SHERLOCK

Security Review For BMX DeFi

Public Best Efforts Audit Contest Prepared For: **BMX DeFi**
Lead Security Expert: **0x73696d616f**
Date Audited: **September 2 - September 16, 2025**

Introduction

BMX is a DeFi ecosystem designed to address a core challenge for long-term users and liquidity providers: the reliance on speculative trading cycles and temporary, unsustainable incentive programs. Deli Swap is a DEX built using Uniswap v4 hooks that exemplifies the ecosystem's unique design, allowing liquidity providers to earn from two sources simultaneously: swap fees and the underlying yield generated by their wBLT in pools. This contest is for the Deli Swap contracts.

Scope

Repository: morphex-labs/deli-swap-contracts

Audited Commit: 85e2a1db2127e6c101989055bc4e3894f8aa57a0

Final Commit: f20f4ce1f3c97da805174e430a483cbcb09f8a50

Files:

- src/base/MultiPoolCustomCurve.sol
- src/DailyEpochGauge.sol
- src/DeliHookConstantProduct.sol
- src/DeliHook.sol
- src/FeeProcessor.sol
- src/handlers/V2PositionHandler.sol
- src/handlers/V4PositionHandler.sol
- src/IncentiveGauge.sol
- src/interfaces/IDailyEpochGauge.sol
- src/interfaces/IFeeProcessor.sol
- src/interfaces/IIncentiveGauge.sol
- src/interfaces/IPoolKeys.sol
- src/interfaces/IPositionHandler.sol
- src/interfaces/IPositionManagerAdapter.sol
- src/interfaces/IRewardDistributor.sol
- src/interfaces/IV2PositionHandler.sol
- src/libraries/DeliErrors.sol
- src/libraries/InternalSwapFlag.sol
- src/libraries/Math.sol

- src/libraries/RangePool.sol
- src/libraries/RangePosition.sol
- src/libraries/TimeLibrary.sol
- src/PositionManagerAdapter.sol
- src/Voter.sol

Final Commit Hash

f20f4ce1f3c97da805174e430a483cbcb09f8a50

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.

Issues Found

High	Medium
8	6

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0x73696d616f
0xDyDx
0xpetern
Bobai23
PratRed
ami
axelot
blockace
cergyk
cpsec
ezsia
globalace
jayjoshix
maigadoh
makeWeb3safe

oct0pwn
r1ver

secret__one
silver_eth

x15

Issue H-1: Gas consumed in `notifyUnsubscribe` is underestimated during tests and is greater than 300,000 without pre-warming

Source: https://github.com/sherlock-audit/2025-09-bmx-deli-swap-judging/issues/27

Found by

cergyk

Description

To protect users against bricked positions by malicious subscriber contracts, `Uniswap V4` notifier allows recovery of a reverting `notifyUnsubscribe` call:

Notifier.sol#L80-L86:

```
if (address(_subscriber).code.length > 0) {
    // require that the remaining gas is sufficient to notify the subscriber
    // otherwise, users can select a gas limit where .notifyUnsubscribe hits
    ↪  OutOfGas yet the
    // transaction/unsubscription can still succeed
    if (gasleft() < unsubscribeGasLimit) GasLimitTooLow.selector.revertWith();
    //@audit try/catch to protect against malicious _subscriber
    try _subscriber.notifyUnsubscribe{gas: unsubscribeGasLimit}(tokenId) {} catch {}
}
```

Conversely, in order to protect `subscriber` against forced failure due to insufficient user gas limit, outer call reverts if less than `unsubscribeGasLimit` is available, and thus guarantees at least `(unsubscribeGasLimit*63)/64` is provided to `notifyUnsubscribe` call (the `63/64` factor is due to EIP 150).

`unsubscribeGasLimit` value on Base is set to `300,000`.

This is why `PositionManagerAdapter.notifyUnsubscribe` must keep gas cost under `(300,000*63)/64 ~= 295000`, to avoid users unsubscribing successfully on `Uniswap V4` while leaving positions with liquidity in Deli gauges accounting.

Unfortunately, this is not the case, and the call to `notifyUnsubscribe` can consume more than `300,000` gas, as shown in the POC below.

Impact

Users can force unsubscribe from `Uniswap V4`, but keep positions accounting intact in Deli gauges. Then the malicious user can either burn or resubscribe his position to `Positi`

`onManagerAdapter` immediately, diluting the rewards for that range by position `liquidity` forever (even if the admin calls `adminForceUnsubscribe`, `liquidity` remains accounted for in global gauge accounting).

DailyEpochGauge.sol#L610-L618:

```
poolRewards[pid].modifyPositionLiquidity(
    RangePool.ModifyLiquidityParams({
        tickLower: tickLower,
        tickUpper: tickUpper,
        //@audit when subscribing, current position liquidity is always added to
        ↪  global liquidity tracking
        liquidityDelta: SafeCast.toInt128(uint256(liquidity)),
        tickSpacing: poolTickSpacing[pid]
    }),
    toks
);
```

POC

Here we show that the already existing test case `testGasAdapterNotifyUnsubTwoWorst` can be adapted to show that `notifyUnsubscribe` fails due to OOG.

To do so, we isolate the "set up" of the test case in a separate transaction by using the special function `beforeTestSetup` (see foundry documentation).

Please run:

1. `git apply POC.patch`
2. `forge test --match-test testGasAdapterNotifyUnsubTwoWorst -vvvv --decode-internal`

```
diff --git a/deli-swap-contracts/test/integration/PositionLifecycleCleanup.t.sol
↪  b/deli-swap-contracts/test/integration/PositionLifecycleCleanup.t.sol
index 07499d3..56ab07e 100644
--- a/deli-swap-contracts/test/integration/PositionLifecycleCleanup.t.sol
+++ b/deli-swap-contracts/test/integration/PositionLifecycleCleanup.t.sol
@@ -594,31 +594,36 @@ contract PositionLifecycleCleanup_IT is Test, Deployers {
         _gasAdapterNotifyUnsub(3, "adapter_notify_unsub_3");
     }

-    /// Worst-case parameters for unsubscribe gas: two extra incentive tokens
↪  (plus base wBLT from _activateStream),
-    /// large elapsed time without prior syncs to force Daily's multi-day
↪  integration and maximize cold reads.
-    /// to force Daily's multi-day integration during unsubscribe and maximize
↪  cold reads.
-    function testGasAdapterNotifyUnsubTwoWorst() public {
+    function worstCaseBeforeSetup() public {
```

```
         // 1) Mint and subscribe two positions
-        uint256 tokenId = _mintAndSubscribe(-1800, 1800, 1e22);
+        uint256 tokenId = _mintAndSubscribe(-2400, 2400, 1e22);
+        //@audit-ok enforce first token minted has id == 2 for consistency with
 ↪  test case
+        assert(tokenId == 2);
         uint256 tokenId2 = _mintAndSubscribe(-1800, 1800, 1e22);

         // 2) Activate Daily stream and base incentive, then add two extra
          ↪  incentive tokens (3 total incentives)
         _activateStream();
-        _addIncentiveTokens(10);
+        //@audit-ok no need to add additional incentive tokens

-        // 3) Warp many days ahead to force DailyEpochGauge._amountOverWindow to
 ↪  iterate across many day boundaries
-        //    and ensure significant elapsed time for incentive calculations,
 ↪  without additional pokes.
-        vm.warp(block.timestamp + 4 days);
+        //@audit-ok no need to even warp
+    }

-        // 4) Measure only the adapter.notifyUnsubscribe path
-        vm.startPrank(address(positionManager));
-        vm.startSnapshotGas("adapter_notify_unsub_2_worst_first");
-        adapter.notifyUnsubscribe(tokenId);
-        vm.stopSnapshotGas();
-        vm.startSnapshotGas("adapter_notify_unsub_2_worst_second");
-        adapter.notifyUnsubscribe(tokenId2);
-        vm.stopSnapshotGas();
-        vm.stopPrank();
+    //@audit-ok special foundry function, isolates calls before test in a separate
 ↪  transaction
+    function beforeTestSetup(bytes4 testSelector) public returns (bytes[] memory
 ↪  beforeTestCalldata) {
+        if (testSelector == this.testGasAdapterNotifyUnsubTwoWorst.selector) {
+            beforeTestCalldata = new bytes[](1);
+            //@audit-ok make the setup for the worst case in an isolated
 ↪  transaction
+            //@audit-ok we do this to avoid warming up all of the storage slots
 ↪  used later in notifySubscribe call
+            beforeTestCalldata[0] =
 ↪  abi.encodeWithSelector(this.worstCaseBeforeSetup.selector);
+        }
+    }
+
+    /// Worst-case parameters for unsubscribe gas: two extra incentive tokens
 ↪  (plus base wBLT from _activateStream),
+    /// large elapsed time without prior syncs to force Daily's multi-day
 ↪  integration and maximize cold reads.
```

```
+    /// to force Daily's multi-day integration during unsubscribe and maximize
 ↪  cold reads.
+    function testGasAdapterNotifyUnsubTwoWorst() public {
+        //@audit-ok use hardcoded token id 2 for simplicity, but this is enforced
 ↪  with assert during preparation
+        positionManager.unsubscribe(2); // @audit-ok run forge test -vvvv to see
 ↪  that notifyUnsubscribe has failed with OOG
     }

     /*////////////////////////////////////////////////////////////////
```

Recommendation

Further gas optimisations must be made to `notifyUnsubscribe` not to hit the limit. A radical change could be to only set a flag ”pendingUnsubscribe” on the given position in both the daily gauge and the incentive gauge, to make it ineligible for claiming at first. Then the heavy work of unsubscribing and cleaning accounting data can be done during a subsequent call outside of the critical path.

