TokenVestingLock - Smart Contract Technical Documentation & Architecture

Smart Contract: TokenVestingLock.sol

References of previous contracts:
https://github.com/hedgey-finance/Locked_VestingTokenPlans/tree/master/technical%20documentation

Background: Hedgey Finance builds onchain tools for automated vesting and token lockups and distributions. One of the core tools built and offered by Hedgey is a vesting tool that allows tokens to be allocated to a specific individual wallet, and automate the vesting process and schedule with smart contracts, where tokens vest along a predefined time based schedule to the beneficiary. The vesting contract was built by Hedgey to be a core primitive, where additional modules and functionality could be added onto the vesting core tool for more complex behaviors. The TokenVestingLock is a module designed specifically for the TokenVestingPlans contract that layers on a second level of time based token locks in addition to vesting. While this may seem redundant, there is a growing trend for teams to require an additional lockup schedule on top of their vesting schedules, where tokens vest following their traditional vesting schedule, but they cannot be unlocked and sold until a second lockup schedule has been passed. This module allows for these more complicated vesting schedules with lockup schedules to be combined and automated in smart contracts on chain. Below is a visual representation of a vesting schedule (green) and its associated lockup schedule (blue);

Purpose: The purpose of the TokenVestingLock contract is to add the additional lockup schedule layer to vesting plans for a more broad and somewhat sophisticated vesting schedule. The contract aims to layer on the new feature and some additional functionality while maintaining the general scope and usefulness of the existing TokenVestingPlans contract, in that it maintains the ability for vesting plans to be revoked, delegated, and redeemed on the predefined schedule. The goal of the TokenVestingLock is not to remove any functionality, but to add additional functionality so that the beneficiary and administrator still have the same overall experience from what they would have with the Hedgey Vesting Plans, but that this layers on more functions with the unlock schedule, but also the ability for the vesting admin, and other

wallets to redeem tokens on behalf of the beneficiary, allowing for even more automation possibilities of the vesting schedule.

Functional Overview

Roles

Role Name	Description	Jobs
Vesting Admin (Creator)	The administrator on the vesting plans, typically the finance or HR team administering the vesting plans for the employees.	- Creating the vesting & lockup plans. - Revoking plans for terminated employees. - Redeeming tokens on behalf of employees. - Acting as emergency agent to transfer plan to a different wallet on behalf of beneficiary if they lose access to their wallet (or its compromised). - Manage transferability of lockups
Beneficiary	The owner of the vesting lockup plan.	- Redeeming tokens as they vest - Unlocking tokens as they unlock (delivered to the beneficiary wallet) - Delegating tokens for voting
Approved Redeemer	An address approved to redeem tokens on behalf of a beneficiary	- Redeeming vesting and unlocking tokens on behalf of beneficiary (tokens always delivered to beneficiary)
Delegator	An address approved to delegate tokens held in a lockup or vesting plan on behalf of the beneficiary	- Delegating vesting or lockup plans for voting
Hedgey Contract Manager	The multisignature wallet used to manage which BatchCreator contract is currently deployed and linked to the TokenVestingLock contract	- Changing and managing the BatchCreator address used to create new vestingLock plans

Core Functions

Function Name	Description	Role / User
createVestingLock	Core function to create a new lock for a vesting plan	Vesting Admin
redeemAndUnlock	Core function to both redeem vested tokens and unlocked tokens in combined function	Beneficiary / approved redeemer
redeemVestingPlans	Core function to redeem only tokens vested in the vesting plan	Beneficiary / approved redeemer
unlock	Core function to unlock tokens that unlocked, assuming they have already been vested and available to be unlocked	Beneficiary / approved redeemer
burnRevokedVesting	Function to burn a vestinglock NFT after the vesting plan NFT has been revoked and thus the vestingLock NFT has no value anymore	Beneficiary
revokePlans	Function to revoke a vesting plan (directly to Vesting Plan, not to VestingLock contract)	Vesting Admin
editLockDetails	Function to change the lock details before the lock has commenced	Vesting Admin
updateTransferability	Function to allow or not allow lock NFTs to be transferred by the Beneficiary	Vesting Admin
delegatePlans	Function to delegate tokens (either onchain or via snapshot NFT delegation)	Beneficiary / Approved Delegator
setAdminRedemption	Function to allow the vesting admin to redeem tokens on behalf of beneficiary	Beneficiary
approveRedeemer	Function to approve a specific wallet as a redeemer	Beneficiary
removeRedeemer	Function to remove an already approved redeemer wallet	Beneficiary
approveRedeemerOperator	Function to approve a wallet that has	Beneficiary

