SHERLOCK

# Security Review For Hydrex

Collaborative Audit Prepared For: **Hydrex**
Lead Security Expert(s): **jennifer37**
**oot2k**

Date Audited: **July 14 - August 4, 2025**
Final Commit: **62f3bdf**

## Introduction

Hydrex is the MetaDEX purpose-built for Base, with a frictionless UX that pipelines users directly into the ve(3,3) flywheel. By combining automated strategies, strategic reserve tokenomics, and Base's powerful distribution channels, Hydrex removes the barriers to yield, and opens the gates to onchain finance for everyone.

## Scope

Repository: hydrexfi/hydrex-contracts

Audited Commit: da0bef71cfb7995d81b2d827a556221d2a69a2af

Final Commit: 62f3bdf1c28e17a18a8cd0cd610f8c5477ca9da3

Files:

- contracts/Constants.sol
- contracts/GaugeV2_CL.sol
- contracts/GaugeV2.sol
- contracts/libraries/DelegateCallLib.sol
- contracts/libraries/Math.sol
- contracts/OptionToken/BribeOptionTokenV2.sol
- contracts/OptionToken/DynamicTwapOracle/IDynamicTwapOracle.sol
- contracts/OptionToken/DynamicTwapOracle/UniswapV3Twap.sol
- contracts/OptionToken/IFloorGuardian.sol
- contracts/OptionToken/IOptionTokenV4.sol
- contracts/OptionToken/OptionFeeDistributor.sol
- contracts/OptionToken/OptionTokenV3.sol
- contracts/OptionToken/OptionTokenV4.sol
- contracts/OptionToken/SimpleFloorGuardian.sol
- contracts/VoterV5/BribeV2.sol
- contracts/VoterV5/IVoterV5_ClaimHelper.sol
- contracts/VoterV5/IVoterV5_ClaimLogic.sol
- contracts/VoterV5/IVoterV5_GaugeLogic.sol
- contracts/VoterV5/IVoterV5_Logic.sol
- contracts/VoterV5/IVoterV5.sol
- contracts/VoterV5/IVoterV5_Storage.sol

- contracts/VoterV5/RewardsDistributorV2.sol
- contracts/VoterV5/VoterV5_ClaimLogic.sol
- contracts/VoterV5/VoterV5_GaugeLogic.sol
- contracts/VoterV5/VoterV5.sol
- contracts/VoterV5/VoterV5_Storage.sol
- contracts/VoterV5/VotingEscrow/IVotingEscrowV2_ApprovalLogic.sol
- contracts/VoterV5/VotingEscrow/IVotingEscrowV2_Data.sol
- contracts/VoterV5/VotingEscrow/IVotingEscrowV2_LockLogic.sol
- contracts/VoterV5/VotingEscrow/IVotingEscrowV2_Logic.sol
- contracts/VoterV5/VotingEscrow/IVotingEscrowV2.sol
- contracts/VoterV5/VotingEscrow/IVotingEscrowV2_Storage.sol
- contracts/VoterV5/VotingEscrow/libraries/Checkpoints.sol
- contracts/VoterV5/VotingEscrow/libraries/EscrowDelegateCheckpoints.sol
- contracts/VoterV5/VotingEscrow/libraries/EscrowDelegateStorage.sol
- contracts/VoterV5/VotingEscrow/VotingEscrowV2_ApprovalLogic.sol
- contracts/VoterV5/VotingEscrow/VotingEscrowV2_LockLogic.sol
- contracts/VoterV5/VotingEscrow/VotingEscrowV2_Storage.sol
- contracts/VoterV5/VotingEscrow/VotingEscrowV2Upgradeable.sol

## Final Commit Hash

**62f3bdf1c28e17a18a8cd0cd610f8c5477ca9da3**

## Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

## Issues Found

| High | Medium | Low/Info |
|---|---|---|
| 4 | 2 | 7 |

## Issues Not Fixed and Not Acknowledged

| High | Medium | Low/Info |
|---|---|---|
| 0 | 0 | 0 |

# Issue H-1: voting can be manipulated

Source: https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/issues/40

## Summary

voting can be manipulated

## Vulnerability Detail

In VoterV5, users can vote for different gauges via their holding ve token's voting power. The voting power depend on past voting power at the start of this epoch.

