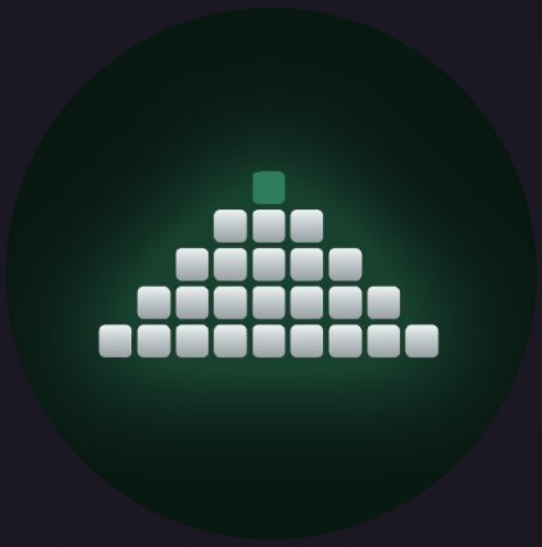
SHERLOCK

Security Review For Delta



Collaborative Audit Prepared For: **Delta**
Lead Security Expert(s): **TessKimy**
vinica_boy
Date Audited: **August 31 - September 4, 2026**

Introduction

This is a pre-deployment revision of DeltaLadderManager, the custodial position manager for Delta Pools. An earlier generation is running in production today.

It includes the following changes and improvements:

- Partial liquidity removal for managed positions (decreaseV3Batch, decreaseV4Batch).
- Three new custody operations: withdrawing a position intact as an NFT, transferring beneficiary rights without moving custody, and an owner-countersigned release for a position whose pool, hook or token blocks the fee harvest.
- Failed ERC20 payouts become claimable credits rather than reverting the exit, and native payouts a recipient rejects are delivered as WETH.
- Distribution of Merkl and other incentive rewards accrued to managed positions, allocated by the owner.
- Removal of six single-position functions, superseded by batch equivalents or by NFT-transfer custody.
- Misc minor improvements and optimizations, including deadlines and per-position hook data on the V4 entrypoints.

Scope

Repository: Scoopz22/delta-sherlock-audit

Audited Commit: 8c0898c4d0d427afd6511ae21c98ea438d6e7d03

Final Commit: 1ff031aecba72d2c038a8bc5d3616169529846c4

Files:

- src/DeltaLadderManager.sol
- src/DeltaPositionBuilder.sol
- src/libraries/PonsLimits.sol

Final Commit Hash

1ff031aecba72d2c038a8bc5d3616169529846c4

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.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	3	8

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: Force-sent ETH permanently bricks Delta-PositionBuilder and disables all V3 ladder minting [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/issues/4

Summary

`mintLadder()` asserts an absolute zero native balance on the way out, and no path spends native ETH the contract did not receive as `msg.value`, so 1 wei force-sent to `DeltaPositionBuilder` reverts every subsequent `mintLadder()` and every `DeltaLadderManager.openV3()`.

Vulnerability Detail

The closing postcondition compares the native balance against zero rather than against the balance the call started with:

```
_requireEmpty(token0);
_requireEmpty(token1);
if (address(this).balance != 0) revert CustodyLeak(address(0),
↪  address(this).balance);
```

`receive()` refuses native ETH from every sender but WETH, which stops ordinary transfers and nothing else. `selfdestruct`, a coinbase payout, and pre-funding the CREATE2 address before deployment all credit the balance without executing code.

Neither funding branch of `mintLadder()` consumes an idle balance. On the ERC-20 path the function never touches native ETH at all. On the native path it wraps exactly `msg.value`, and the refund unwraps and returns `wethRefund`, which is measured from the WETH balance:

```
uint256 wethRefund = token0 == weth ? refund0 : refund1;
if (wethRefund > 0) {
    IWETH(weth).withdraw(wethRefund);
    (bool okRefund,) = msg.sender.call{value: wethRefund}("");
    if (!okRefund) revert EthRefundFailed();
}
```

The donated wei survives both paths, and the builder is stateless with no withdraw, sweep, or rescue function, so the closing check reverts on every call from then on.

Impact

Permanent DoS of all V3 ladder minting, direct and through the manager, for the cost of 1 wei. `DeltaLadderManager.v3Builder` is immutable, so recovery requires redeploying the builder and the manager and migrating existing custody to it.

