AUDITONE

Audit Report

Hedgey.

Vesting Locks

05.06.2024

Table of Contents

CONTENTS

INTRODUCTION

Smart Contract Security Analysis Report

Note: This report may contain sensitive information on potential vulnerabilities and exploitation methods. This must be referred internally and should be only made available to the public after issues are resolved (to be confirmed prior by the client and AuditOne).

INTRODUCTION

Defsec, Minhquanym and X0f-unit, who are auditors at AuditOne, successfully audited the smart contracts (as indicated below) of Hedgey Finance. The audit has been performed using manual analysis. This report presents all the findings regarding the audit performed on the customer's smart contracts. The report outlines how potential security risks are evaluated. Recommendations on quality assurance and security standards are provided in the report.

01-PROJECT DESCRIPTION

Hedgey is a platform that helps decentralized autonomous organizations (DAOs) and onchain organizations distribute tokens securely and efficiently. It combines token streams, periodic release schedules, and administrative controls, such as revocability and optional governance rights, to automate token distribution to team members, contributors, investors, and the community.

The platform has been used by notable teams like Arbitrum DAO, Celo, Gitcoin, Gnosis, Shapeshift, Index Coop, and Collabland to streamline their token management processes. Hedgey's tools are available as free public goods on a self-service basis, optimized for various issuers and recipients.

Hedgey's core products include token vesting, investor lockups, and token claims. Token vesting plans feature flexible schedules, cliffs, and backdated start dates, and are fully onchain and revocable. Investor lockups allow tokens to be distributed to investors on predefined schedules, with options for transferability and voting rights. The token claims product supports large-scale token distributions, enabling users to review, analyze, and claim their tokens efficiently.

Built on robust smart contracts, Hedgey's platform ensures secure and trustless interactions, providing essential tools for onchain teams to manage their digital assets effectively and support their growth.

02-Project and Audit Information

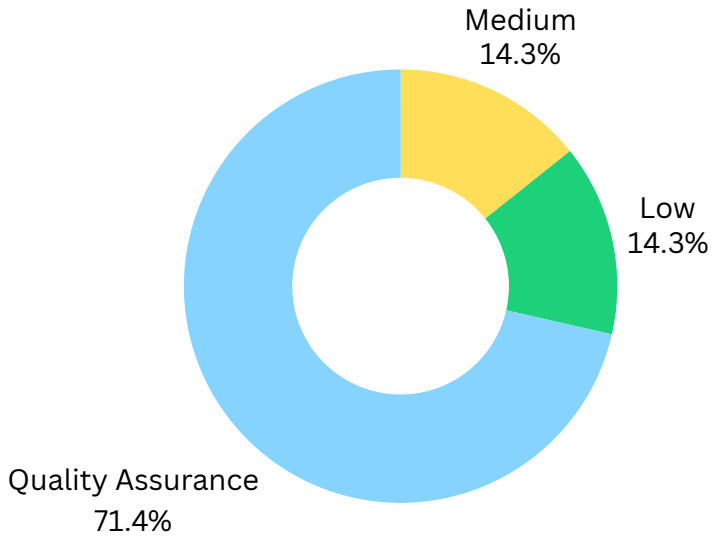
Term	Description
Auditor	Defec, Minhquanym and X0f-unit
Reviewed by	Luis Buendia and Gracious Igwe
Type	Finance
Language	Solidity
Ecosystem	EVM Compatible
Methods	Manual Review
Repository	https://github.com/hedgey-finance/VestingLockups/
Commit hash (at audit start)	972c273f9aaf81f88e43a87d60081f041ce9df52
Commit hash (after resolution)	02027f964df77deed2e85462d175c127b9506781
Documentation	NA
Unit Testing	NA
Website	https://hedgey.finance/
Submission date	20/05/2024
Finishing date	03/06/2024

03-Contracts in Scope

- ERC721Delegate/ERC721Delegate.sol
- ERC721Delegate/PlanDelegator.sol
- libraries/TransferHelper.sol
- libraries/UnlockLibrary.sol
- periphery/BatchCreator.sol
- periphery/VotingVault.sol
- TokenVestingLock.sol

04-Executive summary

Hedgey Finance smart contracts were audited between 20-05-2024 and 03-06-2024 by Defec, Minhquanym and X0f-unit. Manual analysis was carried out on the code base provided by the client. The following findings were reported to the client. For more details, refer to the findings section of the report.



Issue Category	Issues Found	Resolved	Acknowledged
High	0	0	0
Medium	2	2	0
Low	2	2	0
Quality Assurance	10	7	3

05-Severity Definitions

Risk factor matrix	Low	Medium	High
Occasional	L	M	H
Probable	L	M	H
Frequent	M	H	H

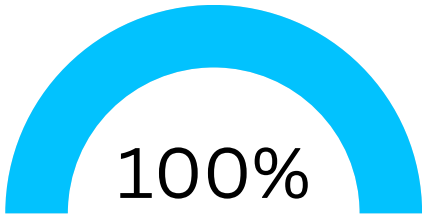
High: Funds or control of the contracts might be compromised directly. Data could be manipulated. We recommend fixing high issues with priority as they can lead to severe losses.