Issue H-2: Users' voting weight can be double-counted when finalize epoch is processed in multiple steps

Source: https://github.com/sherlock-audit/2025-09-bmx-deli-swap-judging/issues/62

Found by

blockace, cergyk

Description

In `Voter.sol`, the admin can call `finalizeEpoch` in order to tally all of the votes when an epoch has ended. Since the list of voters can be unbounded, finalizing can be broken into multiple steps in order to avoid DOS by OOG.

Unfortunately, when tallying the votes, the current balance of `SBF_BMX` is used, which allows a manipulation of the result by malicious actors:

Voter.sol#L338-L369:

```
    /// @dev Processes up to `maxBatch` auto-voters for `ep`, adding live balances
    ↪  to option weights.
    function _tallyAutoVotes(uint256 ep, uint256 maxBatch) internal returns (bool
    ↪  finished) {
        EpochData storage e = epochInfo[ep];
        uint256 processed;
        while (processed < maxBatch && batchCursor[ep] < autoVoterList.length) {
            uint256 i = batchCursor[ep];
            address voterAddr = autoVoterList[i];
            batchCursor[ep] = i + 1;
            processed++;

            uint8 opt = autoOption[voterAddr];
            if (opt >= 3) continue; // disabled
            if (userVoteWeight[ep][voterAddr] > 0) continue;


            //@audit current balance is used for voterAddr
>>          uint256 bal = SBF_BMX.balanceOf(voterAddr);
            if (bal == 0) {
                // queue for removal instead of removing now
                pendingRemovals[ep].push(voterAddr);
                continue;
            }
            e.optionWeight[opt] += bal;
            userVoteWeight[ep][voterAddr] = bal;
            userChoice[ep][voterAddr] = opt;
```

```
        }

        finished = (batchCursor[ep] >= autoVoterList.length);

        // process removals only after loop is complete
        if (finished && pendingRemovals[ep].length > 0) {
            _processPendingRemovals(ep);
        }
    }
```

Concrete scenario

The admin separates the processing of autoVotes into 2 transactions, and Alice controls 2 addresses `voterAddressA` (included in the first batch) and `voterAddressB` (included in the second batch).

Between the 2 admin transactions, Alice unstakes her `sbfBMX` from `voterAddressA` and stakes to `voterAddressB`.

Since the balance of `sbfBMX` at the time of admin calling is used, the `sbfBMX` from `voterAddressA` will be counted again when `voterAddressB` is processed during the second admin call.

Alice has successfully manipulated the result of the vote.

Recommendation

Either restrict staking of sBMX when `finalizationInProgress`, or use a snapshotting system which would get the latest `userVoteWeight[prevEp][voterAddr]` where `prevEp` is the latest epoch where `userVoteWeight` was updated.

Issue H-3: DoSed `Voter::finalize()` due to unbounded pending removals lacking a batch argument variable

Source: https://github.com/sherlock-audit/2025-09-bmx-deli-swap-judging/issues/68

Found by

0x73696d616f, Bobai23, ezsia, maigadoh

Summary

`Voter::finalize()` processes in batches to allow finalizing epochs and distributing rewards without running out of gas. However, `Voter::_processPendingRemovals()` doesn't take a batch variable into account, making it possible for an attacker to register as auto voter with a balance > 0, and then transfer their votes out, being sent to the pending removals array as soon as the epoch is finalized. By spamming this in a huge number of accounts, an attacker is able to DoS epoch finalization for all epochs, effectively making all rewards stuck.

Root Cause

In `Voter.sol:386`, Voter::_processPendingRemovals() is missing a batch variable argument.

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. Attacker calls `Voter::vote()` with a wei balance, enabling auto voting, and then transferring the balance out. Attacker does this for multipler addresses such that `Voter::finalize()` runs out of gas.
2. Admin calls `Voter::finalize()`, and in `Voter::_tallyAutoVotes()`, all the accounts are pushed to the pendingRemovals array. Due to the lack of a batch argument in the `Voter::_processPendingRemovals()` function, it will run out of gas trying to remove all the addresses of the attacker.

Impact

Epoch finalization is impossible and rewards are forever stuck.

PoC

Note `Voter.sol::_processPendingRemovals()`:

```
function _processPendingRemovals(uint256 ep) internal {
    address[] memory toRemove = pendingRemovals[ep];
    for (uint256 i = 0; i < toRemove.length; i++) {
        address addr = toRemove[i];
        uint256 idx = autoIndex[addr];
        if (idx > 0) {
            _removeAutoVoter(addr, idx - 1);
        }
    }
    delete pendingRemovals[ep];
}
```

Mitigation

Send a batch argument, to avoid OOG revert.

Issue H-4: Finalize-window vote-changing vulnerability: auto-voters can alter choices post-epoch to manipulate results

Source: https://github.com/sherlock-audit/2025-09-bmx-deli-swap-judging/issues/103

Found by

0x73696d616f, Bobai23, PratRed, ami, axelot, cpsec, r1ver

Summary

Missing freeze or snapshot of auto-vote choices at epoch end allows attackers to manipulate the voting process. Since auto-voters can enable or change their option after the epoch has ended but before or between finalization batches, they can shift their votes retroactively, resulting in manipulated WETH allocation and distorted outcomes for the safety module and reward recipients.

Root Cause

The root cause of this vulnerability is that the contract records auto-vote information only when `finalize` is called, instead of at the actual end of the epoch. In `_tallyAutoVotes`, the code directly reads the current `autoOption[user]` and `SBF_BMX.balanceOf(user)` and applies them to the target epoch, even if the epoch has already ended:

https://github.com/sherlock-audit/2025-09-bmx-deli-swap/blob/main/deli-swap-contracts/src/Voter.sol#L348-L360

```
uint8 opt = autoOption[voterAddr];
if (opt >= 3) continue;
if (userVoteWeight[ep][voterAddr] > 0) continue;

uint256 bal = SBF_BMX.balanceOf(voterAddr);
if (bal == 0) {
    pendingRemovals[ep].push(voterAddr);
    continue;
}
e.optionWeight[opt] += bal;
userVoteWeight[ep][voterAddr] = bal;
userChoice[ep][voterAddr] = opt;
```

Because there is no check such as `autoEffectiveEpoch[user] <= ep` or validation against `epochEnd(ep)`, users can enable or modify auto-vote after the epoch has already finished but before it is finalized, and still affect the voting outcome of that past epoch. This issue

is aggravated by the batching mechanism in `finalize`, since auto-voters are processed gradually, leaving a window where their current state can still change the results.

Internal Pre-conditions

1. User needs to enable `autoVote` to set `autoOption[user]` to be less than 3 after epoch end but before finalize
2. User needs to hold `SBF_BMX.balanceOf(user)` to be at least 1 token during finalize batch processing
3. `epochInfo[ep].settled` needs to be false so the epoch is still unfinalized

External Pre-conditions

None

Attack Path

1. **User** calls `vote(optionX, true)` **after epoch N has ended but before** `finalize(N, …)` , enabling auto-vote (or switches to desired `optionX` if already enabled).
2. **User** transfers `sbfBMX` to self (ERC20 `transfer`) **to raise** `SBF_BMX.balanceOf(user)` prior to being tallied.
3. **Admin** calls `finalize(N, maxBatch)` in batches; `_tallyAutoVotes(N, …)` begins processing auto-voters but **has not yet reached the user**.
4. **User** can at any time before being processed call `vote(optionY, true)` **to change auto option** (or further adjust balance) based on observed voting trend. Steps 3 and 4 may occur in either order, since the key requirement is simply that the user modifies their vote **after the epoch ends but before they are tallied**.
5. **Admin** eventually calls `finalize(N, maxBatch)` again; when `_tallyAutoVotes` **reaches the user**, it reads the **current** `autoOption[user]` and **current** `SBF_BMX.balanceOf(user)`, and **credits weight to epoch N**.
6. **Admin** completes `finalize(N, …)`; **Finalize** settles epoch N using the **manipulated weights**, sending WETH per the now-influenced winning option.

Impact

The protocol governance process suffers a critical integrity loss, as auto-voters can still change their vote **after the epoch has ended but before or during finalization batches**, effectively rewriting the outcome of a closed vote. This enables an attacker holding significant `sbfBMX` balance to manipulate results retroactively, causing incorrect allocation of WETH between the safety module and reward distributor. The affected

parties are the protocol treasury and honest voters, whose intended distribution is overridden by the attacker's post-epoch vote changes.

