Bunni Security Review

Pashov Audit Group

Conducted by: pontifex, Said, ast3ros

October 17th - October 30th

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **Bunniapp/tokenomics** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Bunni

Bunni v2 is an AMM that allows liquidity providers to manage complex liquidity shapes and shift between them seamlessly. It includes features like constant gas costs for swaps, automatic compounding of fees, and autonomous rebalancing to maintain optimal token ratios. The scope included an executor contract for Uniswap X, Zap contracts for enhanced interaction with Bunni v2, smart contracts for BUNNI token migration and a capped staking pool.

5. Risk Classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hashes :

- 13a77bfa1983336e6fb5980a042d503f0e8b6c25
- 762868c8283ff4812dedab8813a971f77fe14b10
- 2a0e0782776dd753ef88fd23978e18d0cff27ce0

fixes review commit hash :

- 5721ae480dca2b5f85226342466ad5dcce54c4d3
- 762c8835fb56087a3c9317a3965c5fc15f2b39ac
- 45f7e77410bfbc5e12deb086251154743839fa26

Scope

The following smart contracts were in scope of the audit:

- `BunniHookOracle`
- `ERC20Multicaller`
- `RecurPoolId`
- `RecurPoolKey`
- `RushPoolId`
- `RushPoolKey`
- `BUNNI`
- `MasterBunni`
- `OptionsToken`
- `SmartWalletChecker`
- `TokenMigrator`
- `VeAirdrop`
- `LibMulticaller`
- `VotingEscrow`
- `BunniExecutor`
- `BunniZapIn`
- `ReentrancyGuard`

7. Executive Summary

Over the course of the security review, pontifex, Said, ast3ros engaged with Bunni to review Bunni. In this period of time a total of **21** issues were uncovered.

Protocol Summary

Protocol Name	Bunni
Repository	https://github.com/Bunniapp/tokenomics
Date	October 17th - October 30th
Protocol Type	DEX

Findings Count

Severity	**Amount**
Critical	3
High	3
Medium	4
Low	11
Total Findings	**21**

Summary of Findings

ID	Title	Severity	Status
[C-01]	claimRecurPool does not verify the provided incentiveToken is equal to rewardToken	Critical	Resolved
[C-02]	incentiveToken is not verified within incentivizeRecurPool	Critical	Resolved
[C-03]	Previous rewardRate is removed when periodFinish has not yet been reached	Critical	Resolved
[H-01]	joinRecurPool can incorrectly increment userPoolCounts	High	Resolved
[H-02]	_settleCurrency does not properly interact with the poolManager when calling settle	High	Resolved
[H-03]	Lack of receive() function	High	Resolved
[M-01]	depositIncentive and incentivizeRecurPool do not verify if the incentiveToken exists	Medium	Resolved
[M-02]	DOS Attack in joinRushPool	Medium	Resolved
[M-03]	Rewards are permanently locked when totalSupply = 0 in RecurPools	Medium	Resolved
[M-04]	Incorrect tick rounding in TWAP calculation	Medium	Acknowledged
[L-01]	Incorrect domain separator caching could break permit2 integration	Low	Resolved
[L-02]	safeApprove can be reverted if the token is USDT	Low	Resolved

[L-03]	Lack of BunniHookOracle's parameters validation	Low	Resolved
[L-04]	Lack of token decimals check	Low	Resolved
[L-05]	Merkle tree leaf generation is single-hashed and might lead to a second preimage attack	Low	Resolved
[L-06]	Incorrect accounting for fee-on-transfer tokens	Low	Acknowledged
[L-07]	Overflow due to unsafe uint64 casting	Low	Resolved
[L-08]	incentivizeRecurPool does not verify that the added newRewardRate is 0.	Low	Resolved
[L-09]	Missing nonReentrant modifier	Low	Resolved
[L-10]	Lack of address(0) check for recipient address	Low	Resolved
[L-11]	Lack of slippage protection	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] `claimRecurPool` does not verify the provided `incentiveToken` is equal to `rewardToken`

