Hedgeys – FuturesNFT.sol Technical Documentation

Contract: FuturesNFT.sol

Background: Hedgey Finance is the group that develops token tools for DAOs to utilize their treasury tokens along various life cycle events of the DAO, from inception to maturity. A DAO, generally known as Decentralized Autonomous Organization, will be used generally as a term to describe a group of people in the "Web3" ecosystem, namely building products in the Ethereum and similar EVMs Blockchains. DAOs often have a Token, generally an ERC20 standardized Token, that represents some form of governance, purpose, or just general community fun that is associated with the DAO itself. DAOs use these tokens in many different ways, and this specific protocol is targeted at assisting DAOs responsible distribute tokens from their treasury to contributors and investors via a token locking mechanism. Differently than the standard Open Zeppelin ERC20 Token Vesting / Locking contract, however, the FuturesNFT protocol locks tokens and wraps them inside of an NFT. We call this contract the Hedgeys for brand awareness, but the FuturesNFT is an NFT contract (Standard ERC721 Enumerable) that represents an amount of tokens unlockable in the future.

Why an NFT?

The locking of tokens inside an NFT has many advantages of the standard token locking mechanisms. Firstly, the NFT contract can store different tokens, and store tokens for more than one user, unlike the standard vesting contracts. Additionally, the NFT, being owned by a wallet, can be used in other cases where the NFT primitive is used such as; voting with the locked tokens, borrowing against them, and of course selling the NFTs in a secondary OTC marketplace while the tokens remain locked. As a fun element, the Hedgeys contract adds in custom artwork for each NFT which gives them a fun element that owners can view and feel ownership of beyond a simple vesting or locking contract.

Functional Overview

Purpose: The purpose of the Hedgeys (FuturesNFT.sol) is to make it easy and simple for DAOs to distribute locked tokens to contributors, investors, community members, and other parties in the ecosystem – even including other DAOs. The contract plays a pivotal central role in the broader smart contract architecture and ecosystem of Hedgey's protocols. The contract is built to be simple and modular, and usable by other protocols, both developed internally and externally by other partnering teams. For example, in the other HedgeyOTC.sol protocol associated with this contract, DAOs can offer to sell their tokens to investors with a particular lockup period so that when tokens are purchased the buyer actually receives a Hedgeys NFT rather than the underlying tokens – which they can unlock the tokens in the future. Additionally, we have a protocol for batch minting the NFTs in large groups, which can be used for airdrops, or monthly contributor compensation, or to investors paying in cash off-chain with

a special cliff vesting schedule. These are just some of the ways the NFTs are currently and can be used, but the protocol is built in this modular very simple sense so that it can be integrated in a number of ways and protocols – all with the utmost safety by using very simple logic and requirements in the smart contract.

Roles: There are generally three roles in the FuturesNFT protocol, the deployer of the contract, a Creator of NFTs, and a Holder of NFT(s). We'll refer to them as the Deployer, Creator, Holder:

a. **Deployer:** The Deployer is the developer on the Hedgey Team who deploys the NFT smart contract. Their main role is to deploy the contract and immediately update the baseURI used for linking each NFT to metadata & artwork. The contract is setup such that the URI can only be set once, which it is set to the standard of `https://nft.hedgey.finance/${network_name}/${contractAddress}/`. This is the current convention that Hedgey uses for its metadata storage – and the function to set this URI is immediately called as soon as the contract is deployed. This function does not have any influence to the underlying financial logic of the contract, and therefore is not a high risk security concern.

b. **Creator:** The Creator is the DAO, or team, or individual, who creates and mints an NFT. The NFTs are minted through a simple function called createNFT. The creator is responsible for locking the ERC20 tokens into the contract, setting the unlock date, and assigning the NFT to a specific Holder. The Holder can be the same address as the Creator, in which case the Creator is minting an NFT to themselves. This role can also be filled by other smart contracts that utilize the token locking mechanisms; such as the HedgeyOTC or BatchNFTMinter contracts, which call this function when they are locking tokens and distributing the NFTs to addresses based on their contract logic. In all cases the Creator must ensure they have sufficient balances of the ERC20, and have provided sufficient allowance to the FuturesNFT.sol contract address before taking any actions.

