Token Lockup Plans Technical Documentation

Smart Contract(s): TokenLockupPlans.sol; TokenLockupPlans_Bound.sol; VotingTokenLockupPlans.sol; VotingTokenLockupPlans_Bound.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. Before and or after a token is launched by a DAO, they will issue tokens to their investors and contributors, and most often those tokens that are granted should contain an unlocking arrangement. Our decentralized tools are built specifically for this purpose, solving the ability to issue locked tokens to investors and community, while supercharging those token lockups with voting rights and the ability to use them in other DeFi activities.

Functional Architecture:

The purpose of this tool is to automate and effectively manage the token unlock schedules and plans of DAOs and its investors. Unique to Hedgey's tool, we utilize the backbones of NFTs (ERC721) to create a lockup plan for each participant that is defined as an individual and unique NFT. Each NFT in the collection represents a single and unique lockup schedule for the beneficiary of the token lockup plan, issued by the DAO. The NFT backbone empowers additional abilities to the contract, such as greater transparency, managing multiple lockup schedules at a time, and most importantly adding in the ability for beneficiaries to vote or delegate their votes based on the tokens that are contained in their lockup plan. The voting attribute is an addition that DAOs must actively select to use in their snapshot or comes standard for on-chain voting DAOs.

Tokens unlock in a linear and periodic schedule, unlocking a precise amount of tokens every discrete period, where the period is denominated in seconds, as defined by the unlock

schedule. When a plan beneficiary wants to claim some or all of their unlocked tokens, they will interact with the smart contract to redeem whatever amount of tokens that has been unlocked. Owners of the lockups may redeem the entire amount of tokens available to be unlocked, or a partial amount if they choose to claim only a portion of what has become unlocked. Unlock schedules may contain a single cliff date, where any tokens prior to the cliff become fully unlocked in a chunk on the cliff date, and are not claimable prior to the cliff date.

Because the unlock plans are built on top of ERC721 backbone, they can be transferred to other addresses. This allows them to participate in decentralized finance activities such as secondary sales on NFT platforms, or borrowing against the value of the NFTs. Additionally to facilitate these activities owners of the NFTs can segment a single NFT into multiple chunks, and they may also recombine segmented NFTs back into the original whole. This allows for owners to segment large chunks of tokens into smaller chunks; opening up more access to DeFi opportunities, as well as the ability to sub-delegate a large position of tokens to multiple different delegates for voting.

User Interface Overview:

In addition to the smart contracts, the Hedgey team also builds a beautiful front end interface for users to interact with and perform all of the smart contract functions, as well as view their streaming balances and administer them. The application relies on a back end database that stores all of the information from the smart contract each time a transaction occurs, storing the event values and several other key pieces of information. That information is then used to display current information for users about their balances, as well as enable them to properly perform smart contract calls. Because Hedgey's front end applications rely on a database for streamlined viewing, storing initial data in the smart contract storage is unnecessary, and so the contract data is streamlined for purpose, while the application is built to store all previous states and variables for historical data visualization.

1.1 Roles:

a. **Contract Admin**
   Contract Admin is the deployer of the smart contract, who has two primary responsibilities; 1) deploying the smart contract and verifying the authenticity of it and 2) adding a baseURI string to link the NFTs to the metadata that provides an easy way to visualize the data contained in the vesting schedule for front end applications.
b. **Lockup Plan Distributors**
   Generally this is the DAO, and or its finance team. They are in charge of distributing tokens to the investors and beneficiaries of lockup plans, and inputting all of the details necessary to create each unique lockup plan. Once distributed, their job is done, they cannot alter or change anything else about the lockup plan. The Plan Distributors also need to decide whether the lockup plans may be transferable or not, and decide between snapshot optimized voting or on-chain token voting, as each of these decisions will result in a different smart contract they lock tokens inside and distribute NFTs from.
c. **Lockup Plan Beneficiaries**
   Lockup Beneficiaries are the receivers of the token lockup plans, typically project

investors, early employees and or community contributors. Once a lockup plan is received, the beneficiary may claim tokens from their plan anytime an amount is redeemable. They may also transfer or sell their position (if determined this was acceptable by the Plan Distributor). Beneficiaries can vote with their tokens locked in the plans, either using a snapshot optimized strategy, or an on-chain voting strategy, and they can delegate their tokens to other delegatees. Beneficiaries can segment their NFT into smaller chunks, and recombine segments into a whole again should they require smaller token lockup plans for sub-delegation or DeFi activities.

