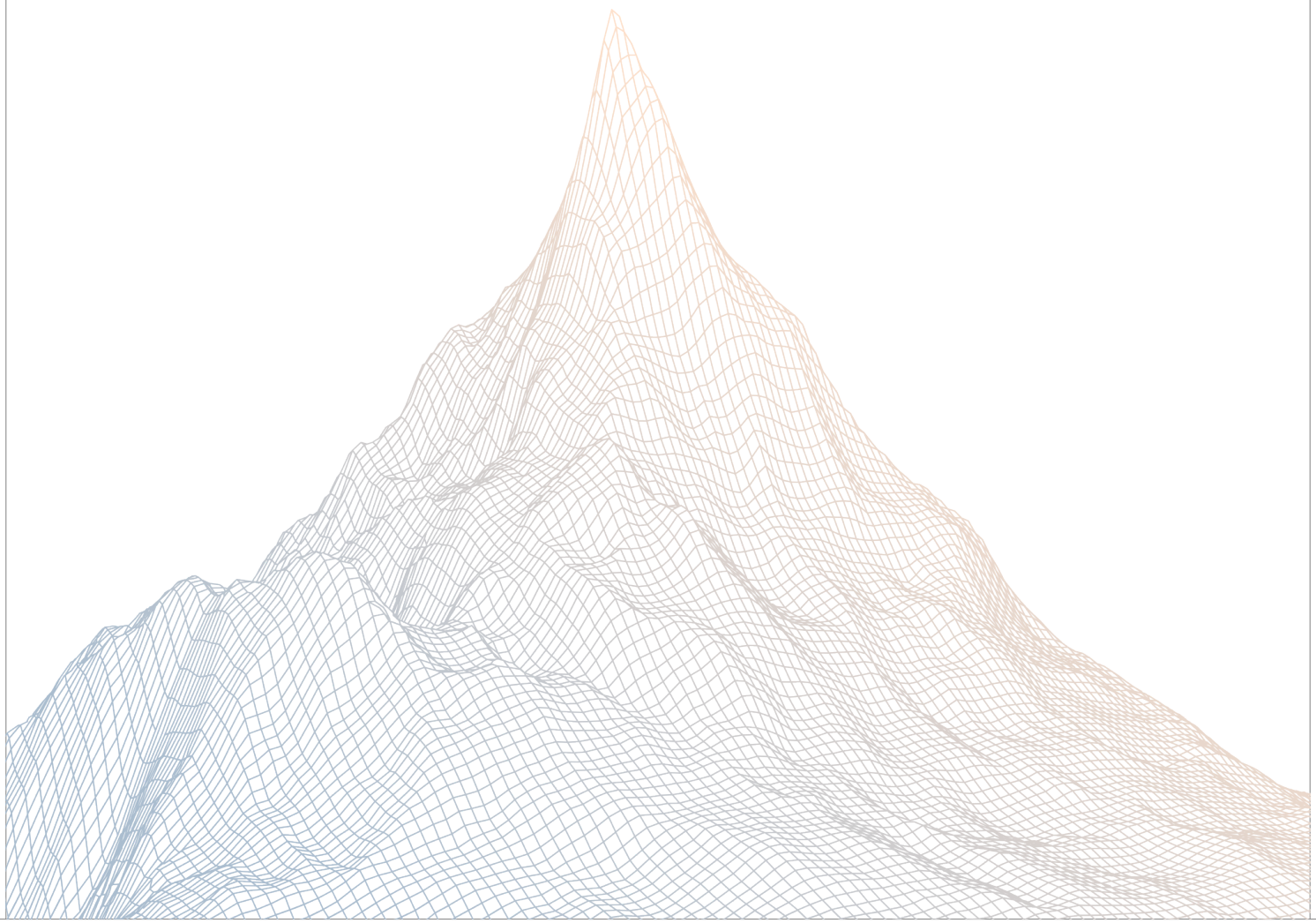
Zenith

Monday.trade

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: November 17th to November 28th, 2025

AUDITED BY: Matte
ether_sky

Contents

1

Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Monday.Trade

Monday Trade is Monad's native spot and perps DEX that offers the best of CEX and DEX trading experience. Monad's low latency enables Monday Trade to execute trades within milliseconds, ensuring traders can make the most of market moves, without giving up their asset ownership to centralized exchanges.

Monday Trade combines the precision of a fully on-chain order book with the simplicity of AMMs into one sleek UI, delivering gas-efficient, high-performance onchain trades with optional advanced trading tools.

2.2 Scope

The engagement involved a review of the following targets:

Target	MondayTrade-Spot
Repository	https://github.com/SynFutures/MondayTrade-Spot
Commit Hash	`6bcd55688c65452bb74c7493c6c677d38ccac192`
Files	`contracts/**/*.sol`

2.3 Audit Timeline

November 17, 2025	Audit start
November 28, 2025	Audit end
December 10, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	2
Medium Risk	5
Low Risk	2
Informational	6
Total Issues	**15**

3

Findings Summary

ID	Description	Status
H-1	Unvalidated size sign in cancel function leads to misapplied fees and attacker profitability	Resolved
H-2	The innerSize is incorrectly reset during nonce synchronization in the place function	Resolved
M-1	The totalTaken can be manipulated to zero or reduced, enabling protocol loss	Resolved
M-2	Users can lose funds in inner order fill when amountIn is 0	Resolved
M-3	The swapCrossRange can skip feeGrowth and liquidity updates	Resolved
M-4	The overflow innerBitmap can happen in the flipTickWithBitmap when tickSpacing is 256	Resolved
M-5	Using Rounding-Up taker fee per order causes protocol fee over-accounting	Resolved
L-1	Changing config allows altering fee parameters, causing inconsistent accounting for previously filled orders	Resolved
L-2	Fee parameter change causes incorrect accounting for existing orders	Acknowledged
I-1	Unused parameter (initialized) in the _stepToNextTick function	Resolved
I-2	Unused variable assignment	Resolved
I-3	Mismatch between comment and code in enableFeeAmount	Resolved
I-4	Tick.initialized comment is inaccurate - should include limit orders	Resolved
I-5	SwapRouter does not support batchPlace	Resolved
I-6	Redundant condition in crossRangeTickOrder	Resolved

4

Findings

4.1 High Risk

A total of 2 high risk findings were identified.