Medium: The impact of medium issues is less critical than high, but still probable with considerable damage. The protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions.

Low: Low issues impose a small risk on the project. Although the impact is not estimated to be significant, we recommend fixing them on a long-term horizon. Assets are not at risk: state handling, function incorrect as to spec, issues with comments.

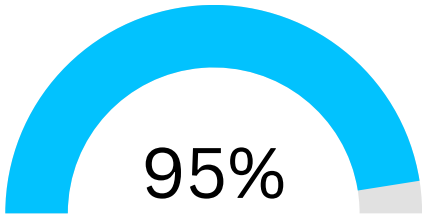
Quality Assurance: Informational and Optimization - Depending on the chain, performance issues can lead to slower execution or higher gas fees. For example, code style, clarity, syntax, versioning, off-chain monitoring (events etc.)

06-Audit Overview



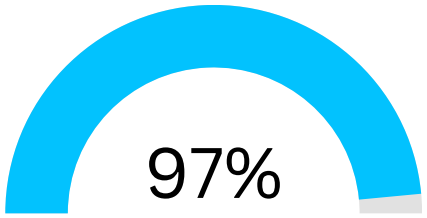
Security score

Security score is a numerical value generated based on the vulnerabilities in smart contracts. The score indicates the contract's security level and a higher score implies a lower risk of vulnerability.



Code quality

Code quality refers to adherence to standard practices, guidelines, and conventions when writing computer code. A high-quality codebase is easy to understand, maintain, and extend, while a low-quality codebase is hard to read and modify.



Documentation quality

Documentation quality refers to the accuracy, completeness, and clarity of the documentation accompanying the code. High-quality documentation helps auditors to understand business logic in code well, while low-quality documentation can lead to confusion and mistakes.

07-Findings

Finding: #1

Issue: _approvedDelegators[planId] isn't deleted when an ERC721 token is transferred

Severity: Medium

Where: contracts/ERC721Delegate/PlanDelegator.sol#L71

Impact: When the Plan NFT is transferred, the approved delegator of a plan isn't reset or removed. This allows the previous delegator to act on behalf of the new owner.

Description: In the **PlanDelegator** of the **VestingLockups** repository, the NFT owner can approve a delegator to act on their behalf through the function **_approveDelegator()**. This function sets **_approvedDelegators[planId] = delegator**, granting the **delegator** the authority to call some functions for plan **planId**.

```
function _approveDelegator(address delegator, uint256 planId) internal virtual {
  _approvedDelegators[planId] = delegator; // @audit not deleted when token is transferred
  emit DelegatorApproved(planId, ownerOf(planId), delegator);
}
```

The **PlanDelegator** contract of the **Locked_VestingTokenPlans** repository has the same mapping for the same purpose. In this contract, **_approvedDelegators[planId]** is removed or reset when the NFT is transferred.

```
function _beforeTokenTransfer(
  address from,
  address to,
  uint256 firstTokenId,
  uint256 batchSize
) internal virtual override {
  super._beforeTokenTransfer(from, to, firstTokenId, batchSize);
  delete _approvedDelegators[firstTokenId];
}
```

However, this behavior isn't present in the **PlanDelegator** of the **VestingLockups** repository. The approved delegator of a plan isn't reset or removed when the Plan NFT is transferred. This allows the previous delegator to act on behalf of the new owner.

Recommendations: When the ERC721 token is transferred, delete or reset the **_approvedDelegators[planId]** through the **_update()** function.

Status: Resolved

Finding: #2

Issue: BatchCreator and **DelegatedClaimCampaigns** are partially incompatible with Bound Lockup Plans

Severity: Medium

Where: BatchCreator.createLockupPlansWithDelegation(), and **DelegatedClaimCampaigns._claimLockedAndDelegate()** functions.

Impact: The **BatchCreator** and **DelegatedClaimCampaigns** functions that include delegation after creation or claim cannot be used with Bound Lockup Plans. Since these tokens cannot be transferred, an incompatibility of this function arises, preventing non-transferable lockups from being created using a CSV upload, which is an undocumented incompatibility.

Description: The **createLockupPlansWithDelegation()** function is not compatible with **TokenLockupPlans_Bound** and **VotingTokenLockupPlans_Bound** plans. This is because the **createLockupPlansWithDeletation()** function mints the plan tokens to the BatchCreator contract, performs the requested delegations, and then transfers the token to the legitimate owner.
As expected, both of these plans will revert when they are transferred.