d. **Plan Delegators and Delegator Operators**
Plan Delegators and Delegator Operators are external parties that act on behalf of plan owners to delegate the underlying locked tokens to other addresses. A Plan Delegator can only act on behalf of a specific planID, a single NFT, to manage the delegation of the tokens locked within. Whereas the Delegator Operator is approved by a wallet address to manage all of delegations across all of the NFT plans owned by the wallet address. The Delegator Operator can also approve other single Plan Delegators to delegate on behalf of the owner. This schema follows the standards created by the ERC721 allowance schema to spend an NFT, allowing either a single plan to be delegated by an address, or giving an address full access to manage the delegations across all the plans. If a plan is transferred the Plan Delegator is deleted from storage and must be reset.

1.2 Functions and Features

I. Core Functions

1. **updateBaseURI**
This is a Contract Admin Only function, and used to link a URI to the NFT collection to easily represent the NFT data on-chain to off-chain metadata for easier application visualizations.
2. **deleteAdmin**
This is a Contract Admin Only function, which deletes the administrator after they have completed the linking of the URI to the NFT collection for metadata purposes.
3. **createPlan**
This function is what creates the lockup schedule. It is performed by the plan distributor, who will input all of the data points of the lockup schedule into the function, lock the tokens inside the contract for escrow, and mint an NFT to the beneficiary representing their unlock schedule. This function will pull the tokens from the distributor's balance and transfer them to the NFT smart contract. The creator should know when they are creating the following parameters:
A. beneficiary / recipient / holder of the NFT
B. address of the DAO token
C. amount of total tokens in the entire vesting schedule
D. the start date when the plan starts vesting (can be back dated, forward dated, or now)
E. An optional cliff date when some of the tokens vest on a single cliff date
F. the per period rate which the tokens vest upon the start date

G. the length of the period, denominated in seconds

4. **redeemPlans**
   This function is used by the beneficiary. This is the core function that the beneficiary will call to redeem the available claimable tokens that have unlocked for a specific plan. They can enter in multiple plan IDs to the function to redeem tokens from more than one unlock plan. If the entire lockup plan balance is claimed and unlocked, then the NFT Plan will be burned when this function is called. If there are no claimable (have not unlocked) for the NFT provided, then nothing is redeemed. Tokens are transferred from the NFT contract to the beneficiary wallet address.
5. **redeemAllPlans**
   This function is used by the beneficiary to redeem all of their lockup plans without needing the individual planIds.
6. **partialRedeemPlans**
   As an alternative to redeeming plans, the beneficiary can pick a discrete time amount to redeem anything unlocked up until that time. The benefit of this is nuanced as it relates to taxes and operationally redeeming only a specific amount of unlocked tokens that might be required as opposed to the entire available unlocked balance.
7. **segmentPlans**
   This function allows a beneficiary to segment a single lockup plan (and NFT) into multiple smaller chunks.
8. **segmentAndDelegatePlans**
   This function allows a beneficiary to segment a single lockup plan, and then simultaneously delegate the tokens in the newly created segment to a delegatee. The initial plan's delegation will not be altered, only the newly created segment.
9. **combinePlans**
   This function allows a beneficiary to combine lockup plans that were previously segmented, or plans that have the exact same lockup schedule. The plans must match in every parameter except the amount and rate to combine two plans together.

