Smart Contract Security Audit Report

Fluence Labs DAO

1. Contents

2. General Information

This report contains information about the results of the security audit of the Fluence Labs (hereafter referred to as “Customer”) DAO smart contracts, conducted by Decurity in the period from 02/05/2024 to 02/12/2024.

2.1. Introduction

Tasks solved during the work are:

- Review the protocol design and the usage of 3rd party dependencies,
- Audit the contracts implementation,
- Develop the recommendations and suggestions to improve the security of the contracts.

2.2. Scope of Work

The audit scope included the contracts in the following repository: https://github.com/fluencelabs/dao.

Initial review was done for the commit 77c2de020b0b09ed5ffcb4e4f5cfaca69463bd02.

The retesting was done for the commit 2f9f1770d7da678f022d031fbb51716601e31c17.

2.3. Threat Model

The assessment presumes the actions of an intruder who might have the capabilities of any role (an external user, token owner, token service owner, or a contract).

The main possible threat actors are:

- User,
- Protocol owner,
- DAO,
- Third parties (Balancer, Uniswap).

2.4. Weakness Scoring

An expert evaluation scores the findings in this report, and the impact of each vulnerability is calculated based on its ease of exploitation (based on the industry practice and our experience) and severity (for the considered threats).

2.5. Disclaimer

Due to the intrinsic nature of the software and vulnerabilities and the changing threat landscape, it cannot be generally guaranteed that a certain security property of a program holds.

Therefore, this report is provided “as is” and is not a guarantee that the analyzed system does not contain any other security weaknesses or vulnerabilities. Furthermore, this report is not an endorsement of the Customer’s project, nor is it an investment advice.

That being said, Decurity exercises the best effort to perform its contractual obligations and follow the industry methodologies to discover as many weaknesses as possible and maximize the audit coverage using limited resources.

3. Summary

As a result of this work, we haven't discovered critical exploitable security issues but outlined suggestions included fixing the low-risk issues and some best practices.

3.1. Suggestions

The table below contains the discovered issues, their risk level, and their status as of February 20, 2024.

Table. Discovered weaknesses

Issue	Contract	Risk Level	Status
Funds may be lost during claim	contracts/contracts/DevRewardDistributor.sol	High	Fixed
Missing zero address check	contracts/contracts/LPController.sol contracts/contracts/DevRewardDistributor.sol	Low	Acknowledged
Potential front-running in LPController	contracts/contracts/LPController.sol	Low	Acknowledged
String error messages used instead of custom errors	contracts/contracts/DevRewardDistributor.sol contracts/contracts/Executor.sol contracts/contracts/Governor.sol contracts/contracts/LPController.sol contracts/contracts/Vesting.sol	Info	Acknowledged
Unspecific compiler version pragma	contracts/contracts/*.sol	Info	Acknowledged
Magic numbers	contracts/contracts/Vesting.sol	Info	Acknowledged
Division before multiplication	contracts/contracts/Vesting.sol	Info	Acknowledged

Issue	Contract	Risk Level	Status
Unnecessary import	contracts/contracts/Executor.sol	Info	Acknowledged
Unused variable	contracts/contracts/LPController.sol	Info	Acknowledged

4. General Recommendations

This section contains general recommendations on how to improve the overall security level.

The Findings section contains technical recommendations for each discovered issue.

4.1. Security Process Improvement

The following is a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level:

- Keep the whitepaper and documentation updated to make it consistent with the implementation and the intended use cases of the system,
- Perform regular audits for all the new contracts and updates,
- Ensure the secure off-chain storage and processing of the credentials (e.g. the privileged private keys),
- Launch a public bug bounty campaign for the contracts.

5. Findings

5.1. Missing zero address check

Risk Level: Low

Status: Not fixed

Contracts:

- contracts/contracts/LPController.sol
- contracts/contracts/DevRewardDistributor.sol

Description:

There is no address validation of the `daoExecutor` in LPController and `executor` in `DevRewardDistributor` which may be crucial because there're transfers to these addresses.

While there're some other address variables without similar checks (e.g. token, Balancer vault and factory, etc), in these particular cases this may go unnoticed until the funds are sent to a wrong address because they're not used for other purposes

Remediation:

Implement the checks to validate that the `daoExecutor` and `executor` addresses are non-zero.

5.2. Potential front-running in LPController

Risk Level: Low

Status: Not fixed

Contracts:

- contracts/contracts/LPController.sol

Location: Lines: 173.

Description:

An attacker can create a Uniswap pool with the same parameters as intended by the protocol owners which may lead to unexpected results.

The function `createUniswapLP` in the `LPController` contract calls `IUniswapNFTManager.createAndInitializePoolIfNecessary` to assign a Uniswap V3 pool. The function accepts the parameter `sqrtPriceX96` and passes it further to the position manager, however, if the pool has already been created, the actual value of `sqrtPriceX96` in the pool storage may be different.

`LPController` does not enforce that the passed `sqrtPriceX96` value is the same as in the pool's storage.

The potential impact of such discrepancy is not very high because the owner can specify the ticks and slippage parameter to avoid losses and error. However, the owner should be aware of such front-running possibility and should carefully construct parameters accordingly.

Remediation:

Cover the potential scenarios with tests (the current tests don't fully test the slippage protection), prepare to calculate the slippage parameters and ticks.

Additionally, consider enforcing the `sqrtPriceX96` check after the pool initialization, e.g.:

```
(uint160 _sqrtPriceX96,,,,,,) = uniswapPool.slot0();
require(_sqrtPriceX96 == sqrtPriceX96, "sqrtPriceX96 mismatch");
```

5.3. Division before multiplication

Risk Level: Info

Status: Not fixed

Contracts:

- contracts/contracts/Vesting.sol

Location: Lines: `152-153`. Function: `getAvailableAmount`.