The problem here is that users can vote at the start of this epoch and then transfer their veToken to another account. Then users can use the same voting power to vote again.

This will cause that we can use the same veToken, same voting power to vote multiple times to manipulate the vote.

```
function _vote(address _voter, address[] memory _poolVote, uint256[] memory
↪  _voteProportions) internal {
    _reset(_voter);
    uint256 _poolCnt = _poolVote.length;
    // Here we have one timestamp.
    uint256 _time = _epochTimestamp();
    uint256 _weight = IVotingEscrowV2(_ve).getPastVotes(_voter, _time);
}
```

## Impact

Malicious users can manipuate the vote.

## Code Snippet

https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/9a1e74294d68a42ff49e2c4ac8c726dec0cc3dce/hydrex-contracts/contracts/VoterV5/VoterV5.sol#L611-L629

## Tool Used

Manual Review

## Recommendation

If the veToken is used to vote for this epoch, we should not allow this veToken transferred.

## Discussion

**lpetroulakis**

Issue has been fixed in PR #140

# Issue H-2: Fail to update `weightsPerEpoch` correctly when we re-vote

Source: https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/issues/41

## Summary

Fail to update `weightsPerEpoch` correctly when we re-vote

## Vulnerability Detail

When users re-vote in the same epoch, we will reset previous voting, and vote again. In `_reset` function, we will deduct the previous voting power from `weightsPerEpoch` if the previous vote is in current epoch.

The problem here is that if `lastVoted[_voter]` equals `_time`, we don't think this is one valid `votedInEpoch`. This is incorrect. Based on this incorrect `votedInEpoch`, we may fail to update the `weightsPerEpoch`. This will cause the incorrect `weightsPerEpoch` calculation and incorrect reward distribution.

```
    function _reset(address _voter) internal {
        address[] storage _poolVote = poolVote[_voter];
        uint256 _poolVoteCnt = _poolVote.length;
        uint256 _totalWeight = 0;
        // the start time for current period.
        uint256 _time = _epochTimestamp();
        bool votedInEpoch = lastVoted[_voter] > _time;
        for (uint256 i = 0; i < _poolVoteCnt; i++) {
            address _pool = _poolVote[i];
            uint256 _votes = votes[_voter][_pool];

            if (_votes != 0) {
@>                  if (votedInEpoch && isAlive[gauges[_pool]])
↪   weightsPerEpoch[_time][_pool] -= _votes;
                // clear voter's vote for this pool.
                votes[_voter][_pool] -= _votes;

                // if is alive remove _votes, else don't because we already done it
                ↪  in killGauge()
                // If this gauge is killed, we will deduct these votes, because
                ↪  when we kill gauge, we will
                if (isAlive[gauges[_pool]]) _totalWeight += _votes;
                emit Abstained(_voter, _votes);

                IBribe(internal_bribes[gauges[_pool]]).withdraw(uint256(_votes),
                ↪  _voter);
```

```
                IBribe(external_bribes[gauges[_pool]]).withdraw(uint256(_votes),
                ↪  _voter);
            }
        }
```

## Impact

Malicious users can manipuate the vote to gain more voting power via re-vote..

## Code Snippet

https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/9a1e74294d68a42ff49e2c4ac8c726dec0cc3dce/hydrex-contracts/contracts/VoterV5/VoterV5.sol#L552-L568

## Tool Used

Manual Review

## Recommendation

```
bool votedInEpoch = lastVoted[_voter] >= _time;
```

## Discussion

**lpetroulakis**

Issue has been fixed in PR #141

# Issue H-3: The `getMinPrice()` function uses `UNDERLYING_TOKEN` instead of `paymentToken` to calculate the floor price

Source: https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/issues/49

## Summary

The `getMinPrice()` function returns either the floor price or a discounted price from twap oracle. Currently the floor price is directly taken from the feeDistributor, which returns the floor price in payment tokens.

When calculating the floor price using decimals, the current code however takes the decimals of the underlying token instead of the payment token.
This leads, in case a token with less decimals (like usdc in current context), to an to low floor price.