II. **Voting Functions**

10. **delegate**
    This function is used by the beneficiary who wants to delegate the tokens locked inside their NFT to another wallet address, specifically used for voting. For the snapshot optimized contract, this will delegate the NFT to a wallet address (which is by default self-delegated). For the on-chain optimized version, this will delegate the tokens held in a separate voting vault contract to a delegatee.
11. **delegateAll**
    This function is used by the beneficiary who wants to delegate all of their lockup plans to a single wallet address for the voting purposes. This iterates through all of the owned plans of a single beneficiary and then delegates the lockup plan tokens.
12. **setupVoting** (*on-chain voting specific*)
    For the on-chain voting version, the tokens are held in an external voting vault contract.

Since this is not a mandatory action, for someone to participate in on-chain governance they would use this function to create the separate voting vault, which transfers the tokens to the voting vault and then delegates them either to the beneficiary, or if they have previously delegated their wallet tokens, then will automatically be delegated to them.

13. **approveDelegator**
    To assign a wallet to be the Plan Delegator for a single NFT, an owner or operator would call this function to assign the plan to a single wallet address as the Delegator. The Delegator can then call the setupVoting and delegate functions on behalf of the owner of the NFT.
14. **setApprovalForAllDelegation**
    This function sets up the Delegator Operator, where an address has full control to manage the delegation activities on behalf of a wallet address - where the operator can delegate tokens for any NFT held by that wallet.

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.

Technical Architecture

2.0 Smart Contract Architecture Overview

There are four versions of the Lockup Plan smart contracts; TokenLockupPlans (“TLP”), its non-transferable version TokenLockupPlans_Bound (“TLP-B”), VotingTokenLockupPlans (“V-TLP”) and its non-transferable version VotingTokenLockupPlans_Bound (“V-TLP-B”), collectively called “Lockup Plans”. The V-TLP and V-TLP-B contracts are built for on-chain governance voting, while the TLP and TLP-B contracts are optimized for snapshot voting. Both pairs leverage the ERC721Enumerable extension as their background, however the snapshot optimized goes a step further and uses a custom built ERC721Delegate contract that allows owners of the NFTs to delegate them (and their entire lockup balance) to their own wallet or another wallet, which we have built custom snapshot strategies specifically to implement. Both contracts effectively hold the tokens in escrow during the lockup period, however when the beneficiaries of the V-TLP and V-TLP-B plans participate in on-chain governance they will set up a voting vault whereby the tokens are physically moved and segregated to the voting vault. Each NFT has one and only one voting vault in this architecture, and the vault can only delegate tokens (the tokens inside the voting vault cannot be physically moved anywhere except back to the NFT contract when redeemed).

This NFT backbone allows the smart contract to actively manage multiple lockup schedules within the same smart contract - allowing for better and more efficient visibility to the DAO and management by the beneficiaries. The contract holds tokens that are locked in the smart contract as escrow - pulled to the contract when new lockup plan NFTs are created. When beneficiaries redeem a plan the tokens in the contract are transferred out to the beneficiary. The contract maps an NFT to a Struct that stores all of the data points of each unique lockup plan.

2.1 File Imports and Dependencies

Both pairs of contracts share the same dependencies, with few exceptions. The contracts that are not openzeppelin imports are part of the core infrastructure and their inherited functionality will be detailed.

Shared Imports:

- **@openzeppelin/contracts/token/ERC721/ERC721.sol**
  The standard ERC721 NFT contract is the backbone of the LockupPlans. The TLP-B and V-TLP-B contracts override the internal _transfr function to prevent any NFTs from being transferred to another address. All other functions are left as is. Each lockup plan is represented by an NFT, so each time a new plan is minted a new NFT is minted and mapped to the plan details.

- **@openzeppelin/contracts/utils/Counters.sol**
  Counters is used simply for incrementing the planId and nft token Id each time a new one is minted.

- **@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.

- **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.

- **URIAdmin.sol**
  This contract provides the basic framework for the URI admin (the contract deployer) to update the NFT uri that each NFT points to for its metadata, and then can delete the admin when ready.