Severity

Impact: High

Likelihood: High

Description

It can be observed that when `claimRecurPool` is called, it doesn't verify if the `incentiveToken` provided is equal to the claimed `RecurPoolKey`'s `rewardToken`.

```
function claimRecurPool(
      RecurClaimParams[]calldataparams,
      addressrecipient
    ) external nonReentrant {
        address msgSender = LibMulticaller.senderOrSigner();
        for (uint256 i; i < params.length; i++) {
>>>         address incentiveToken = params[i].incentiveToken;
            uint256 totalClaimableAmount;

            for (uint256 j; j < params[i].keys.length; j++) {
                RecurPoolKey calldata key = params[i].keys[j];
                RecurPoolId id = key.toId();

                // key should be valid
                if (!isValidRecurPoolKey(key)) continue;

                ///
                // -----------------------------------------------------------------
                /// Storage loads
                ///
                // -----------------------------------------------------------------

                // load state
                RecurPoolState storage state = recurPoolStates[id];
                uint64 lastUpdateTime = state.lastUpdateTime;
                uint64 periodFinish = state.periodFinish;
                uint64 lastTimeRewardApplicable =
                    block.timestamp < periodFinish ? uint64
                      (block.timestamp) : periodFinish;
                uint256 rewardPerTokenUpdated = _rewardPerToken(
                    state.rewardPerTokenStored,
                    state.totalSupply,
                    lastTimeRewardApplicable,
                    lastUpdateTime,
                    state.rewardRate
                );

                ///
                // -----------------------------------------------------------------
                /// State updates
                ///
                // -----------------------------------------------------------------

                // accrue rewards
                uint256 reward = _earned(
                    state.userRewardPerTokenPaid[msgSender],
                    state.balanceOf[msgSender],
                    rewardPerTokenUpdated,
                    state.rewards[msgSender]
                );
                state.rewardPerTokenStored = rewardPerTokenUpdated;
                state.lastUpdateTime = lastTimeRewardApplicable;
                state.userRewardPerTokenPaid[msgSender] = rewardPerTokenUpdated;

                if (reward != 0) {
                    // delete accrued rewards
                    delete state.rewards[msgSender];

                    // accumulate claimable amount
                    totalClaimableAmount += reward;
                }
            }

            // transfer incentive tokens to user
            if (totalClaimableAmount != 0) {
                // @audit - incentiveToken is not checked!
>>>             incentiveToken.safeTransfer(recipient, totalClaimableAmount);
```

```
            }

            // emit event
            emit ClaimReward
              (msgSender, incentiveToken, recipient, totalClaimableAmount);
        }
    }
```

An attacker can exploit this by creating a fake Recur Pool and providing an arbitrary/worthless `rewardToken` to increase its `rewardRate`. Then, when `claimRecurPool` is called, the attacker can set `incentiveToken` to another token that they want to steal from `MasterBunni`.

Recommendations

Validate that the provided `incentiveToken` is equal to the `rewardToken` of each `RecurPoolKey`.

[C-02] `incentiveToken` is not verified within `incentivizeRecurPool`

Severity

Impact: High

Likelihood: High

Description

When `incentivizeRecurPool` is called, it will iterate through params and update `state.rewardRate` based on the provided `incentiveAmount`.

```
function incentivizeRecurPool
     (RecurIncentiveParams[] calldata params, address incentiveToken)
        external
        returns (uint256 totalIncentiveAmount)
    {
        address msgSender = LibMulticaller.senderOrSigner();
        for (uint256 i; i < params.length; i++) {
            ///
            // -----------------------------------------------------------------------
            /// Validation
            ///
            // -----------------------------------------------------------------------

            if (params[i].incentiveAmount == 0) continue;

            RecurPoolKey calldata key = params[i].key;
            if (!isValidRecurPoolKey(key)) continue;

            // ...

            ///
            // -----------------------------------------------------------------------
            /// State updates
            ///
            // -----------------------------------------------------------------------

            // ...

            // record new reward
            uint256 newRewardRate;
            if (block.timestamp >= periodFinish) {
                // current period is over
                // uint256 internal constant REWARD_RATE_PRECISION = 1e6;
                newRewardRate = params[i].incentiveAmount.mulDiv
                  (REWARD_RATE_PRECISION, key.duration);
                state.rewardRate = newRewardRate;
                state.lastUpdateTime = uint64(block.timestamp);
                state.periodFinish = uint64(block.timestamp + key.duration);
            } else {
                // period is still active
                // add the new reward to the existing period
                uint256 remaining = periodFinish - block.timestamp;
                newRewardRate += params[i].incentiveAmount.mulDiv
                  (REWARD_RATE_PRECISION, remaining);

                state.rewardRate = newRewardRate;
                state.lastUpdateTime = uint64(block.timestamp);
            }
            // prevent overflow when computing rewardPerToken
            if (newRewardRate >= ((type
              (uint256).max / PRECISION_DIV_REWARD_RATE_PRECISION) / key.duration)) {
                revert MasterBunni__AmountTooLarge();
            }

            totalIncentiveAmount += params[i].incentiveAmount;
        }

        // transfer incentive tokens from msgSender to this contract
        if (totalIncentiveAmount != 0) {
>>>         incentiveToken.safeTransferFrom2(msgSender, address
  (this), totalIncentiveAmount);
        }

        // ...
    }
```

However, within the loop, it never validates that the current `rewardToken` is equal to `incentiveToken`, allowing an attacker to increase `state.rewardRate` without providing the actual `rewardToken`. This results in users being unable to claim the reward due to the unavailability of `rewardToken` in the contract, even when legitimate incentives exist.