Code Snippet

https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/9f4925e9cfeb408e9405550afa48927e95dda480/delta-sherlock-audit/src/DeltaPositionBuilder.sol#L134-L136

https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/9f4925e9cfeb408e9405550afa48927e95dda480/delta-sherlock-audit/src/DeltaPositionBuilder.sol#L252-L257

https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/9f4925e9cfeb408e9405550afa48927e95dda480/delta-sherlock-audit/src/DeltaPositionBuilder.sol#L267

Tool Used

Manual Review

Recommendation

Take the pre-call baseline at entry and check the closing balance against it, so a forced balance is tolerated and only a leak created by this call is caught:

```
// at function entry
uint256 ethBase = address(this).balance - msg.value;
// in place of the zero check
if (address(this).balance != ethBase) revert CustodyLeak(address(0),
↪  address(this).balance - ethBase);
```

Issue M-2: Merkl rewards earned by managed positions cannot be claimed by anyone, and reach treasury in full if they ever are [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/issues/8

Summary

Merkl attributes campaign rewards to `posm.ownerOf(tokenId)`, which for a managed position is the manager, and the manager has no path to call the Distributor and no path to split a reward token if one arrives.

Vulnerability Detail

The manager has no claim function and no arbitrary-call surface.

The Merkl Distributor is live on chain 4663 at `0x3Ef3D8bA38EBe18DB133cEc108f4D14CE00Dd9Ae` with a non-zero merkle root, and the Merkl API lists roughly twenty `UNISWAP_V4` campaigns for the chain, each with an empty `forwarders` array, so a contract holding the NFT is not forwarded through to its users. `Distributor.claim` reverts `NotWhitelisted` unless the caller is the user, is `tx.origin`, is a registered operator for that user, or is Merkl governance. The manager can be none of these, and it cannot call `toggleOperator` to become one. Nobody can claim.

If Merkl governance claims on the manager's behalf, the tokens rest as a manager balance and `sweepTokens` sends the full balance of any listed token to `treasury`:

```
uint256 bal = IERC20(token).balanceOf(address(this));
if (bal == 0) continue;
IERC20(token).safeTransfer(treasury, bal);
```

Its NatSpec justifies taking the full balance on the grounds that "a resting balance is the retained `feeBps` cut and NOTHING else", which holds only for tokens the contract's own paths produce. A reward token delivered from outside is not one of those, and there is no `_skim` on this path.

Impact

A managed position earns zero Merkl rewards where a self-custodied position in the same pool earns them in full, and those rewards are unclaimable by the protocol as well. Where governance does claim, the beneficiary's share reaches treasury at 100% rather than at `feeBps`. Users can avoid this only by taking the NFT back with `withdrawV4Batch`, which is the custody model the manager exists to provide.

Code Snippet

https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/8c0898c4d0d427afd6511ae21c98ea438d6e7d03/delta-sherlock-audit/src/DeltaLadderManager.sol#L884-L898

Tool Used

Manual Review

Recommendation

Add a claim entry point that calls `Distributor.claim` for the manager.

Issue M-3: `mintLadder`'s price band is checked against `slot0.tick`, which is too coarse and can be off by one [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/issues/14

Summary

The caller's price band for the ladder is enforced by comparing the pool's reported tick against two integer tick bounds:

```
(, int24 tick,,,,,) = IUniswapV3Pool(pool).slot0();
if (tick < minTick || tick > maxTick) revert PriceOutOfBounds(tick, minTick,
↪  maxTick);
```

Note that `slot0()` returns `sqrtPriceX96` as its first value and the code discards it, taking the derived `tick` instead.

Vulnerability Detail

A tick is a `1.0001×` price step,Because the bounds are `int24` ticks, the narrowest band a caller can express is 1 bps wide, and any price movement inside a tick is invisible to the comparison. A caller who wants a tighter tolerance than 1 bps or who wants a bound that falls partway between two ticks cannot express it through this contract at all.

Uniswap has also edge case scenarios on tick rounding if we're exactly tick spacing boundries. Uniswap uses current tick - 1 for these boundries for the next sell swaps and this situation increases the real difference.

