Claim Campaigns Technical Documentation

Smart Contract(s): ClaimCampaigns.sol;

1.0 Overview & Purpose:

Background: Hedgey Finance is a group of builders that develops token tools for treasury managers of crypto companies and decentralized organizations (DAOs). DAOs in this context have a token, ERC20 standard, which represents governance, ownership or utility over the DAO and its ecosystem. Hedgey builds a suite of tools deisgned around the token generation events for on-chain teams, such as vesting, investor lockups, and community token distributions. The ClaimsCampaign is a smart contract tool allowing for on-chain projects to effeciently distribute tokens to their community via a claim process, whereby the project can allocate tokens to a large amount of users that all have the ability to claim tokens based on prequalifying criteria. This tool is different than an airdrop tool, in that recipients have to manually claim tokens rather than simply recieving them via airdrop. The reason for this is to ensure that only active real users are participating in the community distribution, and the project has the opportunity to capture attention of the recipients during the claims process for engagement in things like delegation and governance and other project engagement.

Functional Architecture:

The purpose of the ClaimsCampaign tool is to let any team distribute their tokens to a massive audience. We do this by utilizing the compression and cryptography of merkle trees, leveraging their power to allow a project to efficiently upload a massive amount of recipients in a single transaction, and the cryptography of the merkle tree verifying that only intended recipients can actually claim the tokens from the campaign. We use the term campaign to describe a single distribution event, as a project may do several claim campaigns throughout their lifecycle, each one with a separate and unique list of recipients and amounts allocated to each.

The tool also leverages the power of our other vesting and lockup smart contracts, so that projects can distribute either liquid unlocked tokens, or tokens that when claimed unlock or vest on a time based schedule.

User Interface Overview:

In addition to the smart contract tool for distributing the tokens, Hedgey also builds a beautiful decentralized application with a user interface for easily creating and managing an ongoing campaign, as well as a customizable interface for projects to let their community participate in the claim process.

1.1 Roles:

a. **Campaign Manager**
   The Campaign Manager is the creator of the campaign, and generally the manager who can cancel it at any time (though it is optional to assign a different campaign manager address). The manager is in charge of creating the list of wallets that can claim, and their amounts, and then generating the merkle tree and uploading the root to the smart contract. These functions can all be done via the Hedgey App UI. The manager has to decide up front if the tokens to be claimed are unlocked, or will have a lockup or vesting schedule implemented when users claim tokens.
b. **Community Claimer**
   The Community Claimer is an individual that claims tokens from the campaign. They will have to know the ID of the campaign, the proof, and the amount they are eligible to receive. The Hedgey UI makes this very simple for users to get this data without having to manually look it up or store it themselves.
c. **Hedgey**
   Hedgey deploys the contract and also receives donations from campaign managers when they create campaigns from time to time.

1.2 Functions and Features

I. **Core Functions**

1. **createUnlockedCampaign**
   This function is the core function that allows a Manager to create a new claims campaign. They will need to have already created the merkle tree with the wallets and amounts (the merkle tree's leafs must consist ONLY OF a single wallet and single amount), and have a unique UUID for it. The function takes the UUID, and the campaign details, as we as an optional donation amount for Hedgey, and then stores the campaign details in storage mapped to the UUID. In addition it pulls in all of the tokens for the entire claim campaign into the contract for escrow. Any tokens claimed from a campaigned created by this function will be liquid and unlocked, ie the claimers will receive the tokens directly from the contract.

2. **createdLockedCampaign**
   This function is the core function that allows a Manager to create a new claims campaign. They will need to have already created the merkle tree with the wallets and amounts (the merkle tree's leafs must consist ONLY OF a single wallet and single amount), and have a unique UUID for it. The function takes the UUID, and the campaign details, as we as an optional donation amount for Hedgey, and then stores the campaign details in storage mapped to the UUID. The campaign manager also needs to input the details of the lockup or vesting parameters, which are stored in storage and mapped by the UUID. In addition it pulls in all of the tokens for the entire claim campaign into the contract for escrow. Tokens claimed from a locked campaign will deliver tokens to a hedgey TokenLockPlans or TokenVestingPlans contract, and then mint a lockup or vesting NFT to the recipient with the parameters and details set by the manager during this function.

3. **claimTokens**
   This function is the primary function for claimers to claim their tokens. They call this with the UUID of the campaign they are attempting to claim tokens from, with the merkle tree proof and the amount they are eligible to claim. The inputs are used to verify the eligibility of the claim based on the merkle tree cryptographic proof and leaf calculations. Once they have claimed their tokens, we set a boolean to true that evidences they have already claimed. Then the tokens are distributed to the claimer if unlocked, or if locked they will be delivered to the lockup plan set by the campaign manager, and the claimer will receive an NFT as their token to redeem tokens once vested or unlocked.

