Hybra Finance Audit Report

Decentralized public liquidity layer for Hyperliquid

Version 1.0

Prakash Yadav

October 7, 2025

Hybra Finance Audit Report

Prakash Yadav

October 7, 2025

Prepared by: Prakash Yadav Lead Auditors: - Prakash Yadav

Table of Contents

Protocol Summary

Disclaimer

Prakash Yadav makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of 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.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

The scope of the project involves a subset of the full contract list in this repository, and should be consulted to ensure wardens invest their time and effort solely in the in-scope contracts.

Contract
ve33/contracts/GaugeManager.sol
ve33/contracts/GaugeV2.sol
ve33/contracts/MinterUpgradeable.sol

Contract
ve33/contracts/VoterV3.sol
ve33/contracts/VotingEscrow.sol
ve33/contracts/GovernanceHYBR.sol
ve33/contracts/HYBR.sol
ve33/contracts/RewardHYBR.sol
ve33/contracts/swapper/HybrSwapper.sol
ve33/contracts/CLGauge/GaugeCL.sol
ve33/contracts/CLGauge/GaugeFactoryCL.sol
cl/contracts/core/CLFactory.sol
cl/contracts/core/CLPool.sol
cl/contracts/core/fees/DynamicSwapFeeModule.sol

For a machine-readable version, kindly consult the `scope.txt` file in the repository

Roles

Executive Summary

Issues found

Category	No. of Issues
High	1
Medium	8
Low	1
Total	10

Findings

High

[H-1]: `Multicall.multicall` delegatecall inside loop in payable function.

Description: Delegatecall executes callee code in the context (storage, msg.sender) of the caller. Allowing user-provided calldata to be delegate-called into the same contract effectively permits arbitrary internal/external function invocation with the contract's storage and privilege context. Doing this inside a loop and marking the function payable increases complexity and enables ordering, gas exhaustion, and reentrancy attacks.

```
1
2  function multicall(bytes[] calldata data) public payable override
       returns (bytes[] memory results) {
3      results = new bytes[](data.length);
4  @>  for (uint256 i = 0; i < data.length; i++) {
5  @>      (bool success, bytes memory result) = address(this).
       delegatecall(data[i]);
6
7          if (!success) {
8              // Next 5 lines from https://ethereum.stackexchange.com/a
                   /83577
9  @>          if (result.length < 68) revert();
10 @>            assembly {
11 @>                result := add(result, 0x04)
12             }
13 @>           revert(abi.decode(result, (string)));
14         }
15
16         results[i] = result;
17     }
18 }
```

Proof of Concepts:

Demonstrate reentrancy: create a contract with a vulnerable function that updates a balance and uses an external call; use multicall to first call a function that triggers an external transfer and then call another function that depends on state changed by the first call — show the invariant break. Minimal PoC files: Vulnerable contract exposing state change functions (that Multicall can call) An attacker contract that calls multicall with crafted data to exploit ordering / reentrancy.

```
1
2  contract VulnerableBank is Multicall {
3      mapping(address => uint256) public balances;
4
```

```
    function deposit() external payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw(uint256 amount) external {
        require(balances[msg.sender] >= amount, "Insuff");
        // vulnerable: external call before balance update or non-reentrant guard
        (bool ok, ) = msg.sender.call{value: amount}("");
        require(ok, "send failed");
        balances[msg.sender] -= amount;
    }

    // wrapper so Multicall can call withdraw on behalf of the attacker, etc.
}
```

Create `Attacker.sol`. Add following code in it.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.7.6;

contract Attacker {
    VulnerableBank bank;
    address owner;

    constructor(address _bank) {
        bank = VulnerableBank(_bank);
        owner = msg.sender;
    }

    receive() external payable {
        // during withdrawal re-enter withdraw again if a condition is met
        if (address(bank).balance >= 1 ether) {
            // craft multicall calldata to call withdraw again
            bytes[] memory calls = new bytes[](1);
            calls[0] = abi.encodeWithSelector(bank.withdraw.selector, 1 ether);
            bank.multicall{value:0}(calls);
        }
    }

    function attack() external payable {
        bank.deposit{value: msg.value}();
        // craft multicall to call withdraw (or sequence) and cause reentrancy
        bytes[] memory calls = new bytes[](1);
        calls[0] = abi.encodeWithSelector(bank.withdraw.selector, 1 ether);
        bank.multicall(calls);
```