PoC

I write a POC in `deli-swap-contracts/test/unit/VoterFinalizeWindow.t.sol`. execute the command `forge test --match-path test/unit/VoterFinalizeWindow.t.sol -vvv` to run this test.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

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

import {Voter} from "src/Voter.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IRewardDistributor} from "src/interfaces/IRewardDistributor.sol";
import {TimeLibrary} from "src/libraries/TimeLibrary.sol";

import {MockRewardDistributor} from "test/mocks/MockRewardDistributor.sol";
import {MintableERC20} from "test/mocks/MintableERC20.sol";

contract VoterFinalizeWindowTest is Test {
    MintableERC20 weth;
    MintableERC20 sbfBmx;
    MockRewardDistributor distributor;
    Voter voter;

    address admin = address(0xA11CE);
    address safety = address(0xBEEF);
    address attacker = address(0xA774C);
    address other;

    uint256 constant WEEK = 7 days;

    function setUp() public {
        // Deploy tokens and distributor
        weth = new MintableERC20();
        sbfBmx = new MintableERC20();
        distributor = new MockRewardDistributor();

        // Label for readability
        vm.label(address(weth), "WETH");
        vm.label(address(sbfBmx), "SBF_BMX");
        vm.label(address(distributor), "Distributor");
        vm.label(admin, "Admin");
        vm.label(safety, "Safety");
        vm.label(attacker, "Attacker");
```

```
        // Mint initial balances
        weth.mintExternal(admin, 1_000 ether);
        sbfBmx.mintExternal(attacker, 100 ether);

        // Epoch zero = current time aligned to now for simplicity
        uint256 epochZero = block.timestamp;

        // Deploy voter
        voter = new Voter(
            IERC20(address(weth)),
            IERC20(address(sbfBmx)),
            safety,
            IRewardDistributor(address(distributor)),
            epochZero,
            5_000, // opt0 -> 50% to safety
            3_000, // opt1 -> 30%
            2_000  // opt2 -> 20%
        );

        // Set admin
        vm.prank(voter.owner());
        voter.setAdmin(admin);

        // Approvals for deposit and early auto voter setup
        vm.startPrank(admin);
        weth.approve(address(voter), type(uint256).max);
        voter.deposit(200 ether); // deposit during epoch 0
        vm.stopPrank();

        // Prepare an early auto-voter so they occupy index 0 in autoVoterList
        other = address(0xB0B);
        sbfBmx.mintExternal(other, 100 ether);
        vm.prank(other);
        voter.vote(0, true); // enable auto for 'other' before epoch end
    }

    function _endEpoch(uint256 ep) internal {
        // move time to after end of ep
        uint256 endTs = voter.EPOCH_ZERO() + (ep + 1) * WEEK; // using same math as
        ↪  contract
        if (block.timestamp < endTs) {
            vm.warp(endTs + 1);
        }
    }

    function test_PoC_finalizeWindowAutoVoteManipulation() public {
        // Arrange: move to after epoch 0 ends but do not finalize
        uint256 ep = 0;
        _endEpoch(ep);
```

```
// Attacker enables auto-vote AFTER epoch end choosing option 1 initially
vm.prank(attacker);
voter.vote(1, true); // enable auto; allowed post-epoch

// Attacker boosts balance before being tallied
sbfBmx.mintExternal(attacker, 400 ether); // now balance = 500

// Start finalize with batch size 1
vm.prank(admin);
voter.finalize(ep, 1);

// If attacker already tallied in first batch, reset scenario by creating a
↪  fresh beneficiary is complex; instead we'll assert behavior in both
↪  cases.
// If attacker not tallied yet, userVoteWeight[ep][attacker] == 0; then we
↪  change option and expect that option to be counted when processed next.

(, uint256 weightBefore) = voter.getUserVote(ep, attacker);
bool attackerTallied = (weightBefore > 0);

if (!attackerTallied) {
    // Change auto option in the finalize window based on observed trend
    vm.prank(attacker);
    voter.vote(0, true); // switch from option 1 to 0

    // Also adjust balance again to prove live balance is used
    sbfBmx.mintExternal(attacker, 500 ether); // now balance larger

    // Next batch processes attacker and should read CURRENT option and
    ↪  balance
    vm.prank(admin);
    voter.finalize(ep, 10);
} else {
    // Attacker was processed already; still demonstrate live-balance
    ↪  mutability for a different auto voter
    vm.prank(other);
    voter.vote(2, true); // flip 'other' option in-window
    sbfBmx.mintExternal(other, 200 ether);
    vm.prank(admin);
    voter.finalize(ep, 10);
}

// Complete finalize
vm.prank(admin);
voter.finalize(ep, 1000);

// Assert: epoch is settled
(, , bool settledEp0) = voter.epochData(ep);
assertTrue(settledEp0, "epoch not settled");
```

```
        // Core assertion: for attacker, the recorded userChoice and userVoteWeight
        ↪  reflect the latest values at time of tally
        (uint8 recordedOpt, uint256 recordedWeight) = voter.getUserVote(ep,
        ↪  attacker);
        assertGt(recordedWeight, 0, "attacker must be tallied");

        // The expected option is 0 if attacker switched before being processed;
        ↪  otherwise it's 1.
        // We allow either, but we assert that it reflects the in-window choice
        ↪  actually present when tallied by checking storage directly.
        // In either case, verify that userChoice equals what was used in weights
        ↪  array impact (non-zero in one of options).
        // We cannot read private epochInfo weights for mapping to a user, so we
        ↪  assert option in {0,1,2} and not 3.
        assertTrue(recordedOpt < 3, "invalid recorded option");

        // And ensure the recorded weight equals final live balance at tally time
        ↪  by checking it's at least initial 100 ether and reflects mints (>100).
        assertGe(recordedWeight, 100 ether, "weight should use post-epoch balance");
    }
}
```

Mitigation

A proper mitigation is to ensure that auto-votes are only effective from the next epoch onward, not retroactively applied to an epoch that has already ended. This can be enforced by recording the epoch at which auto-vote becomes active when the user enables it, for example by storing `autoEffectiveEpoch[user] = currentEpoch() + 1`. Then in `_tallyAutoVotes`, the contract should only count a user's vote if `ep >= autoEffectiveEpoch[user]`. Alternatively, a timestamp check can be added so that `autoOption` changes made after `epochEnd(ep)` are ignored for that epoch. Another mitigation is to freeze all modifications to `autoOption` (both enabling and changing) whenever `_hasEndedUnfinalizedEpoch()` returns true, aligning the restriction already applied to disabling auto-vote. Any of these adjustments will prevent users from influencing past epochs with state changes made after the epoch has already ended.

Issue H-5: `RangePool::adjustToTick()` desyncs when `nextTick == tick` leading to stolen fees

Source: https://github.com/sherlock-audit/2025-09-bmx-deli-swap-judging/issues/178

Found by

0x73696d616f

Summary

`RangePool::adjustToTick()` breaks when the `nextTick` equals the current tick:

```
function adjustToTick(State storage self, int24 tickSpacing, int24 tick, address[]
↪  memory tokens) internal {
    int24 currentTick = self.tick;
    int128 liquidityChange = 0;
    bool lte = tick <= currentTick;

    if (lte) {
        while (tick < currentTick) {
            (int24 nextTick, bool initialized) =
                self.tickBitmap.nextInitializedTickWithinOneWord(currentTick,
                ↪  tickSpacing, true);

            if (nextTick <= tick) { //@audit here
                break;
            }
    ...
```

However, this will desync with the Uniswap pool whenever the sqrtPrice reaches the next tick.

For example, consider that a swap moved the price to tick 60, tick spacing is 60, and there is an initialized tick at 60. Looking at the Uniswap pool code, it will cross the tick, and positions from ticks 0 to 60 will now accrue fees, not 60 to 120.

But, this won't happen in the DailyEpochGauge nor IncentiveGauge. These pools will be poked after the swap, so the tick at 60 has been crossed in the Uniswap pool, but in the `RangePool::adjustToTick()` logic, it won't cross the tick, as the nextTick is 60, and so is the tick, which breaks and doesn't flip the tick nor update liquidity.

As a result, the following happens:

1. Positions with tick lower at 60 should be out of range (real Uniswap pool tick becomes 59 when the sqrtPrice reaches 60), but in the DailyEpochGauge and IncentiveGauge they will in reality be in range.

2. Thus, an attacker may exploit this to add liquidity to the Uniswap pool at ticks 60 to 120, out of range, where regular users and bots won't add liquidity because it is out of range. The attacker will get away with most of the fees.
3. An attacker can force this to happen by specifying the sqrtLimit in the swap to be exactly the next initialized tick, and add liquidity to the out of range position to steal fees.

Root Cause

In `RangePool.sol:174`, the break condition is incorrect and makes the pool desync.

Internal Pre-conditions

None.

External Pre-conditions

None.

Attack Path

1. Attacker swaps in `PoolManager.swap()`, setting the sqrtPrice to the next initialized tick. The Uniswap pool will cross the tick, but the DailyEpochGauge/IncentiveGauge won't.
2. Attacker adds liquidity to the position out of range in the Uniswap pool, stealing the fees from honest users that placed liquidity in the in range position.

Impact

100% of the fees are stolen by the out of range positions.

PoC

See above.

Mitigation

No response

Issue H-6: Reward Token Loss for LPs During NFT Position Transfer

Source: https://github.com/sherlock-audit/2025-09-bmx-deli-swap-judging/issues/184

Found by

0x73696d616f, blockace, secret__one

Summary

In **DeliHook v4 pools**, when a liquidity provider (LP) transfers their NFT position to a new owner, the **auto-claiming of rewards** is executed in favor of the new owner instead of the previous owner.

As a result, the previous LP permanently loses their accumulated reward tokens at the time of transfer.

Currently, during **position removal or modification**, the protocol performs an auto-claim of reward tokens to the LP.

However, during a **position transfer** (DeliHook v4 Pools), the `transferFrom` function changes ownership *before* the unsubscribe process is triggered. This ordering causes the reward claim to be credited to the **new owner**, not the previous owner.

Vulnerability Details

When an NFT position is transferred, the following function is called:

```
function transferFrom(
    address from,
    address to,
    uint256 id
) public virtual override onlyIfPoolManagerLocked {
    super.transferFrom(from, to, id);   // Ownership is updated first
    if (positionInfo[id].hasSubscriber()) _unsubscribe(id);
}
```

- Ownership of the NFT position is changed first.
- _unsubscribe is then called, which triggers the notifyUnsubscribe logic.

