MAVERICK TOKENOMICS AUDIT REPORT

hansfriese

March 29th, 2023

Contents

Maverick Tokenomics Audit Report

Version 1.1

Prepared by hansfriese

Disclaimer

I make all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit from me is not an endorsement of the underlying business or product. The audit was time-boxed, and the review of the code is solely on the security aspects of the solidity implementation of the contracts.

About hansfriese

hansfriese is a top 1 warden and judge in **code4rena** and an independent security researcher contributing to blockchain and Web3 security. Please reach out on Twitter @hansfriese

Protocol Summary

Maverick is partitioned into two components:

1. Core AMM
2. PoolPositions and Incentives

The Core AMM component of Maverick allows LPs to stake arbitrary liquidity distributions in either a static or movement mode. All trading fees generated from the LP's position go back to the LP.

The PoolPosition component of Maverick allows protocols and market makers to create a "Pool Position" (PP), which is a distribution of liquidity within a given pool. LPs can join this PP by adding assets proportional to the existing PP asset mix. PP LPs will collect only a portion of the fees generated by their liquidity, but they will be compensated with LP incentives that any party can add to the LP Rewards contract.

Audit Details

Scope Of Audit

Between Februrary 23rd 2023 - Mar 3rd 2023, I conducted an audit on the 13 files in the current audit scope. The scope of the audit was as follows:

1. The files in audit scope are specified:

- Distributor.sol
- Poll.sol
- PoolPositionBase.sol
- PoolPositionDynamic.sol
- PoolPositionRouter.sol
- PoolPositionStatic.sol
- RewardBase.sol
- RewardOpen.sol
- RewardPusher.sol
- RewardSingle.sol
- RewardVote.sol
- VoterToken.sol
- factories/PoolPositionAndRewardFactory.sol

2. Commit hash: 3fd1b92 of token-v1

3. Mitigation reviewed for commit hash 194be8f

Severity Criteria

- High: Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals).
- Medium: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.
- Low: Low impact and low/medium likelihood events where assets are not at risk (or a trivia amount of assets are), state handling might be off, functions are incorrect as to natspec, issues with comments, etc.
- Informational / Non-Critial: A non-security issue, like a suggested code improvement, a comment, a renamed variable, etc. Auditors did not attempt to find an exhaustive list of these.

- Gas: Gas saving / performance suggestions. Auditors did not attempt to find an exhaustive list of these.

Summary Of Findings

- High - 1
- Medium - 6
- Low - 3
- Non-Critical - 6
- Gas - 15

	Findings	Severity
H-1	Users can charge 100% of base rewards using a very long `lockDuration`	High
M-1	Users can bypass the `MAX_DURATION` limit by depositing rewards several times	Medium
M-2	Possible DOS by sending dust	Medium
M-3	In `getRewardBaseNewTokenId.getRewardBaseNewTokenId()`, users would lose some portion of rewards if `lockDuration = 0, proportionToLockD18 > 0`	Medium
M-4	In `RewardOpen.sol`, `BASE_TOKEN_INDEX` shouldn't be used for the base reward token	Medium
M-5	Needless `mulDiv` can cause accuracy loss	Medium
M-6	Voting weights will be changed and an attacker can skew the voting weights when there are duplicated voting targets	Medium

Tools used

- Slither
- 4naly3er
- foundry
- Hardhat
- Solodit

High Findings

[H-1] Users can charge 100% of base rewards using a very long `lockDuration`

Details

According to the documentation, users should lock for 4 years to get 100% of their base rewards.

But in `RewardOpen._getBaseReward()`, there is no validation of `lockDuration` and users can charge 100% of base rewards by manipulating `lockDuration`.

This POC works with the below assumptions and steps.

1. `rewardFactory.minimumLockPeriodOnBaseClaim = 0` so users can use a custom `proportionToLockD18`.
2. User's total rewards should be less than `1e18`. So for `MAV`, it's 1 `MAV` and this amount would be more valuable to manipulate if the reward token decimals < 18.
3. Then a malicious user can charge base rewards with `proportionToLockD18` = 1, `lockDuration` = `1460 days * ONE`
4. In this case, the user can receive 100% of the rewards without locking.

```
function testGetAllRewardsWithoutLocking() public {
        uint256 proportionToLock = 1; // default value = 1e18
        uint256 claimLockPeriod = 1460 days * ONE; // limit = 1460 days
        bool useNewOrCreate = true;

        MockFactory(address(pprFactory)).setPeriod(0); // set minLockDuration = 0 of factory
        ↪  to use custom proportion

        // have other user's stake
        vm.startPrank(otherUser);
        LP.approve(address(r), 1e30);
        r.stake(1e20, otherUser); //other user staked 1e20
        vm.stopPrank();

        // have this user's stake
        LP.approve(address(r), 1e18);
        r.stake(1e9, thisUser); // this user staked 1e9, so this user's reward for 1 day is
↪  less than 1e18

        vm.warp(block.timestamp + 3600 * 24 * 1);
        vm.roll(block.number + 1);

        // prev MAV balance of user
        uint256 preBalance = MAV.balanceOf(thisUser);
        uint256 amountToLock;
        uint256 amountToUser;
        uint256 amountReturned;

        // claim and lock
        if (useNewOrCreate) {
            (amountToLock, amountToUser, amountReturned, ) =
            ↪  r.getRewardBaseNewOrCreateTokenId(claimLockPeriod, proportionToLock, thisUser,
            ↪  thisUser);
        } else {
            (amountToLock, amountToUser, amountReturned, ) =
            ↪  r.getRewardBaseNewTokenId(claimLockPeriod, proportionToLock, thisUser,
            ↪  thisUser);
        }

        console.log("amountToLock: ", amountToLock);
        console.log("amountToUser: ", amountToUser);
        console.log("amountReturned: ", amountReturned);

        // this user received whole reward
        uint256 receivedAmount = MAV.balanceOf(thisUser) - preBalance;
        assertEq(amountToUser, receivedAmount);

        // nothing to be locked
        assertEq(amountToLock, 0); // this value should be 0 to bypass lockDuration limit of
        ↪  ve

        // receive almost 100% of rewards
        assertLt(amountReturned, 2);
    }
```

This is the output.

```
[PASS] testGetAllRewardsWithoutLocking() (gas: 391297)
Logs:
  amountToLock:  0
  amountToUser:  999999999989999999
  amountReturned:  1
```

As `amountToLock = 0`, users can bypass the `MAXTIME(4 years)` validation of the voting escrow.