## Vulnerability Detail

The getMinPrice() function determines the floor price using the UNDERLYING_TOKENs decimals.

```
function getMinPrice(uint256 _amount, uint256 _discount) public view returns
↪  (uint256) {
    uint256 floorPriceAmount = (getMinPaymentAmount() * _amount) / (10 **
    ↪  IERC20Metadata(address(UNDERLYING_TOKEN)).decimals());
```

https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/main/hydrex-contracts/contracts/OptionToken/OptionTokenV4.sol#L337

The `getMinPaymentAmount()` function however returns an amount in paymentToken decimals:

```
 * @return The minimum payment amount in payment token units
 */
function getMinPaymentAmount() public view returns (uint256) {
    return feeDistributor.floorPrice();
}
```

https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/main/hydrex-contracts/contracts/OptionToken/OptionTokenV4.sol#L322

In current context it is expected that:

- UNDERLYING_TOKEN is Hydrex, also named protocol token
- paymentToken is Base USDC

- option token is the option token contract

This means that hydrex which has 18 decimals will be used to calculate an usdc amount which has 6 decimals. The returned floor price will be much smaller then expected.

## Impact

The returned floor price will be much smaller then expected.

## Code Snippet

https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/main/hydrex-contracts/contracts/OptionToken/OptionTokenV4.sol#L337

## Tool Used

Manual Review

## Recommendation

Calculate the floor price using `paymentTokens` decimals instead of UNDERLYING_TOKENs.

## Discussion

**lpetroulakis**

Issue has been fixed in PR #153

# Issue H-4: Rebase in `RewardsDistributorV2.sol` can release unbacked tokens

Source: https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/issues/51

## Summary

The `claim` and `claim_many` functions in `RewardsDistributorV2.sol` either increase the balances of non-permalocked assets or release HYDX tokens directly into ciruclation upon rebases under the right conditions.

## Vulnerability Detail

As a rule of thumb, Protocol Tokens should not be released into circulation without the minimum floor price of the Options Token being acquired by the protocol. The current rebase mechanic creates a 2 situations where this is a possibility.

Situation #1 - This condition is met `_locked.endTime <= block.timestamp && _locked.lockType == IVotingEscrowV2_Data.LockType.NON_PERMANENT` and the liquid rebased tokens get minted directly to the end user. Situation #2 - There is a vanilla `IVotingEscrowV2(voting_escrow).increaseAmount(_tokenId, amount);` paired with a `NON_PERMANENT` which adds tokens to the liquid balance, and can be later acquired after the vesting time without paying the options token fee.

## Impact

Complete erosion of one of the core mecahnics of the system - requiring payment before issuing a liquid token.

## Code Snippet

RewardsDistributorV2.sol:187-199 and RewardsDistributorV2.sol:208-1223

## Tool Used

Manual Review

## Recommendation

Exclusively handle rebases always adding to or creating new permalocked positions.

## Discussion

**lpetroulakis**

Issue has been fixed in PR #148

# Issue M-1: Some users with permanent lock will be forced to claim rewards with one penalty

Source: https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/issues/43

This issue has been acknowledged by the team but won't be fixed at this time.

## Summary

Some users with permanent lock will be forced to claim rewards with one penalty

## Vulnerability Detail

In Hydrex, we have one new lock type, permanent lock. In option v3, users have to pay some base token to convert their oToken to ve lock. In option v4, users don't need to pay some extra base token, however, the lock type will be the permanent type. When we create one permanent lock, the voting power will be less than actual oToken amount, 1/1.3 option token amount. We can take this as some penalty for this kind of convertion.

veToken holders can claim some rewards via the reward distributor. And the claimable rewards should be related with this escrow id's voting power.

The problem here is that if the lock is one permanent lock, the rewards have to be increased to the lock amount, and we should notice that the rewards will be less than rolling lock, non-permanent. Because when we increase amount for the permanent lock, the actual amount will be less than input amount, 1/1.3. When users convert oToken to veLock, users have already taken some penalty to receve less locked amount. After this convertion, we should not keep deduct users' rewards.

```
    function claim(uint _tokenId) external returns (uint) {
        if (block.timestamp >= time_cursor) _checkpoint_total_supply();
        uint _last_token_time = last_token_time;
        _last_token_time = (_last_token_time / WEEK) * WEEK;
        uint amount = _claim(_tokenId, voting_escrow, _last_token_time);
        if (amount != 0) {
            // if locked.end then send directly
            IVotingEscrowV2.LockDetails memory _locked =
             ↪  IVotingEscrowV2(voting_escrow).lockDetails(_tokenId);
            // if the lock is non-permanent, and we pass the end time. We will
             ↪  transfer to the owner directly.
            if (_locked.endTime <= block.timestamp && _locked.lockType ==
             ↪  IVotingEscrowV2_Data.LockType.NON_PERMANENT) {
                address _nftOwner =
                 ↪  IVotingEscrowV2(voting_escrow).ownerOf(_tokenId);
                IERC20(token).transfer(_nftOwner, amount);
            } else {
```

```
                // If the lock is rolling, permanent or not-expired, we should
                ↪  increase the amount.
@>                IVotingEscrowV2(voting_escrow).increaseAmount(_tokenId, amount);
            }
            token_last_balance -= amount;
        }
        return amount;
    }
```