Recommendations

Validate that `rewardToken` is equal to `incentiveToken` within the `incentivizeRecurPool` calls.

[C-03] Previous `rewardRate` is removed when `periodFinish` has not yet been reached

Severity

Impact: High

Likelihood: High

Description

When `incentivizeRecurPool` is called and the `periodFinish` has not yet been reached, it incorrectly sets `newRewardRate`, which consists of only the new `incentiveAmount`, to `state.rewardRate` instead of adding it to `state.rewardRate`.

```
function incentivizeRecurPool
      (RecurIncentiveParams[] calldata params, address incentiveToken)
        external
        returns (uint256 totalIncentiveAmount)
    {
        // ...

            // record new reward
>>>         uint256 newRewardRate;
            if (block.timestamp >= periodFinish) {
                // current period is over
                // uint256 internal constant REWARD_RATE_PRECISION = 1e6;
                newRewardRate = params[i].incentiveAmount.mulDiv
                  (REWARD_RATE_PRECISION, key.duration);
                state.rewardRate = newRewardRate;
                state.lastUpdateTime = uint64(block.timestamp);
                state.periodFinish = uint64(block.timestamp + key.duration);
            } else {
                // period is still active
                // add the new reward to the existing period
                uint256 remaining = periodFinish - block.timestamp;
                // @audit - this is removing previous rewardRate
>>>             newRewardRate += params[i].incentiveAmount.mulDiv
  (REWARD_RATE_PRECISION, remaining);

>>>             state.rewardRate = newRewardRate;
                state.lastUpdateTime = uint64(block.timestamp);
            }
            // prevent overflow when computing rewardPerToken
            if (newRewardRate >= ((type
              (uint256).max / PRECISION_DIV_REWARD_RATE_PRECISION) / key.duration)) {
                revert MasterBunni__AmountTooLarge();
            }

            totalIncentiveAmount += params[i].incentiveAmount;
        }

        // ...
    }
```

This effectively removes the previous `rewardRate` from the other incentive provider, causing the reward to be lost.

Recommendations

Add the `newRewardRate` to `state.rewardRate`, instead of replacing `state.rewardRate` with `newRewardRate`.

```
function incentivizeRecurPool
      (RecurIncentiveParams[] calldata params, address incentiveToken)
        external
        returns (uint256 totalIncentiveAmount)
    {
        // ...

            // record new reward
            uint256 newRewardRate;
            if (block.timestamp >= periodFinish) {
                // current period is over
                // uint256 internal constant REWARD_RATE_PRECISION = 1e6;
                newRewardRate = params[i].incentiveAmount.mulDiv
                  (REWARD_RATE_PRECISION, key.duration);
                state.rewardRate = newRewardRate;
                state.lastUpdateTime = uint64(block.timestamp);
                state.periodFinish = uint64(block.timestamp + key.duration);
            } else {
                // period is still active
                // add the new reward to the existing period
                uint256 remaining = periodFinish - block.timestamp;
                // @audit - this is removing previous rewardRate
                newRewardRate += params[i].incentiveAmount.mulDiv
                  (REWARD_RATE_PRECISION, remaining);

-                state.rewardRate = newRewardRate;
+                state.rewardRate += newRewardRate;
                state.lastUpdateTime = uint64(block.timestamp);
            }
            // prevent overflow when computing rewardPerToken
            if (newRewardRate >= ((type
              (uint256).max / PRECISION_DIV_REWARD_RATE_PRECISION) / key.duration)) {
                revert MasterBunni__AmountTooLarge();
            }

            totalIncentiveAmount += params[i].incentiveAmount;
        }

        // ...
    }
```

