AUDITONE

Audit Report

Hedgey.

Delegated Token claims

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/DelegatedTokenClaims/
Commit hash (at audit start)	4446cb36cc45345f979135051206bbe4fdf36c13
Commit hash (after resolution)	674c22ce0c270fbc276f6603d11303b3572a8a16
Documentation	NA
Unit Testing	NA
Website	https://hedgey.finance/
Submission date	20/05/2024
Finishing date	03/06/2024

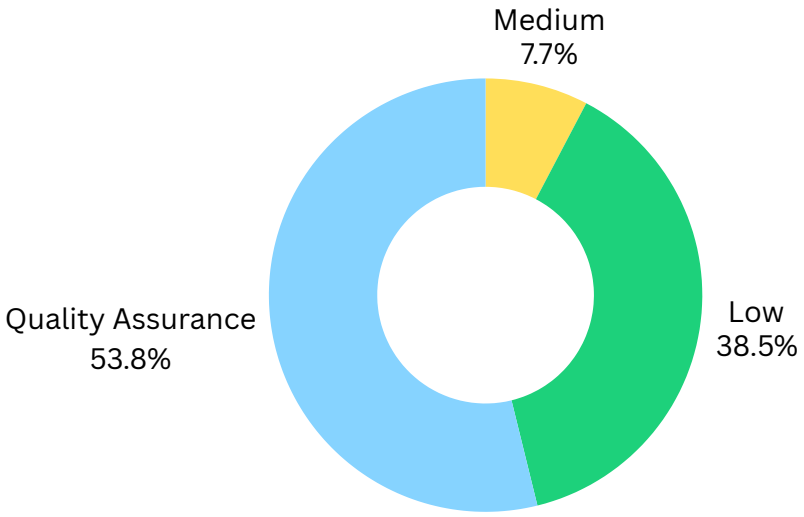
03-Contracts in Scope

- DelegatedClaimCampaigns.sol
- libraries/TransferHelper.sol

After the audit, the Hedgy finance team included a change from a finding not identified during the audit time. This change is not included on the audit scope. The change can be observed on the commit hash provided on the project audit information.

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	1	1	0
Low	5	1	4
Quality Assurance	7	2	5

05-Severity Definitions

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

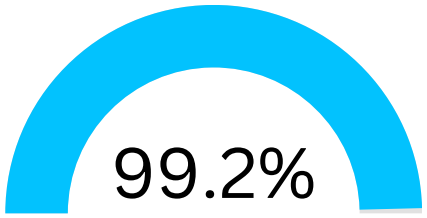
High: Funds or control of the contracts might be compromised directly. Data could be manipulated. We recommend fixing high issues with priority as they can lead to severe losses.

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.

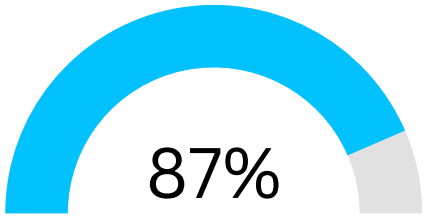
Quality Assurance: Informational and Optimization - Depending on the chain, performance issues can lead to slower execution or higher gas fees. For example, code style, clarity, syntax, versioning, off-chain monitoring (events etc.)

06-Audit Overview



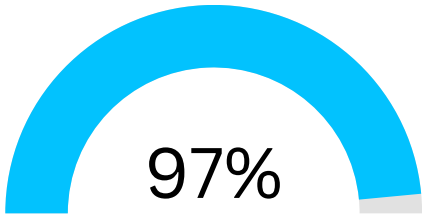
Security score

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: An attacker could use the **delegationSignature** to delegate directly, causing a Denial of Service (DOS) for users attempting to claim

Severity: Medium

Where: contracts/DelegatedClaimCampaigns.sol#L338
contracts/DelegatedClaimCampaigns.sol#L376

Impact: Users will not be able to claim a campaign that requires them to delegate token.

Description: When a campaign is created, users can specify the **delegating** parameter, which is a boolean that determines whether claims need to be delegated when claimed. If **delegating == true**, users will not be able to claim without delegating their tokens.

When users call **claimAndDelegate()** or **claimAndDelegateWithSig()** to claim an Unlocked campaign, they need to provide the **delegationSignature** signed by the users to perform the delegation.

However, an attacker could use this **delegationSignature** and call the **ERC20Votes.delegateBySig()** function directly. This action will render the **delegationSignature** invalid and unusable again. As a result, the user's transaction to claim and delegate will revert because the **delegationSignature** has been used.