```
29      }
30  }
```

Recommended Mitigation:

1. Make multicall non-payable unless absolutely required. Remove payable keyword if not needed.
2. Add nonReentrant modifier from ReentrancyGuard:
3. function multicall(…) public payable nonReentrant returns (…) 4.Restrict allowed selectors for multicall using a whitelist mapping:

```
+      mapping(bytes4 => bool) public multicallWhitelist;
+
+      function addToWhitelist(bytes4 sel) external onlyOwner {
+         multicallWhitelist[sel] = true;
+       }
```

Note: retrieving selector from calldata in solidity requires slicing; using inline assembly or reading first 4 bytes of data[i] is needed. 5. Prefer external calls (this.call) instead of delegatecall if calling public/external functions (they run in their own context). Delegatecall should only be used when intentionally running library code in your contract's storage context. 6. Limit the number of calls or total calldata size to avoid gas exhaustion attacks. 6. Add thorough unit tests that model malicious sequences.

Example safe implementation (quick pattern):

```
function multicall(bytes[] calldata data) public payable nonReentrant
    returns (bytes[] memory results) {
-      results = new bytes[](data.length);
-     for (uint256 i = 0; i < data.length; i++) {
-         (bool success, bytes memory result) = address(this).
    delegatecall(data[i]);

-         if (!success) {
-             // Next 5 lines from https://ethereum.stackexchange.com/a
    /83577
-             if (result.length < 68) revert();
-             assembly {
-                 result := add(result, 0x04)
-             }
-             revert(abi.decode(result, (string)));
-         }

-         results[i] = result;
-     }
+        bytes4 selector;
+             selector := calldataload(data.offset)
+          }
+          require(multicallWhitelist[selector], "NOT_ALLOWED");
```

```
+         // call external function (not delegatecall)
+         (bool success, bytes memory result) = address(this).call(data[i]);
+         require(success, "CALL_FAILED");
+         results[i] = result;
+     }
+ }
```

Note: switching to call changes semantics; ensure intended functions are external and behave correctly.

Medium

[M-1]: Arbitrary from in transferFrom in `PeripheryPayments.pay`.

Description:

The pay function in `PeripheryPayments.sol` uses `TransferHelper.safeTransferFrom(token, payer, recipient, value)` where payer is a user-provided parameter. This allows the caller to specify any payer address, potentially enabling unauthorized token transfers if not proper.

```

function pay(address token, address payer, address recipient, uint256 value) internal {
    if (token == WETH9 && address(this).balance >= value) {
        // pay with WETH9
        IWETH9(WETH9).deposit{value: value}(); // wrap only what is needed to pay
        IWETH9(WETH9).transfer(recipient, value);
    } else if (payer == address(this)) {
        // pay with tokens already in the contract (for the exact input multihop case)
        TransferHelper.safeTransfer(token, recipient, value);
    } else {
        // pull payment
        TransferHelper.safeTransferFrom(token, payer, recipient, value);
    }
}
```

Impact: This could allow an attacker to transfer tokens from any address that has approved the contract, leading to loss of funds for users.

Proof of Concepts: An attacker could call a function that invokes pay with a payer address of a victim who has approved the contract, transferring their tokens to the attacker's recipient.Deploy a malicious

ERC20 that implements transferFrom(from, to, amount) as: ignore allowance and, if the call originates from this Periphery contract, transfer tokens from some victim-held balance (or simply return true regardless). Then call the higher-level function that ultimately calls pay(token, victimAddress, attackerRecipient, amount) with payer==victimAddress. The malicious token's transferFrom will succeed and move victim tokens to attackerRecipient. Alternatively, a token that always returns true from transferFrom but does no transfer will allow the flow to proceed while giving a false sense of success.