8.2. High Findings

[H-01] `joinRecurPool` can incorrectly increment `userPoolCounts`

Severity

Impact: High

Likelihood: Medium

Description

Users can call `joinRecurPool` to update their staked balance when their current `stakeToken` balance increases.

```
function joinRecurPool(RecurPoolKey[] calldata keys) external nonReentrant {
        address msgSender = LibMulticaller.senderOrSigner();
        for (uint256 i; i < keys.length; i++) {
            RecurPoolKey calldata key = keys[i];

            ///
            // -----------------------------------------------------------------------
            /// Validation
            ///
            // -----------------------------------------------------------------------

            // key should be valid
            if (!isValidRecurPoolKey(key)) continue;

            // user should have non-zero balance
            uint256 balance = ERC20(address(key.stakeToken)).balanceOf
              (msgSender);
            if (balance == 0) {
                continue;
            }

            // user's balance should be locked with this contract as the
            // unlocker
            if (
                !key.stakeToken.isLocked(msgSender)
                    || key.stakeToken.unlockerOf(msgSender) != IERC20Unlocker
                      (address(this))
            ) {
                continue;
            }

            ///
            // -----------------------------------------------------------------------
            /// Storage loads
            ///
            // -----------------------------------------------------------------------

            RecurPoolId id = key.toId();
            RecurPoolState storage state = recurPoolStates[id];
            uint256 stakedBalance = state.balanceOf[msgSender];

            // can't stake in a pool twice
>>>         if (balance <= stakedBalance) {
                continue;
            }

            // ...

            // stake
            state.totalSupply = totalSupply - stakedBalance + balance;
            state.balanceOf[msgSender] = balance;

            // increment user pool count
            // @audit - this should check balance before is 0
            unchecked {
>>>             ++userPoolCounts[msgSender][key.stakeToken];
            }

            // emit event
            emit JoinRecurPool(msgSender, keys[i]);
        }
    }
```

However, it will also increment `userPoolCounts` even when the operation is only updating `stakedBalance` and not the first time joining the recur pool.

Incorrectly incrementing `userPoolCounts` here will prevent users from unlocking their tokens.

Recommendations

Increment `userPoolCounts` only if the previous `stakedBalance` is 0.

[H-02] `_settleCurrency` does not properly interact with the `poolManager` when calling `settle`

Severity

Impact: Medium

Likelihood: High

Description

After the swap is performed within the executor's `unlockCallback`, it will settle all the currency by triggering `_settleCurrency`. However, when settling currency on Uniswap V4's `poolManager`, the correct `settle` function is not used, causing the interaction with Uniswap V4 to always fail.

```
function _settleCurrency(Currency currency) internal {
        int256 amount = poolManager.currencyDelta(address(this), currency);
        if (amount < 0) {
            // address(this) owes PoolManager currency
            // contract already has input tokens in its balance
            // so we directly transfer tokens to PoolManager
            uint256 absAmount = uint256(-amount);
            if (currency.isNative()) {
                // native currency (e.g. ETH)
                poolManager.settle{value: absAmount}();
            } else {
                // ERC20 token
                poolManager.sync(currency);
                currency.transfer(address(poolManager), absAmount);
                // @audit - should use poolManager.settle()
                IPoolManagerOld(address(poolManager)).settle
                //(currency); // TODO: compatible with old sepolia v4 deploy, use pool
            }
        } else if (amount > 0) {
            // address(this) has positive balance in PoolManager
            // take tokens from PoolManager to address(this)
            // the reactor will use transferFrom() to take tokens from address
            //(this)
            poolManager.take(currency, address(this), uint256(amount));
        }
    }
```

Recommendations

Use the mainnet `settle` :

```
function _settleCurrency(Currency currency) internal {
        int256 amount = poolManager.currencyDelta(address(this), currency);
        if (amount < 0) {
            // address(this) owes PoolManager currency
            // contract already has input tokens in its balance
            // so we directly transfer tokens to PoolManager
            uint256 absAmount = uint256(-amount);
            if (currency.isNative()) {
                // native currency (e.g. ETH)
                poolManager.settle{value: absAmount}();
            } else {
                // ERC20 token
                poolManager.sync(currency);
                currency.transfer(address(poolManager), absAmount);
                // @audit - should use poolManager.settle()
-               IPoolManagerOld(address(poolManager)).settle
- (currency); // TODO: compatible with old sepolia v4 deploy, use poolManager.settle()
+               poolManager.settle();
            }
        } else if (amount > 0) {
            // address(this) has positive balance in PoolManager
            // take tokens from PoolManager to address(this)
            // the reactor will use transferFrom() to take tokens from address
            //(this)
            poolManager.take(currency, address(this), uint256(amount));
        }
    }
```

[H-03] Lack of `receive()` function

Severity

Impact: Medium

Likelihood: High

Description

The `BunniZapIn` contract should be able to receive native ETH as a result of the `unwrapEthOutput` function execution and refunded amounts from deposits and swaps. In order to receive this ETH there should be either `fallback` or `receive` method in the contract. However, this contract has neither.

```
function unwrapEthOutput() external nonReentrant {
        weth.withdraw(weth.balanceOf(address(this)));
    }
```