Mitigation

We should check if `lockDuration < 4 years` in `_getBaseReward()`.

Mitigation Review

This issue was resolved by validating lockDuration <= FOUR_YEARS.

Medium Findings

[M-1] Users can bypass the `MAX_DURATION` limit by depositing rewards several times

Details

In `RewardBase.transferAndNotify()`, it checks if the reward duration shouldn't be greater than `MAX_DURATION`.

```
    function transferAndNotify(address rewardTokenAddress, uint256 amount, uint256 duration)
↪   public nonReentrant {
        if (duration > MAX_DURATION) revert DurationOutOfBounds();
        _notifyRewardAmount(rewardTokenAddress, amount, duration);
        IERC20(rewardTokenAddress).safeTransferFrom(msg.sender, address(this), amount);
    }
```

But when it recalculates the duration in `_notifyRewardAmount()`, it extends the duration without any validation if the deposited amount is less than the remaining rewards.

```
    function _notifyRewardAmount(address rewardTokenAddress, uint256 amount, uint256 duration)
↪   internal {
        uint8 rewardTokenIndex = _checkAndAddRewardToken(rewardTokenAddress);
        RewardData storage data = rewardData[rewardTokenIndex];
        updateReward(address(0), data);
        uint256 remainingRewards = MavMath.clip(data.rewardToken.balanceOf(address(this)),
↪   data.escrowedReward);
        if (amount > remainingRewards || data.rewardRate == 0) {
            // if notifying new amount, notifier gets to set the rate
            if (duration == 0 && totalSupply != 0) {
                data.rewardPerTokenStored += Math.mulDiv(amount + remainingRewards, ONE,
↪   totalSupply);
                data.escrowedReward += (amount + remainingRewards);
            } else if (duration != 0) {
                data.rewardRate = (amount + remainingRewards) / duration;
            }
        } else {
            // if notifier doesn't bring enough, we extend the duration at the
            // same rate
            duration = (amount + remainingRewards) / data.rewardRate; //@audit can bypass max
↪   duration
        }
        data.finishAt = block.timestamp + duration;
        data.updatedAt = block.timestamp;
        emit NotifyRewardAmount(msg.sender, rewardTokenAddress, amount, duration,
        ↪  data.rewardRate);
    }
```

So users can bypass the `MAX_DURATION` limit easily like the below.

1. A user started a reward thread of 30 days with 100 amount of reward tokens.
2. Right after that, he deposited another 100 and called `transferAndNotify()`. The reward duration will be 60 days now.
3. If he replays step 2 again with 200 amount of reward tokens, the duration will be 120 days.

Mitigation

We should validate the duration again after extending.

```
    function _notifyRewardAmount(address rewardTokenAddress, uint256 amount, uint256 duration)
↪   internal {
        uint8 rewardTokenIndex = _checkAndAddRewardToken(rewardTokenAddress);
        RewardData storage data = rewardData[rewardTokenIndex];
        updateReward(address(0), data);
        uint256 remainingRewards = MavMath.clip(data.rewardToken.balanceOf(address(this)),
↪   data.escrowedReward);
        if (amount > remainingRewards || data.rewardRate == 0) {
            // if notifying new amount, notifier gets to set the rate
            if (duration == 0 && totalSupply != 0) {
                data.rewardPerTokenStored += Math.mulDiv(amount + remainingRewards, ONE,
↪   totalSupply);
                data.escrowedReward += (amount + remainingRewards);
            } else if (duration != 0) {
                data.rewardRate = (amount + remainingRewards) / duration;
            }
        } else {
            // if notifier doesn't bring enough, we extend the duration at the
            // same rate
            duration = (amount + remainingRewards) / data.rewardRate;

            if (duration > MAX_DURATION) revert DurationOutOfBounds(); //+++++++++++++++++++++
        }
        data.finishAt = block.timestamp + duration;
        data.updatedAt = block.timestamp;
        emit NotifyRewardAmount(msg.sender, rewardTokenAddress, amount, duration,
        ↪  data.rewardRate);
    }
```

Mitigation Review

This issue was resolved by validating the duration again.

[M-2] Possible DOS by sending dust

Details

In `RewardBase.sol`, there are some functions to be done after the reward thread is finished.

But this logic wouldn't work properly by a malicious user.

DOS scenario 1 Stale tokens wouldn't be removed forever.

```
function removeStaleToken(uint8 rewardTokenIndex) public virtual nonReentrant {
    RewardData storage data = rewardData[rewardTokenIndex];
    if (block.timestamp < STALE_INTERVAL + data.finishAt) revert TokenNotStale();
    // remove token from list
    globalActive.unset(rewardTokenIndex);
    delete tokenIndex[address(data.rewardToken)];
    // take left over tokens
    data.rewardToken.safeTransfer(address(rewardFactory.vault()),
↪  data.rewardToken.balanceOf(address(this)));
    delete data.rewardToken;
    delete data.escrowedReward;
    delete data.rewardPerTokenStored;
    delete data.rewardRate;
    delete data.finishAt;
}
```

1. The reward period of `tokenA` was finished and the remaining rewards were withdrawn using `recoverERC20()`.
2. This token can be removed as stale after `STALE_INTERVAL = 30 days`.
3. Within these 30 days, a malicious user calls `transferAndNotify()` with `amount = 1 wei, duration = 30 days`.
4. At L230 in `_notifyRewardAmount()`, new `rewardRate` will be 0 for 30 days as `1 wei > remainingRewards(= 0)`.
5. So `tokenA` will be active for another 30 days without actual rewards.
6. If malicious users keep doing this per 30~60 days, any tokens can't be removed and it will affect the protocol as there is a `MAX_REWARD_TOKENS` limit.