```
function notifyUnsubscribe(
    uint256 tokenId
) external override onlyAuthorizedCaller {
    NotifyContext memory c = _buildContextFromToken(tokenId, true);
```

```
    dailyEpochGauge.notifyUnsubscribeWithContext(
        c.posKey,
        c.pidRaw,
        c.currentTick,
        c.owner,
        c.tickLower,
        c.tickUpper,
        c.liquidity
    );

    ...
}
```

During unsubscribe, context is built for the position, including the owner:

```
function _buildContextFromToken(
    uint256 tokenId,
    bool includeOwner
) internal view returns (NotifyContext memory ctx) {
    IPositionHandler handler = getHandler(tokenId);

    ...

    if (includeOwner) {
        ctx.owner = handler.ownerOf(tokenId);
    }

}
```

```
function ownerOf(uint256 tokenId) external view override returns (address) {
    return IERC721(address(positionManager)).ownerOf(tokenId);
}
```

Because the transfer already occurred, ownerOf(tokenId) returns the new owner's address.

The unsubscribe function then passes this owner to notifyUnsubscribeWithContext:

```
function notifyUnsubscribeWithContext(
    bytes32 posKey,
    bytes32 poolIdRaw,
    int24 currentTick,
    address ownerAddr,
    int24 tickLower,
    int24 tickUpper,
    uint128 liquidity
) external onlyPositionManagerAdapter {
    ....
```

```
    _claimRewards(posKey, ownerAddr);

    _removePosition(pid, posKey);

}
```

Finally, rewards are transferred:

```
function _claimRewards(
    bytes32 posKey,
    address recipient
) internal returns (uint256 amount) {
    amount = positionRewards[posKey].claim();

    if (amount > 0) {
        BMX.transfer(recipient, amount);

        emit Claimed(recipient, amount);
    }
}
```

Since recipient is the new owner, the previous LP loses all their accumulated rewards.

Root Cause

The reward claim in notifyUnsubscribeWithContext (https://github.com/sherlock-audit/2025-09-bmx-deli-swap/blob/main/deli-swap-contracts/src/DailyEpochGauge.sol#L706-L707) uses the current NFT owner (post-transfer) instead of the previous LP (position owner):

```
function notifyUnsubscribeWithContext(
    bytes32 posKey,
    bytes32 poolIdRaw,
    int24 currentTick,
    address ownerAddr,
    int24 tickLower,
    int24 tickUpper,
    uint128 liquidity
) external onlyPositionManagerAdapter {

    ...

    _claimRewards(posKey, ownerAddr); // root-cause
    _removePosition(pid, posKey);
}
```

Internal Pre-conditions

1. LP holds an NFT position in a DeliHook v4 pool.
2. LP transfers their NFT position to another account.

External Pre-conditions

None

Attack Path

1. An attacker purchases an existing NFT position from an LP.
2. The transferFrom function executes, updating the NFT ownership to the attacker.
3. _unsubscribe is triggered, building context with the attacker's address as the new owner.
4. notifyUnsubscribeWithContext calls _claimRewards with the attacker's address.
5. The attacker receives all reward tokens that belonged to the previous LP.

Impact

- Loss of accumulated rewards: The previous LP permanently loses all their unclaimed rewards.
- High Impact – direct and irreversible economic loss for LPs.

PoC

None

Mitigation

The reward claim logic should be updated to reference the recorded position owner (tracked from subscription) rather than the current NFT owner.

```
```solidity
    function notifyUnsubscribeWithContext(
        bytes32 posKey,
        bytes32 poolIdRaw,
        int24 currentTick,
        address ownerAddr,
        int24 tickLower,
```
```

```
        int24 tickUpper,
        uint128 liquidity
    ) external onlyPositionManagerAdapter {

        ...

        @ audit Bug
-     _claimRewards(posKey, ownerAddr);

       @ audit Fix

+     address owner = positionOwner[posKey];
+     _claimRewards(posKey, ownerAddr);



    }
```

Issue H-7: In the `IncentiveGauge._upsertIncentive()` function, `_updatePoolByPid()` should be called outside the `if` statement

Source: https://github.com/sherlock-audit/2025-09-bmx-deli-swap-judging/issues/240

Found by

0xpetern, ami, cpsec, jayjoshix, maigadoh, oct0pwn, x15

Summary

In the `IncentiveGauge._upsertIncentive()` function, if `rewardRate == 0`, then `_updatePoolByPid()` is not executed. This results in the `lastUpdated` timestamp of the pool remaining at its original, older timestamp. Meanwhile, the reward token being upserted begins its release from the current timestamp.

Consequently, during the next update, the released reward amount for the token will be inaccurately calculated. The released amount is determined by the timestamp difference between the pool's `lastUpdated` and the moment of the update, incorrectly assuming that the reward started its release at the pool's `lastUpdated` timestamp.

Root Cause

As shown at line 486, the function `IncentiveGauge._upsertIncentive()` calls `_updatePoolByPid()` only if `info.rewardRate > 0`.

```
    function _upsertIncentive(PoolId pid, IERC20 token, uint256 amount) internal
    ↪  returns (uint128 rate) {
        IncentiveInfo storage info = incentives[pid][token];

        uint256 total = amount;
        if (info.rewardRate > 0) {
            // For active schedules, first update pool accounting, then add
            ↪  leftover budget
486          _updatePoolByPid(pid);
            if (block.timestamp < info.periodFinish) {
                uint256 remainingTime = info.periodFinish - block.timestamp;
                uint256 leftover = remainingTime * info.rewardRate;
                if (amount <= leftover) revert DeliErrors.InsufficientIncentive();
                total += leftover;
            }
        }

        rate = SafeCast.toUint128(total / TimeLibrary.WEEK);
```

```
        info.rewardRate = rate;
497     info.periodFinish = uint64(block.timestamp + TimeLibrary.WEEK);
        info.lastUpdate = uint64(block.timestamp);
        info.remaining = SafeCast.toUint128(total);
    }
```

Consider the case where `info.rewardRate == 0`:

1. `_updatePoolByPid()` is not called. Since the pool is not updated, the pool's `lastUpdated` remains an older timestamp.
2. As indicated at line 497, the release of the reward token begins from the current timestamp.
3. During the next update:

   The released reward amount is calculated using active seconds: `endTs - poolLast`, but this is incorrect because `poolLast` is an older timestamp that predates the reward token's release.

```
    function _updatePool(PoolId pid, int24 currentTick) internal {
        ...

            if (info.rewardRate > 0) {
                uint256 endTs = nowTs < info.periodFinish ? nowTs :
                ↪  info.periodFinish;
                if (endTs > poolLast) {
@>                  uint256 activeSeconds = endTs - poolLast;
                    amt = uint256(info.rewardRate) * activeSeconds;
                }
            }

            ...
```

As a result, a larger amount is incorrectly captured as the released amount.

Internal pre-conditions

External pre-conditions

Attack Path

Impact

This issue leads to an excess reward amount being allocated to the pool, which can result in a lack of available rewards for claiming, hinder position burning, and make it impossible to remove liquidity.

PoC

Mitigation

`_updatePoolByPid()` should be called outside the `if` statement:

```
    function _upsertIncentive(PoolId pid, IERC20 token, uint256 amount) internal
    ↪  returns (uint128 rate) {
        IncentiveInfo storage info = incentives[pid][token];

+       _updatePoolByPid(pid);

        uint256 total = amount;
        if (info.rewardRate > 0) {
            // For active schedules, first update pool accounting, then add
            ↪  leftover budget
-            _updatePoolByPid(pid);
            if (block.timestamp < info.periodFinish) {
                uint256 remainingTime = info.periodFinish - block.timestamp;
                uint256 leftover = remainingTime * info.rewardRate;
                if (amount <= leftover) revert DeliErrors.InsufficientIncentive();
                total += leftover;
            }
        }

        ...
    }
```

Issue H-8: `Voter::finalize()` incorrect rewards distribution due to transfering WETH before calling `distributor::setTokensPerInterval()`

Source: https://github.com/sherlock-audit/2025-09-bmx-deli-swap-judging/issues/242

Found by

0x73696d616f

Summary

`Voter::finalize()` transfers the WETH, and then calls `distributor.setTokensPerInterval()`. However, this order of operations is incorrect and it will lead to the previous reward rate being applied for the transferred WETH, doing an instantaneous rewards increase that will be arbitraged and stolen.

As can be seen below, `RewardsDistributor::setTokensPerInterval()` calls `RewardTracker::updateRewards()`, which calls `RewardsDistributor::distribute()`. Now, if `pendingRewards()` will apply the last `tokensPerInterval`, accrue for the last time called, and send to users if there is enough balance. Due to having sent the WETH before calling `setTokensPerInterval()`, it will instantly distribute this rewards as soon as the function is called, and there won't be enough rewards for the next week. Thus, past stakers (and bots or attackers that knew about this) will steal instant rewards, and honest stakers will not collect them over the full week.

RewardsDistributor