Recommendations

Consider implementing the `receive()` function:

```
receive() external payable {}
```

8.3. Medium Findings

[M-01] `depositIncentive` and `incentivizeRecurPool` do not verify if the `incentiveToken` exists

Severity

Impact: High

Likelihood: Low

Description

Both `depositIncentive` and `incentivizeRecurPool` use solady's `SafeTransferLib` to transfer the `incentiveToken` to the `MasterBunni`.

```
function depositIncentive(
        RushIncentiveParams[]calldataparams,
        addressincentiveToken,
        addressrecipient
    )
        external
        nonReentrant
        returns (uint256 totalIncentiveAmount)
    {
        // ...

        // transfer incentive tokens to this contract
        if (totalIncentiveAmount != 0) {
            // @audit - doesn't use the latest version of solady
            incentiveToken.safeTransferFrom2(msgSender, address
              (this), totalIncentiveAmount);
        }

        // emit event
        emit DepositIncentive
          (msgSender, incentiveToken, recipient, params, totalIncentiveAmount);
    }
```

Inside solady's implementation, if there is no return data, the function will always success:

```
function trySafeTransferFrom
      (address token, address from, address to, uint256 amount)
        internal
        returns (bool success)
    {
        /// @solidity memory-safe-assembly
        assembly {
            let m := mload(0x40) // Cache the free memory pointer.
            mstore(0x60, amount) // Store the `amount` argument.
            mstore(0x40, to) // Store the `to` argument.
            mstore(0x2c, shl(96, from)) // Store the `from` argument.
            mstore(0x0c, 0x23b872dd000000000000000000000000) // `transferFrom
            //(address,address,uint256)`.
            success :=
                and( // The arguments of `and` are evaluated from right to left.
                    or(eq(mload(0x00), 1), iszero(returndatasize
                    //()))), // Returned 1 or nothing.
                    call(gas(), token, 0, 0x1c, 0x64, 0x00, 0x20)
                )
            mstore(0x60, 0) // Restore the zero slot to zero.
            mstore(0x40, m) // Restore the free memory pointer.
        }
    }
```

This means that an attacker can provide a non-contract to the functions, and the function will succeed. This is problematic in cases where the `incentiveToken` is a soon-to-be-created contract with a predictable address, such as a Bunni LP token. For instance, if users want to create a reward pool for staking a Bunni LP with another soon-to-be-created Bunni LP token, the attacker can front-run the operation, provide fake rewards, and disrupt the pool rewards accounting.

Recommendations

Consider checking the code size of `incentiveToken`, or simply use the latest version of solady, where the code size is also verified within the library.

[M-02] DOS Attack in joinRushPool

Severity

Impact: High

Likelihood: Low

Description

The `joinRushPool` function in MasterBunni allows users to stake tokens up to a maximum cap. However, the lack of time restrictions between joining and exiting a pool creates a vulnerability where malicious actors can execute sandwich attacks to prevent legitimate users from staking.

```
function joinRushPool(RushPoolKey[] calldata keys) external nonReentrant {
        ...
            {
                uint256 balance = ERC20(address(keys[i].stakeToken)).balanceOf
                  (msgSender);

                                stakeAmountUpdated = remainderStakeAmount + balance >
                     ? keys[i].stakeCap - remainderStakeAmount
                     : balance;
            }

            // ensure there is capacity left and that we're increasing the
            // user's stake
            // the user's stake may increase when either
            // 1) the user isn't staked yet or
            // 2) the user staked & hit the stake cap but more capacity has
            // opened up since then
            if
              (stakeAmountUpdated == 0 || stakeAmountUpdated <= userState.stakeAmount)
                continue;
            }
        ...
    }
```

Attacker's steps:

- Attacker observes a pending `joinRushPool` transaction
- Attacker front-runs by calling `joinRushPool` to fill the pool to its cap
- Victim's transaction reverts due to no remaining capacity
- Attacker back-runs by calling `exitRushPool` to withdraw their stake

The attacker can repeat this pattern to consistently block other users from joining the pool. The only cost is gas fees for the sandwich transactions.

Recommendations

It's recommended that minimum stake duration be added or an unstaking delay implemented.