```
contract PoCExploit is PeripheryPayments {
    constructor(address _factory, address _WETH9)
        PeripheryImmutableState(_factory, _WETH9) {}

    // Hypothetical public version of pay to demonstrate the
        vulnerability
    function publicPay(address token, address payer, address recipient,
         uint256 value) external {
        pay(token, payer, recipient, value);
    }

    // Attacker calls this to steal from victim
    function exploit(address token, address victim, uint256 value)
        external {
        // Assumes victim has approved this contract for 'token'
        // Attacker sets recipient to themselves or another address
        publicPay(token, victim, msg.sender, value);
    }
}
```

Recommended Mitigation: Ensure that payer is validated to be the msg.sender or a trusted address. Alternatively, use a different mechanism for payments that doesn't rely on arbitrary payer inputs.

[M-2]: PeripheryPayments.pay: ignoring return value of IWETH9.transfer(...).

Description: The code ignores the boolean return of transfer. Some token implementations may return false on failure rather than reverting. Ignoring the return allows the function to continue as if transfer succeeded when it did not.

If transfer silently fails (returns false) funds may remain in the contract while subsequent logic assumes recipient was paid. This leads to incorrect accounting and potential stuck funds.

```
function pay(address token, address payer, address recipient, uint256
    value) internal {
        if (token == WETH9 && address(this).balance >= value) {
            // pay with WETH9
            IWETH9(WETH9).deposit{value: value}(); // wrap only what is
                needed to pay
@>          IWETH9(WETH9).transfer(recipient, value);
```

```
        } else if (payer == address(this)) {
            // pay with tokens already in the contract (for the exact
                input multihop case)
            TransferHelper.safeTransfer(token, recipient, value);
        } else {
            // pull payment
            TransferHelper.safeTransferFrom(token, payer, recipient,
                value);
        }
    }
```

Proof of Concepts: 1. MaliciousAlwaysTrueERC20 (or a WETH-like contract that returns false) demonstrates how transfer return values can be deceptive. Minimal test: Replace WETH9 with a WETH-like contract that returns false for transfer or returns true with no transfer; show that recipient doesn't receive expected tokens.

Recommended Mitigation:

1. Use require(IWETH9(WETH9).transfer(recipient, value), "WETH_TRANSFER_FAILED"); Or call the safe wrapper: TransferHelper.safeTransfer(WETH9, recipient, value); which handles different token semantics.
2. Alternatively unwrap WETH to ETH and send via TransferHelper.safeTransferETH(recipient, value); if the flow permits (the function already uses deposit — but better to call withdraw then safeTransferETH if the contract is holding WETH).

```
-   IWETH9(WETH9).transfer(recipient, value);
+   TransferHelper.safeTransfer(WETH9, recipient, value);
// or
+   require(IWETH9(WETH9).transfer(recipient, value), "
    WETH_TRANSFER_FAILED");
```

[M-3]: `abi.encodePacked` with multiple dynamic args (NFTDescriptor / NFTSVG).

Description: `abi.encodePacked` concatenates the inputs in packed form without length prefixes for dynamic types. Different sequences of dynamic inputs can produce identical packed bytes, leading to hash collisions or ambiguous concatenation results.

```
return string(
@>    abi.encodePacked('\"name\":\"', name, '\", \"description\":\"',
    descriptionPartOne, descriptionPartTwo, '\"')
);
```

And in NFTSVG.sol — generateTopText uses:

```

```

```
string memory poolId = string(abi.encodePacked("CL", tickToString(
    tickSpacing), "-", quoteTokenSymbol, "/", baseTokenSymbol));
string memory tokenIdStr = string(abi.encodePacked("ID #", tokenId.
    toString()));
string memory id = string(abi.encodePacked(poolId, tokenIdStr));
```

Impact: For display-only metadata the immediate risk is limited, but if packed bytes are used as hash inputs, identifiers, or stored in a context where uniqueness is required, collisions can lead to spoofed metadata or identifier collisions. Exploitability:If an attacker can craft token symbols or inputs leading to collisions. For example, inputs (“ab”,“c”) and (“a”,“bc”) collide under abi.encodePacked.

Proof-of-Concept: Show two different inputs that produce identical abi.encodePacked outputs:

```
bytes memory a = abi.encodePacked("ab", "c"); // "abc"
bytes memory b = abi.encodePacked("a", "bc"); // "abc"
assert(keccak256(a) == keccak256(b)); // identical
```

With poolId and tokenIdStr concatenation, crafted symbols could cause collisions in id.

Recommended Mitigation: Use `abi.encode` for unambiguous encoding (it includes length prefixes). Add explicit separators/delimiters between dynamic pieces that cannot appear in inputs, or escape inputs reliably. For JSON tokenURI building, prefer constructing JSON with explicit separators or base64-encode the entire JSON payload to avoid ambiguity. For SVG IDs, include a separator like | or : and ensure values are escaped. Notes: Practical severity depends on usage; still recommended to fix for correctness and to avoid future edge-case issues. Use abi.encode instead of abi.encodePacked when you need unambiguous encoding (it adds length prefixing). Or inject a unique delimiter/separator that cannot appear in inputs (for token symbols, ensure safe escaping). For example: use a non-printable separator unlikely to appear in token symbols or escape the inputs. For tokenURI JSON building: use abi.encodePacked only when you ensure input can’t cause ambiguity, otherwise build JSON with clear separators or using abi.encode then base64 encode the whole JSON (common pattern). For ids in SVG: use string(abi.encodePacked(poolId, “|”, tokenIdStr)) and ensure | is escaped if necessary. NFTDescriptor.constructTokenURI:

```
return string(
    abi.encodePacked(
        '"name":"',
        name,
        '", "description":"',
        descriptionPartOne,
        '|||',
        descriptionPartTwo,
        '"'
    )
);
```

Alternatively,build the JSON with abi.encode and then string():

```
- return string(
-   abi.encodePacked('\"name\":\"', name, '\", \"description\":\"', descriptionPartOne,     descriptionPartTwo, '\"')
- );
+ bytes memory json = abi.encode(
+   '{"name":"', name, '","description":"', descriptionPartOne, descriptionPartTwo, '"}'
+ );
+ return string(json);
```

For NFTSVG id:

```
+ string memory id = string(abi.encodePacked(poolId, "|", tokenIdStr)); // ensure `|` cannot appear
```

Alternatively,

```
+ string memory id = string(abi.encode(poolId, tokenIdStr)); // deterministic: include length prefixes (not readable directly)
```

Note: abi.encode returns bytes with length prefixes; converting to string may need base64 or careful processing if producing human-readable strings.

[M-4]: Reward Truncation in Cross-Epoch Claiming

Description: The _claim function limits reward processing to a maximum of 50 weeks per claim call, causing permanent loss of rewards for users who do not claim for extended periods exceeding 50 weeks.

The loop in _claim is capped at 50 iterations, each processing one week's rewards, without a mechanism to handle claims spanning more than 50 weeks.

```

for (uint i = 0; i < 50; i++) {
    if (week_cursor >= _last_token_time) break;
    // ... reward calculation
    week_cursor += WEEK;
}
```

Impact: Users with long-term locks (e.g., permanent locks) who forget to claim for over a year may lose all rewards beyond the 50-week window, leading to significant financial losses and reduced protocol participation incentives.

Proof of Concepts:

A user locks tokens for 2 years without claiming. After 52 weeks, their claim call processes only the first

50 weeks, permanently forfeiting the remaining 2 weeks' rewards. Subsequent claims cannot recover the lost rewards due to the fixed loop limit.