```
function setTokensPerInterval(uint256 _amount) external onlyAdmin {
    require(lastDistributionTime != 0, "RewardDistributor: invalid
    ↪  lastDistributionTime");
    IRewardTracker(rewardTracker).updateRewards();
    tokensPerInterval = _amount;
    emit TokensPerIntervalChange(_amount);
}

function pendingRewards() public view override returns (uint256) {
    if (block.timestamp == lastDistributionTime) {
        return 0;
    }

    uint256 timeDiff = block.timestamp.sub(lastDistributionTime);
    return tokensPerInterval.mul(timeDiff);
}

function distribute() external override returns (uint256) {
```

```
    require(msg.sender == rewardTracker, "RewardDistributor: invalid msg.sender");
    uint256 amount = pendingRewards();
    if (amount == 0) { return 0; }

    lastDistributionTime = block.timestamp;

    uint256 balance = IERC20(rewardToken).balanceOf(address(this));
    if (amount > balance) { amount = balance; }

    IERC20(rewardToken).safeTransfer(msg.sender, amount);

    emit Distribute(amount);
    return amount;
}
```

RewardTracker

```
function updateRewards() external override nonReentrant {
    _updateRewards(address(0));
}

function _updateRewards(address _account) private {
    uint256 blockReward = IRewardDistributor(distributor).distribute();
...
```

Root Cause

In `Voter.sol:262`, it transfers WETH before calling setTokensPerInterval().

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. Admin finalizes an epoch, sends WETH and sets the tokens per interval to distribute this amount of WETH over 1 week.
2. Due to the logic above, part of this WETH, if not all, will be consumed instantly and rewards won't be distributed over the full week.

Impact

Stolen rewards.

PoC

No response

Mitigation

First call setTokensPerInterval(), then transfer the WETH.

Issue M-1: Protocol fee conversion uses pre‑swap price snapshot

Source: https://github.com/sherlock-audit/2025-09-bmx-deli-swap-judging/issues/4

Found by

blockace, cergyk, globalace, silver_eth, x15

Description

In the concentrated‑liquidity hook (`DeliHook`), the fee is denominated in the user's specified token using the pre‑swap price snapshot, then converted to the fee currency (wBLT) before the swap executes.

The swap then moves the price, but the fee amount in wBLT is not re‑valued after the move. This lets an attacker bias the conversion rate ("prime") right before the large swap and unwind after, so the protocol under‑collects fees in wBLT. The documented floor/ceil asymmetry on token0→token1 conversions can be chosen by the attacker to further cheapify the fee.

The codepath reads the price before the swap and computes a base fee in the specified token. _beforeSwap pulls `sqrtPriceX96` from `StateLibrary.getSlot0` (pre‑swap), and computes

```
(uint160 sqrtPriceX96,,, uint24 poolFee) = StateLibrary.getSlot0(poolManager,
↪  key.toId());
// fee in specified-token units
uint256 baseFeeSpecified = FullMath.mulDiv(absAmountSpecified, uint256(poolFee),
↪  1_000_000);
```

This is explicitly done before the swap in the concentrated hook.

Next it converts that base fee to wBLT using the pre-swap price and asymmetric rounding. If the specified token isn't wBLT, the hook converts via p = token1 per token0 = $\frac{sqrt^2}{2^{192}}$ with:

- token0 → token1: `floor(base * p)` (two `mulDiv` floors)
- token1 → token0: `ceil(base / p)` (two `mulDivRoundingUp`)

These exact paths are implemented in `_beforeSwap` and stored into `_pendingFee`.

It then charges the fee amount computed pre‑swap after the swap without re‑pricing. `_afterSwap` simply takes `feeOwed = _pendingFee` (the amount computed off the pre‑swap snapshot) and transfers that many wBLT to the `FeeProcessor`. There is no recomputation against the post‑swap price.

As a result If the swap moves price materially (common on steep concentrated curves), the conversion done at the pre‑swap price can be far from the fair post‑swap valuation. An attacker can

1. Prime the price with a small swap so that the pre-swap `p` used for conversion is favorable (for example, make `p` high to minimize `ceil(base/p)` when paying fee in token0; or make `p` low to minimize `floor(base*p)` when paying fee in token1).
2. Execute a large swap; the fee (in wBLT) is charged at the pre-swap conversion.
3. Unwind the prime (or do an opposite-leg cycle across two pools).

Because the hook allows either rounding floor (token0→token1) or ceil (token1→token0), an attacker can choose orientation and exactInput/Output so the rounding direction plus the stale price both reduce the wBLT amount actually paid.

Note: This issue is specific to the concentrated hook shown above. The constant‑product hook computes/forwards fees differently and does not use `sqrtPriceX96` for fee conversion; this finding targets the concentrated path.

Recommendation

Store `baseFeeSpecified` and whether it's token0/token1 in `_beforeSwap`, but delay the conversion to wBLT until `_afterSwap` and convert using `sqrtPriceX96` read after the swap. This aligns the fee's valuation point with the deduction point. (All the plumbing you need is already there: `_pendingFee` and `_pendingCurrency` handoffs, and `_afterSwap` is already where the fee is transferred.)

Issue M-2: Unconditional lastUpdated advance in RangePool.sync leads to loss of streamed BMX when pool liquidity == 0

Source: https://github.com/sherlock-audit/2025-09-bmx-deli-swap-judging/issues/53

Found by

0x73696d616f, blockace, maigadoh, makeWeb3safe

Summary

A logic ordering issue in RangePool::sync causes the pool's lastUpdated timestamp to advance even when liquidity == 0. If the gauge consumes its N+2 day-bucket during a period with zero active liquidity, the streamed BMX is never credited to the pool's rewards-per-liquidity accumulator and therefore cannot be claimed later, even after liquidity returns. The tokens remain in the gauge contract balance but are unreachable by LP positions.

Root Cause

https://github.com/sherlock-audit/2025-09-bmx-deli-swap/blob/main/deli-swap-contracts/src/libraries/RangePool.sol#L282 https://github.com/sherlock-audit/2025-09-bmx-deli-swap/blob/main/deli-swap-contracts/src/libraries/RangePool.sol#L287

```
    /// @notice High-level helper used by gauges: initialise (if needed),
    ↪  accumulate rewards and adjust to new tick.
    /// @param self        Pool state storage pointer.
    /// @param perTokenAmounts  Token amounts accrued over the window since last
    ↪  update (0 allowed).
    /// @param tickSpacing Pool tick spacing (passed to adjustToTick).
    /// @param activeTick  Current active tick from PoolManager.slot0.
    function sync(
        State storage self,
        address[] memory tokens,
        uint256[] memory perTokenAmounts,
        int24 tickSpacing,
        int24 activeTick
    ) internal {
        // Bootstrap state on first touch so accumulate sees dt = 0
        if (self.lastUpdated == 0) {
            self.initialize(activeTick);
            // no need to accumulate or adjust because lastUpdated = now and tick
            ↪  == activeTick
```

```
            return;
        }
        // 1. Update lastUpdated and credit per-token amounts
@>>         self.lastUpdated = uint64(block.timestamp);

        if (self.liquidity > 0) {
            uint256 len = tokens.length;
            for (uint256 i; i < len; ++i) {
  @>>               _accumulateToken(self, tokens[i], perTokenAmounts[i]);
            }
        }
        // 2. If price moved out of current range adjust liquidity & tick, flipping
        ↪  per-token outside
        if (activeTick != self.tick) {
            self.adjustToTick(tickSpacing, activeTick, tokens);
        }
    }
}
```

RangePool::sync sets self.lastUpdated = uint64(block.timestamp) before applying per-token accumulation. It only executes `_accumulateToken` when self.liquidity > 0.

When self.liquidity == 0, lastUpdated advances but accumulation is skipped so, `DailyEpochGauge::_syncPoolStateCore's` computed `_amountOverWindow` is effectively consumed and lost for distribution.

```
    function _syncPoolStateCore(PoolId pid, int24 activeTick, int24 tickSpacing)
    ↪  internal {
        RangePool.State storage pool = poolRewards[pid];
        uint256 t0 = pool.lastUpdated;
        uint256 t1 = block.timestamp;
        address[] memory toks = new address[](1);
        toks[0] = address(BMX);
        uint256[] memory amts = new uint256[](1);
        if (t1 > t0) {
@>>             amts[0] = _amountOverWindow(pid, t0, t1);
        } else {
            amts[0] = 0;
        }
@>>       pool.sync(toks, amts, tickSpacing, activeTick);
    }
```

Internal Pre-conditions

- DailyEpochGauge has a non□zero day bucket for a pool (via DailyEpochGauge::addRewards called by FeeProcessor).
- The pool's RangePool.State.lastUpdated is older than the target streaming window.

- RangePool.State.liquidity == 0 at the time DailyEpochGauge syncs the pool (no active in-range liquidity).
- DailyEpochGauge::pokePool (or any sync path) is invoked, causing RangePool::sync to run.

External Pre-conditions

- A position that previously provided liquidity was unsubscribed (via PositionManagerAdapter::notifyUnsubscribeWithContext), resulting in the pool having zero active liquidity.
- The scheduled streaming day arrives (N+2), and an external actor or hook triggers a pool sync (e.g., DeliHook calls pokePool on swaps).
- The fee flow has previously transferred BMX into a gauge balance (via FeeProcessor → DAILY_GAUGE.addRewards), so tokens exist but are unallocated.

Attack Path

No response

Impact

- Funds meant for distribution (BMX) are retained in the gauge balance but never credited to any pool accumulator for claim by LP positions.
- LPs present after liquidity returns cannot claim past streaming amounts; protocol revenue intended for LPs can be effectively sidelined.
- Denial of reward for LPs; accounting mismatch between gauge token balance and claimable amounts.
- This is not an immediate theft but a correctness/availability failure with lasting distribution impact.

PoC

1. Place the test in the `GaugeStream.t.sol` test file.
2. Run the test using this `forge test --mt testLostStreamingWhenZeroLiquidity -vvvv`