[M-03] Rewards are permanently locked when totalSupply = 0 in RecurPools

Severity

Impact: Medium

Likelihood: Medium

Description

In the MasterBunni contract, when calculating rewards for RecurPools using the `_rewardPerToken` function, any rewards allocated during periods where `totalSupply = 0` become permanently locked in the contract. This occurs because:

1. When `totalSupply = 0`, the function simply returns the existing rewardPerTokenStored without accounting for the elapsed time:

```
function _rewardPerToken(
        uint256 rewardPerTokenStored,
        uint256 totalSupply,
        uint256 lastTimeRewardApplicable,
        uint256 lastUpdateTime,
        uint256 rewardRate
    ) internal pure returns (uint256) {
        if (totalSupply == 0) {
            return rewardPerTokenStored; // @audit rewardPerTokenStored isn't
            // updated but lastUpdateTime is updated
        }
        // mulDiv won't overflow since we check that rewardRate is less than
        //(type(uint256).max / PRECISION_DIV_REWARD_RATE_PRECISION / duration)
        return rewardPerTokenStored
            + FixedPointMathLib.mulDiv(
                (
                 lastTimeRewardApplicable-lastUpdateTime
                ) * PRECISION_DIV_REWARD_RATE_PRECISION, rewardRate, totalSupply
            );
    }
```

2. After `_rewardPerToken` is called, the `lastUpdateTime` is always updated:

```
state.lastUpdateTime = lastTimeRewardApplicable;
```

This means that for any period where the pool has incentives allocated (`rewardRate > 0`) and the `total supply is 0` (no stakers) and time passes between `lastUpdateTime` and `periodFinish`, the rewards meant for distribution during this period become permanently locked in the contract as:

- They are not distributed to any stakers
- There is no mechanism for the incentive depositor to recover them
- The time period is marked as processed due to `lastUpdateTime` being updated

Recommendations

It's recommended to

- skip updating `lastUpdateTime` when `totalSupply = 0` to allow the rewards to accumulate for future stakers or
- allow incentive providers to reclaim undistributed rewards after the period ends

[M-04] Incorrect tick rounding in TWAP calculation

Severity

Impact: Low

Likelihood: High

Description

In `BunniHookOracle._queryTwap()`, the arithmetic mean tick is calculated by dividing `tickCumulativesDelta` by the window size. However when `tickCumulativesDelta` is negative, Solidity's integer division rounds towards zero (upward). It can lead to tick values being off by 1, affecting price calculations. It should consistently round down towards negative infinity.

```
function _queryTwap(
        PoolKeymemorypoolKey,
        uint32twapSecondsAgoStart,
        uint32twapSecondsAgoEnd
    )
        internal
        view
        returns (int24 arithmeticMeanTick)
    {
        ...
        int56 tickCumulativesDelta = tickCumulatives[1] - tickCumulatives[0];
        return int24(tickCumulativesDelta / int56(uint56
        //(windowSize))); // @audit rounding towards zero if tickCumulativesDelta < 0
    }
```

Recommendations

Implement consistent rounding-down behavior for the tick calculation:

```
function _queryTwap(
      PoolKeymemorypoolKey,
      uint32twapSecondsAgoStart,
      uint32twapSecondsAgoEnd
    )
        internal
        view
        returns (int24 arithmeticMeanTick)
    {
        ...
        int56 tickCumulativesDelta = tickCumulatives[1] - tickCumulatives[0];
+       arithmeticMeanTick = int24(tickCumulativesDelta / int56(uint56
+ (windowSize)));
+       // Always round to negative infinity
+       if (tickCumulativesDelta < 0 &&
+ (tickCumulativesDelta % windowSize != 0)) arithmeticMeanTick--;
+       return arithmeticMeanTick;
-       return int24(tickCumulativesDelta / int56(uint56(windowSize)));
    }
```

8.4. Low Findings

[L-01] Incorrect domain separator caching could break permit2 integration

The `BunniZapIn` contract caches the Permit2 domain separator as an immutable variable during construction. This can cause signature verification failures if the chain undergoes a hard fork that changes the `chainID`, as the domain separator would no longer match the one used by Permit2.

```
constructor(address payable zeroExProxy_, WETH weth_, IBunniHub bunniHub_) {
        zeroExProxy = zeroExProxy_;
        weth = weth_;
        bunniHub = bunniHub_;
        permit2DomainSeparator = IPermit2
        //(SafeTransferLib.PERMIT2).DOMAIN_SEPARATOR(); // @audit permit2DomainSeparat
    }
```

The Permit2 contract dynamically recalculates its domain separator when the chain ID changes:

```
/// @notice Returns the domain separator for the current chain.
    /// @dev Uses cached version if chainid and address are unchanged from
    // construction.
    function DOMAIN_SEPARATOR() public view returns (bytes32) {
        return block.chainid == _CACHED_CHAIN_ID
            ? _CACHED_DOMAIN_SEPARATOR
            : _buildDomainSeparator(_TYPE_HASH, _HASHED_NAME);
    }
```

It's recommended instead of caching the domain separator immutably, the contract should query it dynamically from Permit2 when needed.