DOS scenario 2 A malicious user can make `recoverERC20()` revert by sending dust.

```
    function recoverERC20(uint8 rewardTokenIndex) external nonReentrant {
        RewardData storage data = rewardData[rewardTokenIndex];
        if (data.finishAt != data.updatedAt) revert RewardStillActive();
        uint256 remainingRewards = MavMath.clip(data.rewardToken.balanceOf(address(this)),
↪   data.escrowedReward);
        data.rewardToken.safeTransfer(address(rewardFactory.vault()), remainingRewards);
        emit Recovered(msg.sender, rewardTokenIndex, address(data.rewardToken),
        ↪  remainingRewards);
    }
```

1. This function works only if the reward period is finished.
2. But when `recoverERC20()` is called to withdraw unused rewards, a malicious user can call `transferAndNotify()` with dust and extends the `finishAt` by frontrunning.

Mitigation

I have 2 suggestions. 1. Add a whitelist of callers for `transferAndNotify()`. 2. Add a minimum threshold of amount in `transferAndNotify()` to prevent dust amounts.

Mitigation Review

This issue was resolved by checking minimumRewardAmount.

[M-3] In `getRewardBaseNewTokenId.getRewardBaseNewTokenId()`, users would lose some portion of rewards if `lockDuration = 0, proportionToLockD18 > 0`

Details

In `getRewardBaseNewTokenId()`, users can set both `lockDuration` and `proportionToLockD18` as they like.

```
    function getRewardBaseNewTokenId(
        uint256 lockDuration,
        uint256 proportionToLockD18,
        address veRecipient,
        address baseRecipient
    ) public override returns (uint256 amountToLock, uint256 amountToUser, uint256
    ↪   amountReturned, uint256 veTokenId) {
        (lockDuration, amountToLock, amountToUser, amountReturned) =
        ↪  _getBaseReward(lockDuration, proportionToLockD18, baseRecipient);
        // create lock pulls mav into ve contract for lock
        if (lockDuration != 0 && amountToLock != 0) { //@audit lose rewards when lockDuration
        ↪   = 0, amountToLock > 0
            base.approve(address(ve), amountToLock);
            veTokenId = ve.create_lock_for(amountToLock, lockDuration, veRecipient);
        }
    }
```

And this function locks funds to voting escrow only if `lockDuration != 0, amountToLock != 0`.

So if `lockDuration = 0, amountToLock > 0`, users lose `amountToLock` forever as it's not locked to `ve`.

Logically, `amountToLock` will be 0 when `proportionToLockD18 = 0` and we should validate this for the user's preference.

Mitigation

We can modify `getRewardBaseNewTokenId()` like the below.

```
    function getRewardBaseNewTokenId(
        uint256 lockDuration,
        uint256 proportionToLockD18,
        address veRecipient,
        address baseRecipient
    ) public override returns (uint256 amountToLock, uint256 amountToUser, uint256
    ↪   amountReturned, uint256 veTokenId) {
        if (lockDuration == 0) { //++++++++++++++++
            require(proportionToLockD18 == 0, "Invalid proportion");
        }

        (lockDuration, amountToLock, amountToUser, amountReturned) =
        ↪   _getBaseReward(lockDuration, proportionToLockD18, baseRecipient);
        // create lock pulls mav into ve contract for lock
        if (lockDuration != 0 && amountToLock != 0) {
            base.approve(address(ve), amountToLock);
            veTokenId = ve.create_lock_for(amountToLock, lockDuration, veRecipient);
        }
    }
```

Mitigation Review

This issue was resolved by removing `lockDuration != 0` condition in getRewardBaseNewTokenId().

[M-4] In `RewardOpen.sol`, `BASE_TOKEN_INDEX` shouldn't be used for the base reward token

Details

In `RewardOpen.sol`, it uses `BASE_TOKEN_INDEX` to calculate the base rewards.

```
    function getReward(address recipient, uint8[] calldata rewardTokenIndices) external
↪   override {
        for (uint8 i = 0; i < rewardTokenIndices.length; i++) {
            uint8 ind = rewardTokenIndices[i];
            if (ind == BASE_TOKEN_INDEX) revert MustCallBaseRewardsSeperately(); //@audit
            ↪  dangerous validation
            _getReward(msg.sender, recipient, ind);
        }
    }
```

At the first time, the token index of the base reward is `BASE_TOKEN_INDEX = 1` but it might be changed if the base token is removed as stale and added again.

Furthermore, there is a benefit for normal users if the real base token index is changed because they can charge full rewards without locking.

So users might collaborate to manipulate the base token index by removing and adding again.

Mitigation

We should save `BASE_TOKEN_ADDRESS` instead of `BASE_TOKEN_INDEX` and use the correct index every time.

Mitigation Review

This issue was resolved by making impossible to remove the base token here.