```
// PoC: Simulate a user claiming after 52 weeks
function testRewardTruncation() public {
    // Assume user locks tokens and doesn't claim for 52 weeks
    uint userTokenId = 1;
    // Fast forward time by 52 weeks
    vm.warp(block.timestamp + 52 * 1 weeks);

    // User calls claim
    uint claimedAmount = rewardsDistributor.claim(userTokenId);

    // Only 50 weeks processed, rewards for weeks 51-52 are lost
    // Expected: All 52 weeks' rewards claimed
    // Actual: Only 50 weeks' rewards, loss of ~4% of total rewards
    assert(claimedAmount < expectedTotalRewardsFor52Weeks);
}
```

Recommended Mitigation: Implement pagination by allowing multiple claim calls or modifying the loop to process all pending weeks, potentially with gas limits. Update the function to track and resume from the last processed week.

```
function _claim(uint _tokenId, address ve, uint _last_token_time)
   internal returns (uint) {
+    uint to_distribute = 0;
    // ... existing code ...
+    uint max_iterations = 50; // or make configurable
-   for (uint i = 0; i < 50; i++) {
+   for (uint i = 0; i < max_iterations; i++) {
    if (week_cursor >= _last_token_time) break;
    // ... reward calculation
    week_cursor += WEEK;
}
    // Store week_cursor for pagination
+    time_cursor_of[_tokenId] = week_cursor;
    // ... rest of function ...
}
```

[M-5]:Potential Reward Loss in Rollover Mechanism

Description: When stakedLiquidity is zero, rewards are accumulated in the rollover variable but are never redistributed to future stakers, effectively burning those rewards.

Code Issue: In _updateRewardsGrowthGlobal, rewards are added to rollover when no staked liquidity exists, but there's no subsequent redistribution logic.

```
if (stakedLiquidity > 0) {
    rewardGrowthGlobalX128 += FullMath.mulDiv(reward, FixedPoint128.
        Q128, stakedLiquidity);
} else {
@>   rollover += reward;  // These rewards are never redistributed
}
```

Impact: During periods of zero staked liquidity, earned rewards are permanently lost, reducing the total reward pool available to users and potentially discouraging participation during low-activity periods.

Proof of Concepts: Demonstrate reward loss during zero staked liquidity

```
function testRolloverLoss() public {
    // Set up pool with zero staked liquidity
    clPool.stake(0, tickLower, tickUpper, false); // stakedLiquidity =
        0

    // Simulate reward update
    uint initialRollover = clPool.rollover();
    clPool.updateRewardsGrowthGlobal();

    // Rewards accumulate in rollover
    uint finalRollover = clPool.rollover();
    assert(finalRollover > initialRollover);

    // Later, when liquidity is staked
    clPool.stake(stakedLiquidityDelta, tickLower, tickUpper, false);
    clPool.updateRewardsGrowthGlobal();

    // Rollover rewards are not redistributed, effectively lost
    assert(clPool.rollover() == finalRollover); // Still accumulated,
        not distributed
}
```

Recommended Mitigation:

Redistribute rollover rewards when staked liquidity becomes available, or integrate rollover into future reward calculations.

```
function _updateRewardsGrowthGlobal() internal {
    // ... existing code ...
    if (stakedLiquidity > 0) {
        uint256 totalReward = reward + rollover;
        rewardGrowthGlobalX128 += FullMath.mulDiv(totalReward,
            FixedPoint128.Q128, stakedLiquidity);+
+        rollover = 0; // Reset rollover after redistribution
    } else {
        rollover += reward;
    }
```

```
10      // ... rest ...
11  }
```

[M-6]:Potential Manipulation via Tick Price Movements.

Description: Users can manipulate price movements during swaps to cross ticks at advantageous times, affecting reward accrual timing for specific positions and creating unfair advantages.

Code Issue: The swap function's tick crossing logic allows price manipulation to influence when rewards are accrued, as tick transitions update reward growth.