Recommendations: If full compatibility is desired, it is recommended to implement an special case in **_beforeTokenTransfer()** hook to allow the token transfer if the **msg.sender** or the owner of the token is **BatchCreator** or **DelegatedClaimCampaigns** contract.

Status: Resolved.

Finding: #3

Issue: Lack of two-step ownership transfer pattern.

Severity: Low

Where: TokenVestingLock.changeManager()

Impact: Since the role is transferred in a single step, any fail when calling **changeManager()** (i.e. because of a typo in the address) might render these functions unusable.

Description: The **changeManager()** function directly transfers the **manager** role to any selected user. In the case of any error while calling **changeManager()**, any function using the **onlyManager** modifier could remain locked.

Recommendations: Implement a two-step role transfer in which **changeManager()** upgrades a **proposedManager** variable, and the transfer is not fully completed until **proposedManager** performs a call to **claimManager().**

Status: Resolved.

Finding: #4

Issue: Incompatibility with Fee-On-Transfer tokens

Severity: Low

Where: TransferHelper.sol:L28,44

Impact: The plans would be locked if any token being used in a plan enables this feature.

Description: The **transferTokens()** and **withdrawTokens()** functions in **TransferHelper** contracts include a require statement that ensures that the total **amount** balance transferred has been received, reverting if a Fee-On-Transfer token is used or if the amount received is lower than the expected due to any reason.

This is behavior is described in the comments in **TransferHelper** contract from VestingLockups repository, but not in the one found in Locked_VestingTokenPlans.

However, certain tokens, such as USDT or USDC, already include Fee-On-Transfer mechanisms, only that they are currently set to 0. If any of these tokens enabled the fee-on-transfer functionality, any existing plan using these tokens would remain locked in the contract since the require statement of line 44 would always revert.

Recommendations: It is recommended to implement an emergency withdraw function in case tokens remain locked in the contract, similar to the emergency admin transfer feature from **TokenVestingPlans** contract.

Status: Resolved.

Finding: #5

Issue: Missing ETH Wrapping and Unwrapping Functionality in **TransferHelper** Library.

Severity: QA

Where: TransferHelper.sol#L8

Impact: The library fails to meet the expectations set by the specification, which states that it should handle ETH wrapping and unwrapping. This inconsistency can cause confusion and misunderstandings for developers relying on the library.

Description: The provided TransferHelper library includes functions for safely transferring ERC20 tokens using the transferTokens and withdrawTokens functions. However, the library lacks functionality for wrapping and unwrapping ETH (Ether) to and from WETH (Wrapped Ether), despite the specification mentioning the library's intention to handle ETH wrapping and unwrapping.

```
/// @notice Library to help safely transfer tokens and handle ETH wrapping and unwrapping of WETH
library TransferHelper {
  using SafeERC20 for IERC20;
```

Recommendations: Consider fixing the spec on the library.

Status: Resolved.

Finding: #6

Issue: Contract whitelist cannot be modified

Severity: QA

Where: `BatchCreator.initWhiteList()``

Impact: If the whitelist has to be modified (adding any new contract or removing any due to a logic flaw identified, for example), a whole new set of contracts must be deployed, which uses more gas.

Description: The **BatchCreator** contract implements a whitelist, including all of the authorized plan contracts that will be used. However, the function initializing the whitelist, **initWhiteList(),** can be called only once since it will delete the **_manager** address at the end of the execution.

Recommendations: Including a function to modify the whitelist if needed will be cheaper than deploying the contract again.

Status: Resolved

Finding: #7

Issue: Commented-out code

Severity: QA

Where: **UnlockLibrary**:L85

Impact: Including commented-out lines in the final code can lead to confusion, increase the codebase's size, and make it harder to maintain and debug. In addition, issues like commented-out lines, typos, and inconsistent formatting make the code look sloppy and unfinished.

Description: A line of commented-out code was detected in **UnlockLibrary**:L85: **uint256 availablePeriods = availableAmount / rate;.**

Recommendations: Remove any commented-out lines present in the code.

Status: Resolved.

Finding: #8

Issue: Unused function

Severity: QA

Where: UnlockLibrary.min() function from VestingLockups report.

Impact: Unused functions reduce code readability and increase gas usage.

Description: The **min()** function included in the **UnlockLibrary** contract determines the minimum value between input parameters **a** and b. The **UnlockLibrary** contract probably is derived from the **TimelockLibrary** contract from the Locked_VestingTokenPlans repository, in which **min()** is used, but in the case of **UnlockLibrary** is not. Since it is an internal function, there is no reason to keep it.

Recommendations: Remove any unused functions, variables or libraries from the code.

Status: Unresolved.

Finding: #9

Issue: Use of **public** instead of **external** visibility.