c. **Holder:** The Holder is the recipient of an NFT. This is generally an individual, but may also be a DAO, or other smart contract that is utilizing the functionality of the locked tokens. The Holder will receive an NFT after the Creator has created and minted an NFT for them – they do not have to call any contract interactions to simply receive the NFT. However, after the unlock date has passed on their futures NFT, they can call a function to redeem the NFT. The redeemNFT function will burn the NFT (thus destroying any linkage to the metadata and artwork) and remit the locked tokens to the owner of the NFT – the Holder.

Functions: There are three external writable functions, one for each of the roles as we will lay out below. There are several other internal and view only functions that are outlined in the technical overview section.

1. updateBaseURI
   This function is called immediately after deployment by the Deployer role. As outlined above it simply changes the baseURI of the contract to point at the new hyperlink. This has no financial implications to it. The baseURI is a reference point for each NFT, as is standard for ERC721 contracts. Each token is associated to a specific tokenURI which is the concatenation of the baseURI + tokenID. With this function, each tokenID is linked to the metadata folder location where Hedgey Finance stores all of its metadata and artwork. This function simply updates the link to point to the proper storage place, given that the contract address is a necessary input, the function is called after deployment and creation of the contract. Once this function has been called once, it cannot be called again by the simple use of a counter that is set to 1.

2. createNFT
   This function is the primary function for the Creator role. The Creator will call this function to crate and mint an NFT. The Creator will input the address of the Holder, the amount of ERC20 tokens to lock, the ERC20 token address, and the unlock date (represented as block timestamp ie Unix timestamp in seconds). When called, the function will pull tokens from the Creator's wallet, create the locked tokens struct called a Future, held in storage, with the details input, and additionally mint an ERC721 NFT token to the Holder.

3. redeemNFT
   This function is the primary function called by the Holder. The Holder can call this function ONLY after the unlock date has passed. Once the unlock date has passed, the Holder can call this function to redeem and unlock their tokens. This function will first burn the NFT, and then return the locked tokens held by the NFT back to the Holder. After this both the Future and the NFT are burned / deleted from storage.

Technical Overview:

Contract Dependencies & Imports

1. OpenZeppelin **ERC721.sol**
   Purpose: Base standard ERC721 NFT contract is used as the primary standard for the NFT contract.

2. OpenZeppelin **ERC721Enumerable.sol**

Purpose: Adds in enumerable features for easy viewing of the NFTs owned by a single Holder, should they own multiple

3. OpenZeppelin **Counters.sol**
   Purpose: Convenient and easy to use counting tool for incrementing tokenIDs

4. OpenZeppelin **ReentrancyGuard.sol**
   Purpose: Prevent reentrancy attacks for contract function calls that transfer tokens and change storage

5. **TranferHelper.sol**
   Purpose: Assist with transferring tokens between Creators and Holders and the smart Contract. This is a helper library that ensures the correct amount of tokens is transferred and that the before and after balances match with what is the expected outcome. Additionally the library checks to ensure sufficient balances are owned by the transferrer prior to attempting to transfer the tokens.

Global Variables

1. *string* **name**
   This is the string name of the collection, which is set to 'Hedgeys'.

2. *string* **symbol**
   This is the string symbol of the collection, which is set to 'HDGY'.

3. *address payable public* **weth**
   This is the address of the contract 'WETH' or Wrapped Ether, which assists with storing Ether in this contract as an ERC20. It was chosen to use WETH to store in the contract to make moving tokens back and forth consistent – as well as ensuring that the proper amount of ETH is received when creating an NFT.

4. *string private* **baseURI**
   This is a string of the hyperlink that is the baseURI. The baseURI is used to link a specific NFT with its tokenID to a metadata location. The baseURI + tokenID concatenation can be used to lookup the metadata of a given NFT, assuming the NFT exists. If the NFT does not exist then the tokenURI will not return.

