audit process

We started the audit on December 05, 2022 and finished on December 26, 2022.

We inspected the materials provided for the audit. Then, we contacted the developers for an introduction to the project. After a discussion, we performed preliminary research of the protocols that the project integrates with:

- Curve,
- Convex,
- Balancer,
- Aura.

In the process of work, we kept in touch with the developers and sent the found issues twice.

We manually analyzed all the contracts within the scope of the audit and checked their logic. Among other, we verified the following properties of the project:

- Whether using `delegatecall` in adapters is safe.
- Whether integrations with Aura and Convex protocol are correct.
- Whether processing UniswapV2 pairs is correct.
- Whether integration with Curve pools is correct.
- Whether MasterMind works correctly with tokens (FRAX, CRV, CVX, FXS, AURA, BAL).
- Whether the structures are used correctly in contracts.
- Whether depositing is performed using Boosters, not via pools in Convex and Aura.
- Whether the contracts perform ERC20 calls correctly.
- Whether slippage protection is implemented.

We scanned the project with the static analyzer Slither with our own set of rules and then manually verified all the occurrences found by the tool.

We ran tests and calculated the code coverage.

We combined in a private report all the verified issues we found during the manual audit or discovered by automated tool.

After the initial audit, we discussed the results with the developers. The developers provided us with an updated version of the code. In this update, they fixed some of the issues from our report and improved tests coverage. We did not manage to discover new issues in the update.

Finally, we updated the report.

Manual analysis

The contracts were completely manually analyzed, their logic was checked. Besides, the results of the automated analysis were manually verified. All the confirmed issues are described below.

Critical issues

Critical issues seriously endanger project security. They can lead to loss of funds or other catastrophic consequences. The contracts should not be deployed before these issues are fixed.

C01. Deleting a logic contract via delegatecall (fixed)

A malicious user can gain access to any method of **MasterMind** logic contract using the `init` function and a proper malicious contract. As a result, the user can, among others, delete the logic contract.

An attacker can delete the **MasterMind** contract with the following sequence:

1. For any pool, set its target address to a malicious contract using the `updateTargetInfo` function. The malicious contract should contain the `selfdestruct` call in its fallback function.
2. Call the `updatePool` function of the **MasterMind** contract and providing the given pool id as an argument. As a result, the **MasterMind** contract will perform a `delegatecall` to the malicious contract and trigger a `selfdesctruct` of the calling contract.

The issue has been fixed and is not present in the latest version of the code.

Medium severity issues

Medium issues can influence project operation in the current implementation. Bugs, loss of potential income, and other non-critical failures fall into this category, as well as potential problems related to incorrect system management. We highly recommend addressing them.

M01. Outdated address (fixed)

In the Aura project documentation, there is a different address for the `Booster` contract. For the address `0x7818A1DA7BD1E64c199029E86Ba244a9798eEE10` in the **AuraDelegate** contract, calling the function `isShutdown` returns the value `true`. We recommend using the current address `Booster`.

The issue has been fixed and is not present in the latest version of the code.

M02. Overpowered role (fixed)

In the **IFO** contract in the `finalWithdraw` function, the admin role can transfer all the tokens from the contract balance to an arbitrary address at any moment. In the current implementation, the system depends heavily on the admin role. Thus, there are scenarios that can lead to undesirable consequences for the project and its users, e.g., if admin's private keys become compromised. We recommend designing contracts in a trustless manner or implementing proper key management, e.g., setting up a multisig.

The issue has been fixed and is not present in the latest version of the code.

M03. Tests issues

The project has 8 tests. All these tests pass successfully. However, the code coverage is only 52.12%. Many critical scenarios remain uncovered.

Testing is crucial for project security, and the audit does not replace tests in any way. We highly recommend covering all the important scenarios with tests.

M04. Token inflation (addressed)

The **SpiralStaking** contract has `stake` and `unstake` functions for users to exchange COIL and SPR tokens. When users call these functions, the contract accepts the given amount of tokens from the user and sends a decreased amount of complementary tokens in exchange. The contract calculates the tokens amount to send back to a user using the `initialIndex` constant and the `index` variable that does not depend on the tokens balance.

The minting amount of COIL in the `rebase` function depends on the `totalSupply` value of SPR token. But for the SPR token, the `totalSupply` value can change due to interaction with other contracts. Thus, for the **SpiralStaking** contract, the amount of `Coil` tokens can be minted that is in no way related to user activity in this contract. Users from other projects influence the rewards of users of this project.

> *Comment from the developers: We understand that it is very important in principle for the project that there are no other possibilities to mint SPR tokens. Because SPR is exactly the share of single staking and the wrapper of COIL, like gOHM / OHM. And so far nothing more. We did not plan to create any issues or complications, especially related to minting SPR tokens in other places, without making significant changes to the staking and the connection between COIL and SPR.*

M05. Operator skipped (fixed)

In the **ConvexMainAdapter** contract, the `earnedReward` function lacks a `return` statement at line 340. As a result, it returns an incorrect value for the `index == 1` case.

The issue has been fixed and is not present in the latest version of the code.