[L-02] `safeApprove` can be reverted if the token is USDT

In `BunniExecutor` contract, when approving the swap output tokens to the reactor, if the token is USDT and it's already approved to an amount different from 0, the next approval would revert.

```
function unlockCallback
      (bytes calldata data) external onlyPoolManager returns (bytes memory) {
        ...
        // approve swap output tokens to the reactor
        for (uint256 i; i < tokensToApproveForReactor.length; i++) {
            tokensToApproveForReactor[i].safeApprove(address(reactor), type
            //(uint256).max); // @audit revert if token is USDT
        }
        ...
    }
```

It's recommended to set the approval to 0 before approving the new amount.

[L-03] Lack of `BunniHookOracle`'s parameters validation

The `BunniHookOracle` contract parameters should fit in reasonable ranges to prevent unexpected behavior of the contract: `multiplier` should be less than `MULTIPLIER_DENOM`, `secs_` and `ago_` should be reasonable.

```
constructor(
        IBunniHook bunniHook_,
        PoolKey memory poolKey,
        address paymentToken_,
        address underlyingToken_,
        address owner_,
        uint16 multiplier_,
        uint32 secs_,
        uint32 ago_,
        uint128 minPrice_
    ) {
        bunniHook = bunniHook_;
        paymentToken = paymentToken_;
        underlyingToken = underlyingToken_;

>>      multiplier = multiplier_;
>>      secs = secs_;
>>      ago = ago_;
        minPrice = minPrice_;
<...>
    function setParams(
      uint16multiplier_,
      uint32secs_,
      uint32ago_,
      uint128minPrice_
    ) external onlyOwner {
>>      multiplier = multiplier_;
>>      secs = secs_;
>>      ago = ago_;
        minPrice = minPrice_;
        emit SetParams(multiplier_, secs_, ago_, minPrice_);
    }
```

[L-04] Lack of token decimals check

Though the `OptionsToken` contract is compatible only with 18 decimals tokens there are no checks, which could prevent usage of tokens with other decimals neither in the `OptionsToken` nor `BunniHookOracle` contract

```
/// @dev Assumes the underlying token and the payment token both use 18
// decimals.
contract OptionsToken is ERC20Multicaller, Ownable {
```

Consider implementing a corresponding check either in the `OptionsToken` or `BunniHookOracle` contract.

[L-05] Merkle tree leaf generation is single-hashed and might lead to a second preimage attack

Merkle trees whose leaves are just single-hashed are vulnerable to second preimage attack. The correct way is to double-hash them as OpenZeppelin suggests. The problem exists in the `VeAirdrop` contract:

```
function claim(uint256 amount, bytes32[] calldata proof) external {
<...>
        bytes32 leaf = keccak256(abi.encodePacked(msgSender, amount));
```

Consider following the OpenZeppelin recommendations:

```
bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(addr, amount))));
```

[L-06] Incorrect accounting for fee-on-transfer tokens

The `MasterBunni` contract fails to properly account for fee-on-transfer tokens when they are used as incentive tokens. The contract records the pre-fee amount rather than the actual received amount after fees are deducted.

In the depositIncentive function:

```
function depositIncentive(
       RushIncentiveParams[]calldataparams,
       addressincentiveToken,
       addressrecipient
    )
        ...
                        rushPoolIncentiveAmounts[id][incentiveToken] += params[i].inc

            // add incentive to depositor
            rushPoolIncentiveDeposits[id][incentiveToken][recipient] +=
            // params[i].incentiveAmount; // @audit Records full amount before fees
        ...

        // transfer incentive tokens to this contract
        if (totalIncentiveAmount != 0) {
            incentiveToken.safeTransferFrom2(msgSender, address
            //(this), totalIncentiveAmount); // @audit Actual received amount will be
        }
        ...
    }
```

This creates a discrepancy between the recorded incentive amounts in contract state and the actual token balance held by the contract. It can lead to:

- Some users will be unable to withdraw their full recorded incentive amounts since the contract holds less tokens than accounted for.
- The last users to withdraw may have their transactions revert due to insufficient balance.

```
function withdrawIncentive(
       RushIncentiveParams[]calldataparams,
       addressincentiveToken,
       addressrecipient
    )
        external
        nonReentrant
        returns (uint256 totalWithdrawnAmount)
    {

        // transfer incentive tokens to recipient
        if (totalWithdrawnAmount != 0) {
            incentiveToken.safeTransfer
            //(recipient, totalWithdrawnAmount); // @audit Will revert if contract bal
        }
        ...
    }
```

Track actual received amounts by calculating the token balance difference before and after transfer.