## Impact

Users' reward with one permanent lock will be less than expected. This is unfair for users who hold one permanent lock.

## Code Snippet

https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/9a1e74294d68a42ff49e2c4ac8c726dec0cc3dce/hydrex-contracts/contracts/VoterV5/RewardsDistributorV2.sol#L182-L199

## Tool Used

Manual Review

## Recommendation

Suggest refactor this mechanism. Users with permanent lock should have the same rewards compared with rolling lock and non-permanent lock.

## Discussion

**lpetroulakis**

Issue is acknowledged. Ties in to #51 where everyone is forced to claim with penalty but the team cranks up rebase emissions to compensate for it.

# Issue M-2: `distribute()` in `OptionFeeDistributor.sol` should use decimals of payment token to determine `floorGuardianAmount`

Source: https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/issues/48

## Summary

As a code comment correctly states, the `distribute()` function should use the decimals of payment token. In current code context in case only usdc is used on base which has 6 decimals instead of the hardcoded 18.

## Vulnerability Detail

In OptionFeeDistributor a hardcoded decimal of 18 is used for payout tokens. As the code comment correctly states, this should be removed and replaced by a decimals call to the token contract.

```
function distribute(address token, uint256 payoutAmount, uint256 amount) external
↪  override {
    uint256 floorGuardianAmount = (floorPrice * payoutAmount) / (1e18); /// @dev
    ↪  needs to match decimals of payout token
```

https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/main/hydrex-contracts/contracts/OptionToken/OptionFeeDistributor.sol#L154

## Impact

Currently the contract assumes that the option token and the payment token have the same decimals. In case this assumption is broken the contract will pay out less to the floor guardian. Because in current context we expect to use usdc as the payment token, the payed amount is lower then expected.

## Code Snippet

https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/main/hydrex-contracts/contracts/OptionToken/OptionFeeDistributor.sol#L154

## Tool Used

Manual Review

## Recommendation

Add a call to get the decimals of the payment token:

```
uint256 floorGuardianAmount =  (floorPrice * payoutAmount) / (10 **
↪  IERC20Metadata(address(token)).decimals());
```

The fix involves another issue related to `paymentAmount` being incorrect calculated.

## Discussion

**lpetroulakis**

Issue has been fixed in PR #154

# Issue L-1: Users may fail to claim rewards via claim_many

Source: https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/issues/42

## Summary

Users may fail to claim rewards via claim_many

## Vulnerability Detail

When users claim their rewards, we will transfer reward tokens to the recipient directly if the lock is non-permanent and the lock is expired. Otherwise, we will increase the lock amount.

The problem here is that there is one edge case when `_locked.endTime` equals `block.timestamp`, we will try to increase the lock amount. But the operation will be revereted because we don't allow to increase amount when `_locked.endTime` equals `block.timestamp`. Actually, we should take this case as expired. Refer to the `claim`'s implementation, we should transfer rewards to the recipients directly.

```
    function claim_many(uint[] memory _tokenIds) external returns (bool) {
        if (block.timestamp >= time_cursor) _checkpoint_total_supply();
        uint _last_token_time = last_token_time;
        _last_token_time = (_last_token_time / WEEK) * WEEK;
        address _voting_escrow = voting_escrow;
        uint total = 0;

        for (uint i = 0; i < _tokenIds.length; i++) {
            uint _tokenId = _tokenIds[i];
            if (_tokenId == 0) break;
            uint amount = _claim(_tokenId, _voting_escrow, _last_token_time);
            if (amount != 0) {
                // if locked.end then send directly
                IVotingEscrowV2.LockDetails memory _locked =
                ↪  IVotingEscrowV2(_voting_escrow).lockDetails(_tokenId);
                if (_locked.endTime < block.timestamp && _locked.lockType ==
                ↪  IVotingEscrowV2_Data.LockType.NON_PERMANENT) {
                    address _nftOwner =
                    ↪  IVotingEscrowV2(_voting_escrow).ownerOf(_tokenId);
                    IERC20(token).transfer(_nftOwner, amount);
                } else {
@>                      IVotingEscrowV2(_voting_escrow).increaseAmount(_tokenId,
↪   amount);
                }
            }
```

```
        }
    }
```