Recommendations: First, check if the **delegates()** are pointed to the correct address. If they are, there's no need to call **delegateBySig()**.

Status: Resolved.

Finding: #2

Issue: Rounding Up Allows Users to Claim Faster Than Expected.

Severity: Low

Where: contracts/DelegatedClaimCampaigns.sol#L592
contracts/DelegatedClaimCampaigns.sol#L521

Impact: The **rate** rounds up when claiming locked tokens in **DelegatedClaimCampaigns**. This could lead to a faster unlock schedule than anticipated.

Description: The contract uses the **claimAmount** and the number of periods **c.periods** to calculate the rate of the plan when claiming locked tokens in **DelegatedClaimCampaigns**.

```
uint256 rate;
if (claimAmount % c.periods == 0) {
  rate = claimAmount / c.periods;
} else {
  rate = claimAmount / c.periods + 1; // @audit rounding up
}
```

If **claimAmount** is not divisible by **c.periods**, the **rate** value will round up. This could result in a faster unlock schedule.

Consider the following scenario:

1. The admin creates a campaign with **totalAmount = 1000 X** and **c.periods = 200**. The overall expected rate the admin wants is **rate = 1000 / 200 = 5.**
2. The campaign has two recipients. Alice will receive **499 X** and Bob will receive **501 X.**
3. When Alice and Bob claim, the calculation will be:

```
Alice_rate = 499 / 200 + 1 = 3
Bob_rate = 501 / 200 + 1 = 3
```

In the end, the sum of Alice and Bob's rates would be **6**, which means the users are claiming faster than the admin intended (the admin wanted a total rate of 5).

Recommendations: Round down when calculating the rate.

Status: Acknowledged.

Finding: #3

Issue: claimMultipleWithSig() does not incorporate all claims into the signed hash

Severity: Low

Where: contracts/DelegatedClaimCampaigns.sol#L295

Impact: The caller can claim different campaigns than the user intended.

Description: The **DelegatedClaimCampaigns** contract allows users to sign a signature that permits anyone possessing the signature to claim on their behalf. This signed claim also supports claiming multiple campaigns in a single transaction through the **claimMultipleWithSig()** function.

However, as we see in the following code snippet, the signed hash only includes **campaignIds[0]** and **claimAmounts[0]**. Therefore, if a user wants to claim, for instance, four campaigns A, B, C, D, the caller could use the signature to claim A, X, Y, Z instead.

```
function claimMultipleWithSig(
  bytes16[] calldata campaignIds,
  bytes32[][] calldata proofs,
  address claimer,
  uint256[] calldata claimAmounts,
  SignatureParams memory claimSignature
) external nonReentrant {
  require(campaignIds.length == proofs.length, 'length mismatch');
  require(campaignIds.length == claimAmounts.length, 'length mismatch');
  require(claimSignature.expiry > block.timestamp, 'claim expired');
  address signer = ECDSA.recover(
    _hashTypedDataV4(
      keccak256(
        abi.encode(
          MULITCLAIM_TYPEHASH,
          campaignIds[0],
          claimer,
          claimAmounts[0],
          claimSignature.nonce,
          claimSignature.expiry,
          campaignIds.length
        )
      )
    ),
    claimSignature.v,
    claimSignature.r,
    claimSignature.s
  );
  require(signer == claimer, 'invalid claim signature');
```

Recommendations: Incorporate all data into the signed hash.

Status: Acknowledged.

Finding: #4

Issue: Functions **createUnlockedCampaign()** and **createLockedCampaign()** are vulnerable to DoS attacks by front-running to create spam campaigns with the same **id.**

Severity: Low

Where: contracts/DelegatedClaimCampaigns.sol#L145 contracts/DelegatedClaimCampaigns.sol#L174

Impact: Users may be unable to create any campaign in **DelegatedClaimCampaigns**.

Description: The **createUnlockedCampaign()** and **createLockedCampaign()** functions allow anyone to create new campaigns. Users must provide an **id**, which is the UUID or CID of the file storing the Merkle tree. The new campaign will be identified by this **id**.

```
function createLockedCampaign(
  bytes16 id,
  Campaign memory campaign,
  ClaimLockup memory claimLockup,
  address vestingAdmin,
  uint256 totalClaimers
) external nonReentrant {
  require(!usedIds[id], 'in use');
  usedIds[id] = true;
```

However, attackers could exploit this by initiating a DoS attack on new campaign creation. Since the contract prohibits the creation of campaigns with duplicate IDs, an attacker could front-run other users and create a spam campaign with the same ID. This will cause the other user's transaction to fail because the campaign ID already exists.

