Vesting Token Plans Technical Documentation

Smart Contract(s): TimeVestingTokenPlans.sol; TimeVestingVotingTokenPlans.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 employees and contributors, and most often those tokens that are granted will contain some legal vesting arrangement. This is common in the traditional startup space with equity plans, and for DAOs a token vesting plan is established for employees and early investors. Currently many DAOs keep these vesting plans on legal documents and excel sheets, but are not enforceable in any way on-chain. Hedgey's Vesting Plan smart contract tools aim to provide a decentralized tool and infrastructure to support the token vesting plans for DAOs, bringing the off-chain agreements on-chain, in a transparent, trusted and decentralized tool.

As with any plan, we imagine that a DAO has a group of people, or even the entire DAO itself, that have predefined and determined the token vesting schedules for those beneficiaries, and now just need a simple and elegant tool to generate those plans on-chain and administer them as necessary.

Functional Architecture:

The purpose of this tool is to automate and effectively manage the token vesting schedules and plans of DAOs. Unique to Hedgey's tool, we utilize the backbones of NFTs (ERC721) to create a vesting plan for each participant that is defined as an individual and unique NFT. Each NFT in the collection represents a single and unique vesting schedule between the DAO vesting administrator and the vesting plan beneficiary (the holder of the NFT). The NFT backbone empowers additional abilities to the contract, such as greater transparency, managing multiple vesting 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 vesting 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 vest in a linear schedule, vesting the same precise amount every discrete period, where the period is denominated in seconds, as defined by the vesting schedule. When a plan beneficiary wants to claim some or all of their vested tokens, they will interact with the smart contract to redeem whatever amount of tokens has been vested. Vesting schedules may contain a single cliff date, where any tokens prior to the cliff become fully vested in a chunk on the cliff date.

The DAO will establish a Vesting Administrator (which may be the DAO or multi-sig address that holds the tokens itself), who is responsible for revoking the plans should the DAO and beneficiary part ways prior to the fully vested date. The vesting admin can revoke tokens at any time prior to the fully vested date. There is an optional toggle as well, which allows for the vesting admin to transfer an NFT on behalf of the beneficiary to a different address. This is a key feature for new DAOs with an HR team that manages everything on behalf of the employees, and wants to safeguard in case employees lose their wallet private key, there is a backup to moving the plan to a different address in case of emergency. This is an optional toggle however. Otherwise the vesting plans are not transferable, and if the optional toggle is turned off, then the plans are never transferable.

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. **Vesting Plan Creators**
   Generally these are the DAO in general or its executive / HR team. This group will generate and agree upon the overall token vesting plan and then subsequent individual vesting schedules for employees and other beneficiaries. The creators are responsible

for creating the on-chain plans via creation of NFTs that represent each unique vesting schedule, which requires them to deliver tokens from the DAO treasury to the StreamVestingNFT smart contract.

c. **Vesting Plan Administrator**
The Plan Administrator is generally a group of individuals, the Plan Creators, or the collective DAO, that are responsible for revoking vesting plans when beneficiaries are no longer part of the DAO (ie fired or quit). The administrators have a huge responsibility, because they can revoke the vesting NFT from a beneficiary at any time before it becomes fully vested. When an NFT is revoked, it will result in the unvested tokens being delivered to the vesting administrator, and any un-claimed but vested tokens will be delivered to the beneficiary. Administrators can revoke a single NFT, or they can revoke multiple at the same time, which is useful if employees have received multiple grants and upon termination multiple vesting NFTs need to be revoked.

d. **Vesting Plan Beneficiaries**
Vesting plan beneficiaries are the end employee who is receiving the vesting tokens from the DAO. The cannot transfer their schedule to anyone else. The beneficiary can claim (redeem) any vested tokens, subject to the lockup period, which will pull tokens from the NFT contract and deliver them to the beneficiary wallet. The beneficiary can also delegate the tokens locked inside their vesting plan to an external wallet, specifically for snapshot voting strategies implemented by the DAO. Beneficiaries can redeem their tokens individually, as a group, or they can redeem and claim all tokens that are vested across all of their vesting NFTs.