M06. Incorrect calculation of the reward (fixed)

In the **Rewarder** contract, when updating the `rewardPerBlockPerOneEtherOfDeposit` parameter, it is necessary to call `updatePool` so that the reward for the past blocks is calculated with the old parameter value. At the moment, this is not taken into account in the `updatePoolRewards` and `massUpdatePoolRewards` functions.

The issue has been fixed and is not present in the latest version of the code.

M07. Incompatibility of pools

In the **ConvexMainAdapter** contract, the deposit on line 273 can be made not only in pools with two underlying tokens. For example, a call to `_zapAny` with a parameter pointing to `tbtc` Curve pool `0x64eda51d3Ad40D56b9dFc5554E06F94e1Dd786Fd` will stop reverting on line 273, because the pool interface is waiting for four tokens.

M08. Incorrect reward calculation (fixed)

The `index` parameter in the `earnedReward()` function of **ConvexMainAdapter** contract defines a token that the function calculates a reward for. So the function is supposed to return a reward amount for token `rewardToken(index)`, but in case `index` > 1 the `for` loop is executed and `index` is not taken into account. Consider replacing `address(rewardToken(i))` with `address(rewardToken(index))` on line 343.

The issue has been fixed and is not present in the latest version of the code.

Low severity issues

Low severity issues do not directly affect project operation. However, they might lead to various problems in future versions of the code. We recommend fixing them or explaining why the team has chosen a particular option.

L01. Using UniswapV2 to determine tokens exchange rate

The **ConvexMainAdapter** contract uses UniswapV2 protocol to determine the tokens exchange rate in the pool. This protocol is susceptible to a price manipulation via flash loans. Consider using other methods to assess an exchange rate, e.g., Chainlink oracles or TWAP.

L02. Unaccounted token

The `_zapAny` function of the **ConvexMainAdapter** contract does not verify that the provided `sellToken` address is whitelisted. Before calling, an attacker can create a `WETH` UniswapV2 pair with this token. The main problem is that there is no verification in the function that a `sellToken` is one of the acceptable tokens, listed in the **ConvexMainAdapter** contract. And for `depInfo`, real data is already being taken. The condition on line 173 will be met in case there is no suitable Curve pool. Next, the attacker can exchange `sellToken` for `WETH`, and then for `USDT` or `WBTC`. Before the exchange, the attacker can manipulate the liquidity of the pool and the price of the token to pump up the value of his token. It is necessary to limit the use of unaccounted tokens.

L03. CEI pattern violation (fixed)

The `distributeFee` function of the **Rewarder** contract violates the CEI pattern: it performs external calls at lines 154–155, but updates storage variables after this at lines 156–157. We highly recommend following CEI pattern to improve the code execution predictability. Another occurances of this issue are listed below:

- `userInfo[msg.sender].claimed` is set after external calls in the `harvest()` function of **IFO** contract.
- `epoch.endBlock` and `epoch.number` state variables are set after external calls in the `rebase()` function of **SpiralStaking** contract.

The issues have been fixed and are not present in the latest version of the code.

L04. Missing checks during the contract initialization

For the **IFO** contract to work correctly, it is necessary to transfer `offeringToken` tokens in the amount of `offeringAmount` to the contract balance in the `initialize` function. Or, in the initialization function, check the availability of tokens on the contract balance. The same condition should be for the `setOfferingAmount` function.

L05. Dependency management

1. Consider moving tools from `dependencies` section to `devDependencies` in **package.json**.
2. We recommend removing .lock files from **.gitignore** to ensure that dependecies versions match across local repositories.
3. Running tests requires `tenderly` dependency that is missing in **package.json** file.

L06. Extra storage interactions (fixed)

Since every reading from storage requires significant amount of gas, consider minimizing interactions with storage:

1. In the `updatePool` function of the **Rewarder** contract, consider replacing `poolInfo[poolId]` at line 142 with `pool`.
2. The `drain` function of the **MasterMind** contract reads the `pool.adapter` value from storage multiple times. Consider saving this value to a local variable.
3. The `addBulk` function of the **MasterMind** contract reads `_poolInfo().count` value multiple times. Consider saving it to a local variable.
4. The `_burnFrom()` function of the **COIL** and **Spiral** contracts reads `allowance(account_, msg.sender)` value from storage multiple times. Consider saving it to a local variable.

The issues have been fixed and are not present in the latest version of the code.

L07. Redundant code

The project contains the following redundant code:

1. **(fixed)** The `{ Ownable }` import at line 12 of the **MasterMind** contract is unused.
2. **(fixed)** The `FEE_BASE` constant declared at line 22 of the **MasterMind** contract is unused.
3. **(fixed)** The `averageDepositTime` property of the `UserInfo` in the **IMasterMind** interface is never used.
4. The code on lines 221-226 of the **ConvexSingleAdapter** contract does not affect the result of the function as the return value if defaulted to zero and an execution cannot be continued after this exact block of code.

L08. Duplicating code

The **MasterMind** contract contains duplicating code blocks that significantly decreases code maintainability. E.g., functions `_zapAndDeposit` and `_zapAndDepositRouted`, `withdraw` and `withdrawAndUnzapRouted` have the same logic with minor modifications. We recommend moving repeating logical blocks to separate internal functions.