Severity: QA

Where:

- VestingStorage.planEnd()
- TokenVestingLock.currentTokenId()
- TokenVestingLock.getVestingLock()
- TokenVestingLock.getLockBalance()
- TokenVestingLock.currentTokenId()

Impact: Using public visibility in functions that are not being called within the same contract increases the gas usage.

Description: Some functions are using **public** visibility instead of **external** and they are not being called within the same contract.

Recommendations: Use **external** visibility if functions are not being called within the same contract.

Status: Resolved.

Finding: #10

Issue: Incomplete NATSPEC

Severity: QA

Where: Every scoped contract.

Impact: Incomplete NATSPEC reduces code readability and can affect UX.

Description: It has been detected that some functions (including public/external functions) are lacking of a proper NATSPEC documentation.

Recommendations: It is recommended to have a complete and detailed NATSPEC documentation, even for non-public functions, since it increases code readability.

Status: Unresolved.

Finding: #11

Issue: Revert Strings increase gas usage

Severity: QA

Where: Every revert string used in the scoped contracts.

Impact: The gas usage is increased and can be reduced.

Description: Since Solidity 0.8.4, custom errors can be used. Each custom error saves around 50 gas every time they are hit, since they avoid the need for allocating and storing the revert string.

Recommendations: Use custom errors instead of revert strings to save gas.

Status: Acknowledged.

Finding: #12

Issue: Redundant check can be removed to reduce gas usage

Severity: QA

Where: **PlanDelegator.sol**:L48

Impact: The gas usage is increased due to a redundant check in **approveSpenderDelegator()** function.

Description: The **approveSpenderDelegator()** function from **PlanDelegator** contract includes the following function call: **_approve(spender, planId, msg.sender);**

This _approve implementation includes the "**auth**" parameter, which if different from zero, will check that "**auth**" is indeed the owner or approved to operate on all tokens held by the owner. In this case, **msg.sender** is sent as **auth**. However, this check is already being performed in the lines above, so it's redundant:

```
function approveSpenderDelegator(address spender, uint256 planId) public virtual {
  address owner = ownerOf(planId);
  require(
    msg.sender == owner || (isApprovedForAllDelegation(owner, msg.sender) && isApprovedForAll(owner, msg.sender)
    '!ownerOperator'
  );
  require(spender != msg.sender, '!self approval');
  _approveDelegator(spender, planId);
  _approve(spender, planId, msg.sender);
}
```

Recommendations: Since the relevant ownership or approval checks have already been performed, **auth** can be set to **0** in the **_approve()** call to save some gas.

Status: Resolved.

Finding: #13

Issue: For loops gas optimizations

Severity: QA

Where: Every **for** loop in the scoped contracts.

Impact: By using a non-locked pragma version, contracts could be accidentally deployed using a different pragma, which could introduce bugs or different behaviors that could negatively affect the contracts.

Description: Multiple gas optimizations have been found in the **for** loop used in the scoped contracts:

1. uint256 index **i** is being checked while incremented: It is not necessary to use a safe increment on **i** if a **uint256** variable is used since the function will run out of gas way before overflowing.
2. Postfix operators (**i++**) are being used instead of prefix operators (**++i**).
3. The array length is read in every iteration instead of being cached outside the loop.
4. **i** is initialized to **0**.

Recommendations:

1. Increment **i** using **unchecked: unchecked {++i}**
2. Use prefix operators instead of postfix operators.
3. Cache the array length in a variable outside the loop as long as the size will not change during the loop.
4. Do not initialize the value of **i**, since **0** (or **false**) are the default values in Solidity.

Status: Resolved.

Finding: #14

Issue: Unlocked pragma

Severity: QA

Where: VestingLockups repository

Impact: By using a non-locked pragma version, contracts could be accidentally deployed using a different pragma, which could introduce bugs or different behaviors that could negatively affect the contracts.

Description: It has been detected that contracts in **VestingLockups** repository are using an unlocked compiler pragma (**pragma solidity ^0.8.20**).

Recommendations: Use locked pragma versions, matching the version used during development and testing.

Status: Resolved.

08 - Disclaimer

The smart contracts provided to AuditOne have been analyzed by the best industry practices at the date of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions). The ethical nature of the project is not guaranteed by a technical audit of the smart contract. Any owner-controlled functions should be carried out by the responsible owner. Before participating in the project, all investors/users are recommended to conduct due research.

The focus of our assessment was limited to the code parts associated with the items defined in the scope. We draw attention to the fact that due to inherent limitations in any software development process and 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 cannot be free from any errors or failures. These preconditions can impact 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 correctly executing the included third-party technology stack itself. Report readers should also consider that over the life cycle of any software product, changes to the product itself or the environment in which it is operated can have an impact leading to operational behaviors other than initially determined in the business specification.

Contact

auditone.io

@auditone_team

hello@auditone.io

A trust layer of our
multi-stakeholder world.