	control over approvals for the entire wallet	

Smart Contract Technical Definitions & Design

The contract itself is an ERC721 contract, that both holds ERC721 tokens, and then issues an NFT representing ownership of the underlying vesting NFT, it is a “back to back” NFT contract. In inherits the ERC721 Enumerable functionality from Open Zeppelin to record states and ownership of positions. The contract maps the vesting NFT held to another NFT that is owned by the beneficial owner, allowing the owner to perform the necessary functions that they can do at the base vesting contract, in addition to the functionality that is layered on top with the lockup schedule. A struct is mapped to the NFT that contains all of the unlock logic and schedule, as well as the underlying vesting NFT that is held.
From an architectural perspective, the vestingLock contract owns the vesting plan NFT, and when it redeems tokens it pulls those tokens into the vestingLock contract. This updates the amount of tokens that are available to be unlocked at a given time, based on the unlock schedule. Then when tokens unlock a user can unlock them transferring the tokens from the vestingLock contract to the beneficial user. One way to think about this metaphorically is that the vestingLock contract is like a bucket, and it redeems tokens from the vesting plan filling up its bucket, whereby when tokens unlock they are pulled from the bucket, and so only the amount of tokens that have filled up the bucket can be unlocked. In this manner the vesting schedule periodicity is not forced to match the periodicity of the unlock schedule, but each discrete period of the unlock schedule is a single bucket, and so when unlocking only when the bucket has been filled up by the vesting redemption can the user unlock tokens. This means that if tokens have been redeemed from the vesting schedule but have not “filled” the unlock bucket period, the user cannot unlock those tokens. While less than ideal, this does support the actual intention of unlocking tokens only on the predefined discrete schedule. Alternatively, if users choose a per second streaming style, then tokens are generally always unlockable whenever some amount have been redeemed from the vesting plan.

Contract Inheritance

Open Zeppelin Imports:
@openzeppelin/contracts/token/ERC721/ERC721.sol
@openzeppelin/contracts/utils/ReentrancyGuard.sol
@openzeppelin/contracts/token/ERC20/IERC20.sol
@openzeppelin/contracts/token/ERC721/utils/ERC721Holder.sol

Hedgey Contract Imports:
ERC721Delegate.sol - Contract for delegating NFTs for snapshot and offchain or NFT voting, does not require ERC20Votes standard but works with standard ERC20 tokens by delegating an entire NFT

VotingVault.sol - Contract that holds tokens specifically for onchain voting, requires that a token conforms to the ERC20Votes standard

Libraries:

UnlockLibrary.sol - Library that contains all of the logic to calculate how and when tokens unlock
TransferHelper.sol - Library that controls transferring tokens into and out of smart contracts, and beneficiaries