- **LockupStorage.sol**
  This contract is the shared contract that contains the core global variables like the struct of what comprises an individual vesting plan, the mapping from the NFT token ID to the vesting plan struct, and the events for each of the functions. It also leverages the

TimelockLibrary to get key read functions for the vesting plan based on the storage, like a specific plans end date, or the balance of a specific vesting plan.

- **PlanDelegator.sol**
  This is an extension of the ERC721Enumerable.sol contract from OpenZeppelin. This contract takes the functionality but adds in additional functionality around approving Delegators and Delegator Operators, who, once approved, can manage delegations on behalf of a lockup plan beneficiary. The contract works similarly to that of the spend and allowance logic, but strictly for delegation instead of spending the NFT.

Plans Only Dependency:

- **ERC721Delegate.sol**
  This contract inherits the ERC721Enumerable OpenZeppelin contract and adds the functionality to delegate NFTs to another wallet. There is a separate document outlining the technical details of this specific contract because of its core usage by the Plans contract.

VotingPlans Only Dependencies:

- **VotingVault.sol**
  The VotingVault is a small specific contract that is created by the VotingPlans contract. It allows for owners to segregate tokens into an instance of this contract, creating a new VotingVault that is associated with each NFT individually. Its only capability is to delegate tokens to another wallet address, and otherwise all storage is still controlled by the master VotingPlans contract so that there is never anything out of alignment (ie read reentrency attack vectors).

2.2 Global Variables

2.2.1 Public Global Variables

Plan:

```
struct Plan {
  address token;
  uint256 amount;
  uint256 start;
  uint256 cliff;
  uint256 rate;
  uint256 period;
}

mapping(uint256 => Plan) public plans;
```

The Plan is the primary storage variable that contains all of the necessary information about an individual vesting plan. Each vesting plan is unique and so a Plan object is used to represent one unique plan, and so each NFT is mapped uniquely to a single Plan.

Token: the address of the ERC20 token that is vesting

Amount: the total amount of locked and unlocked yet unclaimed tokens (this is not the total amount of the plan that was initiated, this data is updated each time tokens are claimed to keep it current with the remaining amount that is locked)

Start: This is the date when the lockup plan begins, but each time a redemption occurs, this start date updates to the most recent redemption time.

Cliff: A date that represents a discrete cliff whereby the beneficiary cannot redeem any tokens prior to this date. This does not change.

Rate: The tokens per period rate which the tokens unlock. If the period is 1 for example, then this rate represents the amount of tokens that vest each second. If the period is (86400 = 1 day), then this is the amount of tokens that vest in a single day.

Period: the seconds in each period that determine the unlock schedule. Schedules may unlock tokens every second, others each day, and others on a 30 day plan. This allows for flexibility in the unlocking schedules with longer discrete periods.

votingVaults (*mapping for Voting Plans)*:

```
mapping(uint256 => address) public votingVaults;
```

This is specific for the V-TLP and V-TLP-B. This maps the NFT planId to the specific voting vault address so that tokens can be delegated and transferred back and forth as required.

2.2.2 Private and Internal Global Variables

_planIds

```
using Counters for Counters.Counter;
 Counters.Counter private _planIds;
```

This is a simple counter tool that increments each time a new NFT is minted. The counters is a tool that lets us store a current number, and each time we mint a new NFT and vesting plan, we increment the _planIds by one.

2.3 Contract Admin Only Functions

```
function updateBaseURI(string memory _uri) external
```

This is the function called by the admin to update the baseURI after deployment.

```
function deleteAdmin() external
```

This function can be called by the admin after updating the baseURI to delete itself from storage, thus not allowing any further changes to the baseURI.

