PlanDelegator Contract Updates

The PlanDelegator.sol smart contract is an efficient way to allow third party wallets or smart contracts to manage your delegations on your behalf. The idea behind this is to give permissions to your NFT vesting and lockup plans as it relates to delegation of the underlying tokens, similar to giving approval permissions to spend your NFT or approveAll of your NFT spend actions. The architecture is almost identical to the well worn approval of the ERC721 standard, using an approve function, and a setApprovalForAll style function to allow another wallet to manage your delegations. The storage variables allow for the delegate function to check whether the msg.sender is an owner of the NFT, or is an approved delegator. In this context the delegator is the wallet that has been approved to delegate tokens on behalf of the underlying NFT owner.

Business Case:

First this will allow for a wider range of use cases and applications to build ontop of the Hedgey protocols to assist with management of vesting and lockup plan delegations. For example, a employee can store an NFT in cold storage and call a single function from cold storage to approve a delegator (or operator) of their 'hot wallet' or even an external contract, that can now manage their delegations of the NFT(s) held by the employee in Cold Storage. This ensure security of the NFT, and separates the financial risk with the delegation of the tokens, which may be a more frequent activity depending on the vibrancy of the governance of the token ecosystem.

Additionally, it would allow for owners of locked tokens to effectively sell their delegation rights to another wallet, without having to sell the locked tokens themselves. This could enable a more diverse ecosystem of delegation and bribe markets to arise by the possibility of token delegations being sold, and separated at the smart contract level from the financial aspects of the locked tokens.

Another similar use case is in the case of a Custodian relationship with multiple owners of locked token plans, all held by the same Custodian. A custodian could effectively store all of the locked token positions in a single contract wallet, but give approvalForAllDelegation ability to a different smart contract wallet that handles delegation specifically. This mechanism, which the Custodian would have to build and still manage their individual relationships to call functions on behalf of specific tokens, can allow a broader and more effective management of the locked positions and delegations, while adding in the security benefits of separating the financial jobs and the delegation jobs.

Technical Architecture:

The technical architecture is very similar to the standard ERC721 approval and operator system.

Storage Variables:

```
mapping(uint256 => address) private _approvedDelegators;

function approveDelegator(address delegator, uint256 planId) public
virtual

function _approveDelegator(address delegator, uint256 planId) internal
virtual

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

_approvedDelegators is a mapping of the tokenId to a specific address that can delegate on its behalf. This function acts similar to the ***_tokenApprovals*** in the ERC721 contract. This variable is updated when an owner or operator calls the public function approveDelegator, which triggers the internal function (after some checks) to update the mapping in storage and emit an event. If the user wants to revoke the delegation, they do so by setting the zero address as the delegator address.

Then there is a public view function that returns the address of the Delegator, which is used in a bigger internal function to evaluate if the msg.sender is approved or owner.

```
mapping(address => mapping(address => bool)) private
_approvedOperatorDelegators;

function setApprovalForAllDelegation(address operator, bool approved)
public virtual

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

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

_approvedOperatorDelegators is the other storage variable that mirrors the ***_operatorApprovals*** variable in the ERC721 standard. This variable stores the boolean for when an address sets another address as its operator delegator, which has all of the rights and abilities of the owner of the NFT itself. The Operator Delegator can approve another Delegator with the approveDelegator function, it can also set delegations itself because the isApprovedForAllDelegation view function will return a boolean check when a delegation function is called to determine whether that msg.sender is a delegator operator. The variable is updated by the public setApprovalForAllDelegation function, which performs some checks and calls the internal _setApprovalForAllDelegation function. These functions do not require a msg.sender be an owner, because the msg.sender is directly passed into the argument with the boolean approved into the function, so it just checks that the owner is not the operator address. This can be turned off by calling the same function with the operator address and setting the approved parameter as false.

```
function _isApprovedDelegatorOrOwner(address delegator, uint256 planId)
internal view returns (bool) {
    address owner = ownerOf(planId);
    return (delegator == owner ||
      isApprovedForAllDelegation(owner, delegator) ||
      getApprovedDelegator(planId) == delegator);
  }
```

The final output is that we have an internal checking mechanism now to determine whether an address calling the external delegate or delegatePlans function is allowed to do so, by checking whether they are the owner, or an approved delegator, or an approved delegator operator.

This check is then added to the internal delegation functions in the TokenVoting (plans);

```
function _delegate(uint256 planId, address delegatee) internal {
    require(_isApprovedDelegatorOrOwner(msg.sender, planId),
'!delegator');
```

```
function _setupVoting(uint256 planId) internal returns (address) {
    require(_isApprovedDelegatorOrOwner(msg.sender, planId),
'!delegator');
```

And to the ERC721Delegate contract which is inherited by the TokenLockupPlans and TokenVestingPlans:

```
function _delegateToken(address delegate, uint256 tokenId) internal {
    require(_isApprovedDelegatorOrOwner(msg.sender, tokenId),
'!delegator');
    _transferDelegate(delegate, tokenId);
  }
```

***Note**: The **‘delegateAllPlans’** function should not generally be used by the operator because it will pull the NFTs owned by the msg.sender, it does not iterate through a list of wallets that have approved that wallet to perform delegator functions.*

Finally, we added a combined function that allows a wallet to approve another address as both the ‘spender’ and as the ‘delegator’, and similarly combining the setApprovalForAll and setApprovalForAllDelegation, so that if a person wants another contract to have entire control, or wants to allow an intermediary contract (like a sales platform) to both spend and delegate its token once the NFT is sold, it can do so because it has been approved for both activities. Adding this to a single function seemed efficient for an end user perspective rather than having to make two almost identical function calls.

```
function approveSpenderDelegator(address spender, uint256 planId) public
virtual {
    address owner = ownerOf(planId);
    require(msg.sender == owner || (isApprovedForAllDelegation(owner,
msg.sender) && isApprovedForAll(owner, msg.sender)), '!ownerOperator');
    require(spender != msg.sender, '!self approval');
    _approveDelegator(spender, planId);
    _approve(spender, planId);
  }
```

This will call the internal _approveDelegator and _approve functions to update the internal storage for both simultaneously. The check here is that the msg.sender must be the owner, or it must be the operator **and** delegator operator for the wallet.

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

This function sets the operator for both spending and delegating activities in a single transaction, calling both the _setApprovalForAllDelegation and as well the _setApprovalForAll function as inherited by the ERC721 contract standard.