```
/// @notice Demonstrates that if a pool has zero liquidity when the gauge
   ///         sync runs, the per-day bucket amounts are not applied to the
   ///         pool accumulator (they become effectively unallocated to any
   ///         position). This reproduces the "lost streaming when
   ///         liquidity==0" behaviour: tokens remain in the gauge balance
```

```
///         but positions can't claim them.
function testLostStreamingWhenZeroLiquidity() public {
    // Precondition: setUp has funded the gauge and added a day-bucket.
    uint256 bucket = 1000 ether;

    // Ensure initial gauge balance contains the bucket
    assertGe(bmx.balanceOf(address(gauge)), bucket);

    // 1) Remove the only tracked liquidity so pool active liquidity == 0
    positionManager.unsubscribe(wideTokenId);

    // Confirm pool has zero active liquidity now via gauge view
    (,, uint128 activeLiq) = gauge.getPoolData(pid);
    assertEq(activeLiq, 0, "pool liquidity should be zero");

    // 2) Advance to the streaming day (N+2) so the gauge will try to credit
    uint256 dayEnd = TimeLibrary.dayNext(block.timestamp);
    vm.warp(dayEnd + 1 days);

    // Sanity: streamRate should be non-zero because the day-bucket exists
    assertGt(gauge.streamRate(pid), 0, "streamRate should be active for the
    ↪  day");

    // 3) Trigger pool sync while liquidity == 0. Because RangePool.sync
    //    updates lastUpdated before accumulating, the amounts for the
    //    elapsed window are not applied when liquidity==0.
    vm.prank(address(hook));
    gauge.pokePool(key);

    // 4) Re-add liquidity (mint and subscribe a fresh position after the
    //    missed window). This new position cannot recover the previously
    //    scheduled streaming for the earlier window.
    uint256 tokenIdNew;
    (tokenIdNew,) = EasyPosm.mint(
        positionManager,
        key,
        -60000,
        60000,
        1e21,
        type(uint256).max,
        type(uint256).max,
        address(this),
        block.timestamp + 1 hours,
        bytes("")
    );
    positionManager.subscribe(tokenIdNew, address(adapter), bytes(""));

    // 5) Sync now that liquidity > 0. Only amounts since the previous
    //    lastUpdated will be applied - the bucket that streamed during
    //    The earlier zero-liquidity window is not credited to positions.
```

```
        vm.prank(address(hook));
        gauge.pokePool(key);

        // 6) Claim for owner: should receive zero (or very small) because the
        // Earlier, the streaming window was missed when liquidity was 0.
        PoolId[] memory arr = new PoolId[](1);
        arr[0] = pid;
        uint256 balBefore = bmx.balanceOf(address(this));
        gauge.claimAllForOwner(arr, address(this));
        uint256 claimed = bmx.balanceOf(address(this)) - balBefore;

        // The test demonstrates the bug: the bucket is still sitting in the
        // gauge contract balance but positions received nothing for the
        // streaming window that occurred while liquidity was zero.
        assertEq(claimed, 0, "expected no rewards allocated to position");
        assertGe(bmx.balanceOf(address(gauge)), bucket, "gauge should still hold the
        ↪  bucket funds");
    }
```

Mitigation

Do not advance lastUpdated and therefore do not consume the elapsed-window amounts. when `self.liquidity == 0`. Return early so the pending per-day amounts remain available and are applied once liquidity appears.

Apply changes in the `RangePool.sol::sync`

```
@@
-        // 1. Update lastUpdated and credit per-token amounts
-        self.lastUpdated = uint64(block.timestamp);
-
-        if (self.liquidity > 0) {
-            uint256 len = tokens.length;
-            for (uint256 i; i < len; ++i) {
-                _accumulateToken(self, tokens[i], perTokenAmounts[i]);
-            }
-        }
+        // If no active liquidity, do not advance lastUpdated or consume amounts.
+        // Preserve the time window so amounts are processed later when liquidity
 ↪  exists.
+        if (self.liquidity == 0) {
+            // Adjust price movement if needed, but keep lastUpdated unchanged.
+            if (activeTick != self.tick) {
+                self.adjustToTick(tickSpacing, activeTick, tokens);
+            }
+            return;
+        }
+
+        // 1. Update lastUpdated and credit per-token amounts
```

```
+        self.lastUpdated = uint64(block.timestamp);
+
+        uint256 len = tokens.length;
+        for (uint256 i; i < len; ++i) {
+            _accumulateToken(self, tokens[i], perTokenAmounts[i]);
+        }
```

Issue M-3: Integer Truncation in Incentive Rate Permanently Locks Unstreamed Rewards

Source: https://github.com/sherlock-audit/2025-09-bmx-deli-swap-judging/issues/141

Found by

0x73696d616f, ami, blockace, cergyk, cpsec, r1ver

Summary

The integer division in `_upsertIncentive` will cause unstreamable “dust” accumulation for reward tokens as every remainder from `total / WEEK` is truncated. This results in permanently locked tokens for the protocol as any caller who tops up incentives will repeatedly create dust that cannot be streamed. The issue is more severe for tokens with fewer decimals, since each truncated remainder represents a larger absolute value, making the amount of unstreamable dust proportionally greater.

Root Cause

Root Cause

In `IncentiveGauge._upsertIncentive`, the incentive rate is computed with **integer division** against a 7-day window, so any fractional part is truncated and never streamed. With `uint256 TimeLibrary.WEEK = 7 days`, the code is: https://github.com/sherlock-audit/2025-09-bmx-deli-swap/blob/main/deli-swap-contracts/src/IncentiveGauge.sol#L483-L499

```
// IncentiveGauge::_upsertIncentive
uint256 total = amount; // (+ leftover if topping up an active stream)
rate = SafeCast.toUint128(total / TimeLibrary.WEEK); // <-- integer truncation
info.rewardRate = rate;
info.periodFinish = uint64(block.timestamp + TimeLibrary.WEEK);
info.lastUpdate = uint64(block.timestamp);
info.remaining = SafeCast.toUint128(total);
```

Streaming later accounts as `dt * info.rewardRate`, and when the period ends `_updatePool` **zeros** any leftover instead of paying it out: https://github.com/sherlock-audit/2025-09-bmx-deli-swap/blob/main/deli-swap-contracts/src/IncentiveGauge.sol#L723-L727