[M-5] Needless `mulDiv` can cause accuracy loss

Details

In `Poll.updateVoteWeightsToMatchVeAmount`, when a user revotes with current balances, `newVoteWeights` is calculated by `currentVoteWeights * veWeight / totalVoteWeight`.

```
    function updateVoteWeightsToMatchVeAmount(uint256 veTokenId) public {
        (IReward[] memory voteTargets, uint256[] memory currentVoteWeights) =
        ↪  getCurrentVoteWeights(veTokenId);
        uint256[] memory newVoteWeights = new uint256[](currentVoteWeights.length);
        address veTokenIdAsAddress = address(SafeCast.toUint160(veTokenId));
        uint256 totalVoteWeight = voterToken.balanceOf(veTokenIdAsAddress);
        uint256 veWeight = ve.balanceOfNFT(veTokenId);
        for (uint256 i = 0; i < currentVoteWeights.length; i++) {
            newVoteWeights[i] = Math.mulDiv(currentVoteWeights[i], veWeight, totalVoteWeight);
↪   //@audit needless calculation
        }
        _vote(veTokenId, voteTargets, newVoteWeights);
    }
```

`newVoteWeights` are used in `_vote` and the scales of `newVoteWeights` are used because these weights are divided by the total weights in the `_vote` method. So the `mulDiv` is needless and `newVoteWeights[i] = currentVoteWeights[i]` is enough. This needless calculation can cause accuracy loss. For example, when `veWeight` is small and `totalVoteWeight` is large, `newVoteWeights[i]` can be 0 and so there will be no vote for the target.

Mitigation

Get rid of `mulDiv` and just set the `currentVoteWeights` from `getCurrentVoteWeights`

```
        newVoteWeights[i] = currentVoteWeights[i];
```

Mitigation Review

All needless code blocks are removed now so this issue is resolved.

```
    function updateVoteWeightsToMatchVeAmount(uint256 veTokenId) public {
        (IReward[] memory voteTargets, uint256[] memory currentVoteWeights) =
↪   getCurrentVoteWeights(veTokenId);
        _vote(veTokenId, voteTargets, currentVoteWeights);
    }
```

[M-6] Voting weights will be changed and an attacker can skew the voting weights when there are duplicated voting targets

Details

When a voter votes with duplicated targets, voting weights will be changed when other users call `Poll.updateVoteWeightsToMatchVeAmount` to revote and an attacker can skew the voting weights using this exploit.

For instance, a voter votes [1, 1, 1] for [A, A, B], he will vote 0.66 for A and 0.33 for B. When `updateVoteWeightsToMatchVeAmount` is called, the voting weights are from `getCurrentVoteWeights` and `getCurrentVoteWeights` gets these weights from `voteReward` balance.

```
    function updateVoteWeightsToMatchVeAmount(uint256 veTokenId) public {
        (IReward[] memory voteTargets, uint256[] memory currentVoteWeights) =
        ↪  getCurrentVoteWeights(veTokenId);
        ...
        for (uint256 i = 0; i < currentVoteWeights.length; i++) {
            newVoteWeights[i] = Math.mulDiv(currentVoteWeights[i], veWeight, totalVoteWeight);
        }
        _vote(veTokenId, voteTargets, newVoteWeights);
    }


    function getCurrentVoteWeights(uint256 veTokenId) public view returns (IReward[] memory
↪   voteTargets, uint256[] memory voteWeights) {
        ...
        for (uint256 i = 0; i < voteTargetCnt; i++) {
            IReward voteReward = voteTargets[i];
            voteWeights[i] = voteReward.balanceOf(veTokenIdAsAddress);
        }
    }
```

So the voting weight will [0.66, 0.66, 0.33] and 4/3 for A, and 1/3 for B. So the weights are changed to 0.8 for A and 0.2 for B.

Mitigation

The easiest solution is to restrict duplicated targets in `Poll.vote`. If `Poll.vote` still accepts duplicated targets, we can modify the implementation of `getCurrentVoteWeights`. It can return 0 for duplicated targets, [0.66, 0, 0.33], for example.

Mitigation Review

Maverick decided not to fix this issue.

Low Findings

	Issue	Instances
L-1	`abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`	2
L-2	Empty Function Body - Consider commenting why	4
L-3	Unsafe ERC20 operation(s)	10

[L-1] `abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`

Use `abi.encode()` instead which will pad items to 32 bytes, which will prevent hash collisions (e.g. `abi.encodePacked(0x123,0x456)` => `0x123456` => `abi.encodePacked(0x1,0x23456)`, but `abi.encode(0x123,0x456)` => `0x0...1230...456`). "Unless there is a compelling reason, `abi.encode` should be preferred". If there is only one argument to `abi.encodePacked()` it can often be cast to `bytes()` or `bytes32()` instead. If all arguments are strings and or bytes, `bytes.concat()` should be used instead

Instances (2):

```
File: factories/PoolPositionAndRewardFactory.sol

113:        bytes32 hash = keccak256(abi.encodePacked(pool, binIds, ratios));

130:        bytes32 hash = keccak256(abi.encodePacked(pool, binIds, ratios));
```

Link to code

[L-2] Empty Function Body - Consider commenting why

Instances (4):

```
File: PoolPositionBase.sol

125:     function migrateBinLiquidity() external virtual {}

127:     function claimFeeToVoters() external nonReentrant checkBin checkpointLiquidity {}
```

Link to code

```
File: PoolPositionDynamic.sol

25:     ) PoolPositionBase(_pool, _binIds, _ratios, factoryCount,
↪   _poolPositionAndRewardFactory, false, _protocolEscrow) {}
```

Link to code

```
File: RewardSingle.sol

26:     function removeStaleToken(uint8 rewardTokenIndex) public override {}
```

Link to code

[L-3] Unsafe ERC20 operation(s)

Instances (10):

```
File: Distributor.sol

40:         mav.approve(address(reward), monthRewardAmount);
```

Link to code

```
File: PoolPositionRouter.sol

75:             WETH9.transfer(recipient, value);

128:         position.approve(address(poolPosition), routerTokenId);

130:         position.approve(address(poolPosition), routerTokenId);

153:         poolPosition.transferFrom(msg.sender, address(this), lpAmountStaked);

154:         poolPosition.approve(address(lpReward), lpAmountStaked);
```

Link to code

```
File: RewardOpen.sol

76:             base.transfer(baseRecipient, amountToUser);

92:             base.approve(address(ve), amountToLock);

107:             base.approve(address(ve), amountToLock);
```