Description:

The calculation of `amount` first performs division before multiplication which could lead to a loss of precision due to integer truncation. This could result in fewer tokens being allocated than expected.

```
contracts/contracts/Vesting.sol:
  152:             uint256 amountBySec = locked / totalTime;
  153:             amount = past * amountBySec - released;
```

However, there's no real impact in this particular case, the error in vesting duration can only reach 1 second.

Remediation:

To avoid loss of precision, the multiplication should be performed before the division. This can be achieved by modifying the calculation of `amount` to `amount = ((locked * past) / totalTime) - released;`.

5.4. String error messages used instead of custom errors

Risk Level: Info

Status: Not fixed

Contracts:

- contracts/contracts/DevRewardDistributor.sol
- contracts/contracts/Executor.sol
- contracts/contracts/Governor.sol
- contracts/contracts/LPController.sol
- contracts/contracts/Vesting.sol

Description:

The contracts make use of the `require()` to emit an error. While this is a perfectly valid way to handle errors in Solidity, it is not always the most efficient.

```
contracts/contracts/DevRewardDistributor.sol:
  121:         require(isClaimingActive() == isActive, "Claiming status is not
as expected");
  139:         require(!isClaimed(userId), "Tokens already claimed");
  143:         require(MerkleProof.verify(merkleProof, merkleRoot, leaf),
"Valid proof required");
  148:         require(signer == temporaryAddress, "Invalid signature");
  172:         require(msg.sender == canceler, "Only canceler can withdraw");

contracts/contracts/Executor.sol:
  36:         require(

contracts/contracts/Governor.sol:
  75:         require(
```

```
contracts/contracts/LPController.sol:
   49:         require(token0_ < token1_, "Token0 must be less than token1");
   71:         require(address(liquidityBootstrappingPool) == address(0), "LBP
already exists");
   73:         require(
  123:         require(bptBalance > 0, "No BPT balance");
  163:         require(address(uniswapPool) == address(0), "Uniswap pool
already exists");
  165:         require(token0Amount != 0, "Invalid amount of token0");
  166:         require(token1Amount != 0, "Invalid amount of token1");
  201:         require(token.balanceOf(address(this)) != 0, "Invalid
balance");

contracts/contracts/Vesting.sol:
   81:         require(accounts.length == amounts.length, "accounts and
amounts must have the same length");
   83:         require(bytes(name_).length <= 31, "invalid name length");
   84:         require(bytes(symbol_).length <= 31, "invalid symbol length");
  182:         require(to == address(0x00), "Transfer allowed only to the zero
address");
  210:         require(releaseAmount > 0, "Not enough the release amount");
  213:             require(amount <= releaseAmount, "Not enough the release
amount");
```

Remediation:

Consider using custom errors as they are more gas efficient while allowing developers to describe the error in detail using NatSpec.

References:

- https://blog.soliditylang.org/2021/04/21/custom-errors/

5.5. Unspecific compiler version pragma

Risk Level: Info

Status: Not fixed

Contracts:

- contracts/contracts/*.sol

Description:

The contracts use the following pragma which is not specific:

```
pragma solidity >=0.8.15;
```

While floating pragmas make sense for libraries to allow them to be included with multiple different versions of applications, it may be a security risk for application implementations.

A known vulnerable compiler version may accidentally be selected or security tools might fall-back to an older compiler version ending up checking a different EVM compilation that is ultimately deployed on the blockchain.

It is recommended to avoid floating pragmas for non-library contracts and pin to a concrete compiler version.

Remediation:

Consider specifying a concrete Solidity version.

5.6. Magic numbers

Risk Level: Info

Status: Not fixed

Contracts:

- contracts/contracts/Vesting.sol

Description:

The contract uses magic numbers, 31 in this case, for the maximum length of the `name_` and `symbol_` fields.

Although during this review it was clear why these numbers exist in the code, generally magic numbers can lead to errors as they are not self-explanatory or easily adjustable.

```
contracts/contracts/Vesting.sol:
  83:         require(bytes(name_).length <= 31, "invalid name length");
  84:         require(bytes(symbol_).length <= 31, "invalid symbol length");
```

In the `DevRewardDistributor.sol` contract, the magic numbers 20, 256 are used, which may result in confusion and potential errors. They represent the length of the signed message and the bit index respectively, but could benefit from being defined as constants for clarity.

```
contracts/contracts/DevRewardDistributor.sol:
    145:         bytes32 msgHash = keccak256(abi.encodePacked("\x19Ethereum
Signed Message:\n20", msg.sender));
  183:         uint256 claimedWordIndex = index / 256;
```

```
  184:         uint256 claimedBitIndex = index % 256;
  221:         uint256 claimedWordIndex = index / 256;
  222:         uint256 claimedBitIndex = index % 256;
```

Remediation:

Consider using constants instead of raw numeric values.

5.7. Unnecessary import

Risk Level: Info

Status: Not fixed

Contracts:

- contracts/contracts/Executor.sol

Location: Lines: 7.

Description:

The following import is unused:

```
contracts/contracts/Executor.sol:
  7: import "./Governor.sol";
```

Remediation:

Remove unnecessary imports.

5.8. Unused variable

Risk Level: Info

Status: Not fixed

Contracts:

- contracts/contracts/LPController.sol

Location: Lines: 22.

Description:

The following public variable is set in the constructor and never used in the code or tests:

```
contracts/contracts/LPController.sol:
  22:     IUniswapV3Factory public immutable uniswapFactory;
```

This could be a dead code bloating the contract size.

Remediation:

Review if the variables not used in the contract are needed. Remove them if they are not necessary.

6. Appendix

6.1. About us

The Decurity team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we've gained expertise in the blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.