```
// IncentiveGauge::_updatePool
if (nowTs >= info.periodFinish || info.remaining == 0) {
    info.remaining = 0;            // <-- residual is discarded
    info.rewardRate = 0;
    emit IncentiveDeactivated(pid, tok);
}
```

This means `total % WEEK` (the remainder from `total / WEEK`) becomes **permanently locked** in the contract each time an incentive is created or topped up.

The effect worsens for tokens with **fewer decimals** (coarser granularity): the same truncated remainder represents a larger absolute token amount. Because the remainder is truncated on every funding/top-up and later wiped, dust accumulates and remains unclaimable, The smaller the decimal places. The higher the loss.

Internal Pre-conditions

1. **Any user** needs to call `createIncentive()` to set `amount` of reward tokens to be greater than `TimeLibrary.WEEK` (604,800 seconds).
2. **Reward token** needs to have decimals less than 18 (e.g., 6 for USDT/USDC or 4 for some tokens).

External Pre-conditions

None

Attack Path

1. **Any user** calls `createIncentive(key, USDC, 100e6)` (USDC has 6 decimals; `amount` not a multiple of `WEEK = 604,800` seconds).
2. **Contract** pulls `amount` and calls `_upsertIncentive`, where it computes `rate = uint128(total / WEEK)` using integer division (fraction truncated).
3. **Contract** sets `info.rewardRate = rate`, `info.remaining = total`; streaming accrues later as `dt * rate`.
4. **During the 7-day period**, `_updatePool` repeatedly credits rewards using `dt * rate`, leaving a **remainder** `dust = total - rate * WEEK` unstreamed.
5. **At/after period finish**, `_updatePool` deactivates the incentive and does: `info.remaining = 0; info.rewardRate = 0;` – the **unstreamed remainder is discarded** (never paid).
6. **Result:** The truncated remainder (e.g., for 6-decimals tokens like USDT/USDC this can be noticeable) stays **permanently locked** in the contract.
7. **If topped up repeatedly**, steps 1–6 repeat; each new schedule creates additional truncation dust that cumulatively remains unclaimable.

Impact

The protocol suffers a guaranteed loss of the “fractional dust” whenever incentives are created. Since rewards are streamed per second over exactly 604,800 seconds (7 days),

any amount that cannot be evenly divided is truncated. For tokens with 6 decimals like USDT/USDC, this means that each second up to 0.000001 can be discarded. Over a full week, the cumulative loss can approach 0.4 USDC for every 1 USDC allocated as rewards. Tokens with fewer decimals (e.g., 4) will suffer even larger dust losses.

PoC

I write a poc in `deli-swap-contracts/test/unit/IncentiveGaugeRoundingDust.t.sol` run this command to test`forge test --match-contract IncentiveGaugeRoundingDustTest -vvv`

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

import {Test} from "forge-std/Test.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

import {IncentiveGauge} from "src/IncentiveGauge.sol";
import {MockPoolManager} from "test/mocks/MockPoolManager.sol";
import {IPoolManager} from "@uniswap/v4-core/src/interfaces/IPoolManager.sol";

import {PoolKey} from "@uniswap/v4-core/src/types/PoolKey.sol";
import {PoolId, PoolIdLibrary} from "@uniswap/v4-core/src/types/PoolId.sol";
import {Currency} from "@uniswap/v4-core/src/types/Currency.sol";
import {IHooks} from "@uniswap/v4-core/src/interfaces/IHooks.sol";

import {EfficientHashLib} from "solady/utils/EfficientHashLib.sol";
import {IPositionManagerAdapter} from "src/interfaces/IPositionManagerAdapter.sol";
import {PositionInfo} from "v4-periphery/src/libraries/PositionInfoLibrary.sol";

contract USDC6 is IERC20 {
    string public name = "Mock USDC";
    string public symbol = "USDC";
    uint8 public constant decimals = 6;
    mapping(address => uint256) public override balanceOf;
    mapping(address => mapping(address => uint256)) public override allowance;
    uint256 public override totalSupply;

    function transfer(address to, uint256 amount) external override returns (bool) {
        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;
        emit Transfer(msg.sender, to, amount);
        return true;
    }
    function approve(address spender, uint256 amount) external override returns
    ↪  (bool) {
        allowance[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
```

```
    }
    function transferFrom(address from, address to, uint256 amount) external
    ↪  override returns (bool) {
        uint256 allowed = allowance[from][msg.sender];
        if (allowed != type(uint256).max) allowance[from][msg.sender] = allowed -
        ↪  amount;
        balanceOf[from] -= amount;
        balanceOf[to] += amount;
        emit Transfer(from, to, amount);
        return true;
    }
    function mint(address to, uint256 amount) external {
        totalSupply += amount;
        balanceOf[to] += amount;
        emit Transfer(address(0), to, amount);
    }
}

contract IncentiveGaugeRoundingDustTest is Test {
    using PoolIdLibrary for PoolKey;

    MockPoolManager pm;
    IncentiveGauge gauge;
    USDC6 usdc;

    address owner = address(this);
    address hook = address(this);
    address funder = address(0xF00D);
    address lpOwner = address(0xDEADBEEF);

    PoolKey key;
    PoolId pid;

    function setUp() public {
        pm = new MockPoolManager();
        // Adapter is not used in this PoC; pass this contract address
        gauge = new IncentiveGauge(IPoolManager(address(pm)),
        ↪  IPositionManagerAdapter(address(this)), hook);
        usdc = new USDC6();

        // Build a minimal PoolKey
        key = PoolKey({
            currency0: Currency.wrap(address(0xC0)),
            currency1: Currency.wrap(address(0xC1)),
            fee: 0,
            tickSpacing: 60,
            hooks: IHooks(address(0))
        });
        pid = key.toId();
```

```
        // Initialize pool via hook and set whitelist
        gauge.initPool(key, 0);
        gauge.setWhitelist(IERC20(address(usdc)), true);

        // Prepare PoolManager slot0 for updates
        pm.setPoolSlot0(PoolId.unwrap(pid), 79228162514264337593543950336, 0); //
        ↪  sqrtPriceX96=1, tick=0

        // Fund depositor
        usdc.mint(funder, 100_000_000); // 100 USDC with 6 decimals
        vm.prank(funder);
        usdc.approve(address(gauge), type(uint256).max);
    }

    function test_PoC_incentive_dust_is_discarded_and_locked() public {
        uint256 amount = 100_000_000; // 100 USDC (6 decimals)

        // Create incentive (pulls tokens)
        vm.prank(funder);
        gauge.createIncentive(key, IERC20(address(usdc)), amount);

        // Verify integer division on rate
        (uint256 rate,, uint256 remainingBefore) = gauge.incentiveData(pid,
        ↪  IERC20(address(usdc)));
        assertEq(remainingBefore, amount, "remaining should equal amount at start");
        uint256 expectedRate = amount / 604800; // WEEK seconds
        assertEq(rate, expectedRate, "rate should be floor(amount/WEEK)");
        uint256 dust = amount - expectedRate * 604800;
        assertGt(dust, 0, "dust must be > 0 for non-multiple amounts");

        // Advance to after finish and update once via hook
        vm.warp(block.timestamp + 604800 + 1);
        gauge.pokePool(key);

        // After finish, incentive is deactivated and remaining is set to 0
        (uint256 rateAfter,, uint256 remainingAfter) = gauge.incentiveData(pid,
        ↪  IERC20(address(usdc)));
        assertEq(rateAfter, 0, "rewardRate should be 0 after finish");
        assertEq(remainingAfter, 0, "remaining set to 0 (dust discarded)");

        // Contract still holds the full amount (no positions to claim)
        uint256 bal = usdc.balanceOf(address(gauge));
        assertEq(bal, amount, "all tokens remain in contract; dust is
        ↪  irrecoverable");
    }

    function test_PoC_dust_persists_after_claiming_streamed_amount() public {
        // Create incentive and a single LP position that will claim streamed amount
        uint256 amount = 100_000_000; // 100 USDC
        vm.prank(funder);
```

```
        gauge.createIncentive(key, IERC20(address(usdc)), amount);

        // Register one position directly via adapter-only path (adapter is this
        ↪  contract)
        uint256 tokenId = 1;
        bytes32 posKey = EfficientHashLib.hash(bytes32(tokenId),
        ↪  bytes32(PoolId.unwrap(pid)));
        gauge.notifySubscribeWithContext(tokenId, posKey, PoolId.unwrap(pid), 0,
        ↪  -120, 120, 1_000_000, lpOwner);

        // Advance to finish and update
        vm.warp(block.timestamp + 604800 + 1);
        gauge.pokePool(key);

        // Claim to owner via admin force-unsubscribe
        uint256 ownerBalBefore = usdc.balanceOf(lpOwner);
        gauge.adminForceUnsubscribe(posKey, true);
        uint256 ownerBalAfter = usdc.balanceOf(lpOwner);

        // Owner received streamed ~= rate * WEEK (allow 1 unit rounding difference)
        (uint256 rateAfter,,) = gauge.incentiveData(pid, IERC20(address(usdc)));
        assertEq(rateAfter, 0, "rate zero after finish");
        uint256 expectedRate = amount / 604800;
        uint256 expectedStreamed = expectedRate * 604800;
        uint256 claimed = ownerBalAfter - ownerBalBefore;
        bool withinRounding = (claimed == expectedStreamed) || (claimed + 1 ==
        ↪  expectedStreamed);
        assertTrue(withinRounding, "owner received streamed amount within rounding
        ↪  tolerance");

        // Gauge retains exactly the dust (plus any unclaimed streamed, but with
        ↪  one LP it should be zero)
        uint256 gaugeBal = usdc.balanceOf(address(gauge));
        assertEq(gaugeBal, amount - claimed, "gauge holds unclaimed portion (dust +
        ↪  rounding)");
        assertGe(gaugeBal, amount - expectedStreamed, "gauge holds at least
        ↪  theoretical dust");
    }

    // Minimal adapter method used by adminForceUnsubscribe
    function getPoolAndPositionInfo(uint256 /*tokenId*/ ) external view returns
    ↪  (PoolKey memory k, PositionInfo info) {
        k = key; // return the pool key used in this test
        // info left as default zeros; not used by the code path under test
    }
}
```

Mitigation

When funding an incentive, **round the deposit down to a full-week multiple** and handle the remainder immediately so no dust can ever form:

1. Compute `effective = (amount / WEEK) * WEEK; remainder = amount - effective;`
2. Use `effective` to derive `rate = effective / WEEK` and start the 7-day stream.
3. **Return** `remainder` to the funder (or **carry it forward** into the next funding call via a per-pool `pendingRemainder` accumulator until it reaches at least one full `WEEK`).

This guarantees `rate * WEEK == effective` and the entire streamed budget is paid out exactly, while the remainder is never trapped as unstreamable dust.

Issue M-4: Users always pay fee on the full swapped amount in the `DeliHook`, even if the swap is smaller

Source: https://github.com/sherlock-audit/2025-09-bmx-deli-swap-judging/issues/154

Found by

0x73696d616f, blockace, cergyk

Summary

Users can specify a price limit in their swap, limiting the amount of funds actually swapped, or there may not be enough assets of one of the tokens to swap. The `DeliHook` doesn't take this into account, always paying fee over the full amount specified, overcharging the user by a very significant amount, link.

Root Cause

In `DeliHook.sol:231`, the fee is always calculated on the full input amount, regardless of the actual amount swappd.

Internal Pre-conditions

None.

External Pre-conditions

None.

Attack Path

1. User does a swap, specifying a sqrt limit price that will be hit without using the full input/output amount. Or, alternatively, the pool runs out of one of the tokens.
2. `DeliHook` will apply the fee on the full input amount, instead of just the amount swapped, link to where the fee is applied.

Impact

If the user specifies a sqrtPrice limit that only swaps 50% of the input/output funds, and the fee is calculated on 100% of the amount, it would be an error of 100%, high severity, and scales with the amount swapped.

PoC

None

Mitigation

Adjust the fee for the actual amount swapped.

Issue M-5: `DeliHookConstantProduct` swapping `exactOutput` and `_feeFromOutput` is incorrect

Source: https://github.com/sherlock-audit/2025-09-bmx-deli-swap-judging/issues/200

Found by

0x73696d616f, cergyk, jayjoshix, silver_eth, x15

Summary

`DeliHookConstantProduct` intends to preserve K in all swaps in order to forward fees to the `FeeProcessor`, and is achieved for all swap types. However, this is performed with an incorrect calculation when doing **exactOutput oneForZero**, in which the K is preserved, but the fee charged is incorrect, as will be demonstrated.