Link to code

```
File: RewardPusher.sol

23:         base.approve(address(lpReward), amount);
```

Link to code

Informational / Non-Critical Findings

	Issue	Instances
NC-1	Missing checks for `address(0)` when assigning values to address state variables	2
NC-2	`require()` / `revert()` statements should have descriptive reason strings	5
NC-3	Return values of `approve()` not checked	7
NC-4	Event is missing `indexed` fields	3
NC-5	Constants should be defined rather than using magic numbers	1
NC-6	Functions not used internally could be marked external	28

[NC-1] Missing checks for `address(0)` when assigning values to address state variables

Instances (2):

```
File: RewardSingle.sol

24:         rewardPusherAddress = _rewardPusherAddress;
```

Link to code

```
File: VoterToken.sol

25:         minter = minter_;
```

Link to code

[NC-2] `require()` / `revert()` statements should have descriptive reason strings

Instances (5):

```
File: Poll.sol

28:         require(msg.sender == address(factory));

75:         require(voteTargetCnt == weights.length);
```

Link to code

```
File: PoolPositionBase.sol

73:         require(msg.sender == address(poolPositionAndRewardFactory));
```

Link to code

```
File: PoolPositionRouter.sol

85:         require(factory.isFactoryPool(IPool(msg.sender)));

86:         require(msg.sender == address(data.pool));
```

Link to code

[NC-3] Return values of `approve()` not checked

Not all IERC20 implementations `revert()` when there's a failure in `approve()`. The function signature has a boolean return value and they indicate errors that way instead. By not checking the return value, operations that should have marked as failed, may potentially go through without actually approving anything

Instances (7):

```
File: Distributor.sol

40:         mav.approve(address(reward), monthRewardAmount);
```

Link to code

```
File: PoolPositionRouter.sol

128:         position.approve(address(poolPosition), routerTokenId);

130:         position.approve(address(poolPosition), routerTokenId);

154:         poolPosition.approve(address(lpReward), lpAmountStaked);
```

Link to code

```
File: RewardOpen.sol

92:             base.approve(address(ve), amountToLock);

107:             base.approve(address(ve), amountToLock);
```

Link to code

```
File: RewardPusher.sol

23:         base.approve(address(lpReward), amount);
```

Link to code

[NC-4] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

Instances (3):

```
File: Distributor.sol

11:     event Initialize(IERC20 mav, uint256 startTimestamp, IReward reward);

12:     event SetMonthlyDisbursement(uint256 value);

13:     event Disburse(uint256 absoluteMonthNumber, uint256 monthRewardAmount);
```

Link to code

[NC-5] Constants should be defined rather than using magic numbers

Instances (1):

```
File: RewardOpen.sol

15:     uint256 constant FOUR_YEARSD18 = (365 days) * 4 * 1e18;
```

Link to code

[NC-6] Functions not used internally could be marked external

Instances (28):

```
File: Distributor.sol

28:     function setMonthDisbursement(uint256 value) public onlyOwner {

33:     function disburse() public returns (uint256 monthRewardAmount) {
```

Link to code

```
File: Poll.sol

38:     function voterAdvantage(uint256 veTokenId) public view returns (int256 voteVeDiff) {

57:     function addVotingRewardsContractToTokenPermissions(address rewardContract) public
↪   onlyFactory {

64:     function attachRewardPusher(IRewardPusher _rewardPusher) public onlyFactory {

130:     function updateVoteWeightsToMatchVeAmount(uint256 veTokenId) public {
```

Link to code

```
File: PoolPositionDynamic.sol

27:     function initializeContract(IReward _voteReward) public override {
```

Link to code

```
File: PoolPositionRouter.sol

50:     function unwrapWETH9(uint256 amountMinimum, address recipient) public payable override
↪   {

58:     function sweepToken(IERC20 token, uint256 amountMinimum, address recipient) public
↪   payable {

189:     function disburse(IDistributor distributor) public returns (uint256) {
```

Link to code

```
File: PoolPositionStatic.sol

30:     function initializeContract(IReward _voteReward) public override {
```

Link to code

```
File: RewardBase.sol

90:     function earnedList(address account) public view returns (EarnedInfo[] memory
↪   earnedInfo) {

208:     function transferAndNotify(address rewardTokenAddress, uint256 amount, uint256
↪   duration) public nonReentrant {
```

Link to code

```
File: RewardOpen.sol

112:     function getRewardBaseNewOrCreateTokenId(
```

Link to code

```
File: RewardPusher.sol

27:     function push(address lpReward, address voteReward) public {

31:     function push(address lpReward) public {
```

Link to code

```
File: RewardSingle.sol

23:     function setPusher(address _rewardPusherAddress) public onlyFactory {

26:     function removeStaleToken(uint8 rewardTokenIndex) public override {}
```

Link to code

```
File: RewardVote.sol

17:     function notifyRewardAmount(address rewardTokenAddress, uint256 amount, uint256
↪   duration) public override nonReentrant {
```

Link to code

```
File: VoterToken.sol

36:     function totalSupply() public view returns (uint256) {

39:     function balanceOf(address account) public view returns (uint256) {

42:     function addTransferer(address transferer) public onlyMinter {

45:     function removeTransferer(address transferer) public onlyMinter {

48:     function burnAll(address account) public onlyMinter {

58:     function mint(address account, uint256 amount) public onlyMinter {

67:     function transfer(address to, uint256 amount) public returns (bool) {

72:     function transferFrom(address from, address to, uint256 amount) public returns (bool)
↪   {
```

Link to code

```
File: factories/PoolPositionAndRewardFactory.sol

109:     function owner() public view override(IPoolPositionAndRewardFactory, Ownable) returns
↪  (address) {
```

Link to code

Gas Findings

	Issue	Instances
GAS-1	Use `selfbalance()` instead of `address(this).balance`	3
GAS-2	Use assembly to check for `address(0)`	5
GAS-3	Using bools for storage incurs overhead	7
GAS-4	Cache array length outside of loop	16
GAS-5	State variables should be cached in stack variables rather than re-reading them from storage	2
GAS-6	Use calldata instead of memory for function arguments that do not get mutated	8
GAS-7	Use Custom Errors	24
GAS-8	Don't initialize variables with default value	20
GAS-9	Long revert strings	10
GAS-10	Functions guaranteed to revert when called by normal users can be marked `payable`	14
GAS-11	`++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)	23
GAS-12	Using `private` rather than `public` for constants, saves gas	4
GAS-13	Splitting require() statements that use && saves gas	3
GAS-14	Use != 0 instead of > 0 for unsigned integer comparison	4
GAS-15	`internal` functions not called by the contract should be removed	4

[GAS-1] Use `selfbalance()` instead of `address(this).balance`

Use assembly when getting a contract's balance of ETH.