## Impact

Users may fail to claim reward in one edge case. This issue will disappear when users re-try the claim in next block.

## Code Snippet

https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/9a1e74294d68a42ff49e2c4ac8c726dec0cc3dce/hydrex-contracts/contracts/VoterV5/RewardsDistributorV2.sol#L201-L215

## Tool Used

Manual Review

## Recommendation

When `_locked.endTime` equals `block.timestamp`, we should transfer rewards to recipients directly.

## Discussion

**lpetroulakis**

Issue has been fixed in PR #142

# Issue L-2: Miss updating `balanceWithLock` in emergency withdraw

Source: https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/issues/44

## Summary

Miss updating `balanceWithLock` in emergency withdraw

## Vulnerability Detail

In GaugeV2, we support deposing with lock. The related locked deposit balance can be withdrawn after the lock is expired.

If there is something wrong, and we need to withdraw asset emergently, users can withdraw all deposit, including locked amount. The problem here is that we don't clear the locked balance. When we recover from the emergent status, and users deposit some LP shares with non-lock status. These non-lock deposit may fail to be withdrawn because of the remaining locked balance.

```
function emergencyWithdraw() external nonReentrant {
    // Only emergency condition, we can withdraw via this method.
    if (!emergency) revert IsEmergency(emergency);
    if (_balances[msg.sender] <= 0) revert NoBalances();

    uint256 _amount = _balances[msg.sender];
    _totalSupply = _totalSupply - _amount;
    _balances[msg.sender] = 0;
    _updateRewardForAllTokens(address(0));

    if (gaugeRewarder != address(0)) {
        IRewarder(gaugeRewarder).onEmergencyWithdrawAmount(msg.sender, _amount);
    }

    stakeToken.safeTransfer(msg.sender, _amount);

    emit Withdraw(msg.sender, _amount);
}
```

## Impact

When users deposit after we withdraw in emergent status, the normal deposit without lock may be locked and not allowed to be withdrawn.

## Code Snippet

https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/9a1e74294d68a42ff49e2c4ac8c726dec0cc3dce/hydrex-contracts/contracts/GaugeV2.sol#L410-L426

## Tool Used

Manual Review

## Recommendation

Suggest update the locked balance when we withdraw in emergency.

## Discussion

**lpetroulakis**

Issue has been fixed in PR #143

# Issue L-3: delegate may be dos because of the grief attack

Source: https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/issues/45

## Summary

delegate may be dos because of the grief attack

## Vulnerability Detail

In ve contract, users can delegate their vote to one delegatee. When users delegate, users can delegate all escrow ids to the delegatee.

The problem here is that malicious users can create lots of tiny escrow ids to cause out of gas.

Although, we have another interface `delegate(uint256 _tokenId, address delegatee)` to delegate by token Id. Considering that if users have dozens of escrow id, it's not convenient to delegate ve tokens one by one.

```
function _delegate(address delegator, address delegatee) internal {
    // Nft amount for this delegator.
    uint256 balance = balanceOf(delegator);
    address fromDelegate = address(0);
    // Here we will loop the delegate for all tokens that we own.
    for (uint256 i = 0; i < balance; i++) {
        // all token ids that this delegator owns.
        uint256 tokenId = tokenOfOwnerByIndex(delegator, i);
        (address oldDelegate, address newDelegate) = edStore.delegate(
            tokenId,
            delegatee, // new delegatee.
            __lockDetails[tokenId].endTime // We will delegate to the endTime.
        );
    }
}
function delegate(uint256 _tokenId, address delegatee) external
↪  checkAuthorized(_tokenId) {
    (address fromDelegatee, address toDelegatee) = edStore.delegate(
        _tokenId,
        delegatee,
        __lockDetails[_tokenId].endTime
    );
    emit LockDelegateChanged(_tokenId, _msgSender(), fromDelegatee, toDelegatee);
}
```

## Impact

Users may fail to delegate multiple ve tokens via function `delegate`.