Recommendations: Generate the ID on-chain instead of using it as an input. For instance, use a counter to generate the next campaign ID.

Status: Acknowledged.

Finding: #5

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: #6

Issue: Using **IERC721.transferFrom()** instead of **safeTransferFrom().**

Severity: Low

Where: DelegatedClaimCampaigns.sol#L622

Impact: There are certain smart contracts that do not support ERC721, using transferFrom()lead the NFT may get stuck in the contract that does support ERC721.

Description: As per the documentation of EIP-721:
A wallet/broker/auction application MUST implement the wallet interface if it will accept safe transfers.

Reference: https://eips.ethereum.org/EIPS/eip-721, https://docs.openzeppelin.com/contracts/3.x/api/token/erc721#IERC721-transferFrom-address-address-uint256-, https://docs.openzeppelin.com/contracts/3.x/api/token/erc721#IERC721-safeTransferFrom-address-address-uint256-

Recommendations: Consider using **safeTransferFrom()** instead of **transferFrom().**

Status: Acknowledged.

Finding: #7

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: #8

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: #9

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: #10

Issue: **PUSH0** is not supported by all chains

Severity: QA

Where: DelegatedClaimCampaigns.sol#L2

Impact: Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than mainnet like L2 chains that may not support **PUSH0**, otherwise deployment of your contracts will fail.

Description: The compiler for Solidity **0.8.20** switches the default target EVM version to Shanghai, which means that the generated bytecode will include **PUSH0** opcodes. Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.

Recommendations: Make sure to specify the target EVM version when using Solidity **0.8.20**, especially if deploying to L2 chains that may not support the **PUSH0** opcode. Stay informed about the opcode support of different chains to ensure smooth deployment and compatibility.

Status: Acknowledged.

Finding: #11

Issue: claimLockups[] is not deleted

Severity: QA

Where: contracts/DelegatedClaimCampaigns.sol#L449 contracts/DelegatedClaimCampaigns.sol#L514

Impact: The data of a non-existing campaign will persist in **claimLockups[]** even though it is already deleted from **campaigns[].**

Description: campaigns[] and **claimLockups[]** are used to track all active campaigns using the **campaignId** key. However, when users fully claim a campaign, **campaigns[campaignId]** is deleted, but **claimLockups[campaignId]** is not.

```
if (campaigns[campaignId].amount == 0) {
  delete campaigns[campaignId];
}
```

Recommendations: Also delete **claimLockups[campaignId].**

Status: Resolved.

Finding: #12

Issue: Misleading comments

Severity: QA

Where:
- **DelegatedClaimCampaigns**:L161
- **TokenVestingPlans**:L18

Impact: Misleading comments might lead users or developers to make false assumptions about the code, that could lead to unexpected behaviors.

Description: The comments in **DelegatedClaimCampaigns.createLockedCampaign()** function states that **additionally it will check that the lockup details are valid, and perform an allowance increase to the contract for when tokens are claimed they can be pulled.**

However, it was observed that the allowance increases are performed in the claiming functions, not in the creation, and the remaining allowance is checked to be **0** at the end of these.

Another instance was detected in **TokenVestingPlans** contract, in which the following is stated: **3. Governance optimized for snapshot voting: These are built to allow beneficiaries to vote with their unvested tokens on snapshot, or delegate them to other delegates**. However, that is not exactly true, since users do not vote with their unvested tokens, but with the unclaimed ones (which include unvested and vested but unclaimed tokens).

Recommendations: Fix the code comments to adhere to the actual code logic implemented. It is important to update the comments when the code is changed, as well.

Status: Unresolved.

Finding: #13

Issue: Unnecessary allowance check

Severity: QA

Where: DelegatedClaimCampaigns:L203

Impact: Unnecessary checks increase the gas usage of the functions.

Description: The **cancelCampaigns()** function from **DelegatedClaimCampaigns** contract performs a check to ensure that the token allowance of the **DelegatedClaimCampaigns** to the **tokenLocker** contract is **0**.

This require statement is not necessary since allowance increases are only performed in the internal claim functions (**_claimLockedTokens()** and **_claimLockedAndDelegate()**), and a require statement that enforces that allowance is **0** after each function is performed, so under no circumstance, the allowance will be different than **0.**

This statement is probably a trace of earlier stages of the code, just like the previous finding (Misleading Comment) in which the allowance increase was performed when the campaign was created, not when the claim was performed.

Recommendations: Remove any unnecessary statement to reduce gas usage.

Status: Acknowledged.

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.