You can use `selfbalance()` instead of `address(this).balance` when getting your contract's balance of ETH to save gas. Additionally, you can use `balance(address)` instead of `address.balance()` when getting an external contract's balance of ETH.

Saves 15 gas when checking internal balance, 6 for external

Instances (3):

```
File: PoolPositionRouter.sol

66:         if (address(this).balance > 0) _safeTransferETH(msg.sender,
↪   address(this).balance);

66:         if (address(this).balance > 0) _safeTransferETH(msg.sender,
↪   address(this).balance);

73:         if (IWETH9(address(token)) == WETH9 && address(this).balance >= value) {
```

Link to code

[GAS-2] Use assembly to check for `address(0)`

Saves 6 gas per instance

Instances (5):

```
File: RewardBase.sol

117:         if (account != address(0)) {
```

Link to code

```
File: VoterToken.sol

49:         require(account != address(0), "ERC20: burn from the zero address");

59:         require(account != address(0), "ERC20: mint to the zero address");

77:         require(from != address(0), "ERC20: transfer from the zero address");

78:         require(to != address(0), "ERC20: transfer to the zero address");
```

Link to code

[GAS-3] Using bools for storage incurs overhead

Use uint256(1) and uint256(2) for true/false to avoid a Gwarmaccess (100 gas), and to avoid Gsset (20000 gas) when changing from 'false' to 'true', after having been 'true' in the past. See source.

Instances (7):

```
File: PoolPositionBase.sol

29:     bool public immutable isStatic;
```

Link to code

```
File: VoterToken.sol

18:     mapping(address => bool) public transferers;
```

Link to code

```
File: factories/PoolPositionAndRewardFactory.sol

42:     mapping(address => bool) public isPoolPosition;

49:     bool public postDeploy = false;

63:     mapping(address => bool) public isApprovedRewardToken;

156:     mapping(IReward => bool) public isFactoryVoteReward;

169:     mapping(IReward => bool) public isFactoryLpReward;
```

Link to code

[GAS-4] Cache array length outside of loop

If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

Instances (16):

```
File: Poll.sol

136:         for (uint256 i = 0; i < currentVoteWeights.length; i++) {
```

Link to code

```
File: PoolPositionBase.sol

59:         for (uint256 i = 0; i < _binIds.length; i++) {

107:         for (uint256 i = 0; i < binIds.length; i++) {

117:         for (uint256 i = 1; i < binIds.length; i++) {

160:         for (uint256 i = 0; i < binIds.length; i++) {
```

Link to code

```
File: PoolPositionRouter.sol

107:         for (uint256 i = 0; i < binIds.length; i++) {
```

Link to code

```
File: PoolPositionStatic.sol

34:         for (uint256 i = 0; i < binIds.length; i++) {

59:         for (uint256 i = 0; i < binIds.length; i++) {

74:         for (uint256 i = 0; i < binIds.length; i++) {
```

Link to code

```
File: RewardBase.sol

84:         info = new RewardInfo[](rewardData.length);

85:         for (uint8 i = 1; i < rewardData.length; i++) {

142:         for (uint8 i = 1; i < rewardData.length; i++) {

201:         for (uint8 i = 0; i < rewardTokenIndices.length; i++) {
```

Link to code

```
File: RewardOpen.sol

43:         for (uint8 i = 0; i < rewardTokenIndices.length; i++) {
```

Link to code

```
File: factories/PoolPositionAndRewardFactory.sol

174:         rewardInfos = new RewardInfos[](rewardList.length);

175:         for (uint256 i = 0; i < rewardList.length; i++) {
```

Link to code

[GAS-5] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second+ access of a state variable within a function. Caching of a state variable replaces each Gwarmaccess (100 gas) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

Saves 100 gas per instance

Instances (2):

```
File: PoolPositionDynamic.sol

33:         uint256 sqrtPriceUpperEdge = BinMath.tickSqrtPrice(poolTickSpacing, lowerTick +
↪   1);
```

Link to code

```
File: PoolPositionRouter.sol

129:         mintedPoolPositionTokenAmount = poolPosition.mint(recipient, routerTokenId,
↪   SafeCast.toUint128(lpTokenAmount));
```

Link to code

[GAS-6] Use calldata instead of memory for function arguments that do not get mutated

Mark data types as `calldata` instead of `memory` where possible. This makes it so that the data is not automatically loaded into memory. If the data passed into the function does not need to be changed (like updating values in an array), it can be passed in as `calldata`. The one exception to this is if the argument must later be passed into another function that takes an argument that specifies `memory` storage.