[H-1] Unvalidated size sign in cancel function leads to misapplied fees and attacker profitability

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- OrderHandler.sol

Description:

In the `cancel` function, the `size` parameter determines whether the `order` is a `token0 order` (`size > 0`) or a `token1 order` (`size < 0`).

- OrderHandler.sol#L151

```
function cancel(
    address user,
    int128 size,
    int24 targetTick,
    uint32 deadline,
    bool needNativeToken
) public override {
    ...
    // Check have order
▷   bool isToken0 = size > 0 ? true : false;
    if (
        (isToken0 && size > userRecordMap[targetTick][user].totalPlaceSize)
    ||
        (!isToken0 && size < userRecordMap[targetTick][user].totalPlaceSize)
    ) {
        size = userRecordMap[targetTick][user].totalPlaceSize;
    }
    ...
    (uint128 amount0, uint128 amount1) =
```

```
        Broker.cancel(
            ticks,
            tickBitmap,
            targetTick,
            tickSpacing,
            nonce,
            size > 0 ? uint128(size) : uint128(-size),
            size > 0 ? true : false
        );

    // update user record
    if (isToken0) {
        ...
    } else {
        ...
    }
}
```

However, the function never validates that the sign of the user-provided `size` matches the actual stored `order direction` (`userRecordMap[targetTick][user].totalPlaceSize`). This allows a malicious user to intentionally pass a `size` with the wrong `sign`.

Although the `contract` later corrects the `size` here:

- OrderHandler.sol#L156

```
if (
    (isToken0 && size > userRecordMap[targetTick][user].totalPlaceSize) ||
    (!isToken0 && size < userRecordMap[targetTick][user].totalPlaceSize)
) {
    size = userRecordMap[targetTick][user].totalPlaceSize;
}
```

the variable `isToken0` is _never updated_ and remains based on the user-provided malicious sign, not the corrected size.

When the cancel operation proceeds:

- OrderHandler.sol#L171

```
(uint128 amount0, uint128 amount1) =
    Broker.cancel(
        ticks,
        tickBitmap,
        targetTick,
        tickSpacing,
```

```
        nonce,
        size > 0 ? uint128(size) : uint128(-size),
        size > 0 ? true : false
    );
```

the broker returns correct token amounts, but the `fee logic` later still relies on the original, incorrect `isToken0`:

- OrderHandler.sol#L184

```
if (isToken0) {
    ...
    amount1 = uint128(_updateProtocolFees(amount1, true));
} else {
    ...
    amount0 = uint128(_updateProtocolFees(amount0, false));
}
```

Meaning:

- If `isToken0 = true`, the contract applies `maker fee` on `amount1`
- If `isToken0 = false`, the contract applies `maker fee` on `amount0`

If an attacker flips the `sign` intentionally, the `fee` is applied to the wrong token.

Attacker Profits because `maker fees` can be positive or negative, this mis-classification allows the `attacker` to choose the fee direction.

- If `makerFee` is `positive` (charging `maker`), `attacker` wants the `fee` applied to the smaller token, minimizing `fee` paid. They invert `isToken0` so `fee` is charged on the opposite token.
- If `makerFee` is negative (rebate to `maker`), `attacker` wants the `fee` applied to the larger token, increasing reward. Again they invert `isToken0`.

This can be exploited whenever `partial fills` occur, because the two returned amounts (`amount0`, `amount1`) differ in size.

Recommendations:

Add validation of `size` parameter.

```
function cancel(
    address user,
    int128 size,
    int24 targetTick,
    uint32 deadline,
```

```
    bool needNativeToken
) public override {
    // Check user
    require(msg.sender == user, "User Wrong");
    // Check deadline
    require(block.timestamp <= deadline, "Order expired");
    require(targetTick % int128(orderTickSpacing) == 0, "Illegal Order
    Tick");
    require(
        int256(size) *
            int256(userRecordMap[targetTick][user].totalPlaceSize) >= 0,
        "Illegal size"
    );

    ...
}
```

Monday.trade: Resolved with @1e584ff6cd ...

Zenith: Verified.