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 basic vesting schedule. It is performed by the plan creator, who will input all of the data points surrounding the vesting schedule into the function, and create an NFT that is minted to the beneficiary representing their vesting schedule. This function will pull the DAO tokens from the creator’s balance and transfer them to the NFT smart contract. This function does not allow for the optional lockup period, it will create a vesting plan that does not include the lockup period. 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
H. the vesting administrator who is responsible for revoking the tokens, should the DAO and beneficiary part ways.

4. **redeemPlans**
   This function is used by the beneficiary. This is the core function that the beneficiary will call to redeem all of the tokens that have vested for a specific plan. They can enter in multiple plan IDs to the function to redeem tokens from more than one vesting plan. If all of the tokens have vested, then the NFT Plan will be burned when this function is called. If there are no tokens vested 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 vesting plan NFTs. This will pull all of the vesting NFTs the beneficiary owns, and only redeem any tokens that have vested (ie ignore any schedules with 0 vested tokens). If the tokens are fully vested, then the NFT will be burned.
6. **partialRedeemPlans**
   As an alternative to redeeming plans where the beneficiary can pick a discrete time amount to redeem anything vested up until that time. The benefit of this is nuanced as it relates to taxes and operationally redeeming only a specific amount of vested tokens that might be required as opposed to the entire available vested balance.
7. **revokePlans**
   This function is the core function used by the vesting administrator. It will take a single or multiple NFTs as inputs. This function revokes the vesting NFTs - burning the NFTs and delivering any un-vested tokens to the vesting admin and any vested tokens to the beneficiary.
8. **transferFrom** (*override*)
   The standard transferFrom function is overridden for the vesting plans, allowing the special case for the vesting administrator to transfer a vesting plan NFT from the original beneficiary and owner, to a new wallet owner. This is only possible if the toggle to allow transferring on behalf of (OBO) is set to true, otherwise it is not allowable to transfer.

II. **Voting Functions**

9. **delegate**
   This function is used by the beneficiary who wants to delegate the tokens vesting inside their NFT to another wallet address used 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 the delegatee.

10. **delegateAll**
    This function is used by the beneficiary who wants to delegate all of their NFTs to a single wallet address for the voting purposes. This iterates through all of the owned plans of a single user and then delegates them, either delegating the NFT or delegating the tokens in a separate voting vault.
11. **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.

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 two versions of the Vesting Plan smart contracts; TimeVestingTokenPlans (“Plans”) and TimeVestingVotingTokenPlans (“VotingPlans”), collectively called “VestingPlans”. TimeVestingVotingTokenPlans is built for on-chain governance voting, while the former is optimized for snapshot voting. Both 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 vesting 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 vesting period, however when the users of the voting contract setup tokens for on-chain voting, the tokens are physically moved and segregated to a separate and unique voting vault. Each NFT has one and only one voting vault in this architecture, and the vault can only delegate tokens (it cannot use them in any defi activities).
This NFT backbone allows the smart contract to actively manage multiple vesting schedules within the same smart contract - allowing for better and more efficient management by the vesting administrators and the beneficiaries. The contract holds tokens that are vesting in the smart contract as escrow - pulled to the contract when new vesting NFTs are created. When beneficiaries redeem NFTs or vesting administrators revoke NFTs, the tokens in the contract are transferred out to the beneficiary, or to the admin, or both based on what has been vested and what remains unvested. The contract combines an NFT with a special struct that stores all of the data points of each unique vesting schedule - each vesting schedule can be described with a single integer identifier, that maps to the struct that contains all of the necessary information

about the vesting schedule, such as the recipient, token to be locked, total amount vesting, start date of vesting schedule, cliff date, rate that tokens vest per period, the length of the period in seconds, and the vesting admin.

2.1 File Imports and Dependencies

Both 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 VestingPlans. This is the standard imported from Open Zepellin, however the VestingPlans override the transferFrom functionality and the safeTransfer functionality. All other functions are left as is. Each individual plan within the VestingPlans 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 VestingPlans. It performs checks and calculates the end date of a vesting plan, as well as the balance of a vesting 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.

- **VestingStorage.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.