Contract Objects, Variables & Functions

Structs

VestingLock

This is the primary struct defining the vesting lockup and its schedule
token is the address of the token that is locked up and vesting
totalAmount is the total amount - this comes from the vesting plan at inception and has to match
availableAmount is the actual amount of tokens that have vested and been redeemed into this contract that are maximally available to unlock at any time
start is the start date of the lockup schedule in block time
cliff is the cliff date of the lockup schedule in block time
rate is the rate at which tokens unlock per period. So if the rate is 100 and period is 1, then 100 tokens unlock per 1 second
period is the length of each discrete period. a "streaming" version uses a period of 1 for 1 second but daily is 86400 as example
vestingTokenId is the specific NFT token ID that is tied to the vesting plan
vestingAdmin is the administrator on the vesting plan, this is the only address that can edit the lockup schedule
transferable this is a toggle that the admin can define and allow the NFT to be transferred to another wallet by the owner of the lockup
adminTransferOBO this is a toggle that would led the vestingAdmin transfer the lockup NFT to another wallet on behalf of the owner in case of emergency

```
struct VestingLock {
    address token;
    uint256 totalAmount;
    uint256 availableAmount;
    uint256 start;
    uint256 cliff;
    uint256 rate;
    uint256 period;
    uint256 vestingTokenId;
    address vestingAdmin;
    bool transferable;
    bool adminTransferOBO;
  }
```

Recipient

This struct is used for creating vestingLocks, and is not stored in storage itself
beneficiary is the address of the beneficiary, the address the lock NFT will be minted to
adminRedeem is the boolean that initial toggles on or off the ability for the vestingAdmin to redeem tokens on behalf of the beneficiary

```
struct Recipient {
   address beneficiary;
   bool adminRedeem;
 }
```

Global Variables

IVesting public hedgeyVesting: The address and implementation of the Hedgey Vesting Contract that this contract will hold and implement the lockup logic for.

address public hedgeyPlanCreator: A special BatchCreator contract that is allowed to create vestingLocks because it specifically will create vesting plans and vestingLocks simultaneously.

uint256 internal _tokenIds: The internal tracking of tokenIDs that each NFT and VestingLock is mapped to.

mapping(uint256 => VestingLock) internal _vestingLocks: mapping of the VestingLock struct to the tokenIds which represent each NFT

mapping(uint256 => bool) internal _allocatedVestingTokenIds:this is a mapping of the vestingTokenIds that have been allocted to a lockup NFT so that lockup NFTs are always mapped ONLY one to one to a vesting plan

mapping(uint256 => mapping(address => bool)) internal _approvedRedeemers: this is a mapping of the approved redeemeers that can redeem the vested tokens or unlock the unlocked tokens, used as a mechanism for redeeming on behalf. Note: if the user sets the zero address to be true, then it is a global approval for anyone to redeem

mapping(address => mapping(address => bool)) internal _redeemerOperators: This is a mapping of addresses that can act as the operator on behalf of a wallet, controlling the ability to redeem on its behalf, as well as approve redeemers or dis-approve redeemers and the vestingAdmin redemption.

mapping(uint256 => bool) internal _adminRedeem: separate mapping specifically defining if the vestingAdmin can redeem on behalf of end users

mapping(uint256 => address) public votingVaults: this is a mapping of the voting vaults owned by the locked NFTs specifically used for onchain voting and delegation

State Changing Functions

function createVestingLock(
Recipient memory recipient,
uint256 vestingTokenId,
uint256 start,
uint256 cliff,
uint256 rate,
uint256 period,
bool transferable,
bool adminTransferOBO
) external nonReentrant returns (uint256 newLockId)

This function is the core function which creates a new vesting lock NFT. The function requires that the vestingLock contract already owns the NFT, and then it will mint the recipient a vestingLock NFT, and store the inputs in storage. It doesn't require an 'amount' to be input, as the function will look up the amount of tokens held by the vesting NFT and then store that amount in the total amount field. Because the hedgey vesting NFTs are not transferable by default, they cannot use the "transferFrom" function, and so only a special batch creator address, or the vesting admin can create a vesting lock once the vesting NFT has been transferred into the contract. The batch creator contract creates a vesting plan and vesting lock in a single transaction for security purposes.

function redeemAndUnlock(uint256[] calldata lockIds) external nonReentrant returns (uint256[] memory redeemedBalances, uint256[] memory vestingRemainder, uint256[] memory unlockedBalances)

This function will call the internal _redeemVesting and _unlock to redeem any vested tokens and unlock any available unlocked tokens. Redeeming them from the vesting contract will pull tokens from the vesting contract into the vestingLock contract. The unlock function will deliver unlocked tokens to the holder of the vestingLock NFT.