[H-2] The innerSize is incorrectly reset during nonce synchronization in the place function

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- Broker.sol

Description:

When placing an `order`, the `contract` performs `nonce synchronization` with neighboring `range ticks`:

- Broker.sol#L203

```
function place(
    mapping(int24 ⇒ Tick.Info) storage ticks,
    mapping(int16 ⇒ uint256) storage tickBitmap,
    int24 tick,
    int24 slot0Tick,
    int24 tickSpacing,
    uint128 amount,
    bool isZero // place token0
) public returns (UserTmpRecord memory tmpRecord) {
    ...
    // 3. clear innerSize and update tick
    {
        int24 nonce0UpdateRangeTick = LibOrder.calNextRangeTick(tick, true,
    tickSpacing);
        int24 nonce1UpdateRangeTick = LibOrder.calNextRangeTick(tick, false,
    tickSpacing);
▷       if ((ticks[nonce0UpdateRangeTick].orderRecord.totalNonce0 >
    ticks[tick].orderRecord.nonce0)) {
            ticks[tick].orderRecord.innerSize = 0;
            ticks[tick].orderRecord.partialFilledSize = 0;
            ticks[tick].orderRecord.nonce0
    = ticks[nonce0UpdateRangeTick].orderRecord.totalNonce0;
        }
```

```
▷       if (ticks[nonce1UpdateRangeTick].orderRecord.totalNonce1 >
    ticks[tick].orderRecord.nonce1) {
            ticks[tick].orderRecord.innerSize = 0;
            ticks[tick].orderRecord.partialFilledSize = 0;
            ticks[tick].orderRecord.nonce1
    = ticks[nonce1UpdateRangeTick].orderRecord.totalNonce1;
        }
    }
```

The logic unconditionally resets `innerSize` to 0, regardless of whether the current `order` is `token0` or `token1` (`isZero`). This wipe happens even when the current `tick` already has a valid `order`.

Scenario

1. `Current tick = 80, tickSpacing = 60, range tick = 0, 60, 120, ..., order tick spacing = 10`
2. User places a `token1 order` at `tick 70`
3. User places `token0 order` at `tick 100`, size = `1 ETH`
4. A `swap` moves the `price tick` from `80` → `50`, crossing `range tick 60`.
5. User places another `token0 order` at `tick 100`, size = `1 ETH`
   This destroys the **previous 1 ETH** `token0 order` at `tick 100`.
   `innerSize` now reflects only the last `order`, not the accumulated total.

This breaks core `order` accounting, causing real fund impact and incorrect matching behavior.

Recommendations:

```
function place(
    mapping(int24 ⇒ Tick.Info) storage ticks,
    mapping(int16 ⇒ uint256) storage tickBitmap,
    int24 tick,
    int24 slot0Tick,
    int24 tickSpacing,
    uint128 amount,
    bool isZero // place token0
) public returns (UserTmpRecord memory tmpRecord) {
    ...
    // 3. clear innerSize and update tick
    {
        int24 nonce0UpdateRangeTick = LibOrder.calNextRangeTick(tick, true,
```

```
    tickSpacing);
        int24 nonce1UpdateRangeTick = LibOrder.calNextRangeTick(tick, false,
    tickSpacing);
        if ((ticks[nonce0UpdateRangeTick].orderRecord.totalNonce0 >
    ticks[tick].orderRecord.nonce0)) {
            if (isZero) {
                ticks[tick].orderRecord.innerSize = 0;
                ticks[tick].orderRecord.partialFilledSize = 0;
            }
            ticks[tick].orderRecord.innerSize = 0;
            ticks[tick].orderRecord.partialFilledSize = 0;
             ticks[tick].orderRecord.nonce0
    = ticks[nonce0UpdateRangeTick].orderRecord.totalNonce0;
        }
        if (ticks[nonce1UpdateRangeTick].orderRecord.totalNonce1 >
    ticks[tick].orderRecord.nonce1) {
            if (!isZero) {
                ticks[tick].orderRecord.innerSize = 0;
                ticks[tick].orderRecord.partialFilledSize = 0;
            }
            ticks[tick].orderRecord.innerSize = 0;
            ticks[tick].orderRecord.partialFilledSize = 0;
             ticks[tick].orderRecord.nonce1
    = ticks[nonce1UpdateRangeTick].orderRecord.totalNonce1;
        }
    }
```

PoC

```
function testNonceUpdateScenario() public {
    // Swap to move price to tick 80
    token1.mint(address(this), 100 ether);
    int256 amount0Out = -1 ether;
    IUniswapV3Pool(address(pool)).swap(
        address(this),
        false,
        amount0Out,
        TestHelper.getSqrtPriceLimitFromTick(80),
        ''
    );

    // Verify we're at tick 80 (or close)
```

```
    (, int24 currentTick, , , , , ) = pool.slot0();
    console.log('Initial tick:', int256(currentTick));

    // Step 2: Place orders at tick 70 and 100
    IUniswapV3Pool(address(pool)).place(
        address(this),
        -int128(1 ether), // token1 order at tick 70
        70,
        uint32(block.timestamp + 1 hours),
        ''
    );

    IUniswapV3Pool(address(pool)).place(
        address(this),
        int128(1 ether), // token0 order at tick 100
        100,
        uint32(block.timestamp + 1 hours),
        ''
    );

    (, , , , , , , Tick.OrderRecord memory orderRecord) = pool.ticks(100);
    console.log('InnerSize at tick 100 after first place:',
    int256(orderRecord.innerSize));

    // Step 3: Swap token0→token1 to move price to tick 50 (with limitPrice
    of 50)
    token0.mint(address(this), 100 ether);
    int256 amount1Out = -2 ether;
    IUniswapV3Pool(address(pool)).swap(
        address(this),
        true, // zeroForOne = true, token0 → token1 (price moves down)
        amount1Out,
        TestHelper.getSqrtPriceLimitFromTick(50), // Limit price to tick 50
        ''
    );

    // Step 4: Place order at tick 100 again
    IUniswapV3Pool(address(pool)).place(
        address(this),
        int128(1 ether), // token0 order at tick 100
        100,
        uint32(block.timestamp + 1 hours),
        ''
    );

    (, , , , , , , orderRecord) = pool.ticks(100);
    console.log('InnerSize at tick 100 after second place:',
```

```
    int256(orderRecord.innerSize));
}
```

```
Logs:
  Initial tick: 80
  InnerSize at tick 100 after first place: 1000000000000000000
  InnerSize at tick 100 after second place: 1000000000000000000
```

Monday.trade: Resolved with @c68f903195 ...