Impact

Minting can be executed on unfavorable prices

Code Snippet

https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/9f4925e9cfeb408e9405550afa48927e95dda480/delta-sherlock-audit/src/DeltaPositionBuilder.sol#L184

Tool Used

Manual Review

Recommendation

Use sqrtPriceX96 instead of tick for price boundries which has exact accuracy

Issue L-1: Replicated Pons launch caps block ladder operations the token itself allows [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/issues/5

Summary

`_pull()` and `_push()` clamp every transfer against the token's `maxTxAmount()`/`maxWalletAmount()`, but a Pons launcher token applies those caps only to buys out of the pair pool, so `mintLadder()` reverts with `ExceedsTokenLimit` on transfers the token would execute.

Vulnerability Detail

`PonsLimits` models the launch parameters as limits on any transfer of the token. `PonsLauncherToken._update` applies them only when the sender is a registered pair pool: it reads `_isPairPool(from)` and returns through `super._update` for everything else, and the cumulative counter `_restrictedPoolBuys[to]` is written on pool buys only. The published launch rules say the same, that buys from the pool are capped for the first two blocks while selling and wallet-to-wallet transfers are never restricted.

No leg of a ladder is a pool buy. The deposit is `transferFrom(msg.sender, builder)`, the mint pays the position manager, and the refund goes from the builder back to the caller. The library refuses two of the three anyway, and `allowance` measures wallet headroom against the recipient's live balance:

```
uint256 allowed = requested < maxTx ? requested : maxTx;

(bool ok, bytes memory data) =
    token.staticcall(abi.encodeWithSelector(bytes4(keccak256("balanceOf(address)"))
    ↪  , recipient));
if (ok && data.length >= 32) {
    uint256 held = abi.decode(data, (uint256));
    uint256 headroom = held >= maxWallet ? 0 : maxWallet - held;
    if (headroom < allowed) allowed = headroom;
}
```

A caller who already holds `maxWalletAmount` therefore has zero headroom, and `refund0`/`refund1` are measured from the builder's balance rather than computed, so a dust remainder is enough to fail the ladder:

```
function _push(address token, uint256 amount) internal {
    if (amount == 0) return;
    uint256 allowed = PonsLimits.allowance(token, msg.sender, amount);
    if (allowed < amount) revert ExceedsTokenLimit(token, amount, allowed);
    IERC20(token).safeTransfer(msg.sender, amount);
```

```
}
```

With the deployed parameters, 5% max wallet and 5.5% max buy on a 1e27 supply, a holder of 6% of supply cannot be refunded 1 wei and cannot mint a ladder at all, and a deposit above 5.5% of supply fails at `_pull` the same way. The creator and the early large holders are the parties most likely to be seeding launch liquidity.

Impact

Ladders on a Pons token revert for callers holding at least `maxWalletAmount` or depositing more than `maxTxAmount`, though the token permits both transfers. Anyone can produce the same revert for every caller by sending the builder more than `maxTxAmount` of the token, since the refund push then carries the donation and exceeds the clamp. `openV3()` reverts with `mintLadder()` in both cases. Restriction windows are two blocks in the current launch configuration, so the outage is bounded by that.

Code Snippet

https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/9f4925e9cfeb408e9405550afa48927e95dda480/delta-sherlock-audit/src/libraries/PonsLimits.sol#L65-L79

https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/9f4925e9cfeb408e9405550afa48927e95dda480/delta-sherlock-audit/src/DeltaPositionBuilder.sol#L312-L329

https://github.com/ponsdotdev/ponsfamily/blob/78b3ff7b8d161732d37b7c05f026490dbe529c3b/contractsV1/src/PonsLauncherToken.sol#L181-L212

Tool Used

Manual Review

Recommendation

Consider removing the `PonsLimits` checks from `_pull` and `_push` and letting the token enforce its own rules. `PonsLauncherToken` reverts by name with `MaxWalletExceeded` and `MaxTxExceeded` on the paths it does restrict, and the builder is never on the receiving side of one.

Issue L-2: PonsLimits misreads three of the Pons launch token's rules [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/issues/6

Summary

`limits()` and `allowance()` read `maxTxAmount()` as a per-transfer cap, map a zero limit to "unlimited", and close the restriction window one block before the token does, so the headroom the library reports is not the headroom `PonsLauncherToken` enforces.

During the review `PonsLimits` was determined to be obsolete and is scheduled for removal, so the library was not searched for bugs beyond the three items reported here and no fix is proposed for them.