function unlock(uint256[] calldata lockIds) external nonReentrant returns (uint256[] memory unlockedBalances)

Function to unlock tokens that are currently held by the vestingLock contract and deliver them to the owner of the NFT, the beneficiary. The tokens must be in the vestingLock contract for them to be unlocked. If there are no tokens to be unlocked it will not revert, but just return 0,0.

function redeemVestingPlans(uint256[] calldata lockIds) external nonReentrant returns (uint256[] memory balances, uint256[] memory remainders)

Function to redeem tokens from the vesting plan contract. This will pull tokens from the vesting contract into the vestingLock contract. If there is nothing to be redeemed it will just return 0,0, it will not revert.

function burnRevokedVesting(uint256 lockId) external

This function is available for beneficiaries if their underlying vesting plan has been revoked. This function will check that the vesting plan is no longer owned by the contract, and that there is nothing left to be redeemed and then burn the vestingLock NFT and delete the struct mapped in storage.

function _unlock(uint256 lockId) internal returns (uint256 unlockedBalance)
This is an internal function that will unlock tokens. It withdraws and transfers to the owner of the NFT the amount of tokens available to be unlocked based on the UnlockLibrary. This function will at max unlock the availableAmount of tokens held in the VestingLock struct, as that is the amount of tokens physically in the contract that can be at max redeemed. The function will not revert if there is nothing to unlock, but will just return a 0 amount. The function will update the available amount and total amount of tokens in storage after unlocking them. If the total amount is 0, this means that the entire balance has been redeemed from vesting and unlocked, and then the NFT will be burned and storage deleted. Only the beneficiary or an approved redeemer can use this function.

function _redeemVesting(uint256 lockId) internal returns (uint256 balance, uint256 remainder)
Internal function to redeem tokens from the vesting contract, it pulls tokens from the vesting contract based on the vesting plan's schedule and receives those tokens into the vestingLock contract. It then updates the storage values of the vestingLock struct, increasing the amount of available tokens, and performing a total amount check for the instance where something may have changed, like a revoke has occurred. It "fills" up the vestingLock bucket.

function updateVestingAdmin(uint256[] memory lockIds, address newAdmin) external
Function for the vestingAdmin to change their address if they need to adjust things.

function updateTransferability(uint256[] memory lockIds, bool transferable) external
Function for the vestingAdmin to update the transferability of the vestingLock NFT.

function editLockDetails(uint256 lockId, uint256 start, uint256 cliff, uint256 rate, uint256 period) external
Function for the vesting admin to edit the lock details. Only certain parameters can be updated, and it has to happen before the later of the start or cliff date - the effective date of when the lockup schedule starts.

function updateAdminTransferOBO(uint256 lockId, bool adminTransferOBO) external
Function for the beneficiary to update whether the vesting admin can transfer their NFT on their behalf. This is a helpful function for users who are not comfortable with taking self custody or may lose access to their wallets and the admin can act on their behalf to transfer their NFT to a new wallet address.

function updateVestingTransferability(uint256 lockId, bool transferable) external
This is a nuanced function. The vesting plans have the ability for the vesting admin to transfer them, by default when the vestingLock is created this is always false. However, the beneficiary can turn this function on for emergency backup whereby they need to undo the vesting lockup plan entirely, and the vesting admin can transfer the vesting plan NFT out of the vestingLock contract to make changes. This is only toggleable by the beneficiary.

function delegatePlans(uint256[] calldata lockIds, address[] calldata delegatees) external
This function will call the delegatePlans() function on the vesting contract, and delegate the tokens held in escrow by the vesting contract to the delegatee.

function delegateLockNFTs(uint256[] calldata lockIds, address[] calldata delegatees) external
This function will delegate the vestingLock NFT, utilizing the ERC721Delegate contract and call the function to delegate the NFT. This function is useful for when snapshot voting is setup that supports this style of voting.