```
1  // In swap function, during tick crossing:
2  @> if (step.initialized) {
3      // ... tick cross logic ...
4   @>   Tick.LiquidityNets memory nets = ticks.cross(
5          step.tickNext,
6          // ... fee growth and reward growth updates ...
7          rewardGrowthGlobalX128
8      );
9      // ... liquidity updates ...
10 }
```

Impact: Sophisticated users may time transactions to maximize rewards or minimize penalties, creating an uneven playing field and potentially reducing overall protocol fairness.

Recommended Mitigation:

```
1  // Add a check in swap to limit manipulation
2  function swap(...) external override returns (int256 amount0, int256
       amount1) {
3      // ... existing code ...
4      // Before tick crossing, check for manipulation attempts
5  -    if (step.initialized) {
6  +    if (step.initialized && amountSpecified > largeSwapThreshold) {
7  -        Tick.LiquidityNets memory nets = ticks.cross(
8  -        step.tickNext,
9  -        // ... fee growth and reward growth updates ...
10 -        rewardGrowthGlobalX128
11 -    );
12           // Delay or limit the swap impact
13 +        require(block.timestamp > lastLargeSwap + manipulationCooldown
       , "Potential manipulation detected");
14 +        lastLargeSwap = block.timestamp;
15     }
16     // ... rest of function ...
17 }
```

[M-7]:Inconsistent Protocol Fee Handling

Description: In calculateFees, when stakedLiquidity is zero, protocol fees are deducted from unstaked fees, but the logic doesn't consistently account for protocol fees across all scenarios, potentially understating treasury revenue.

Code Issue: The function handles protocol fees only in the unstaked liquidity case, but doesn't ensure consistent application when both staked and unstaked liquidity exist.

```
function calculateFees(uint256 feeAmount, uint128 _liquidity, uint128
    _stakedLiquidity)
    internal
    view
    returns (uint256 feeGrowthGlobalX128, uint256 stakedFeeAmount,
       uint256 protocolFeeAmount)
{
    // ... existing code ...
    else if (_stakedLiquidity == 0) {
        (uint256 unstakedFeeAmount, uint256 _stakedFeeAmount, uint256
           _protocolFeeAmount) = applyUnstakedFees(feeAmount, 0 );
        feeGrowthGlobalX128 = FullMath.mulDiv(unstakedFeeAmount,
           FixedPoint128.Q128, _liquidity);
        stakedFeeAmount = _stakedFeeAmount;
        protocolFeeAmount = _protocolFeeAmount;
    }
    // ... rest ...
}
```

Impact: Protocol revenue may be incorrectly calculated or omitted in mixed liquidity scenarios, leading to understated treasury earnings and potential disputes over fee distribution.

Proof of Concepts: Show inconsistent protocol fee handling

```
function testProtocolFeeInconsistency() public {
    uint feeAmount = 1000;
    uint128 liquidity = 100;
    uint128 stakedLiquidity = 50; // Mixed scenario

    // In mixed liquidity, protocol fees might not be properly deducted
        before splitting
    (uint256 feeGrowth, uint256 stakedFee, uint256 protocolFee) =
       clPool.calculateFees(feeAmount, liquidity, stakedLiquidity);

    // Expected: Protocol fee deducted from total before distribution
    // Actual: May vary depending on implementation, potentially under-
       collecting
    uint expectedProtocolFee = (feeAmount * protocolFeeRate) / 1e6;
    assert(protocolFee != expectedProtocolFee); // May fail if
       inconsistent
}
```

Recommended Mitigation:

Ensure protocol fees are consistently applied across all fee distribution scenarios, deducting them before splitting between staked and unstaked liquidity.

