Hedgey DAO Swap Technical Documentation

Contract: HedgeyDAOSwap.sol

Background: Hedgey Finance is the group that developed the Protocol for the general purpose of two parties, typically DAOs, to swap tokens with each other. 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, but Hedgey's Protocol looks to tackle and assist two DAOs interacting in a very specific way with each other, in what is commonly referred to as a "DAO to DAO Swap" (D2DSwap for shorthand). A D2DSwap is an event that occurs when one DAO swaps some amount of their tokens with another DAO's tokens, exchanging a bulk group of ERC20 tokens for another chunk of ERC20 tokens. Of course any two parties can participate and use the open protocol for this purpose, but it is generally targeted towards DAOs performing this one function. The protocol aims to maximize security and efficiency for DAOs to undergo such a swap.

A Swap is generally defined by a few parameters, the ERC20 token and amounts that each DAO wishes to swap with each other, of course the two DAOs themselves, and in some instances a locking mechanism whereby the tokens once swapped are locked inside another contract until a certain date, at which point both DAOs can unlock and receive the tokens they swapped for and utilize as they please.

Functional Overview

Purpose: The purpose of HedgeyDAOSwap is to automate the D2DSwap process on behalf of two DAOs. It handles standard ERC20 tokens that are swapped between two unique addresses, and can lock those tokens into a vesting contract on behalf of the DAOs as well. There are no fees or tokenomics related to this contract, it does not cost anything for two DAOs to interact and use the contracts beyond standard EVM gas costs.

Roles: There are two roles in a D2DSwap, the initiator and the executor:

a. **Initiator**: This DAO is the one who will setup the Swap based upon their agreed parameters. This means that they are responsible for inputting all of the necessary information, such as Token Addresses of each DAO, the amounts that are to be swapped, the address of the counterparty DAO (executory) and any locking details should the tokens need to be locked upon swapping.
b. **Executor**: This DAO will execute the swap, after confirming all of the details accurate setup by the Initiator, they will execute the swap, delivering each other's tokens to the appropriate DAO, or delivering each other an NFT representing their ownership of locked tokens.

Functions: There are public global variables that are accessible through the standard public "getter" functions, which can read the elements inside a particular Swap to confirm its validity.

There are three Write functions that will alter the storage state of the contract, and transfer tokens from one party to another:

1. initSwap

   This function will initiate the Swap. This is performed by the Initiator DAO, who inputs all of the parameters of the Swap into the function. This function will update the storage of the contract, and pull funds from the Initiator DAO that they will be delivering into the contract, until it is executed (or cancelled).

2. executeSwap

   This function is callable only by the Executor DAO in the Swap. This function just requires that the DAO knows the index that represents the Swap and is mapped to it in the storage of the contract. This function will transfer the tokens from the Executor DAO to the Initiator DAO, and will withdraw the tokens initially sent by the Initiator DAO into the contract and deliver those tokens to the Executor DAO. In the case that the tokens have a future unlock date, the tokens will both be locked into the NFT Locking contract (FuturesNFT.sol), and then two NFTs minted, one to each DAO representing their ownership of the locked tokens that they received from the Swap.

3. cancelSwap

   This function is callable only by the Initiator DAO after they have already called the initSwap function and created a Swap for the contract in storage. This function is used to cancel a Swap that has not been executed yet, in the case that a mistake was made or the DAOs diverged from their original agreement and the Swap is no longer executed. This will cancel the Swap so that it cannot be executed anymore, thus deleting the Swap in storage, and withdrawing the tokens held by the contract and delivering back to the Initiator DAO.

Technical Overview:

Contract Dependencies & Imports:

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

2. TransferHelper.sol
   Purpose: Assist with transferring tokens between users and contracts. 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.

3. NFTHelper.sol

Purpose: Assist with locking tokens into a FuturesNFT contract, which mints an NFT and locks tokens with a precise unlock date. The NFT acts as the key for the owner to unlock the tokens after the unlockDate has been passed.