Suppose the state of the v2 pool is: Token0 = 1000 Token1 = 1000 FeeCurrency = token0

ExactInput, oneForZero swap (zeroForOne == false)

exactInput = 100 token1 specifiedAmount = 100 Now, let's calculate the _pendingFeeAmount from _calculateOutputFee():

_getAmountOut(false, 100, 1000, 1000, 1e4): feeBasis = 1e6 - 1e4 = 990000 amountInWithFee = 100 * 990000 = 99000000 numerator = 99000000 * 1000 = 99000000000 denominator = 1000 * 1e6 + 99000000 = 1099000000 amountOut = 99000000000 / 1099000000 = 90.0818926297 outputWithFee = 90.0818926297

_getAmountOut(false, 100, 1000, 1000, 0): feeBasis = 1e6 - 0 = 1e6 amountInWithFee = 100 * 1e6 = 1e8 numerator = 1e8 * 1000 = 1e11 denominator = 1000 * 1e6 + 1e8 = 1100000000 amountOut = 1e11 / 1100000000 = 90.9090909091 outputWithoutFee = 90.9090909091

outputFee = outputWithoutFee - outputWithFee = 90.9090909091 - 90.0818926297 = 0.8271982794 So, _pendingFeeAmount = 0.8271982794

We need to get the unspecified amount, which is the amountOut in this case, and is implemented by the DeliHookConstantProduct, which internally calls getAmountOut. We have already calculated this value above, which is the outputWithFee, 90.0818926297. Hence, unspecifiedAmount = 90.0818926297.

returnDelta becomes (100, -90.0818926297).

Then, in DeliHookConstantProduct::_updateReservesAfterSwap():

specifiedAmount = 100 amountOut = 90.0818926297 delta0 = -90.0818926297 delta1 = 100 delta0 -= 0.8271982794 = -90.9090909091 (_pendingFeeAmount = 0.8271982794) reserves = (1000 - 90.9090909091, 1000 + 100) = (909.090909091, 1100)

And the final K is 909.090909091 * 1100 = 1000000, same as initially, which adds up.

Then, in the Hooks.sol code of Uniswap v4, swapDelta becomes swapDelta = swapDelta - hookDelta = 0 - (hookDeltaUnspecified, hookDeltaSpecified) = 0 - (-90.0818926297, 100) = (+90.0818926297, -100), which comes from returnDelta above, but switched due to the Hooks.sol logic.

As a result, the user will receive +90.0818926297 token0 and pay 100 token1, and the K of the reserves is kept intact, while the FeeProcessor will collect all the fees.

However, it doesn't work correctly for exactOutput, zeroForOne == false, the K will be correct but the fee charged is incorrect.

ExactOutput, oneForZero swap (zeroForOne == false)

exactOutput = 90.0818926297 token0 (amount received from 1st example) specifiedAmount = 90.0818926297 Now, let's calculate the _pendingFeeAmount, which in this case is (amountSpecified * uint256(key.fee)) / 1_000_000 = 90.0818926297 * 1e4 / 1e6 = 0.90081892629, already different from the example above and is incorrect.

We need to get the unspecified amount, which is the amountIn in this case, and is implemented by the DeliHookConstantProduct, which does:

```
targetOut = amountSpecified * (1e6 + fee) / 1e6 = 90.0818926297 * (1e6 + 1e4) / 1e6
↪  = 90.982711556.
amountUnspecified = _getAmountIn(false, 90.982711556, 1000, 1000, 0) = (reserveIn *
↪  amountOut * 1e6) / ((reserveOut - amountOut) * feeBasis) = (1000 *
↪  90.982711556) / (1000 - 90.982711556) = 100.089088197
```

returnDelta becomes (-90.0818926297, 100.089088197).

Then, in DeliHookConstantProduct::_updateReservesAfterSwap():

specifiedAmount = 90.0818926297 amountIn = 100.089088197 (same as before) delta0 = -90.0818926297 delta1 = 100.089088197 delta0 -= 0.90081892629 = -90.982711556 (_pendingFeeAmount = 0.90081892629) reserves = (1000 - 90.982711556, 1000 + 100.089088197) = (909.090909091, 1100.0890882)

The final K is 909.017288444 * 1100.0890882 = 1000000, same as initially, which again adds up.

Then, in the Hooks.sol code of Uniswap v4, swapDelta becomes swapDelta = swapDelta - hookDelta = 0 - (hookDeltaSpecified, hookDeltaUnspecified) = 0 - (-90.0818926297, 100.089088197) = (+90.0818926297, -100.089088197), which comes from returnDelta above, but switched due to the Hooks.sol logic.

As a result, the user will receive +90.0818926297 token0 and pay 100.089088197 token1.

Final result

From the 2 analogous examples above, in both the user gets 90.0818926297 token0, but pays 100 token1 in the first example (exactInput) and 100.089088197 in the second example (exactOutput).

In the second example the user is overpaying a significantly higher fee than supposed.

Instead of being 0.8271982794, it is 0.90081892629, a very significant error of (0.90081892629 - 0.8271982794) / 0.8271982794 = 8.9 %, high severity.

Root Cause

There are several instances where it needs to be changed and the root cause can be attributed to it (calculation takes place in several steps)

1. https://github.com/sherlock-audit/2025-09-bmx-deli-swap/blob/main/deli-swap-contracts/src/DeliHookConstantProduct.sol#L317-L318
2. https://github.com/sherlock-audit/2025-09-bmx-deli-swap/blob/main/deli-swap-contracts/src/DeliHookConstantProduct.sol#L576-L577
3. https://github.com/sherlock-audit/2025-09-bmx-deli-swap/blob/main/deli-swap-contracts/src/DeliHookConstantProduct.sol#L654

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. User does an exactOutput, zeroForOne == false swap and pays too many fees.

Impact

User suffers a loss of 8.9%.

PoC

Add the 2 following tests to SwapLifecycle_V2.t.sol:

```
function testExactInputSwapZeroForOne() public {
    uint256 inputAmount = 5 ether;

    (uint128 r0Before, uint128 r1Before) = hook.getReserves(pid);
    uint256 bmxBefore = bmx.balanceOf(address(this));
    uint256 wbltBefore = wblt.balanceOf(address(this));

    // Perform exact input swap
```

```
    poolManager.unlock(abi.encode(address(bmx), inputAmount, true));

    (uint128 r0After, uint128 r1After) = hook.getReserves(pid);

    // Verify token balances
    assertEq(bmxBefore - bmx.balanceOf(address(this)), inputAmount, "BMX spent
    ↪  should match input");
    uint256 outputAmount = wblt.balanceOf(address(this)) - wbltBefore;

    assertEq(outputAmount, 4748297375815592703);
}

function testExactOutputZeroForOne() public {
    uint256 outputAmount = 4748297375815592703;

    (uint128 r0Before, uint128 r1Before) = hook.getReserves(pid);
    uint256 bmxBefore = bmx.balanceOf(address(this));
    uint256 wbltBefore = wblt.balanceOf(address(this));

    // Perform exact output swap
    poolManager.unlock(abi.encode(address(bmx), outputAmount, false));

    (uint128 r0After, uint128 r1After) = hook.getReserves(pid);

    // Verify output received
    assertEq(wblt.balanceOf(address(this)) - wbltBefore, outputAmount, "wBLT
    ↪  received should match output");
    uint256 inputAmount = bmxBefore - bmx.balanceOf(address(this));

    assertEq(inputAmount, 5000702855111981996); //@audit bigger than above of 5
    ↪  ether
}
```

Mitigation

Hinted at the solution in the root cause, to verify the fix, the fees have to match when swapping the same amounts.

Issue M-6: Attacker can swap wBTC in the DeliHook multiples times to not pay / pay less swap fees

Source: https://github.com/sherlock-audit/2025-09-bmx-deli-swap-judging/issues/203

Found by

0x73696d616f, 0xDyDx

Summary

`DeliHook::_beforeSwap()` rounds down the `baseFeeSpecified` instead of rounding up, allowing an attacker to abuse the low gas fees on Base and the high btc price to not pay any fees (or half).

The attacker swaps in a pool with the DeliHook and wBTC, with a fee of for example 300 (0.03%). For a wBTC amount of 3000 wei or 3 USD, `baseFeeSpecified = 3000 * 300 / 1e6 = 0.9 = 0`, and the user pays 0. If a 0 fee is problematic for any reason, the attacker can specify 6000 wei, and still save 1 wBTC due to rounding down each time.

The attack is economic viable because 1 wei of wBTC = 1 * 100_000 / 1e8 = 0.001 USD, and the gas cost of a single swap, when bundled together to optimize hot storage, is many times smaller. As an example, a sample swap takes a fee of 0.001566 USD, without using the hot storage properties nor fixed costs, and is already close to profitability. If swaps are made in a loop, the gas price of each swap will go down a lot due to not having fixed tx costs and hot storage loading, making the attack viable. In times of lower gas usage, it is even more profitable.

This causes a 100% fee loss for users of the protocol, hence high severity.

Note: it should also round up when converting the baseFeeSpecified to the fee currency, when needed.

Root Cause

DeliHook.sol:231 rounds down instead of up.

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. Attacker swaps in a loop wBTC, paying 1 less wBTC each time due to rounding down, so the lps (FeeProcessor, DailyEpochGauge) receive much less fees.

Impact

A single attacker can cause a significant loss, if all users do this, they would eliminate all lp fees, or very significantly reduce them, and it is profitable.

PoC

See above.

Mitigation

Round up the fee.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.