2.4 Read Functions

2.4.1 Public Read Functions

```
  function planBalanceOf(
    uint256 planId,
```

```
    uint256 timeStamp,
    uint256 redemptionTime
  ) public view returns (uint256 balance, uint256 remainder, uint256
latestUnlock)
```

This function will take the planId, any timestamp (block time), and another variable redemptionTime (block time), and return the balance that is unlocked and claimable, the remainder which is still locked, and the latest discrete unlock time. The timeStamp and redemptionTime are distinct to allow beneficiaries to claim partial amounts, even if their cliff date is past but they want to claim a partial amount prior to the cliff date. The latestUnlock is the actual discrete timestamp when redemption occurs. For example a plan unlocks with a period of 30 days, then a redemption date that is 65 days will return a latestUnlock of the 60th day timestamp - thus returning the date when the actual redemption of the latest unlock occurred. This is necessary for updating the storage so that tokens are not missed when redeeming multiple times over the course of their lockup plan.

```
  function planEnd(uint256 planId) public view returns (uint256 end)
```

This function returns the end date of the lockup plan based on the planId.

```
function getApprovedDelegator(uint256 planId) public view returns
(address)
```

PlanDelegator: This function is part of the plan delegator schema, and returns an address that was approved by the owner of an NFT to delegate the locked tokens on behalf of the owner of the plan.

```
function isApprovedForAllDelegation(address owner, address operator)
public view returns (bool)
```

PlanDelegator: This function is part of the plan delegator schema. The Owner is the owner of the plan and NFT, the operator is a Delegator Operator, who has control and the ability to delegate NFTs and tokens on behalf of the owner of the plans, for all plans owned by that address.

2.4.2 External Read Functions

```
  function lockedBalances(address holder, address token) external view
returns (uint256 lockedBalance)
```

This function is useful for snapshot voting, and other forms of viewing the amount of tokens a specific wallet has locked inside of the contract. It takes the address of a wallet (holder), and the address of the token they have locked and returns the total amount of that specific token that

this specific wallet is the beneficiary of in the lockup plans. It sums the balance of all the NFT plans that this wallet owns.

```
  function delegatedBalances(address delegate, address token) external
view returns (uint256 delegatedBalance)
```

This function is only used in the TLP and TLP-B. This is a function specifically used for snapshot voting where it pulls the balances of the NFTs that have been delegated to a specific address, based on a specific token. If multiple NFTs have been delegated to the same address, it will return the sum of all of those NFT balances for the same token address.

2.4.3 Internal Read Functions

```
function endDate(uint256 start, uint256 amount, uint256 rate, uint256
period) internal pure returns (uint256 end)
```

The is part of the TimelockLibrary used by the contract to calculate the end timestamp based on the input parameters. This function is used by the 'planEnd' function to calculate the end for a specific planId.

```
function validateEnd(uint256 start, uint256 cliff, uint256 amount, uint256
rate, uint256 period) internal pure returns (uint256 end, bool valid)
```

This function is part of the TimelockLibrary used to calculate the end date, and also verify that the inputs do not violate critical contract checks. It is used when minting a new Plan each time so that it passes the require statements in a single validation function.

```
function balanceAtTime(
    uint256 start,
    uint256 cliffDate,
    uint256 amount,
    uint256 rate,
    uint256 period,
    uint256 currentTime,
    uint256 redemptionTime
  ) internal pure returns (uint256 unlockedBalance, uint256 lockedBalance,
uint256 unlockTime)
```

This function is part of the TimelockLibrary used by the contract to calculate the unlockedBalance, lockedBalance and unlockTime that each contract uses to calculate how

much to redeem, revoke and how to reset the start date and amount for the Plan. This function is called by the ‘planBalanceOf’ function to get the necessary data when redeeming and revoking tokens for users.
periods_elapsed = (redemptionTime - start) / period
*Balance = periods_elapsed * rate*
Then we take the minimum of the Balance and the amount, so that its never more than what is part of the vesting plan, and the remainder is calculated by the amount - balance. The unlocktime is the last time when a redemption happens, which if periods happen every second then it would be the block.timestamp, but otherwise it would be the most recent discrete time at the end of a period.

```
function _isApprovedDelegatorOrOwner(address delegator, uint256 planId)
internal view returns (bool)
```

This function is part of the PlanDelegator schema. It returns a true or false check for whether an address can delegate tokens or not for the planId. If the delegator address is the owner, then it returns true. If the delegator address is an approved Delegator, or Delegator Operator, then it will return true, and otherwise will return false.