function delegateLockPlans(uint256[] calldata lockIds, address[] calldata delegatees) external returns (address[] memory)
This function will delegate the tokens held in the vestingLock contract by a beneficiary to a votingVault for onchain voting.

function _setupVoting(uint256 lockId) internal returns (address)
This is the internal voting vault setup function that deploys a specific voting vault contract and maps it to the vestingLock NFT. It will self delegate if there is not an existing delegate of the wallet address calling the function, but if the msg.sender has pre-delegated its tokens it will automatically delegate the tokens to that address. It returns the voting vault address that has been deployed.

function _delegate(uint256 lockId, address delegatee) internal returns (address)
This is an internal function to delegate tokens of a voting vault. It will check if a voting vault has been setup, and if not set one up, and then delegate the tokens. If the delegatee is already the delegated address of the voting vault, it will skip delegation and just return.

function approveRedeemer(uint256 lockId, address redeemer) external
Function to approve a specific redeemer address for an lock NFT. If the user wants anyone to publicly be able to redeem their vesting and unlock, they can set the 0x0 address to true. This is fine as the 0x0 address is by default set to false, and so they have to set it to true for it to be a public redemption.

function removeRedeemer(uint256 lockId, address redeemer) external
Function to remove an approved redeemer or turn the public redemption to false.

function approveRedeemerOperator(address operator, bool approved) external
Function to set a specific wallet to be an approved redeemer operator that can redeem on behalf of all of the beneficial owners NFTs, but also set approved redeemers or the vestingAdmin redemption function to true or false.

function setAdminRedemption(uint256 lockId, bool enabled) external
Function to specially set the vesting admin as an approved redeemer. This is a useful function as it is envisioned that in many circumstances vesting admins will redeem and unlock tokens on behalf of their entire employee set, and so employees can enable this whereby the vesting admin can redeem tokens on behalf of everyone on a regular basis.

View Functions

function adminCanRedeem(uint256 lockId, address admin) public view returns (bool)
This function will return true if the vesting admin redemption has been set to true.

function isApprovedRedeemer(uint256 lockId, address redeemer) public view returns (bool)
This function will return true if the redeemer address of a specific lockId is set to true, or if the public address of 0x0 has been set to true.

function getLockBalance(uint256 lockId) public view returns (uint256 unlockedBalance, uint256 lockedBalance, uint256 unlockTime)
Function to get the balances of a specific vestingLock. The unlocked balance will return at maximum the availableAmount (which is physically held in the contract). The calculation of unlocked balance is based on the amount of tokens that unlock (rate) in a given amount of time (period), and the amount of time that has passed since the last unlock (start). When the unlock is coming to its end, if the totalAmount left and availableAmount are both less than the calculated amount, then it will unlock the available amount that is left remaining and this signifies the end of the unlock. The algorithm also requires that the available amount is greater than the rate, or else it will not unlock anything. This is the idea that the bucket must be filled before it can be drained, as it is insecure to drain a partially filled bucket.

function getLockEnd(uint256 lockId) public view returns (uint256 end)
This function gets the end date of the vestingLock period based on the amount of total tokens in the vesting and lock plan, and the lock rate and period.

function getVestingLock(uint256 lockId) public view returns (VestingLock memory)
This function is the public getter function that returns the VestingLock struct held in storage.

function lockedBalances(address holder, address token) external view returns (uint256 lockedBalance)
This function will get all of the tokens locked by a specific holder and ERC20 token across multiple NFTs. This is useful function for voting and other defi use cases.

function delegatedBalances(address delegatee, address token) external view returns (uint256 delegatedBalance)
This function returns and aggregate balance of all of the tokens that have been delegated to a specific delegatee based on the delegatee address and the ERC20 token address. This will count through all of the NFTs that have been delegated to that address in the ERC721Delegate contract, and sum up the tokens locked in those NFTs that match the ERC20 token address. This is a specific function used for the snapshot strategy and ERC721 delegation.