Zenith: Verified.

4.2 Medium Risk

A total of 5 medium risk findings were identified.

[M-1] The totalTaken can be manipulated to zero or reduced, enabling protocol loss

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- Broker.sol

Description:

The `totalTaken` for a `range-tick order` is accumulated using a `round-down` calculation when placing and canceling `orders`.

- Broker.sol#L43

```
function _placeSingleTickOrder(
    mapping(int24 => Tick.Info) storage ticks,
    int24 tick,
    uint128 amount,
    bool isZero, // place token0 or not
    bool isAdd // place or cancel
)
    internal
    returns (
        uint128 deltaAmount0,
        uint128 deltaAmount1,
        bool flipped
    )
{
    Tick.Info storage tickInfo = ticks[tick];

    if (isZero) {
        // place token0
```

```
▷       uint128 token1Amount = LibOrder._calAmountAtCurrentTick(tick,
    amount, true);
        ...
```

- Broker.sol#L141

```
function _updateRangeTick(
    mapping(int24 ⇒ Tick.Info) storage ticks,
    uint128 deltaAmount0,
    uint128 deltaAmount1,
    int24 tick,
    int24 nextTick,
    int24 tickSpacing,
    bool isZero,
    bool isAdd,
    bool flipped
) internal returns (bool rangeFlipped) {
    Tick.Info storage tickInfo = ticks[nextTick];
    if (isZero) {
        // place token0
        require(tickInfo.orderRecord.totalSize >= 0, 'size side');
        uint128 size0Before = uint128(tickInfo.orderRecord.totalSize);
        if (isAdd) {
            tickInfo.orderRecord.totalSize += int128(deltaAmount0);
▷           tickInfo.orderRecord.totalTaken += deltaAmount1;
        } else {
            require(tickInfo.orderRecord.totalSize >= int128(deltaAmount0),
    'insufficient liquidity');
            tickInfo.orderRecord.totalSize -= int128(deltaAmount0);
▷           tickInfo.orderRecord.totalTaken
    = tickInfo.orderRecord.totalTaken >= deltaAmount1
                ? tickInfo.orderRecord.totalTaken - deltaAmount1
                : 0;
        }
```

Because rounding down loses precision, a malicious user can place many small `token0` `orders` in the same `tick` (e.g., 100 amounts × 100 tokens each = 10,000 total size).

Then the attacker cancels them `all at once`, but cancels slightly less than the full size (e.g., cancels 9,901 out of 10,000). This leaves:

- `totalSize = 99`
- `totalTaken = 0` (due to cumulative rounding-down losses)

During the next swap, the range-order handler uses:

- LibOrder.sol#L193

```
▷       amountIn = uint256(ticks[tick].orderRecord.totalTaken);  // = 0
        feeAmount = FullMath.mulDivRoundingUp(amountIn, orderFee, 1e6);
        amountOut = zeroForOne
            ? uint256(-ticks[tick].orderRecord.totalSize)
            : uint256(ticks[tick].orderRecord.totalSize);        // = 99
```

Meaning:

- The attacker supplies `0 token1`
- And receives `99 token0 for free`

After `swap`:

- The `order owner` withdraws `token1` corresponding to `99 token0`
- `LPs lose funds; order owner loses nothing`

This exploit works even when other users' `orders` exist at the same `tick` - `attacker` repeatedly places many micro-orders and cancels them in large batches to push `totalTaken` down, causing systematic protocol loss.

Recommendations:

The rule for calculating `totalTaken` should be updated so that taken is always computed using a value derived from the `post-operation innerSize`, using `_calAmountAtTickRoundingUp()`.

PoC

```
function testCancelPartialTotalTaken() public {
    // Swap to move price to tick -24083
    token1.mint(address(this), 1 ether);
    int256 amount0Out = 5 ether;
    IUniswapV3Pool(address(pool)).swap(
        address(this),
        true, // zeroForOne = false, token1 → token0
        amount0Out,
        TestHelper.getSqrtPriceLimitFromTick(-24083),
        ''
    );
    // Get current tick
    (, int24 currentTick, , , , , ) = pool.slot0();
    console.log('Current tick:', currentTick);
    // Place token0 orders at tick 60 (100 times, 100 amounts each = 10000
    total)
```

```
int24 orderTick = -24081;
uint128 orderAmount = 100; // 100 amounts per order
uint256 numOrders = 100; // 100 orders

// Mint enough token0 for all orders
token0.mint(address(this), 1 ether);

// Place orders multiple times
for (uint256 i = 0; i < numOrders; i++) {
    IUniswapV3Pool(address(pool)).place(
        address(this),
        int128(orderAmount), // token0 order
        orderTick,
        uint32(block.timestamp + 1 hours),
        ''
    );
}

// Verify totalSize after placing all orders
(, , , , , , , , Tick.OrderRecord memory orderRecord)
= pool.ticks(-24060);
console.log('TotalSize after placing orders:',
uint256(orderRecord.totalSize));
console.log('TotalTaken after placing orders:',
uint256(orderRecord.totalTaken));

// Cancel 8900 amounts at once
uint128 cancelAmount = 8900;
IUniswapV3Pool(address(pool)).cancel(
    address(this),
    int128(cancelAmount), // cancel token0 order
    orderTick,
    uint32(block.timestamp + 1 hours),
    false // needNativeToken = false
);

// Check final state
(, , , , , , , orderRecord) = pool.ticks(-24060);
console.log('TotalSize after cancel:', uint256(orderRecord.totalSize));
console.log('TotalTaken after cancel:',
uint256(orderRecord.totalTaken));

// User swaps to consume token0 orders
(int256 swapAmount0, int256 swapAmount1)
= IUniswapV3Pool(address(pool)).swap(
    address(this),
    false, // zeroForOne = false, token1 → token0
```

```
        amount0Out,
        TestHelper.getSqrtPriceLimitFromTick(-24000), // Move price up to
    fill orders at -24081
        ''
    );

    console.log('User received token0 amount:', -swapAmount0);
    console.log('User consumed token1 amount:', swapAmount1);
}
```

```
Logs:
  Current tick: -24083
  TotalSize after placing orders: 10000
  TotalTaken after placing orders: 800
  TotalSize after cancel: 1100
  TotalTaken after cancel: 0
  User received token0 amount: 1100
  User consumed token1 amount: 0
```

