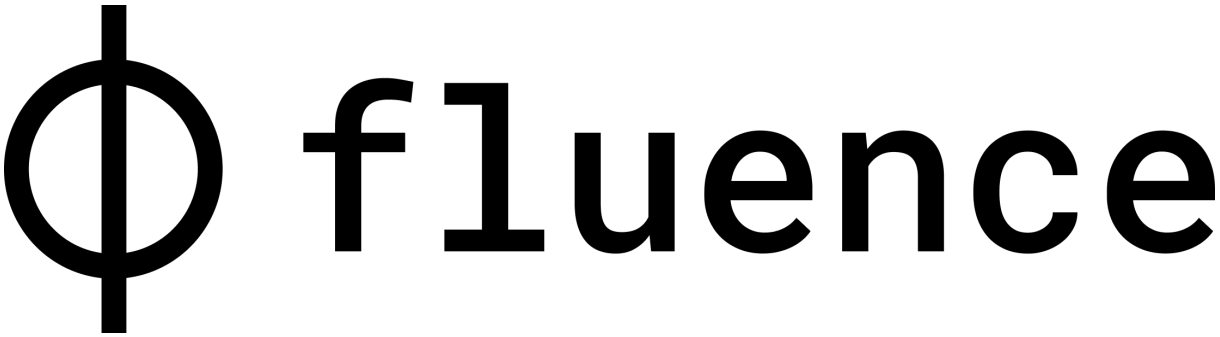

Fluence DAO Security Analysis

by Pessimistic

This report is public

February 26, 2024

Abstract

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

The initial audit showed several issues of medium severity: Discrepancies with the documentation, Multiplication after division during reward calculation, No guarantees for rewards, Incorrect integration, Project Roles, Possible attempt to deploy already existing pool. Also, several low-severity issues were found.

Following the initial audit, the developers provided us with a new version of the code. This update fixed several medium and all low-severity issues, with comments provided for some of them.

After the first recheck, the developers provided us with a new version of the code. The logic of developers' reward distribution was updated. That version introduced one new issue of critical severity Griefing receiving of rewards, one new issue of low severity ERC20 standard is not implemented correctly, and one new note Inconvenient unlock of developers' rewards.

The developers fixed C01 issue in a subsequent update. The exact commit hash can be found in the respective issue section.

The overall code quality of the project is good.

General recommendations

We recommend updating the documentation for the project.

Project overview

Project description

For the audit, we were provided with Fluence DAO project on a public GitHub repository, commit 535d8d93b686b4aeeffb1a81a67b7bab638f5a74.

The scope of the audit included everything excluding **contracts/contracts/dev** folder.

The documentation for the project included **README.md** and **DOC.md**.

All 45 tests pass successfully. The code coverage is 77.01%.

The total LOC of audited sources is 860.

Codebase update #1

After the audit, the developers supplied an update, commit e62fa3698ebc94852b84c758da7b82f64b268d1c. This update includes fixes and commentary for most of the issues, and enhances code coverage.

All 45 tests pass successfully. The code coverage is 96.79%.

Codebase update #2

After the audit, the developers supplied an updated version of the codebase with commit ce2c1b9e612b2a50cce8dee72b00df27afc883b5. Before the recheck started, developers updated us with a final commit 0bfc2087404c5fd849ef40a780bbfc54d5c0c609 that included some fixes for the previously provided version. This update changes the logic of developers' reward distribution. Along with the update, several new issues were introduced, including one issue of critical severity. The number of tests increased.

All 49 tests pass successfully. The code coverage is 96.12%.

After the update, the developers fixed the C01 issue in commit 555dc1f84df38ff30fa499acd3e29ece62962d2d.

Audit process

We started the audit on January 22, 2024 and finished on January 26, 2024.

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 and specified those parts of the code and logic that require additional attention during an audit:

- **LPController** cannot send tokens to the owner;
- Pools creation in **LPController** always succeed;
- All roles participating in the governance assigned properly;
- Remains of the tokens cannot be withdrawn before the end of distribution;
- Voting power for governance is calculated correctly.

During the work, we stayed in touch with the developers and discussed confusing or suspicious parts of the code.

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

- Whether the code corresponded to the documentation;
- Whether the code met best practices;
- Whether all the integrations with external contracts are correct.

We scanned the project with the following tools:

- Static analyzer Slither;
- Our plugin Slitherin with an extended set of rules;
- Semgrep rules for smart contracts. We also sent the results to the developers in the text file.