Vulnerability Detail

1. `maxTxAmount()` is a cumulative per-recipient budget, not a per-call limit.

`PonsLauncherToken._update` accumulates every restricted pool buy in `_restrictedPoolBuys[to]` and compares the running total, not `value`, against `maxTxLimit()`:

```
uint256 cumulative = _restrictedPoolBuys[to] + value;
uint256 cumulativeLimit = maxTxLimit();
if (cumulative > cumulativeLimit) {
    revert MaxTxExceeded(to, cumulative, cumulativeLimit);
}
_restrictedPoolBuys[to] = cumulative;
```

The counter is written on every restricted buy and never reset, and the getter's own NatSpec calls it a "Cumulative pool-buy cap during the restricted window". `allowance()` spends it once per call:

```
uint256 allowed = requested < maxTx ? requested : maxTx;
```

A recipient that has already exhausted its budget is still reported `maxTx` of headroom, and a recipient that has spent none is clamped per call rather than in total.

2. A zero limit means exactly "nothing may move".

```
// A zero limit means "no limit" in the Pons template, not "nothing may move".
maxTx = okTx && tx_ > 0 ? tx_ : type(uint256).max;
maxWallet = okWallet && wallet_ > 0 ? wallet_ : type(uint256).max;
```

`maxWalletLimit()` returns `(totalSupply() * maxWalletBps) / 10_000` and the token compares against it with a strict >:

```
uint256 walletLimit = maxWalletLimit();
uint256 resultingBalance = balanceOf(to) + value;
if (resultingBalance > walletLimit) {
    revert MaxWalletExceeded(to, resultingBalance, walletLimit);
}
```

A launch configured with `maxWalletBps == 0` therefore reverts on any non-zero restricted buy, and the cumulative `maxTxLimit()` check behaves the same way. There is no constructor bound on either bps value. The library maps the most restrictive configuration the token accepts to `type(uint256).max`, the least restrictive one it can represent.

3. The window closes one block early.

```
(bool okEnd, uint256 endBlock) = _tryUint(token,
↪  bytes4(keccak256("restrictionEndBlock()")));
if (okEnd && endBlock != 0 && block.number >= endBlock) return (false, 0, 0);
```

The token restricts while `block.number <= restrictionEndBlock`. At `block.number == restrictionEndBlock` the library reports the token as unbounded while the token is still enforcing both caps.

Impact

`allowance()` is wrong in both directions: it withholds headroom the token grants, and it reports full headroom for a zero-bps launch and on the final restricted block, where the token refuses every restricted buy. All three are latent under the current deployment, where the token restricts pool buys only and the builder is never the recipient of one.

Code Snippet

https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/9f4925e9cfeb408e9405550afa48927e95dda480/delta-sherlock-audit/src/libraries/PonsLimits.sol#L41-L55

https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/9f4925e9cfeb408e9405550afa48927e95dda480/delta-sherlock-audit/src/libraries/PonsLimits.sol#L65-L79

https://github.com/ponsdotdev/ponsfamily/blob/78b3ff7b8d161732d37b7c05f026490dbe529c3b/contractsV1/src/PonsLauncherToken.sol#L145-L167

https://github.com/ponsdotdev/ponsfamily/blob/78b3ff7b8d161732d37b7c05f026490dbe529c3b/contractsV1/src/PonsLauncherToken.sol#L181-L212

Tool Used

Manual Review

Recommendation

None, the library is being removed.

Issue L-3: A blacklisted beneficiary strands the position's fees for the protocol [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/issues/7

Summary

Every path that touches a managed position forwards to `m.owner` with `safeTransfer`, so a beneficiary blacklisted by one of the position's currencies reverts all of them, including the ones that would have let the protocol take its own cut.

Vulnerability Detail

`_push` transfers directly to the beneficiary with no fallback, and `_pushV4` does the same on each side:

```
function _push(address token, address to, uint256 amount) internal {
    if (amount == 0) return;
    if (token == address(weth)) {
        _payEth(to, amount);
        return;
    }
    IERC20(token).safeTransfer(to, amount);
}
```

`collectV3Batch`, `collectV4Batch`, both close paths and both withdraw paths all run a harvest that ends in one of these, so once `to` is blacklisted none of them can execute. Handing the position to a clean address does not help: `transferBeneficiary` settles fees to the OUTGOING beneficiary before moving the record, so it reverts on the same transfer. `rescueNft` is gated on the record being empty and cannot reach a managed id.