Monday.trade: Resolved with @4f5254e5c8 ...

Zenith: Verified.

[M-2] Users can lose funds in inner order fill when amountIn is 0

SEVERITY: Medium	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Low

Target

- SwapHandler.sol

Description:

When a `swap` cannot fully consume `liquidity` and `orders`, it enters the inner `while` loop to fill `orders` step by step.

For each `order tick`, the `swap` calls `LibOrder.fillSingleTickOrder()` to fill the `order` and there is a special-case handling:

- SwapHandler.sol#L430

```
// fill order tick
if (step.initialized) {
    // amountIn and amountOut didn't include fee
    (step.amountIn, step.amountOut, step.feeAmount, filled)
    = LibOrder.fillSingleTickOrder(
        ticks,
        step.tickNext,
        state.amountSpecifiedRemaining,
        zeroForOne,
        //feeParams.orderFee
        IConfig(IUniswapV3Factory(factory).
            config()).getOrderTakerFee(address(this))
    );
    // for a special case, in totalTaken, restAmount didn't filled.
    // But in singleTickFilled, restAmount filled and have rest
    // rest amount as fee
▷   if(step.amountIn == 0 && exactInput) step.feeAmount
    = uint256(state.amountSpecifiedRemaining);
```

This is intended to handle situations where no input is consumed for a step.

However, this creates a vulnerability:

- The order's `innerSize` can become extremely small (e.g., `1 wei`) - either because a user intentionally `places` a tiny `order`, or because partial filling during previous `swaps` reduces it to `1 wei`.
- During a later `swap`, this results in `amountOut` being extremely small (e.g., `1 wei`) while `amountIn = 0`.

The logic above assigns the entire remaining `swap` amount to `feeAmount`, regardless of the actual `order` size. As a result, the user receives far less than expected, while the protocol records an excessive `fee`. This is not just an accounting error - it can cause significant loss to users.

Recommendations:

There are two possible fixes:

- Always round up `amountIn` during calculation so that it never becomes `0`, or
- Remove this special-case branch and compute fees strictly based on the actual filled amount.

PoC

```
function testAmountInZero() public {
    // Swap to move price to tick 120
    token1.mint(address(this), 100 ether);
    int256 amount0Out = -1 ether; // Swap token1 → token0 to move price up
    IUniswapV3Pool(address(pool)).swap(
        address(this),
        false, // zeroForOne = false, token1 → token0
        amount0Out,
        TestHelper.getSqrtPriceLimitFromTick(120),
        ''
    );

    // Verify we're at tick 120
    (, int24 currentTick, , , , , ) = pool.slot0();
    console.log('Initial tick:', int256(currentTick));

    IUniswapV3Pool(address(pool)).place(
        address(this),
        int128(-1), // 1 wei token1 order
        100,
        uint32(block.timestamp + 1 hours),
        ''
    );
```

```
    IUniswapV3Pool(address(pool)).place(
        address(this),
        -int128(1 ether), // 1 ether token1 order at range tick 60
        60,
        uint32(block.timestamp + 1 hours),
        ''
    );

    (, , , , , , , , Tick.OrderRecord memory orderRecord) = pool.ticks(100);
    console.log('Order size at tick 100 before swap:',
    int256(orderRecord.innerSize));
    (, , , , , , , , orderRecord) = pool.ticks(60);
    console.log('Order size at tick 60 before swap:',
    int256(orderRecord.innerSize));

    // Perform swap (exactInput = true)
    // Swap token0 → token1 to fill the token1 order
    token0.mint(address(this), 100 ether);
    int256 amount0In = 0.5 ether;

    (int256 amount0, int256 amount1) = IUniswapV3Pool(address(pool)).swap(
        address(this),
        true, // zeroForOne = true, token0 → token1
        amount0In,
        TestHelper.getSqrtPriceLimitFromTick(0),
        ''
    );

    (, , , , , , , , orderRecord) = pool.ticks(100);
    console.log('Order size at tick 100 after swap:',
    int256(orderRecord.innerSize));
    (, , , , , , , , orderRecord) = pool.ticks(60);
    console.log('Order size at tick 60 after swap:',
    int256(orderRecord.innerSize));

    console.log('User consumed token0 amount:', amount0);
    console.log('User received token1 amount:', -amount1);
}
```

```
Logs:
  Initial tick: 120
  Order size at tick 100 before swap: -1
  Order size at tick 60 before swap: -1000000000000000000
  Order size at tick 100 after swap: 0
  Order size at tick 60 after swap: -1000000000000000000
  User consumed token0 amount: 500000000000000000
```

```
User received token1 amount: 1
```

Monday.trade: Resolved with @4f5254e5c8 ...