Global Variables

1. uint256 swapId
   Purpose: This is an indexer. It's purpose is simple to count and index the swaps. This indexer is important and used to call the executeSwap and cancelSwap functions. The DAO executing or DAO cancelling needs to know the index of the Swap in order to successfully call those functions.
2. Struct Swap
   Purpose: The struct is the way in which the contract stores each Swap in storage, which has the following elements inside the struct that collectively define a single Swap:
   a. tokenA is the address of the tokens that the initiator DAO will be delivering (and executor DAO will receive)
   b. tokenB is the address of the tokens that the executor DAO will be deliveriny (and the initiator DAO will receive)
   c. amountA is the amount of tokenA that the initiator DAO will deliver (and the amount executor will receive)
   d. amountB is the amount of tokenB that the executor DAO will deliver (and the amount the initiator will receive)
   e. unlockDate is the block timestamp for when the tokens will unlock. if this is set to 0 or anything in the past the tokens will not be locked upon swap
   f. initiator is the initiator DAO address who will initialize the swap and will deliver amountA of tokenA
   g. executor is the executor DAO address that will execute the swap and deliver amountB of tokenB
   h. nftLocker is an address of the Hedgeys NFT contract that will lock the tokens IF the unlock date is in the future
3. mapping(uint256 => Swap) public swaps
   Purpose: This mapping maps the swapId to the Swap (struct) held in storage so that we can access the information held there easily via a public getter function. This is useful for DAOs to confirm the parameters of the Swap are correct before executing, as well as necessary to execute the swap via executeSwap or cancel an initiated swap via cancelSwap function.

Read Functions

The only read functions are the ones generated by the contract via the "getter" functions of the public global variables:

1. function swapId() public view returns (uint256)
   => This function returns the next index of the swapId that will be generated for the next initiated Swap

2. function swaps(uint256 _swapId) public view returns (Swap memory swap)
   => This function returns the Swap struct based on the index input into the function. This is very useful for confirming the information stored in the Swap struct is accurate prior to executing a swap. This function is called by the contract when executing or cancelling to perform the actions of actually swapping tokens, based on what is stored in Storage by this Struct at this _swapId index.

Write Functions

1. function initSwap(address tokenA, address tokenB, uint256 amountA, uint256 amountB, uint256 unlockDate, address executor, address nftLocker) external
=> This function is the function that will initiate the Swap. The DAO that calls this function becomes the Initiator in the roles, and defines all of the parameters of the Swap. These inputs match exactly to the Swap Struct that we defined earlier, with the parameter of initiator being defined as the msg.sender (ie the DAO that calls the function). Thus the function performs the following functions:
A. The DAO calling this becomes the Initiator
B. The function pulls in the amountA of tokenA from the msg.sender (Initiator) DAO
C. The function creates the Swap Struct and stores it in storage at the current indext of swapId
D. The function emits an event that sends out all of the parameters of the Swap defined

2. function executeSwap(uint256 _swapId) external
   => This function executes the Swap. It pulls from storage the Swap mapped by the _swapId, and then places that into memory. Next it checks that the msg.sender is the Executor defined in the Swap Struct. For security, it will next delete the global storage struct, while still maintaining the Struct in memory for further processing.
Then it process the Swap itself. If the unlockDate of the Swap is in the future, it will lock the tokens by first, pulling the amountB of tokenB from the Executor DAO into the contract. Then it will lock amountB of tokenB with the nftLocker (a unique Hedgey Contract that locks tokens inside an NFT), minting an NFT for the Initiator DAO. Next it will lock amountA of tokenA with the same nftLocker, and mint the NFT representing those locked tokens to the Executor DAO, thus Initiator DAO having a locked tokens NFT of tokenB in the amount of amountB, and ExecutorDAO having a locked tokens NFT of

tokenA in the amount of amountA.
If the contract does not require a lock because the unlock date has already passed (or is set to 0 as is customary), then it will simply swap the tokens, delivering the amountA of tokenA to the Executor DAO and the amountB of tokenB to the Initiator DAO. This function also emits an event to signify its success.

3. function cancelSwap(uint256 _swapId) external
=> This function allows the Initiator DAO to cancel a swap that has not been executed, but has already been initiated and is stored in a Swap Struct in Storage. The function can only be called by the Initiator DAO. The function will get the Swap Struct stored in storage by the _swapId, and then will place that in memory to perform the rest of the actions. First it will check that the Initiator DAO is the one calling the function, checking by looking at the msg.sender. Next it will delete the Swap stored in storage for safety. Then it will withdraw the tokens that the Initiator originally delivered into the contract when it called the initSwap function. It will finally emit an event to signify the cancellation. With the deletion of the swap, it can no longer be accessed or used by any party, its storage slot is empty.