2.5 Write Functions

2.5.1 Public Write Functions

PlanDelegator.sol functions:

```
function approveDelegator(address delegator, uint256 planId) public
virtual
```

PlanDelegation: This function is part of the plan delegation schema where an owner or operator of an NFT can approve another address to become a ‘delegator’, who can call the delegation functions on behalf of the owner of the NFT. This function setups the approval step for a single address mapped to a single plan to act as the delegator for that plan.

```
function setApprovalForAllDelegation(address operator, bool approved)
public virtual
```

PlanDelegation: This function sets an operator address to a boolean approved, whereby that address can act as a Delegator Operator on behalf of all of the plans and NFTs owned by a single address. The Delegator Operator can also call the approveDelegator function.

```
function approveSpenderDelegator(address spender, uint256 planId) public
virtual
```

```
_approveDelegator(spender, planId);
_approve(spender, planId);
```

PlanDelegation: This function combines the approveDelegator function and the standard approve internal function into a single function call.

```
function setApprovalForOperator(address operator, bool approved) public
virtual {
_setApprovalForAllDelegation(msg.sender, operator, approved);
    _setApprovalForAll(msg.sender, operator, approved);
```

PlanDelegation: This function combines the standard ERC721 function _setApprovalForAll and the Delegator Operator approval function into a single public function call.

2.5.2 External Write Functions

```
  function createPlan(
    address recipient,
    address token,
    uint256 amount,
    uint256 start,
    uint256 cliff,
    uint256 rate,
    uint256 period,
  ) external nonReentrant returns (uint256 newPlanId) {
```

This is the core function to create a new vesting plan. It takes all of the data required to mint an NFT, and creates the lockup plan Struct that contains all of the lockup plan data. Additionally this function pulls the Amount of Token into the contract for escrow. This function returns the newPlanId that is the token id of the NFT mapped to lockup plan, so that external contracts can use it.

Recipient: Is the address of the recipient and beneficiary of the vesting plan.

Token: the address of the ERC20 token that is vesting

Amount: the total amount of unvested + vested and unclaimed tokens (this is not the total amount of the plan that was initiated, this data is updated each time tokens are claimed to keep it current with the remaining amount to be vested and claimed in the plan)

Start: This is the date when the vesting plan begins, but each time a redemption occurs, this start date updates to the most recent redemption time.

Cliff: A date that represents a discrete cliff whereby the beneficiary cannot redeem any tokens prior to this date. This does not change.

Rate: The tokens per period rate which the tokens vest. If the period is 1 for example, then this rate represents the amount of tokens that vest each second. If the period is (86400 = 1 day), then this is the amount of tokens that vest in a single day.

Period: the seconds in each period that determine the vesting schedule. Some schedules vest tokens every second, others each day, and others on a 30 day plan. This allows for flexibility in the vesting schedules with longer frequency vesting periods.

```
function redeemPlans(uint256[] calldata planIds) external nonReentrant
```

This function lets a beneficiary of a or several lockup plans redeem and claim their tokens. It calls an internal function to perform necessary checks and requirements, and then will transfer the unlocked ERC20 tokens associated with the planIds input to the beneficiary. If any plans are fully unlocked and redeemed, the NFT will be burned in the process. Redeeming in this method will request a redemption of the entire available unlocked balance of the NFT, so up to the second it will be redeemed. (the redemptionTime = block.timestamp)

```
function partialRedeemPlans(uint256[] calldata planIds, uint256
redemptionTime) external nonReentrant
```

This function lets a beneficiary of a or several lockup plans redeem and claim their tokens. It calls an internal function to perform necessary checks and requirements, and then will transfer the unlocked ERC20 tokens associated with the planIds input to the beneficiary. If any plans are fully unlocked and redeemed, the NFT will be burned in the process. Redeeming in this method will request a redemption of the claimable balance up to the redemption time, which can be anytime prior to the current block.timestamp. If the redemptionTime does not return any amount that is unlocked, then nothing will be redeemed for that plan.

```
function redeemAllPlans() external nonReentrant
```