## Code Snippet

https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/9a1e74294d68a42ff49e2c4ac8c726dec0cc3dce/hydrex-contracts/contracts/VoterV5/VotingEscrow/VotingEscrowV2Upgradeable.sol#L531-L552

## Tool Used

Manual Review

## Recommendation

Suggeset adding some flexible parameters, and users can choose to delegate one escrow id range to one delegatee.

## Discussion

**lpetroulakis**

Issue has been fixed in PR #152

# Issue L-4: `OptionFeeDistributor.sol` should use safe-Transfer

Source: https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/issues/46

## Summary

OptionFeeDistributor.sol contract currently uses the standard transfer functions, it is recommended to use safe transfer by open zeppelin to avoid conflicts with tokens that might not return correct data on failed transfers.

## Vulnerability Detail

Several transfers are made without the Safe Transfer lib:

https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/main/hydrex-contracts/contracts/OptionToken/OptionFeeDistributor.sol#L163

https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/main/hydrex-contracts/contracts/OptionToken/OptionFeeDistributor.sol#L171

https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/main/hydrex-contracts/contracts/OptionToken/OptionFeeDistributor.sol#L176-L178

## Impact

In current code context there is no impact, this is just a recommendation.

## Recommendation

Switch to safe transfer like in other parts of the codebase.

## Discussion

**lpetroulakis**

Issue has been fixed in PR #144

# Issue L-5: Missing checks for `totalFeeShare` in `OptionFeeDistributor`

Source: https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/issues/47

## Summary

The `_setFeeReceiver()` function allows to add fee receivers to the fee distributor contract. This function does loop over all feeReceivers, but fails to check that totalFeeShare does not exceed MAX_FEE_SHARE.

## Vulnerability Detail

Currently the `_setFeeReceiver()` function loops over all fee Receivers, adding there total share to the local total shares variable. This loop currently does not perform any checks on the fee share.
https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/main/hydrex-contracts/contracts/OptionToken/OptionFeeDistributor.sol#L138-L140

## Impact

It is possible to set fee Receivers in a way that will make every distribute transaction revert due to exceeding MAX_FEE_SHARE.

## Code Snippet

https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/main/hydrex-contracts/contracts/OptionToken/OptionFeeDistributor.sol#L138-L140

## Tool Used

Manual Review

## Recommendation

Consider checking that totalFeeShare does not exceed MAX_FEE_SHARE. In case this is not intended and should be managed off chain, consider removing the loop.

## Discussion

**lpetroulakis**

Issue has been fixed in PR #145

# Issue L-6: Code improvements and recommendations for `OptionTokenV4.sol` and `OptionFeeDistributor.sol`

Source: https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/issues/50

## Summary

There are several code quality recommendations. These do not have any security related impact and are therefor compiled into one report.

Missing NatSpec: https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/main/hydrex-contracts/contracts/OptionToken/OptionFeeDistributor.sol#L148-L153

Unused code: https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/main/hydrex-contracts/contracts/OptionToken/OptionTokenV4.sol#L43

Missing interface function for `getMinPrice()`: https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/main/hydrex-contracts/contracts/OptionToken/IOptionTokenV4.sol

Missing interface functions for several setters in `IOptionFeeDistributor.sol`: https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/blob/main/hydrex-contracts/contracts/OptionToken/IOptionFeeDistributor.sol

## Tool Used

Manual Review

## Recommendation

Fix minor code quality suggestions.

## Discussion

**lpetroulakis**

Issue has been fixed in PR #146

# Issue L-7: Refine Architecutre for Managing veToken and Claims Delegation

Source: https://github.com/sherlock-audit/2025-07-hydrex-finance-july-14th/issues/52

## Summary

Currently there are many instances in which different delegation, claims, and approval permissions can be granted to users. There is not a good organization system or way to understand who has which permissions.

## Vulnerability Detail

## Impact

## Code Snippet

## Tool Used

Manual Review

## Recommendation

Consolidate all approval and delegation functionalities into a streamlined setup with enhanced event logging for easier management + add any missing delegation functionalities.

## Discussion

**lpetroulakis**

Issue has been fixed in PR #149

# Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.