4. **cancelCampaign**
   This function is used by the campaign manager to cancel an ongoing campaign before all claims have been fulfilled. This will delete the campaign in storage and return any remaining tokens back to the campaign manager. No one can claim after it has been canceled.

1.3 Tokenomics & Fees

Hedgey does not have a governance token as of this product development, and so there are no tokenomics or fees associated with the tool. It is a free to use open tool built for the betterment of the crypto and web3 ecosystem. However, because of the complexity of the UI, Hedgey does accept donations by campaign managers when they are setting up their campaign - a totally optional donation.

Technical Architecture

2.0 Smart Contract Architecture Overview

The ClaimCampaigns.sol contract is a relatively simple contract in its architecture. It simple serves as an escrow contract for managers to create a way to distribute tokens to thousands or millions of wallets via a claim. It leverages the cryptography of a merkle tree to ensure that only users stored in the merkle tree with the exact amount can actually make a claim. The merkle tree is an elegant way to cryptographically and securely store all of the

information, and use the storage of the smart contracts efficiently and cheaply. It would be much more expensive for the manager to airdrop the tokens, or to perform some other form of distribution. The merkle tree works because verify the validity of a Leaf of the tree based on the root, the proof (which is the array of all of the leaves that create the tree up to that Leaf), and the Leaf itself. In our case, each Leaf is the keccak256 hash of the Wallet Address and Amount, in that order. Each leaf is hashed together with another leaf, creating a tree structure that can be followed and verified at each and every leaf node. Because the user wallet is input as the msg.sender when they are claiming tokens, they cannot attempt to claim tokens that are due to another wallet address.

The contract additionally interfaces with the other Hedgey Lockup and Vesting plans in this repository, allowing more functionality and flexibility for campaign managers to enforce lockups or vesting schedules on the tokens that are claimed. This is an entirely new functionality that should bring new uses for community distributions for token projects.

2.1 File Imports and Dependencies

TransferHelper.sol

The TransferHelper is a simple library that helps transfer tokens from one address to another. It additionally performs important checks such as ensuring that the balances prior to a transfer and after a transfer are what they were intended to be. This contract imports the open zeppelin SafeERC20.sol utility contract.

TimelockLibrary.sol

This library performs several critical read library functions that are used by the LockupPlans. It performs checks and calculates the end date of a lockup plan, as well as the claimable unlocked balance, and the remaining locked balance of a lockup plan at a given time. It is used to simply verify the validity of a token lockup strategy when creating the campaign.

IVestingPlans.sol

This Interface allows for the contract to generate vesting plans when tokens are claimed.

ILockupPlans.sol

This interface allows for the contract to generate lockup plans when tokens are claimed.

@openzeppelin/contracts/security/ReentrancyGuard.sol

While there are no external oracles or price feeds, reentrancy guard is used to protect against read and write reentrancy attacks.

'@openzeppelin/contracts/utils/cryptography/MerkleProof.sol

MerkleProof is a core contract import that verifies the validity of a root + proof + leaf, and is used to determine whether a claimer is eligible to claim tokens or not.

2.2 Global Variables

Campaign Struct & Mapping:

```
struct Campaign {
    address manager;
```

```
    address token;
    uint256 amount;
    uint256 end;
    TokenLockup tokenLockup;
    bytes32 root;
  }
mapping(bytes16 => Campaign) public campaigns;
```

The Campaign struct contains the general data about a specific claim campaign.

- The **manager** address is the address (typically of the creator) who can cancel the campaign at any time.
- The **token** is the address of the ERC20 token that is held in escrow and distributed to the claimers.
- The **amount** is the total amount of the campaign, until users start claiming. The amount is updated after each claim and then represents the total tokens remaining in the campaign before completion.
- The **end** is a date in unix time that can be used to prevent claims from occurring after a certain time stamp.
- **Tokenlockup** signifies whether the tokens, upon claim, will be unlocked, locked in a TokenLockupPlan, or vesting and locked in a TokenVestingPlan.
- **Root** is the bytes root of the merkle tree created that stores all of the data of who can claim and of what amount.

The Campaigns are mapped to the UUID in storage that is provided during the creation of the campaign function.

ClaimLockup

```
struct ClaimLockup {
    address tokenLocker;
    uint256 rate;
    uint256 start;
    uint256 cliff;
    uint256 period;
  }
mapping(bytes16 => ClaimLockup) public claimLockups;
```

The ClaimLockup struct contains the information regarding a lockup strategy for Only when tokens are locked. This is ignored and not stored for tokens that are unlocked.

- **Tokenlocker** is the address of the TokenLockupPlan or TokenVestingPlan that will lock the tokens when claimed
- **Rate** is the rate at which tokens will unlock or vest in a given period
- **Start** is the start date when tokens will start unlocking or vesting

- **Cliff** is an optional cliff date for when tokens unlock in a single bulk amount on the cliff date
- **Period** is the time in seconds between each unlock or vest interval

When a campaign has a lockup schedule, the Struct is mapped to the same UUID that the campaign is mapped to in storage.