The protocol's own cut is stranded with the user's share. `_skim` leaves `cut` as a contract balance only after `_push` returns, and a revert there unwinds the collect as well, so nothing ever lands for `sweepTokens` to take.

Impact

The beneficiary losing access is expected, and unblacklisting restores it. What is not intended is that the protocol loses the fee it has already earned on that position and every fee it accrues for as long as the blacklist stands, with no path to collect it and the NFT held in a contract that has no way to release it either.

Code Snippet

https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/8c0898c4d0d427afd6511ae21c98ea438d6e7d03/delta-sherlock-audit/src/DeltaLadderManager.sol#L1225-L1232

https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/8c0898c4d0d427afd6511ae21c98ea438d6e7d03/delta-sherlock-audit/src/DeltaLadderManager.sol#L1096-L1108

Tool Used

Manual Review

Recommendation

Consider a permissioned fee-only collect that harvests the position, retains `cut`, and credits the beneficiary's share to a per-address balance they claim later rather than pushing it. The blacklisted transfer then moves out of the harvest and into a claim the beneficiary calls once they are unblacklisted, which unblocks the protocol's cut without giving the owner any reach into user funds.

Making the beneficiary's whole leg pull-based would also fix this class for the close and withdraw paths, at the cost of an extra call on the normal flow.

Issue L-4: A V4 hook that requires hookData on removes permanently locks any position deposited into the manager [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/issues/9

Summary

The manager encodes `bytes("")` as hookData for every V4 action, so a pool whose hook requires non-empty hookData on removes reverts every harvest, and with it every path that could get the position out.

Vulnerability Detail

The last field of each action's parameters is the hookData, and it is a literal at all four sites: mint in `_encodeMintLadder`, increase in `_encodeIncreaseLadder`, burn in `_encodeCloseLadder`, and the zero decrease the harvest runs on:

```
function _encodeCollectFees(Managed memory m, uint256 tokenId) internal view
↪  returns (bytes memory) {
    bytes memory actions = new bytes(2);
    bytes[] memory params = new bytes[](2);
    actions[0] = bytes1(A_DECREASE_LIQUIDITY);
    params[0] = abi.encode(tokenId, uint256(0), uint128(0), uint128(0), bytes(""));
    actions[1] = bytes1(A_TAKE_PAIR);
    params[1] = abi.encode(m.currency0, m.currency1, address(this));
    return abi.encode(actions, params);
}
```

A zero decrease is not exempt from the remove hook. `Hooks.beforeModifyLiquidity` branches on the sign of `liquidityDelta`, and zero takes the remove arm:

```
if (params.liquidityDelta > 0 && self.hasPermission(BEFORE_ADD_LIQUIDITY_FLAG)) {
    self.callHook(abi.encodeCall(IHooks.beforeAddLiquidity, (msg.sender, key,
    ↪  params, hookData)));
} else if (params.liquidityDelta <= 0 &&
↪  self.hasPermission(BEFORE_REMOVE_LIQUIDITY_FLAG)) {
    self.callHook(abi.encodeCall(IHooks.beforeRemoveLiquidity, (msg.sender, key,
    ↪  params, hookData)));
}
```

So a hook that decodes hookData for a referral, a signature or an exit parameter receives an empty `bytes` and reverts. `collectV4Batch` reverts, `closeV4Batch` reverts on the burn, and `withdrawV4Batch` and `transferBeneficiary` revert on the harvest they run

before releasing the position. `rescueNft` is gated on the record being empty and cannot reach a managed id, so the NFT and its principal stay in the manager permanently.

The hook does not need to cooperate for the position to get in. `_recordV4Deposit` reads `getPoolAndPositionInfo` and writes the record, touching no liquidity, so `onERC721Received` accepts a position in any pool without ever calling its hook. A user who minted a working position themselves and deposits it for management is the path in.

Impact

Permanent loss of the deposited position and all principal in it. The user has no exit and the protocol has no rescue. This needs a hook that gates removes on hookData, which is a legitimate design rather than a malicious one, so the pool need not be hostile for the position to be lost.