Zenith: Verified.

[M-3] The swapCrossRange can skip feeGrowth and liquidity updates

SEVERITY: Medium	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Low

Target

- SwapHandler.sol

Description:

In `_swapCrossRange()`, when swapping `token1` → `token0`:

- SwapHandler.sol#L206

```
function _swapCrossRange(
    bool exactInput,
    bool zeroForOne,
    uint160 sqrtPriceLimitX96,
    SwapState memory state,
    SwapCache memory cache,
    Slot0 memory slot0Start
) private returns (SwapState memory, SwapCache memory) {
    bool traversalOrder = true;
    StepComputations memory step;
    // continue swapping as long as we haven't used the entire input/output
    and haven't reached the price limit
    while (state.amountSpecifiedRemaining ≠ 0 && state.sqrtPriceX96 ≠
    sqrtPriceLimitX96) {

        step.sqrtPriceStartX96 = state.sqrtPriceX96;

        (step.tickNext, step.initialized)
    = tickBitmap.nextInitializedTickWithinOneWord(
            state.tick,
            tickSpacing,
            zeroForOne
        );
        // if slot0.tick is rangeTick, and partialFilled, and 1→0
```

```
▷       if(state.tick % tickSpacing == 0 && !zeroForOne &&
    ticks[state.tick].orderRecord.totalSize > 0) {
            step.tickNext = state.tick;
        }
```

This resets `tickNext` to the `current range tick` if it is `range tick` and has `pending orders`, to avoid bypassing the `tick`.

But `step.initialized` is not updated in this case.

If `step.initialized` was previously `false`, the subsequent `tick-crossing` logic will only execute `liquidity` and `fee` updates when `step.initialized` is `true`.

- SwapHandler.sol#L307

```
// if the tick is initialized, run the tick transition
if (step.initialized) {
    // check for the placeholder value, which we replace with the actual
    value the first time the swap
    // crosses an initialized tick
    if (!cache.computedLatestObservation) {
        (cache.tickCumulative, cache.secondsPerLiquidityCumulativeX128)
    = observations
            .observeSingle(
            cache.blockTimestamp,
            0,
            slot0Start.tick,
            slot0Start.observationIndex,
            cache.liquidityStart,
            slot0Start.observationCardinality
        );
        cache.computedLatestObservation = true;
    }
    int128 liquidityNet =
        ticks.cross(
            step.tickNext,
            (zeroForOne ? state.feeGrowthGlobalX128 : feeGrowthGlobal0X128),
            (zeroForOne ? feeGrowthGlobal1X128 : state.feeGrowthGlobalX128),
            cache.secondsPerLiquidityCumulativeX128,
            cache.tickCumulative,
            cache.blockTimestamp
        );
    // if we're moving leftward, we interpret liquidityNet as the opposite
    sign
    // safe because liquidityNet cannot be type(int128).min
    if (zeroForOne) liquidityNet = -liquidityNet;
```

```
    state.liquidity = LiquidityMath.addDelta(state.liquidity, liquidityNet);
}
```

This leads to incorrect `state` and `missing fee` accounting, even though the `tick` had `valid orders` and non-zero `liquidity`.

Recommendations:

When resetting `tickNext` to the current tick due to pending orders, also set `initialized`.

```
function _swapCrossRange(
    bool exactInput,
    bool zeroForOne,
    uint160 sqrtPriceLimitX96,
    SwapState memory state,
    SwapCache memory cache,
    Slot0 memory slot0Start
) private returns (SwapState memory, SwapCache memory) {
    bool traversalOrder = true;
    StepComputations memory step;
    // continue swapping as long as we haven't used the entire input/output
    and haven't reached the price limit
    while (state.amountSpecifiedRemaining ≠ 0 && state.sqrtPriceX96 ≠
    sqrtPriceLimitX96) {

        step.sqrtPriceStartX96 = state.sqrtPriceX96;

        (step.tickNext, step.initialized)
    = tickBitmap.nextInitializedTickWithinOneWord(
            state.tick,
            tickSpacing,
            zeroForOne
        );
        // if slot0.tick is rangeTick, and partialFilled, and 1→0
        if(state.tick % tickSpacing == 0 && !zeroForOne &&
    ticks[state.tick].orderRecord.totalSize > 0) {
            step.tickNext = state.tick;
            step.initialized = true;
        }
```

Monday.trade: Resolved with `@057f49b5a3 ...`

Zenith: Verified.

[M-4] The overflow innerBitmap can happen in the flipTickWithBitmap when tickSpacing is 256

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- TickBitmap.sol

Description:

The function `enableFeeAmount()` allows `tickSpacing` values up to 256.