This function lets a beneficiary redeem all of their unlock plans at once. It calls an internal function to perform necessary checks and requirements, and then will transfer the claimed and unlocked ERC20 tokens associated with the planIds input to the beneficiary. If any plans are fully unlocked and redeemed, the NFT will be burned in the process. Redeeming in this method will request a redemption of the entire unlocked balance of the NFT, so up to the second it will be redeemed. (the redemptionTime = block.timestamp)

```
function segmentPlans(uint256 planId, uint256[] memory segmentAmounts)
external nonReentrant returns (uint256[] memory newPlanIds)
```

This function allows a beneficiary to segment and break apart their NFT and lockup plan into multiple pieces. The beneficiary inputs the planId of their NFT and the chunk size(s) of the amounts they want to segment. The segment sizes always have to be smaller than the original plan that is being broken up. This function will create a new NFT and lockup plan for each of the segments, and alter the original lockup plan to decrease in size based on the amount segmented. Users should input the segmentAmounts in order of smallest to largest size. This function will iterate through the segment amounts, and call an internal function to process security statements and then perform the segmentation. This function will return the array of the new plan IDs created and NFTs minted for each new segment lockup plan.

```
function segmentAndDelegatePlans(
    uint256 planId,
    uint256[] memory segmentAmounts,
    address[] memory delegatees
  ) external nonReentrant returns (uint256[] memory newPlanIds)
```

This function allows a beneficiary to segment a plan into multiple chunks and simultaneously delegate the segments to a delegatee for voting purposes. This will call the internal function of segment plan to process the segmentation, and then call an internal delegate function to delegate the plan tokens to the delegatee. This function will return the array of the new plan IDs created and NFTs minted for each new segment lockup plan.

```
function combinePlans(uint256 planId0, uint256 planId1) external
nonReentrant returns (uint256 survivingPlanId)
```

This function will let a beneficiary who is the holder of two lockup plans combine them into a single lockup plan - the two lockup plans may have been children of a previously whole plan after segmentation, or two lockup plans that share all of the same details, though the amount and rate may be different. One of the planIds (generally planId0) will be the surviving plan, while the other lockup plan will be deleted from storage and NFT burned. This function will return the planId of the surviving plan.

```
function delegate(uint256 planId, address delegatee) external nonReentrant
```

This function will delegate a specific NFT and Plan to a designated delegatee. If this is the Plans contract, then it will simply delegate the NFT with the ERC721Delegate function. If this is the V-TLP or V-TLP-B contract, then it will create a VotingVault (if not already created) and transfer the tokens on-chain to the VotingVault, and then delegate the tokens there to the specific delegatee.

```
function delegateAll(address delegatee) external nonReentrant
```

This function performs the same functions as the above **delegate** function, but performs it for all of the NFT lockup plans owned by a single beneficiary, and delegating to a single delegatee.

```
function setupVoting(uint256 planId) external nonReentrant returns
(address votingVault)
```

This function is used by the V-TLP and V-TLP-B contracts to setup a VotingVault, and transferring the tokens from the ERC721 contract to the VotingVault contract. When the VotingVault is setup, it will check if the beneficiary address has already delegated tokens from their wallet to another address, if it has then it will delegate the tokens in the vault to the delegate, if it has not, it will self-delegate the tokens to the beneficiary wallet. This function returns the address of the newly created VotingVault

2.5.3 Internal Write Functions

```
function _redeemPlans(uint256[] memory planIds, uint256 redemptionTime)
internal
```

The internal function of redeeming multiple plans. It takes the redemptionTime as an input. For the redeemPlans and redeemAllPlans function, this parameter is set to the block.timestamp. This checks to make sure that there is a redeemable balance, if there isn't that planId is skipped so that it does not revert. Then it iterates through and calls the _redeemPlan function that processes each individual plan redemption.

```
function _redeemPlan(
    address holder,
    uint256 planId,
    uint256 balance,
    uint256 remainder,
    uint256 latestUnlock
  ) internal {
```