Instances (8):

```
File: Poll.sol

70:     function vote(uint256 veTokenId, IReward[] memory voteTargets, uint256[] memory
↪  weights) external senderIsOwner(veTokenId) {

70:     function vote(uint256 veTokenId, IReward[] memory voteTargets, uint256[] memory
↪  weights) external senderIsOwner(veTokenId) {

158:     function getRewardManyTokensOneVoteRewards(uint256 veTokenId, address to, IReward
↪  voteReward, uint8[] memory rewardTokenIndices) external senderIsOwner(veTokenId) {
```

Link to code

```
File: PoolPositionDynamic.sol

20:         uint128[] memory _binIds,

21:         uint128[] memory _ratios,
```

Link to code

```
File: PoolPositionStatic.sol

19:         uint128[] memory _binIds,

20:         uint128[] memory _ratios,
```

Link to code

```
File: factories/PoolPositionAndRewardFactory.sol

173:     function getLpRewardListInfo(IReward[] memory rewardList) external view returns
↪  (RewardInfos[] memory rewardInfos) {
```

Link to code

[GAS-7] Use Custom Errors

Source Instead of using error strings, to reduce deployment and runtime cost, you should use Custom Errors. This would save both deployment and runtime cost.

Instances (24):

```
File: Poll.sol

32:         require(ve.isApprovedOrOwner(msg.sender, veTokenId), "Poll: sender not approved");

96:             require(factory.isFactoryVoteReward(voteReward), "Poll: must be factory vote
↪   reward contract");

99:             require(poolPositionWeight != 0, "Poll: Must vote with non-zero weight");
```

Link to code

```
File: PoolPositionBase.sol

153:         require(amountMinted != 0, "PP: zero mint");
```

Link to code

```
File: PoolPositionRouter.sol

40:         require(IWETH9(msg.sender) == WETH9, "Router: Not WETH9");

43:         require(block.timestamp <= deadline, "Transaction too old");

48:         require(success, "ETH transfer failed");

52:         require(balanceWETH9 >= amountMinimum, "Router: Insufficient WETH9");

60:         require(balanceToken >= amountMinimum, "Router: Insufficient token");

104:         require(tokenAAmount >= minTokenAAmount && tokenBAmount >= minTokenBAmount,
↪   "Router: Too little added");

169:         require(poolPositionFactory.isPoolPosition(address(poolPosition)), "Router: must
↪   be factory PoolPosition");

174:         require(tokenAAmount <= maxTokenAAmount && tokenBAmount <= maxTokenBAmount,
↪   "Router: Max token amount exceeded");

176:         require(mintedPoolPositionTokenAmount >= minLpTokenAmount, "Router: minimum mint
↪   amount not met");

187:         require(tokenAAmount >= minTokenAAmount && tokenBAmount >= minTokenBAmount,
↪   "Router: Too little recieved");
```

Link to code

```
File: RewardOpen.sol

88:         require(ve.ownerOf(veTokenId) == msg.sender, "Must own veTokenId");
```

Link to code

```
File: RewardVote.sol

14:         require(msg.sender == pollAddress, "VoteReward: can only be called by Poll");

18:         require(msg.sender == poolPositionAddress, "VoteReward: only PP can notify");
```

Link to code

```
File: VoterToken.sol

29:         require(msg.sender == minter, "VoterToken: Must be minter");

33:         require(transferers[msg.sender], "VoterToken: Must be transferer");

49:         require(account != address(0), "ERC20: burn from the zero address");

59:         require(account != address(0), "ERC20: mint to the zero address");

77:         require(from != address(0), "ERC20: transfer from the zero address");

78:         require(to != address(0), "ERC20: transfer to the zero address");

80:         require(fromBalance >= amount, "ERC20: transfer amount exceeds balance");
```

Link to code

[GAS-8] Don't initialize variables with default value

Instances (20):

```
File: Poll.sol

49:         for (uint256 i = 0; i < voteTargetCnt; i++) {

89:         uint256 totalVoteWeight = 0;

90:         for (uint256 i = 0; i < voteTargetCnt; i++) {
```

```
94:         for (uint256 i = 0; i < voteTargetCnt; i++) {

119:         for (uint256 i = 0; i < voteTargetCnt; i++) {

136:         for (uint256 i = 0; i < currentVoteWeights.length; i++) {
```

Link to code

```
File: PoolPositionBase.sol

57:         uint128 lastBinId = 0;

59:         for (uint256 i = 0; i < _binIds.length; i++) {

107:         for (uint256 i = 0; i < binIds.length; i++) {

160:         for (uint256 i = 0; i < binIds.length; i++) {
```

Link to code

```
File: PoolPositionRouter.sol

107:         for (uint256 i = 0; i < binIds.length; i++) {

118:         for (uint256 i = 0; i < length; i++) {
```

Link to code

```
File: PoolPositionStatic.sol

34:         for (uint256 i = 0; i < binIds.length; i++) {

58:         uint256 j = 0;

59:         for (uint256 i = 0; i < binIds.length; i++) {

74:         for (uint256 i = 0; i < binIds.length; i++) {
```

Link to code

```
File: RewardBase.sol

201:         for (uint8 i = 0; i < rewardTokenIndices.length; i++) {
```

Link to code

```
File: RewardOpen.sol

43:         for (uint8 i = 0; i < rewardTokenIndices.length; i++) {
```

Link to code

```
File: factories/PoolPositionAndRewardFactory.sol

57:     uint256 public feeProportionSentOutOfPP = 0;

175:         for (uint256 i = 0; i < rewardList.length; i++) {
```

Link to code

[GAS-9] Long revert strings

Instances (10):

```
File: Poll.sol

96:             require(factory.isFactoryVoteReward(voteReward), "Poll: must be factory vote
↪  reward contract");

99:             require(poolPositionWeight != 0, "Poll: Must vote with non-zero weight");
```

Link to code

```
File: PoolPositionRouter.sol

169:         require(poolPositionFactory.isPoolPosition(address(poolPosition)), "Router: must
↪  be factory PoolPosition");

174:         require(tokenAAmount <= maxTokenAAmount && tokenBAmount <= maxTokenBAmount,
↪  "Router: Max token amount exceeded");

176:         require(mintedPoolPositionTokenAmount >= minLpTokenAmount, "Router: minimum mint
↪  amount not met");
```

Link to code

```
File: RewardVote.sol

14:         require(msg.sender == pollAddress, "VoteReward: can only be called by Poll");
```

Link to code

```
File: VoterToken.sol

49:         require(account != address(0), "ERC20: burn from the zero address");

77:         require(from != address(0), "ERC20: transfer from the zero address");

78:         require(to != address(0), "ERC20: transfer to the zero address");

80:         require(fromBalance >= amount, "ERC20: transfer amount exceeds balance");
```

Link to code

[GAS-10] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided.