L09. Magic values

The `_zapAny` function of the **ConvexMainAdapter** contract contains multiple hardcoded addresses without any comments. This makes the codebase extremely hard to read and maintain. Any accidental errors in any of these addresses can disrupt the system operation but are hard to notice. We recommend using constants with descriptive names instead.

L10. Incorrect value in the event (fixed)

The `withdraw` function of the **MasterMind** contract emits an event with the `rewardableAmount` value at line 372. However, this value can differ from an actual withdrawn amount.

The issue has been fixed and is not present in the latest version of the code.

L11. Incorrect initialization of a variable

In the **Rewarder** contract, the `poolInfo[poolId].rewardPerBlockPerOneEtherOfDeposit` value should change only for active pools. However, `updatePoolRewards` and `massUpdatePoolRewards` functions do not verify whether pools are active.

L12. Unnecessary calls

The code performs unnecessary external calls, which could be removed to optimize gas usage:

1. **(fixed)** The `earnedRewards` function of the **MasterMind** contract performs multiple `target.rewardTokenCount()` calls to retrieve a single value that does not change within the execution cycle. Consider saving this value to a local variable to optimize gas consumption.
2. `address(this.lockableToken(poolId))` expression on line 67 in the `_zapAny` function of the **ConvexSingleAdapter** contract always equals the `cvxCRV` constant address.
3. `address(this.lockableToken(poolId))` expression on line 119 in the `_zapAny` function of the **ConvexSingleAdapter** contract always equals the `CVX` constant address.

L13. Unchecked return value

The `deposit` function of the **ConvexMainDelegate** and **AuraDelegate** contracts does not check the returned value from the `Booster.deposit` call. If this is not done, the consistency of data between Spiral and Convex or Aura may be disrupted.

L14. No check of the array length (fixed)

The `massUpdatePoolRewards` function of the **Rewarder** contract iterates through the `poolIds` array but uses the same indices for the `newRewardsPerBlockPerEther` array. But there is no check that these arrays have equal length.

The issue has been fixed and is not present in the latest version of the code.

L15. No events for parameters changing

Consider emmiting proper events when changing important parameters of the system to make sure that users can get notifications and to keep track of the changes.
These issues are listed below:

- `updateDev, updateDao` functions of **Rewarder** contract.
- `changeLength, changeAPR, changeCooldown, changeGuard` functions of **SpiralStaking** contract.
- `setOfferingAmount, setRaisingAmount` functions of **IFO** contract.

L16. Extremely long functions

The `_zapAny` functions of the **ConvexMainAdapter** and **ConvexSingleAdapter** contracts occupy 128 and 96 lines respectively. This significantly complicates the codebase maintainability and often results in bugs and logical issues. We highly recommend refactoring the code and splitting the functions into simpler code blocks.

L17. Lack of necessary functionality

At any time, a situation can arise that the token will lose its value. For example, it is hacking the protocol with this token or changing the value of a stablecoin. In this case, it is advisable to prohibit users from working with this token. But the existing functionality does not allow you to do this.

L18. Immutable addresses for changing contracts

In the adapters **ConvexMainDelegate**, **ConvexMainAdapter**, **AuraAdapter**, **AuraDelegate**, the address of the `Booster` contracts is declared as a constant, but these contracts in `` `Convex Finance `` and `Aura Finance` are not updated, and their address may change. We recommend adding the ability to update these addresses in adapters.

L19. No pool status check

The `poolInfo` function of the `Booster` contract returns the status for the `shutdown` pool. When making a deposit, the `shutdown` status is not checked and the pool can be turned off. You also need to check whether the `Booster` itself is turned off. This can be checked by calling the `isShutdown` function in the `Booster` contract. Such checks can be done here:

1. In the contract **AuraDelegate** at lines 28-32.
2. In the contract **ConvexMainDelegate** at lines 32-36.

L20. Reinitializing a variable

In the **ConvexMainAdapter** contract constructor, the same mapping element with different values is initialized twice.

1. **(fixed)** For the variable `curveDeposit[0x5282a4eF67D9C33135340fB3289cc1711c13638C]` on lines 109 and 118.
2. For the variable `curvePair[0x8E870D67F660D95d5be530380D0eC0bd388289E1]` on lines 65 and 68.

L21. Execution of malicious code

In the **MasterMind** contract for the `_zapAndDepositRouted` function, an attacker can pass the address of his contract with the `transferFrom` function as the `token` parameter. But the implementation of this function may contain malicious code that will be executed on behalf of the **MasterMind** contract. It is recommended to limit the operation of the contract only with tokens from the white list.

L22. Variables should be marked immutable

Variables `Coil` and `Spiral` on lines 31,32 in **SpiralStaking** contract are set during contract deployment and never change later. We recommend declaring them `immutable` to reduce gas consumption.

This analysis was performed by Pessimistic:

Vladimir Pomogalov, Security Engineer
Ivan Gladkikh, Security Engineer
Nikita Kirillov, Security Engineer
Boris Nikashin, Analyst
Irina Vikhareva, Project Manager

January 30, 2023