This internal function redeems an individual lockup plan. It takes the address of the holder (ie beneficiary), the planId, and the values calculated by the _redeemPlans method that input the balance, remainder and latestUnlock. From here it performs a few safety checks and then transfers the balance to the holder address. It also updates the storage of the Plan struct so that

the amount is set to the remainder, and the start is set to the latestUnlock. If the remainder == 0, then it will burn the NFT and delete the Plan in storage.
For the VotingPlans, it performs a check to determine if the tokens are held externally in a VotingVault; if they are then it will withdraw and send the balance of redeemed tokens from the VotingVault rather than the ERC721 contract to the Holder address. If the NFT is burned, the VotingVault contract will self destruct.

```
function _segmentPlan(address holder, uint256 planId, uint256 segmentAmount) internal returns (uint256 newPlanId)
```

This is the internal function that will segment a single planId into two pieces. This function will mint a new NFT and planId that is created. The segment plan parameters will match the original plan, except the amount will equal the segment amount, and the rate will equal a pro rata rate based on the segmentAmount and the original plan amount. The original plan Amount is reduced by the segmentAmount, and the original rate is reduced similarly in proportion to the segment / rate. The altered plan Amount and segmentAmount will always equal the original plan amount, and similarly the altered plan rate and segment rate will always equal the original plan rate.

```
function _combinePlans(address holder, uint256 planId0, uint256 planId1) internal returns (uint256 survivingPlan)
```

This function combines two plans into a single surviving NFT and plan. This function will verify that the token address, start date, cliff date, period and holder are the same for both plans. It will check that the end dates are compatible via being the same or their parent end date being the same. Then it will combine the two plans by adding the two amounts and two rates together.
The V-TLP and V-TLP-B contracts also check if there is a VotingVault created for either plan, and then combine the tokens from the VotingVaults into a single VotingVault. This function then returns the survivingPlanId, the other planID is burned and deleted from storage.

```
function _delegate(address holder, uint256 planId, address delegatee) internal
```

This is the internal method to delegate an NFT or delegate VotingVault tokens to an delegatee. It checks that the holder is the owner, and then either delegates the NFT, or it will call the VotingVault contract address and delegate the tokens held in it to the delegatee. If there is no VotingVault setup, it will first setup a VotingVault contract and then delegate the tokens.

```
function _setupVoting(address holder, uint256 planId) internal returns (address)
```

This is the internal function to setup a voting vault. It will create a new contract of VotingVault.sol, transfer the ‘amount’ parameter of the tokens in the vesting plan to the VotingVault and then delegate them to an existing delegate or to the beneficiary wallet address.

PlanDelegation Functions:

```
function _approveDelegator(address delegator, uint256 planId) internal
virtual
```

This function is part of the PlanDelegator schema, which is the internal component of approving a single address mapped to a single plan to act as the Delegator. This function updates the storage variable mapping the planId to the delegator address. To turn this function off, the owner of the plan would have to call the function and set the delegator address to the 0x0 address to disable the Delegator.

```
function _setApprovalForAllDelegation(address owner, address operator,
bool approved) internal virtual
```

This function is part of the PlanDelegator schema, and is the internal function that updates the Delegator Operator, mapping it either to true or false based on the approved variable. When true, the operator can act on behalf of the NFT owner for all of their NFTs to delegate them, and approve other single plan delegators.

```
function _beforeTokenTransfer(
    address from,
    address to,
    uint256 firstTokenId,
    uint256 batchSize
  ) internal virtual override {
    super._beforeTokenTransfer(from, to, firstTokenId, batchSize);
    if (from != address(0) && to != address(0)) revert('Not
Transferable');
  }
```

This function is an internal override function that makes the NFTs not transferable by the NFT owners. This is only used in the TLP-B and V-TLP-B contracts.