- MondayFactory.sol#L96

```
function enableFeeAmount(uint24 fee, int24 tickSpacing) public override {
    ...
▷   require(tickSpacing > 0 && tickSpacing <= 256); // for innerbitmap
    require(feeAmountTickSpacing[fee] == 0);

    feeAmountTickSpacing[fee] = tickSpacing;
    emit FeeAmountEnabled(fee, tickSpacing);
}
```

However, when flip tick in `flipTickWithBitmap()`,

- TickBitmap.sol#L56

```
function flipTickWithBitmap(
    uint256 bitmap,
    int24 orderTick,
    int24 tickSpacing,
    bool zeroForOne
) internal pure returns (uint256) {
    // Get the interval's range tick (lower bound)
    int24 intervalRangeTick = (orderTick / tickSpacing) * tickSpacing;
    //
    if (orderTick % tickSpacing == 0) {
        intervalRangeTick = zeroForOne ? intervalRangeTick + tickSpacing :
```

```
        intervalRangeTick - tickSpacing;
        }
        // place token0, [slot0.tick | -120 | -76 | -60], want to place -76,
        start from -120
        if (orderTick < 0 && zeroForOne == false && orderTick % tickSpacing != 0)
        intervalRangeTick -= tickSpacing;
        // place token1, [60 | 76 | 120 |slot0.tick], want to place 76, start
        from 120
        if (orderTick > 0 && zeroForOne == true && orderTick % tickSpacing != 0)
        intervalRangeTick += tickSpacing;

        // Calculate absolute difference to handle negative values
        // should in (0, 256)
        int24 diff = abs(orderTick - intervalRangeTick);

        uint24 bitPos = uint24(diff);
        uint256 mask = uint256(1) << bitPos;
        return bitmap ^ mask;
    }
```

If `orderTick` is a multiple of `tickSpacing` = 256, then `diff = 256`.

- `bitPos = 256` → `1 << 256` overflows a `uint256` (valid bit positions are 0—255).
- This makes it impossible to store the tick info in the inner bitmap.

As a result, `orders` at the `256th` interval will fail to update the bitmap correctly, breaking tick tracking and order placement logic.

Recommendations:

Cap `tickSpacing` at ≤ 255 to ensure `diff` never exceeds the 0—255 range of the 256-bit inner bitmap. Or we can use `mask = uint256(1) << (bitPos - 1)`, but need to update some logic.

Monday.trade: Resolved with `@998181b912 ...`

Zenith: Verified.

[M-5] Using Rounding-Up taker fee per order causes protocol fee over-accounting

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- OrderHandler.sol

Description:

In `_updateProtocolFees()`, the `taker fee` is calculated using `mulDivRoundingUp`.

- OrderHandler.sol#L315

```
function _updateProtocolFees(uint128 makerSupposedTake, bool zeroForOne)
    private returns (uint256 takenWithFee) {
    int24 orderMakerFee =
        IConfig(IUniswapV3Factory(factory).config())
            .getOrderMakerFee(address(this));
    uint24 orderMakerFeeUnSigned = orderMakerFee > 0 ? uint24(orderMakerFee)
    : uint24(-orderMakerFee);
    uint24 orderTakerFee =
        IConfig(IUniswapV3Factory(factory).config())
            .getOrderTakerFee(address(this));

▷   uint256 takerFeeAmount = FullMath.mulDivRoundingUp(makerSupposedTake,
    orderTakerFee, 1e6);
    uint256 makerFeeAmount = FullMath.mulDiv(makerSupposedTake,
    orderMakerFeeUnSigned, 1e6);
```

This is correct when there is only one `order` at that `tick`.

However, the problem appears when multiple `orders` exist at the `same tick`:

- The `taker` pays `fee` once on the total traded amount.
- But the `protocol` computes `fees` individually per `maker order`, each using round-up.
- Summing the per-order rounded amounts exceeds the actual fee paid.

This leads to a mismatch between:

- actual token balance in the contract, and
- internally recorded accounting.

PoC

```
function testFeeRoundingMismatch_MultipleOrders() public {
      // Step 1: Create 100 different users and place orders at tick 60
      uint128 orderAmount = 1000; // 1000 amounts per order
      int24 orderTick = 60;
      address[100] memory orderOwners;
      token0.mint(address(this), 1 ether);
      // Place orders for all users
      for (uint256 i = 0; i < 100; i++) {
          address user
    = address(uint160(uint256(keccak256(abi.encodePacked("user", i)))));
          orderOwners[i] = user;
          IUniswapV3Pool(address(pool)).place(
              user,
              int128(orderAmount), // token0 order
              orderTick,
              uint32(block.timestamp + 1 hours),
              ''
          );
      }
      // Verify total orders placed
      (, , , , , , , Tick.OrderRecord memory orderRecord) = pool.ticks(60);
      console.log('Total size at tick 60 after 100 orders place:',
    uint256(orderRecord.totalSize));
      // Step 2: One user swaps to fill all orders at once
      // Swap token1 → token0 to fill all orders
      token1.mint(address(this), 1 ether);
      int256 swapAmount0Out = 1 ether;
      (int256 swapAmount0, int256 swapAmount1)
    = IUniswapV3Pool(address(pool)).swap(
          address(this),
          false, // zeroForOne = false, token1 → token0
          swapAmount0Out,
          TestHelper.getSqrtPriceLimitFromTick(120),
          ''
      );
      console.log('Swap completed: amount0 received:', -swapAmount0);
      console.log('Swap completed: amount1 paid:', swapAmount1);
      // Step 3: Each of the 100 order owners withdraws their proceeds
      uint256 totalToken1Withdrawn;
      for (uint256 i = 0; i < 100; i++) {
```

```
            address user = orderOwners[i];
            uint256 userToken1Before = token1.balanceOf(user);
            IUniswapV3Pool(address(pool)).withdraw(user, orderTick, false);
            uint256 userToken1Received = token1.balanceOf(user)
        - userToken1Before;
            totalToken1Withdrawn += userToken1Received;
            // console.log('User', i, 'withdrew token1:', userToken1Received);
        }
        // Final balances
        console.log('=== Summary ===');
        console.log('User amount1 paid for swap:', swapAmount1);
        console.log('Total token1 withdrawn by users:', totalToken1Withdrawn);
        // Check protocol fees
        (uint128 protocolFee0, uint128 protocolFee1) = pool.protocolFees();
        // console.log('Protocol fee token0:', uint256(protocolFee0));
        console.log('Protocol fee token1:', uint256(protocolFee1));
        console.log('Total difference:', int256(uint256(swapAmount1)
        - totalToken1Withdrawn - uint256(protocolFee1)));
    }
```