Claimed

```
mapping(bytes16 => mapping(address => bool)) public claimed;
```

A simple mapping that tests whether a claim address has already claimed tokens or not. This is set to true after a single successful claim in the claimTokens function.

Donation

```
struct Donation {
    address tokenLocker;
    uint256 amount;
    uint256 rate;
    uint256 start;
    uint256 cliff;
    uint256 period;
  }
```

The donation struct contains all the information for a donation. Users can donate tokens to hedgey and include them with a lockup date and information. The tokens donated cannot be vested, but can be locked up in a tokenlockup plan.

- **Tokenlocker** is the address of the TokenLockupPlan that will lock the tokens
- **Amount** is the amount of the donation. If this is set to 0, no donation will occur.
- **Rate** is the rate at which tokens will unlock in a given period
- **Start** is the start date when tokens will start unlocking. If this is set to 0, then tokens will be donated unlocked.
- **Cliff** is an optional cliff date for when tokens unlock in a single bulk amount on the cliff date
- **Period** is the time in seconds between each unlock interval

2.3 Hedgey Only Functions

Donation Collector

```
 function changeDonationcollector(address newCollector) external {
    require(msg.sender == donationCollector);
    donationCollector = newCollector;
  }
```

Hedgey can change the address where donations are received.

2.4 Write Functions

createUnlockedCampaign

```
function createUnlockedCampaign(
    bytes16 id,
    Campaign memory campaign,
    Donation memory donation
  ) external nonReentrant {
```

This function is used to create an unlocked claim campaign. The manager will need to input the UUID they have self generated, typically the location of the merkle tree file in S3 or IPFS storage. They also input the campaign details contained in the campaign struct, and the donation details from the donation struct.
This function will pull the tokens from the msg.sender into the contract, based on the campaign.token address and the campaign.amount. It will make sure the id isn't already in use by another campaign, and it will then store the campaign details in storage, mapped to the id. It will process donation transfer if there is an amount to transfer.

createLockedCampaign

```
function createLockedCampaign(
    bytes16 id,
    Campaign memory campaign,
    ClaimLockup memory claimLockup,
    Donation memory donation
  ) external nonReentrant
```

This function is used to create an lockedclaim campaign. The manager will need to input the UUID they have self generated, typically the location of the merkle tree file in S3 or IPFS storage. They also input the campaign details contained in the campaign struct, and the donation details from the donation struct. In addition to the campaign details, the manager will provide all of the intended lockup parameters in the claimLockup struct. Any tokens claimed will be locked based on this struct that is not mutable.
This function will pull the tokens from the msg.sender into the contract, based on the campaign.token address and the campaign.amount. It will make sure the id isn't already in use by another campaign, and it will then store the campaign details in storage, mapped to the id. The claimLockup will also be mapped to the same id. It will process donation transfer if there is an amount to transfer.

claimTokens

```
function claimTokens(bytes16 campaignId, bytes32[] memory proof, uint256
claimAmount) external nonReentrant {
function cancelCampaign(bytes16 campaignId) external nonReentrant {
```

This function is the function for users to claim their tokens. They will need to input the campaignID (the UUID that was used during the create methods), as well as their merkle proof and the amount they are intending to claim. The function will verify that the campaign.root + proof + leaf (msg.sender, claimAmount) is a valid leaf in the merkle tree. If it is, it will check that they have not already claimed the tokens, and then process the claim. If the campaign is unlocked type, then it will withdraw the claimAmount of tokens and deliver to the msg.sender. If tokens are locked or vesting, then it will deliver tokens to the associated tokenlocker address, and createPlan method - minting the user an NFT that represents their rights to the locked or vesting tokens based on the schedule. The campaign manager will be the vestingAdmin for vesting plan lockups.

The function also ensures that the campaign is sufficiently funded to process the claim, and then reduces the campaign amount by the claimAmount. If the amount is 0, the campaign is completed and deleted in storage and can no longer be claimed.

cancelCampaign

```
function cancelCampaign(bytes16 campaignId) external nonReentrant {
```

This function lets the campaign manager cancel a campaign at anytime before completion. It will check that it isn't complete, and then delete the campaign in storage and refund the campaign manager all of the remaining tokens that have not been claimed.

2.5 Public Functions

```
function verify(bytes32 root, bytes32[] memory proof, address claimer,
uint256 amount) public pure returns (bool)
```

This verify function is used by the claimTokens function to verify the validity of a claim. It is also a public function that anyone can use to validate a claim before calling the external write method. The function takes the root of a merkle tree, the proof for the leaf, and the claimer address and amount. The claimer address and amount, when hashed together, create the leaf node. Then the function uses a merkle cryptographic validator function to validate the leaf is actually part of the tree, with the inputs of the root, proof and leaf. If any of these are wrong it will return invalid and the function will return false.