```
function calculateFees(uint256 feeAmount, uint128 _liquidity, uint128
    _stakedLiquidity)
     internal
     view
     returns (uint256 feeGrowthGlobalX128, uint256 stakedFeeAmount,
        uint256 protocolFeeAmount)
{
-      else if (_stakedLiquidity == 0) {
-         (uint256 unstakedFeeAmount, uint256 _stakedFeeAmount, uint256
    _protocolFeeAmount) = applyUnstakedFees(feeAmount, 0 );
-         feeGrowthGlobalX128 = FullMath.mulDiv(unstakedFeeAmount,
    FixedPoint128.Q128, _liquidity);
-         stakedFeeAmount = _stakedFeeAmount;
-         protocolFeeAmount = _protocolFeeAmount;
-     }

+     uint24 _protocolFeeConfig = protocolFee();
+     if (_protocolFeeConfig > 0) {
+         protocolFeeAmount = FullMath.mulDivRoundingUp(feeAmount,
    _protocolFeeConfig, 1_000_000);
+         feeAmount -= protocolFeeAmount;
+     }
     // Then proceed with existing logic using adjusted feeAmount
     // ... rest of function ...
}
```

[M-8]:Reward Theft via NFT Transfer After Lock Expiry

Description: Users can exploit NFT transfers after lock expiry to steal accumulated rewards. When a lock expires, the NFT can be transferred to a new owner who can then claim all the rewards that were earned during the original owner's lock period.

Code Issue: The claim function transfers rewards to the current ownerOf(_tokenId) at claim time, but the reward calculation in _claim is based on the NFT's balance history during past weeks. This allows the new owner to claim rewards earned by the previous owner.

```
function claim(uint256 _tokenId) external returns (uint256) {
    // ...
@>    if (_locked.end < block.timestamp && !_locked.isPermanent) {
@>        address _nftOwner = IVotingEscrow(voting_escrow).ownerOf(
    _tokenId);
```

```
@>         IERC20(token).safeTransfer(_nftOwner, amount);  // Transfers
    to current owner
@>     } else {
@>         IVotingEscrow(voting_escrow).deposit_for(_tokenId, amount);
@>     }
    // ...
}
```

Impact: Original NFT owners lose their earned rewards if they transfer the NFT after expiry without claiming first. Malicious users can acquire expired NFTs specifically to claim their accumulated rewards, leading to unfair reward distribution and potential loss of trust in the system.

Proof of Concepts: Demonstrate reward theft through NFT transfer.

```
function testRewardTheft() public {
    // User A locks tokens and earns rewards over time
    uint tokenId = 1;
    // Assume rewards accumulate for 1 year

    // Lock expires
    vm.warp(lockEndTime + 1);

    // User A transfers NFT to User B without claiming
    votingEscrow.safeTransferFrom(userA, userB, tokenId);

    // User B claims rewards
    vm.prank(userB);
    uint claimedAmount = rewardsDistributor.claim(tokenId);

    // User B receives all rewards earned by User A
    assert(claimedAmount > 0);
    // User A gets nothing
}
```

Recommended Mitigation:

Implement a claim restriction where only the address that locked the NFT (or approved addresses) can claim rewards, or require claims before transfers for expired locks. Alternatively, modify the logic to prorate rewards based on ownership periods.

```
function claim(uint256 _tokenId) external returns (uint256) {
+    address locker = IVotingEscrow(voting_escrow).ownerOf(_tokenId);
+    require(msg.sender == locker || /* approved claimer */, "Not
    authorized to claim");

    // ... rest of function ...
}
```

Low

[L-1]: FullMath — ^ (bitwise XOR) flagged as exponentiation misuse (false-positive).

Description:

The expression uint256 inv = (3 * denominator) ^ 2; as a likely accidental use of ^ instead of **. However, this line is from the canonical FullMath modular inverse seed used in Uniswap / Remco Bloemen's implementation. ^ here is bitwise XOR and is intentional. Still, it confuses reviewers and static analyzers. Add an explanatory comment and unit tests for mulDiv edge-cases to make intent explicit and ensure future maintainers don't change it.

```
1 @> uint256 inv = (3 * denominator) ^ 2;
```

The expression (3 * denominator) ^ 2 is from the known algorithm for generating an initial seed for modular inverse calculation via Newton-Raphson iterations in modular arithmetic. It uses bitwise operations to produce a starting inverse with correct low bits; subsequent Newton-Raphson steps double the number of correct bits.

Recommendation

Use the correct operator ** for exponentiation.

Read this for more:https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation

Informational

Gas