[L-07] Overflow due to unsafe uint64 casting

The issue occurs in `MasterBunni.incentivizeRecurPool()` where `block.timestamp + key.duration` is unsafely cast to `uint64`. Since `key.duration` is a user-controlled uint256 parameter, their sum can easily exceed `type(uint64).max`.

```
function incentivizeRecurPool
       (RecurIncentiveParams[] calldata params, address incentiveToken)
         external
         returns (uint256 totalIncentiveAmount)
     {
         ...
                 state.rewardRate = newRewardRate;
                 state.lastUpdateTime = uint64(block.timestamp);
                 state.periodFinish = uint64
                 //(block.timestamp + key.duration); // @audit overflow if block.timest
                 ...
     }
```

It's recommended to add a validation check to ensure the timestamp sum doesn't exceed uint64 maximum value.

[L-08] `incentivizeRecurPool` does not verify that the added `newRewardRate` is 0.

When `incentivizeRecurPool` is called, users can provide parameters, including the `incentiveAmount` that they want to provide. However, the operation does not check if the calculated `newRewardRate` is non-zero. This can cause the caller to fail to increase the reward rate while still sending tokens to the contract. This issue may occur if the user provides dust `incentiveAmount` or if the incentive token has low decimal precision. Consider adding a check to verify that the calculated `newRewardRate` is not zero.

[L-09] Missing `nonReentrant` modifier

The `MasterBunni.incentivizeRecurPool` function does not include a `nonReentrant` modifier when all other functions with external calls have it.

```
function incentivizeRecurPool
      (RecurIncentiveParams[] calldata params, address incentiveToken)
        external
        returns (uint256 totalIncentiveAmount)
    {
```

[L-10] Lack of address(0) check for `recipient` address

The `MasterBunni.depositIncentive` function does not check that the `recipient` address is not zero. This can cause assets to lock in the contract. Consider adding the corresponding check or using the `msgSender` variable as a default value when the `recipient` address is zero.

```
function depositIncentive(
      RushIncentiveParams[]calldataparams,
      addressincentiveToken,
      addressrecipient
    )
        external
        nonReentrant
        returns (uint256 totalIncentiveAmount)
    {
        address msgSender = LibMulticaller.senderOrSigner();

        // record incentive in each pool
        for (uint256 i; i < params.length; i++) {
            if (!isValidRushPoolKey
              (params[i].key) || block.timestamp >= params[i].key.startTimestamp) {
                // key is invalid or program is already active, skip
                continue;
            }

            // sum up incentive amount
            totalIncentiveAmount += params[i].incentiveAmount;

            RushPoolId id = params[i].key.toId();

            // add incentive to pool

                          rushPoolIncentiveAmounts[id][incentiveToken] += params[i].inc

            // add incentive to depositor
>>
            rushPoolIncentiveDeposits[id][incentiveToken][recipient] += params[i].ince
        }
<...>

    function withdrawIncentive(
      RushIncentiveParams[]calldataparams,
      addressincentiveToken,
      addressrecipient
    )
        external
        nonReentrant
        returns (uint256 totalWithdrawnAmount)
    {
        address msgSender = LibMulticaller.senderOrSigner();

        // subtract incentive tokens from each pool
        for (uint256 i; i < params.length; i++) {
            if (!isValidRushPoolKey
              (params[i].key) || block.timestamp >= params[i].key.startTimestamp) {
                // key is invalid or program is already active, skip
                continue;
            }

            // sum up withdrawn amount
            totalWithdrawnAmount += params[i].incentiveAmount;

            RushPoolId id = params[i].key.toId();

            // subtract incentive from pool

                          rushPoolIncentiveAmounts[id][incentiveToken] -= params[i].inc

            // subtract incentive from sender
>>
            rushPoolIncentiveDeposits[id][incentiveToken][msgSender] -= params[i].ince
        }
```

[L-11] Lack of slippage protection

The `TokenMigrator.migrate` function swaps old tokens to new tokens using the `newTokenPerOldToken` variable as a rate. Users risk receiving fewer new tokens than they expect in case the contract owner changes the `newTokenPerOldToken` variable right before their invoke.

```
function migrate(
      uint256oldTokenAmount,
      addressrecipient
    ) external returns (uint256 newTokenAmount
<...>
    function setNewTokenPerOldToken
      (uint256 newTokenPerOldToken_) external onlyOwner {
        newTokenPerOldToken = newTokenPerOldToken_;
        emit SetNewTokenPerOldToken(newTokenPerOldToken_);
    }
```