Code Snippet

https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/8c0898c4d0d427afd6511ae21c98ea438d6e7d03/delta-sherlock-audit/src/DeltaLadderManager.sol#L1371-L1379

https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/8c0898c4d0d427afd6511ae21c98ea438d6e7d03/delta-sherlock-audit/src/DeltaLadderManager.sol#L838-L842

https://github.com/Uniswap/v4-core/blob/46c6834698c48bc4a463a86d8420f4eb1d7f3b75/src/libraries/Hooks.sol#L195-L206

Tool Used

Manual Review

Recommendation

Take hookData as a caller-supplied parameter on the exit paths rather than hardcoding it. `collectV4Batch`, `closeV4Batch` and `withdrawV4Batch` should accept a `bytes` per tokenId and forward it into the encoders, which leaves hookless pools working exactly as they do now while giving a hooked position a way out.

Issue L-5: Overlapping rungs are rejected on V3 but accepted on V4 [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/issues/10

Summary

`DeltaPositionBuilder::_validate()` reverts on any rung whose `tickLower` is below the previous rung's `tickUpper`, so a V3 ladder cannot contain a tight position nested inside a wider one, while `DeltaLadderManager::openV4()` applies no rung validation at all and accepts the same shape.

Vulnerability Detail

The V3 builder enforces strictly ascending, non-overlapping rungs:

```
int24 previousUpper = type(int24).min;
for (uint256 i = 0; i < rungs.length; i++) {
    Rung calldata r = rungs[i];
    if (r.tickLower >= r.tickUpper) revert BadRange(r.tickLower, r.tickUpper);
    if (r.tickLower % spacing != 0) revert RangeNotAligned(r.tickLower, spacing);
    if (r.tickUpper % spacing != 0) revert RangeNotAligned(r.tickUpper, spacing);
    if (i > 0 && r.tickLower < previousUpper) {
        revert RungsOutOfOrder(i, previousUpper, r.tickLower);
    }
    if (r.amount0 == 0 && r.amount1 == 0) revert EmptyRung(i);
    previousUpper = r.tickUpper;
```

The stated reason is that overlapping rungs "would compete for the same range and the totals would no longer describe distinct positions". Neither holds. Each rung is minted as its own `NonfungiblePositionManager::mint()` call and becomes its own NFT, and `total0` /`total1` are plain sums of `amount0`/`amount1` that stay correct whatever ranges the rungs cover. Overlap changes nothing about how the position manager, the pool, or the refund accounting behaves.

The shape being excluded is the standard one for concentrated liquidity: a narrow position around spot that earns most of the fees, plus a wide position that keeps the strategy in range when price moves, both centred on the same tick. A user who wants it on V3 must open the tight rung and the wide rung in two separate `openV3()` calls, paying the pull, approval and refund overhead twice, and cannot express it as one ladder at all.

`openV4()` has no equivalent of `_validate()`. It checks `rungs.length != 0` and then forwards every rung verbatim into `MINT_POSITION` actions:

```
for (uint256 i = 0; i < n; i++) {
    actions[k] = bytes1(A_MINT_POSITION);
```

```
    params[k] = abi.encode(
        key, rungs[i].tickLower, rungs[i].tickUpper, rungs[i].liquidity,
        ↪  rungs[i].amount0Max, rungs[i].amount1Max, recipient, bytes("")
    );
    k++;
}
```

Ordering and overlap are never inspected, and inverted or misaligned ranges are left to `Pool::checkTicks()` and the tick-spacing check inside the `PoolManager`, so a malformed V4 ladder is rejected only after the pull and Permit2 arming, with the pool's error rather than the builder's named revert. This is safe. It is also the opposite policy from V3 for the same product, and the deciding rule (”rungs must ascend and not overlap”) lives in a deployed, ownerless contract that the manager cannot amend.

Impact

V3 ladders cannot express the common tight-plus-wide layout in one transaction. V4 ladders can, so the two open paths accept different inputs for the same product, and V4 surfaces malformed rungs as opaque pool reverts after funds have been pulled.

