Pendle Finance V2 Audit ~ Liquidity Mining

Issues

- GAS: Consider increasing the number of optimisation runs to the default 200 or more.
- INF: Generally, the protocol isn't able to handle rebasing/fee-on-transfer tokens as rewards to stakers.

`VotingControllerStorage.sol`

- LOW: `_setAllPastEpochsAsFinalized()` deducts `deployedWTime` from `wTime` when it should be `WEEK`.

`VotingEscrowPendleMainchain.sol`

- LOW: Anyone can call `withdraw()` and force a user to withdraw when they may wish to call `increaseLockPosition()` on their expired lock.
- LOW: `_increasePosition()` will truncate up to `62899200` tokens representing `6.3e7` Pendle tokens. This can accrue overtime and lead to lost value in the contract.
- MED: Total supply can become out of sync if `_broadcastPosition()` is not called for some time. As a result, the new total supply is not broadcasted to sidechains even though the slope change for a given week has expired. Stakers on these sidechains will be able to earn increased rewards at the expense of other stakers. This is possible because the rate of decay on the total supply outpaces the rate of decay for each user. An expired user will not continue to decay, however, they will contibute to the decay of the total supply. There is no incentive to broadcast new total supply unless a user intends to broadcast their position. Each broadcast costs some ether as per the requirement in `CelerSender.sol`.
- INF: An overly inactive voting escrow contract will become locked if `_updateTotalSupply()` has not been called for some time. Similarly, pool updates may experience the same issue.

`VotingEscrowToken.sol`

- INF: `lastSupplyUpdatedAt` can be set in the past as it calculates the floor of `block.timestamp / WEEK`. Ensure this is understood.

`RewardManager.sol`

- GAS: Heavy optimisation in `_doTransferOutRewards()`. Only perform `userReward[token][user].accrued = 0;` and `rewardState[token].lastBalance -= rewardAmounts[i].Uint128();` if `rewardAmounts[i] != 0`. This should reduce the number of `SLOADS` for the majority of user code paths.
- MED: Users can game token distributions be calling `_updateUserActiveBalance()` on as many inactive accounts as possible. This will reduce the system's `totalActiveSupply`, representing `_rewardSharesTotal()`. As a result, the caller is able to increase their share of rewards by refusing to update the active balance on their own account. Claiming on behalf of another account can be done in two ways:
- Call `redeemRewards()` on behalf of the victim account.

- Send `1 wei` of tokens directly to the victim account.
- INF: Like with most protocols that integrate token distributions, these actions can be front-run by sophisticated users. I personally prefer the use of a token drip contract which distributes based on how much time was spent in the contract. But I understand this might be difficult as rewards are not guaranteed at every point in time.
- INF: `_getRewardTokens()` a potential unbounded array. Might be safer to cap to ensure `_updateRewardIndex()` does not fail and affect the liveness of the protocol.
- LOW: `_updateRewardIndex()` isn't exactly safe from reentrancy because `_beforeTokenTransfer()` will call it. If `_redeemExternalReward()` allowed a user to gain control over the flow of execution, they could bypass `_updateRewardIndex()` intended action to initiate some state changing calls beforehand.

`VeBalanceLib.sol`

- GAS: Checkpointing adds a lot of overhead to `_increasePosition()` operations. Consider allowing users to potentially opt out of this if they want to reduce the gas costs of each of their calls.
- INF: Consider adding an invariant to `getValueAt()` such that `require(!(t % WEEK))` holds true. However, it may be worth noting that this could add some additional and potentially unnecessary gas overhead.

`PendleVotingController.sol`

- LOW: If a pool has been removed from this contract, any pre-existing votes will continue to persist until each user explicitly calls `unvote()` on the inactive pool. As a result, rewards for other pools will be slightly diluted by the unexpired `vePendle` vote. Consider removing inactive pool votes from `weekData[wTime].totalVotes`. Its also worth noting, that by calling `unvote()`, `_subtractVotePoolInfo()` will not be called on the pool but the user's vote will be unset. This will allow them to vote for another pool while continuing to have an existing vote on the removed pool.
- LOW: As `CelerSender._sendMessage()` may revert due to some unknown issue with the target contract, it would be safer to implement try/catch statements to safely handle reverts on `celerMessageBus.sendMessage()` and `celerMessageBus.calcFee()`.
- INF: Usage of local `pendlePerSec` variable shadows storage variable.
- LOW: Upon deployment, the first week can be finalized and broadcasted at any time. This can be sort of abused if `pendlePerSec` has been set. A single user could stake, vote for a given pool, finalize the epoch and broadcast results within a single transaction. Consider updating `_setAllPastEpochsAsFinalized()` to not finalize an epoch if `wTime == deployedTime`. The coniditional loop check could potentially be updated to `wTime > deployedWTime` to fix this.

`PendleGaugeController.sol`

- INF: Because `_receiveVotingResults()` is taking somewhat untrusted data from `CelerReceiver.sol`, it would be useful to be consistent in checking that `wTime == WeekMath.getWeekStartTimestamp(wTime)`.
- INF: If this contract is not consistently funded via `fundPendle()`, then `claimMarketReward()` will revert causing `PendleGauge._redeemExternalReward()` in `RewardManager._updateRewardIndex()` to fail. As the latter function affects `_distributeUserReward()` in `_beforeTokenTransfer()`, the liveness of the gauge contracts

will be dependent on if rewards are readily available. Consider using a try/catch statement to check for failed reward index updates.

`CelerSender.sol`

- INF: `_sendMessage()` does not check that `msg.value == fee`, therefore, a user may overpay when sending a message.