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:

- **@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol**
  The OpenZeppeling ERC721Enumerable contract allows for tokens to be enumerated over for a single wallet that owns many. This allows better functionality for functions that would make a change to an owner who owns multiple vesting plans.

- **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;
  address vestingAdmin;
  bool adminTransferOBO;
}

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 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.
vestingAdmin: The address of the administrator who can revoke the tokens at anytime prior to them being fully vested.
adminTransferOBO: This is a boolean toggle to allow the vestingAdmin to transfer a plan on behalf of (OBO) the plan beneficiary. This is useful for HR teams with a larger group of employees that do not have self custody or would allow for their HR team to transfer the tokens in the case the beneficiary loses their private keys or access to their wallet address.

votingVaults (*mapping for Voting Plans)*:

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

This is specific for the VotingPlans. 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 vested, the remainder which is unvested, 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 vests 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 vesting plan.

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

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

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 vesting 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 Plans contract (not VotingPlans). 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.

2.5 Write Functions

2.5.1 Public Write Functions

transferFrom

```
function transferFrom(address from, address to, uint256 tokenId) public
override(IERC721, ERC721)
```

This is the standard transferFrom function that is overridden. It explicitly allows for the vestingAdmin to transfer an NFT on behalf of its owner IF the Plan allows for it to be transferred on behalf of the owner. Otherwise it is not transferable.

2.5.2 External Write Functions

```
  function createPlan(
    address recipient,
    address token,
    uint256 amount,
    uint256 start,
    uint256 cliff,
    uint256 rate,
    uint256 period,
    address vestingAdmin,
    bool adminTransferOBO
  ) external nonReentrant {
```

This is the core function to create a new vesting plan. It takes all of the data required to mint an NFT, and create the vesting plan struct that contains all of the vesting plan data.

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.

vestingAdmin: The address of the administrator who can revoke the tokens at anytime prior to them being fully vested.

adminTransferOBO: This is a boolean toggle to allow the vestingAdmin to transfer a plan on behalf of (OBO) the plan beneficiary. This is useful for HR teams with a larger group of employees that do not have self custody or would allow for their HR team to transfer the tokens in the case the beneficiary loses their private keys or access to their wallet address.

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

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

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

This function lets a beneficiary of a or several vesting plans redeem and claim their tokens. It calls an internal function to perform necessary checks and requirements, and then will transfer the vested ERC20 tokens associated with the planIds input to the beneficiary. If any plans are fully vested and redeemed, the NFT will be burned in the process. Redeeming in this method will request a redemption of the balance vested 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 vested and unlocked, then nothing will be redeemed for that plan.

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

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

```
function revokePlans(uint256[] memory planIds) external nonReentrant
```

This function lets the vesting administrator revoke one or several vesting plans. It will fail if the vesting plan is fully vested already and nothing to revoke. This calls an internal revoke method to perform checks and then processes the revoking; where tokens that are unvested will be withdrawn and returned to the vestingAdmin, and tokens that have vested up to this point in time will be transferred to the beneficiary. The NFT and vesting plan will be burned and deleted from this method.

```
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 VotingPlans 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 vesting plans owned by a single beneficiary, and delegating to a single delegatee.

```
function setupVoting(uint256 planId) external nonReentrant
```

This function is used by the VotingPlans contract 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.

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 vesting 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 _revokePlan(address vestingAdmin, uint256 planId)  internal
```

This is the internal function to revoke a plan. It performs necessary checks to ensure only the votingAdmin is calling the function, and that there is a remainder that is unvested that can be revoked. Once these safety checks have been validated, then it will burn the NFT and delete the Plan in storage. Then it will return the remainder unvested token balances to the vestingAdmin address, and the balance of the vested tokens will be delivered to the beneficiary. If this is a VotingPlans contract and there is a VotingVault, it will withdraw the tokens from the VotingVault address rather than itself, and the VotingVault will self destruct.

```
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.

```
function _safeTransfer(address from, address to, uint256 tokenId, bytes
memory data) internal override {
    revert('Not transferrable');
  }
```

This function is an internal override function that makes the NFTs not transferable by the NFT owners.