Code Snippet

https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/8c0898c4d0d427afd6511ae21c98ea438d6e7d03/delta-sherlock-audit/src/DeltaPositionBuilder.sol#L288-L298 https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/8c0898c4d0d427afd6511ae21c98ea438d6e7d03/delta-sherlock-audit/src/DeltaLadderManager.sol#L644-L651 https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/8c0898c4d0d427afd6511ae21c98ea438d6e7d03/delta-sherlock-audit/src/DeltaLadderManager.sol#L1346-L1352

Tool Used

Manual Review

Recommendation

Pick one policy for both paths. If overlap is allowed, drop the `RungsOutOfOrder` check in the next builder generation and keep V4 as is. If it is not, add a V4 `_validate()` before the pull that mirrors the builder's ordering, alignment and empty-rung checks, so both paths fail the same way and before value moves.

Issue L-6: Exit paths hardcode `deadline: block.timestamp`, leaving collects and closes unbounded in time [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/issues/11

Summary

Every exit path passes the current block's timestamp as its own deadline, which makes the deadline check a tautology.

Vulnerability Detail

In `_closeV3`, the principal leg is submitted as:

```
v3Npm.decreaseLiquidity(
    INonfungiblePositionManager.DecreaseLiquidityParams({
        tokenId: tokenId,
        liquidity: liq,
        amount0Min: amount0Min,
        amount1Min: amount1Min,
        deadline: block.timestamp
    })
);
```

The NPM validates `require(block.timestamp <= params.deadline)`. Because the argument is `block.timestamp`, evaluated in the same transaction against the same block, the comparison holds in every block the transaction could ever land in.

Impact

A close or collect transaction carries no time bound. Submitted with a low priority fee, it can sit in the mempool and execute at an arbitrarily later block, tearing down the position at whatever price prevails then rather than at the price the user saw when they signed.

Code Snippet

https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/9f4925e9cfeb408e9405550afa48927e95dda480/delta-sherlock-audit/src/DeltaLadderManager.sol#L543-L552

Tool Used

Manual Review

Recommendation

Thread a caller-supplied deadline through the exit paths the same way `openV3` and `topUpV4` already do.

Issue L-7: Native-ETH handling is asymmetric between `openV3` and `openV4` [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/issues/12

Summary

The two open paths disagree about whether a WETH-denominated pool may be funded with native ETH.

V3 accepts it. `openV3` treats `msg.value > 0` as a native open, pulls only the non-WETH side from the caller, and forwards the ETH to the builder. V4 refuses it. `openV4` recognises exactly one native shape `currency0 == address(0)`.

Vulnerability Detail

V4 rejects the tx:

```
bool native = key.currency0 == address(0);
if (!native && msg.value > 0) revert ZeroAddress(); // ETH sent to a non-native V4
↪  pool
```

V3 accepts the tx and convert ETH to WETH:

```
bool paidEth = msg.value > 0;
if (paidEth) {
    address wethAddr = address(weth);
    if (token0 == wethAddr) _pull(token1, total1);
    else if (token1 == wethAddr) _pull(token0, total0);
    else revert ZeroAddress(); // ETH sent to a non-WETH pool
}
```

Impact

In V3, users can mint position with native ETH by automatic WETH conversion but this feature does not exist in V4. V4 rejects tx if `currency0` is not equal to `address(0)`

Code Snippet

https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/9f4925e9cfeb408e9405550afa48927e95dda480/delta-sherlock-audit/src/DeltaLadderManager.sol#L658

Tool Used

Manual Review

Recommendation

Apply automatic WETH conversion to V4 side for WETH pools

Issue L-8: `_skim` rounds the protocol cut down to zero on small collections [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/issues/13

Summary

`Math.mulDiv` without a rounding argument rounds toward zero. Whenever `amount * feeBps < BPS`, the cut is exactly `0` and the beneficiary keeps the entire collection.

Vulnerability Detail

The protocol's cut is computed with a truncating division:

```
function _skim(uint256 amount) internal view returns (uint256 toUser, uint256 cut) {
    cut = Math.mulDiv(amount, feeBps, BPS); // @audit rounds down
    toUser = amount - cut;
}
```

Impact

If the collected amount is dust, cut rounds down to 0 and causes protocol may not collect fee.

Code Snippet

https://github.com/sherlock-audit2/2026-08-delta-august-31st-2026/blob/9f4925e9cfeb408e9405550afa48927e95dda480/delta-sherlock-audit/src/DeltaLadderManager.sol#L1116-L1119

Tool Used

Manual Review

Recommendation

Use round-up for protocol fee calculation

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.