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 tools.

After the initial audit, we discussed the results with the developers. On January 31, 2024, the developers provided us with an updated version of the code. This update fixes and provides commentary for most of the issues identified in our report and enhances code coverage.

After reviewing the updated codebase and running the aforementioned tools, we updated the report accordingly.

For the next review, developers provided us with the updated version of the code on February 22, 2024. This update changes the logic of the **DevRewardDistributor** contract, not transferring rewarded tokens immediately but locking them for some time before the actual transfer.

This update introduced several new issues, including one issue of critical severity, one issue of low severity, and one note. The number of tests has increased.

We performed a manual review of the provided version, and updated the report accordingly.

After we provided the report, the developers fixed the C01 issue and provided comments on the L07 and N04 issues.

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. Griefing receiving of rewards (fixed)

The rewards for developers in **DevRewardDistributor** are available during the limited time identified by `claimingEndTime` constant. Moreover, the rewards are locked for `lockupPeriod` seconds. Considering that anyone can call `withdraw` function when the claiming phase is over, it is possible to transfer all the unclaimed rewards to the executor contract, even if the rewards are actually locked and the `lockupPeriod` has not passed yet.

Since there are no restrictions on the number of calls of the `withdraw` function, the malicious actor is able to grieve all attempts to receive tokens after the `claimingEndTime`.

The issue has been fixed in commit 555dc1f84df38ff30fa499acd3e29ece62962d2d.

Medium severity issues

Medium severity 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. Discrepancies with the documentation (fixed)

The codebase of the project is well-described with NatSpecs and provided documentation. However, there are several discrepancies between the code and the documentation in the project:

- It is said that **Vesting** contract implements `ERC20` standard. However, `transferFrom`, `approve`, and `allowance` methods of the standard were not implemented. Additionally, the `totalSupply` does not decrease when tokens are burned, and events are not emitted during the minting of the tokens in the constructor;
- **FluenceToken** uses inherited `18` decimals. However, the documentation mentions that the decimals must be equal to `10`;
- The documentation specifies that the DAO will determine whether to shift the liquidity to Uniswap V3 once the Balancer bootstrapping event finishes. However, the **LPController** does not enforce this;
- The schema in `README.md` says that the mint of the liquidity occurs just after the deployment of the **LPController.** This logic is not presented in the code.

We suggest addressing any inconsistencies in the documentation to enhance code clarity and reduce confusion.

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

M02. Multiplication after division during reward calculation

Multiplication after division can lead to losing some rewards during claiming. Because the **FLT** token has `10` decimals (according to the documentation), it is essential to perform multiplication before the division at lines 145–146 in `getAvailableAmount` function of **Vesting** contract.

The developers have opted for a decimals value of 18. Although this reduces the issue's impact, we still recommend performing multiplication before division.

M03. No guarantees for rewards (commented)

The **Vesting** and **DevRewardDistributor** contracts handle reward distribution. However, they currently do not make sure that the distributed amounts match the actual balances. It is suggested to include a transfer operation for pool rewards during the creation of the contracts.

> *Comment from the developers: After deployment, we will check manually that we have sent the required number of tokens.*

M04. Incorrect integration (fixed)

According to the Balancer documentation, it is not advised to use `queryExit` and `exitPool` at the same transaction as it makes the contract vulnerable to a sandwich attack. Despite the mitigation attempt at the end of the function, where swaps are disabled using the `setSwapEnabled`, it is strongly advised to adhere to best practices by excluding usage of these functions within the same transaction at lines 169–183 of **LPController** contract.

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

M05. Project Roles (commented)

There are several roles in the project that have privileged access to some parts of the code:

- After the exit from the **LBP**, the Balancer LP token is not transferred to the `daoExecutor`. Consequently, the ownership of the extracted liquidity rests within the contract, and the `owner` decides what to do with it;
- `Canceler` role in **DevRewardDistributor** contract can withdraw any remaining tokens before the end of the claiming period;
- In **Executor** contract the `owner` has `proposer` and `execution` roles. `Proposer` is assigned only to the `owner` during the contract initialization.

In the current implementation, the system heavily depends on privileged roles. Thus, there are scenarios that can lead to undesirable consequences for the project and its users, e.g., if owner's private keys become compromised.