5. *uint8 private* **uriSet**
   uriSet is a simple mechanic to ensure that the baseURI can only be set once. It is set to 0 upon deployment of the contract, and then set to 1 as soon as the URI has been properly reset.

6. *uint256 private* **_tokenIds**
   _tokenIds are inherited from Counters. These are used as the unique individual tokenID for each and every NFT. The _tokenId is also associated with the Future that ties the NFT to its locked token storage position.

7. *struct* **Future**
   Future is a struct that creates the locked token position, and is held in storage. The Future struct contains the following three data pieces:
   uint256 amount: this is the amount of tokens locked inside the NFT
   address token: this is the address of the ERC20 token that is locked inside the NFT
   uint256 unlockDate: this is the Unix timestamp (in seconds) that represents the moment the tokens can be unlocked. It is checked against the block.timestamp for unlocking.

8. *mapping (uint256 => Future) public* **futures**
   this is a mapping of the _tokenId to each struct Future, which is a public getter function futures. The mapping ties the NFT to its dedicated Future, held in storage. Because this is a public getter function users can input a tokenID to view the underlying Future details of their locked tokens.

Write Functions

1. *function* **updateBaseURI**(*string memory* _uri) external
   This function is called only once by the Deployer role and sets the baseURI to the new _uri. Once this is called, it cannot be called again for security purposes – although the function serves no financial uses.

2. *function* **createNFT**(*address* _holder, *uint256* _amount, *address* _token, *uint256* _unlockDate) external nonReentrant returns (*uint256*)
   This is the core function used to create and mint an NFT. The function takes 4 key arguments. The function will pull tokens into it, ensuring that the proper amount is always held inside the contract. In the case that Ether is delivered, the TransferHelper library will help convert the Ether in the msg.value into WETH and then ensure that the correct amount of WETH is stored inside of the contract. The function serves the purpose of incrementing the _tokenIds of the NFT by 1, then pulling in the necessary tokens that are to be locked, minting an NFT to the _holder, and finally creating a Future in storage with the details of the _amount, _token, and _unlockDate. It then emits an event with all of the details in the function parameters and the current tokenId.
   a. _holder: is the recipient of the NFT. This is designed intentionally so that the msg.sender can be another smart contract, DAO, or individual who wants to send a locked token position to someone other than themselves. If they want to

mint themselves an NFT they can simply input their address into this field. The _holder is then minted the NFT, and can do with that NFT any of the functions inherited from the standard ERC721 contract, such as transfer, approve allowance.

b. _amount: this is the amount of tokens that the Creator is going to lock up inside the NFT. The Creator should have already called the 'approve' or 'increaseAllowance' function on their ERC20 contract to ensure that they have sufficient allowance for the contract to pull the amount of ERC20 tokens into the contract.
c. _token: is the ERC20 contract address of the tokens that are to be locked
d. _unlockDate: is the Unix timestamp representing the date in which any time after the tokens may be redeemed and unlocked. Until this date has passed the tokens cannot be redeemed or unlocked.

3. *function* **redeemNFT**(*uint256* _id) external nonReentrant returns (*bool*)
This is a core function used to redeem the tokens locked inside an NFT. The redeem function itself calls another internal _redeemNFT(address payable _holder, uint256 _id) function – where the msg.sender is passed in as the _holder, and _id passed in as the _id. This lets the internal function handle all of the requirement and logic necessary to ensure that the NFT is owned by the msg.sender, and that the tokens are ready to be unlocked. The internal function also burns the NFT and deletes the Future in storage that maintained the locked token position. Finally, the function will withdraw the tokens from the contract and deliver to the _holder (msg.sender). If the tokens are ETH, then the TransferHelper library will convert the WETH back into ETH and deliver it to the _holder (except in the case where the _holder is a smart contract – and then for safety it will just transfer WETH as an ERC20 – this prevents unwanted locked ETH inside of smart contracts that do not have special ETH handling).

Inherited Functions

Please see the documentation for the Openzeppelin contracts of ERC721.sol and ERC721Enumerable.sol for functions inherited into this contract, found here:
https://docs.openzeppelin.com/contracts/4.x/api/token/erc721#IERC721