Instances (14):

```
File: Distributor.sol

28:     function setMonthDisbursement(uint256 value) public onlyOwner {
```

Link to code

```
File: Poll.sol

57:     function addVotingRewardsContractToTokenPermissions(address rewardContract) public
↪   onlyFactory {

60:     function attachGlobalVoteReward(IReward _globalVoteReward) external onlyFactory {

64:     function attachRewardPusher(IRewardPusher _rewardPusher) public onlyFactory {
```

Link to code

```
File: RewardSingle.sol

23:     function setPusher(address _rewardPusherAddress) public onlyFactory {
```

Link to code

```
File: VoterToken.sol

42:     function addTransferer(address transferer) public onlyMinter {

45:     function removeTransferer(address transferer) public onlyMinter {

48:     function burnAll(address account) public onlyMinter {

58:     function mint(address account, uint256 amount) public onlyMinter {
```

Link to code

```
File: factories/PoolPositionAndRewardFactory.sol

83:     function updateFeeProportionSentOutOfPP(uint256 newValue) external onlyOwner {

87:     function updateExtractedFeeShareToVoter(uint256 newValue) external onlyOwner {

91:     function updateMiniumLockPeriod(uint256 newValue) external onlyOwner {

94:     function addNewApprovedRewardToken(address rewardToken) external onlyOwner {

97:     function deployMav(address mintTo) external onlyOwner {
```

Link to code

[GAS-11] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)

Saves 5 gas per loop

Instances (23):

```
File: Poll.sol

49:         for (uint256 i = 0; i < voteTargetCnt; i++) {

90:         for (uint256 i = 0; i < voteTargetCnt; i++) {

94:         for (uint256 i = 0; i < voteTargetCnt; i++) {

119:         for (uint256 i = 0; i < voteTargetCnt; i++) {

136:         for (uint256 i = 0; i < currentVoteWeights.length; i++) {
```

Link to code

```
File: PoolPositionBase.sol

59:         for (uint256 i = 0; i < _binIds.length; i++) {

107:         for (uint256 i = 0; i < binIds.length; i++) {

117:         for (uint256 i = 1; i < binIds.length; i++) {

160:         for (uint256 i = 0; i < binIds.length; i++) {
```

Link to code

```
File: PoolPositionRouter.sol

107:         for (uint256 i = 0; i < binIds.length; i++) {

118:         for (uint256 i = 0; i < length; i++) {
```

Link to code

```
File: PoolPositionStatic.sol

34:         for (uint256 i = 0; i < binIds.length; i++) {

59:         for (uint256 i = 0; i < binIds.length; i++) {

63:                 j++;

74:         for (uint256 i = 0; i < binIds.length; i++) {
```

Link to code

```
File: RewardBase.sol

85:         for (uint8 i = 1; i < rewardData.length; i++) {

93:         for (uint8 i = 1; i < length; i++) {

142:         for (uint8 i = 1; i < rewardData.length; i++) {

154:             for (uint8 i = 1; i < MAX_REWARD_TOKENS + 1; i++) {

163:             _data.globalResetCount++;

201:         for (uint8 i = 0; i < rewardTokenIndices.length; i++) {
```

Link to code

```
File: RewardOpen.sol

43:         for (uint8 i = 0; i < rewardTokenIndices.length; i++) {
```

Link to code

```
File: factories/PoolPositionAndRewardFactory.sol

175:         for (uint256 i = 0; i < rewardList.length; i++) {
```

Link to code

[GAS-12] Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

Instances (4):

```
File: RewardBase.sol

34:     uint256 public constant MAX_DURATION = 30 days;
```

Link to code

```
File: VoterToken.sol

21:     string public constant name = "Maverick Voter Token";

22:     string public constant symbol = "MVT";

23:     uint8 public constant decimals = 18;
```

Link to code

[GAS-13] Splitting require() statements that use && saves gas

Instances (3):

```
File: PoolPositionRouter.sol

104:         require(tokenAAmount >= minTokenAAmount && tokenBAmount >= minTokenBAmount,
↪  "Router: Too little added");

174:         require(tokenAAmount <= maxTokenAAmount && tokenBAmount <= maxTokenBAmount,
↪  "Router: Max token amount exceeded");

187:         require(tokenAAmount >= minTokenAAmount && tokenBAmount >= minTokenBAmount,
↪  "Router: Too little recieved");
```

Link to code

[GAS-14] Use != 0 instead of > 0 for unsigned integer comparison

Instances (4):

```
File: PoolPositionRouter.sol

53:         if (balanceWETH9 > 0) {

61:         if (balanceToken > 0) {

66:         if (address(this).balance > 0) _safeTransferETH(msg.sender,
↪  address(this).balance);
```

Link to code

```
File: RewardBase.sol

193:         if (reward > 0) {
```

Link to code

[GAS-15] `internal` functions not called by the contract should be removed

If the functions are required by an interface, the contract should inherit from that interface and use the `override` keyword

Instances (4):

```
File: PoolPositionBase.sol

138:     function _transferFeesOut(IPool.RemoveLiquidityParams[] memory params) internal {
```

Link to code

```
File: RewardBase.sol

171:     function _stake(address supplier, uint256 amount, address account) internal
↪   nonReentrant checkAmount(amount) {

185:     function _unstakeAll(address account, address recipient) internal {

200:     function _getRewardList(address account, address recipient, uint8[] memory
↪   rewardTokenIndices) internal {
```

Link to code