> *Comment from the developers: All roles are configured in the deployment scripts. There, we transfer all roles to the Executor's address. Our multisig has only the right of veto.*

M06. Possible attempt to deploy already existing pool

The calls `uniswapFactory.createPool` and `balancerLBPFactory.create`, at lines 92 and 211 in **LPController** contract, will revert if the pools with such parameters already exist. The error will not be handled, leading to the revert of the whole transaction and the inability to provide liquidity using this contract.

The developers have implemented a check for the Uniswap pool.

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. Incorrect requirement check (fixed)

The check is intended to validate that the parameters passed to the function are correct. However, the condition will be met when at least one part is satisfied. Consider changing `||` to `&&` to ensure all the conditions are met at lines 80–85 in **LPController** contract.

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

L02. Misleading variable name (fixed)

In the **VestingWithVoting** contract `_cliffDuration` parameter is passed to the base constructor through the `_vestingDelay` variable. To improve code clarity, consider changing the variable's name at line 22 in `constructor` of **VestingWithVoting** contract.

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

L03. Mark event arguments as indexed (fixed)

Consider marking `userId`, `account` and `leaf` arguments in `Claimed` event as `indexed` at lines 70–75 in **DevRewardDistributor** contract.

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

L04. Not all base contracts are initialized (fixed)

The missing initialization of an inherited smart contract can affect its state and behavior. Consider calling `__ERC20Votes_init` and `__ERC20Permit_init` functions in the `constructor` of **FluenceToken** contract.

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

L05. Use safeTransferFrom (fixed)

The `transferFrom` function is used to perform token transfers from the sender to the contract at lines 111 and 114 of **LPController**. However, the result of the call needs to be validated. Consider using the `SafeERC20.safeTransferFrom` from OpenZeppelin library, as it supports reverting on fails for different types of tokens. Additionally, `approve` is used on line 152, but `forceApprove` is used throughout the code in other places.

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

L06. Multiple storage reading (fixed)

There are many places in **LPController** contract where storage variables are read multiple times. It is advisable to read them only once into a local variable to minimize gas consumptions.

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

L07. ERC20 standard is not implemented correctly (commented)

The **DevRewardDistributor** contract implements an interface that is very similar to the interface of `ERC20` token standard. However, `transferFrom`, `approve`, and `allowance` methods of the standard were not implemented. Moreover, the `to` parameter in the `transfer` function is not used.

> *Comment from the developers: We implemented only those methods and events that are needed to display **DevReward** as a token in wallets and scanners.*

Notes

N01. Different fee settings for pools (fixed)

The fee defined for the Uniswap V3 pool is constant. However, the `swapFeePercentage` for the Balancer pool is a parameter. Consider unifying the logic of the fee settings for both pools in **LPController** contract.

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

N02. No minimum amounts for Uniswap V3 liquidity minting (fixed)

When there is a logic for providing liquidity to an existing pool (created by someone else), the minimum amounts should not be set to zero. Leaving values unspecified may result in providing liquidity in unintended quantities, especially vulnerable to price manipulation before the minting process. For instance, liquidity might be supplied in only one token. Additionally, the contract balances could be changed by any transfers before the execution of this function, potentially impacting the `amountDesired` parameters in the subsequent `mint` call at lines 232–233 in `createUniswapLP` function of **LPController** contract.

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

N03. The order of the pool deployments not enforced (commented)

Currently, the **LPController** does not enforce any order of the function calls, which allows the withdrawal of assets at any point in time.

> *Comment from the developers: We have a certain level of trust in the owner of the **LPController**.*

N04. Inconvenient unlock of developers' rewards (addressed)

The `DevRewardDistributor.claimTokens` function ensures that no more than `maxClaimedSupply` tokens are claimed. If this limit is going to be surpassed, then the function reverts, and the caller should wait for the reward to be halved to actually claim it. However, after the `halvePeriod`, only one user will be able to collect their rewards, and the others should be waiting for the next halving to claim their rewards (and only one account would be able to actually do it during the single `halvePeriod`).

> *Comment from the developers: The number of tokens on the contract will be a multiple of `initDeposit`. This means that if the first epoch `totalClaimAmount` is equal to `maxClaimAmout`, the rest of the token amount is zero.*

This analysis was performed by Pessimistic:

Pavel Kondratenkov, Senior Security Engineer
Oleg Bobrov, Security Engineer
Egor Dergunov, Junior Security Engineer
Irina Vikhareva, Project Manager
Konstantin Zherebtsov, Business Development Lead
Alexander Seleznev, Founder

February 26, 2024