Logs:

```
Total size at tick 60 after 100 orders place: 100000
Swap completed: amount0 received: 100000
Swap completed: amount1 paid: 100902
=== Summary ===
User amount1 paid for swap: 100902
Total token1 withdrawn by users: 100500
Protocol fee token1: 500
Total difference: -98 (edited)
```

Recommendations:

A safer design is to use round-down when distributing `protocol fees` per `order`, so the total distributed amount never exceeds the actual `taker fee` paid by the user.

```
function _updateProtocolFees(uint128 makerSupposedTake, bool zeroForOne)
    private returns (uint256 takenWithFee) {
    int24 orderMakerFee = IConfig(IUniswapV3Factory(factory)
        .config()).getOrderMakerFee(address(this));
    uint24 orderMakerFeeUnSigned = orderMakerFee > 0 ? uint24(orderMakerFee)
    : uint24(-orderMakerFee);
    uint24 orderTakerFee = IConfig(IUniswapV3Factory(factory)
        .config()).getOrderTakerFee(address(this));
```

```
uint256 takerFeeAmount =
    FullMath.mulDivRoundingUp(makerSupposedTake, orderTakerFee, 1e6);
uint256 takerFeeAmount =
    FullMath.mulDiv(makerSupposedTake, orderTakerFee, 1e6);
```

Monday.trade: Resolved with @12baa3da44 ...

Zenith: Verified.

4.3 Low Risk

A total of 2 low risk findings were identified.

[L-1] Changing config allows altering fee parameters, causing inconsistent accounting for previously filled orders

SEVERITY: Low	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- MondayFactory.sol

Description:

It is allowed to change config contract.

- MondayFactory.sol#L115

```
function setNewConfig(address newConfig) external {
    require(msg.sender == owner);
    require(newConfig ≠ address(0));

    config = newConfig;
    emit ChangeConfigAddress(newConfig);
}
```

But the `config` contract contains `fee parameters` (including `orderTakerFee`). If the new config defines a different `orderTakerFee`, previously `filled orders` will cause inconsistent accounting.

Recommendations:

Remove this function, or it should be strictly cautioned that the `config` must not be changed during a `live pool`'s lifecycle, as modifying it mid-pool `swapping` will cause inconsistent accounting.

Monday.trade: Resolved with `@50072016f6 ...`

Zenith: Verified.

[L-2] Fee parameter change causes incorrect accounting for existing orders

SEVERITY: Low	IMPACT: High
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- Config.sol

Description:

The `protocol` allows the `owner` to modify `orderTakerFee` and `orderMakerFee` at any time via `resetOrderFee()`.

- Config.sol#L77

```
function resetOrderFee(uint24 newTakerFee, int24 newMakerFee)
    external override onlyOwner {
    if (newMakerFee < 0 && uint24(-newMakerFee) > newTakerFee)
    revert('MakerFee Wrong');
    if (newTakerFee >= 1e6 || newMakerFee >= 1e6) revert('Invalid Fee');
    orderMakerFee = newMakerFee;
    orderTakerFee = newTakerFee;
}
```

If the `taker` has already paid `fees` according to the old rates, but `maker/protocol fee` distribution is computed using the _new_ rates, this will results in inconsistent accounting.

Recommendations:

It is better to make them immutable to ensure consistent and correct fee accounting for all `orders`.

Monday.trade: Acknowledged. But I think if the pool has no pending withdrawal orders, resetting the takerFee wouldn't cause any issues.

4.4 Informational

A total of 6 informational findings were identified.

[I-1] Unused parameter (initialized) in the _stepToNextTick function

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- SwapHandler.sol

Description:

In `_stepToNextTick`, the parameter `initialized` is passed in and assigned:

- SwapHandler.sol#L514

```
function _stepToNextTick(
    bool exactInput,
    bool zeroForOne,
    int24 nextTick,
▷   bool initialized,
    uint160 sqrtPriceLimitX96,
    SwapState memory state,
    SwapCache memory cache
) internal view returns (SwapState memory, SwapCache memory) {
    StepComputations memory step;
    step.tickNext = nextTick;
▷   step.initialized = initialized;
```

However, `step.initialized` is never referenced anywhere else inside the function. This means the parameter has no effect on the function's behavior. The assignment and parameter can be safely removed to reduce gas usage and avoid confusion for future maintainers.

Recommendations:

Consider removing this parameter.

Monday.trade: Resolved with `@c31c84c0a5 ...`

Zenith: Verified.

[I-2] Unused variable assignment

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- SwapHandler.sol

Description:

In `_swapCrossRange`, this line is unnecessary.

- SwapHandler.sol#L198

```
function _swapCrossRange(
    bool exactInput,
    bool zeroForOne,
    uint160 sqrtPriceLimitX96,
    SwapState memory state,
    SwapCache memory cache,
    Slot0 memory slot0Start
) private returns (SwapState memory, SwapCache memory) {
    bool traversalOrder = true;
    StepComputations memory step;
    // continue swapping as long as we haven't used the entire input/output
    and haven't reached the price limit
    while (state.amountSpecifiedRemaining ≠ 0 && state.sqrtPriceX96 ≠
    sqrtPriceLimitX96) {

▷       step.sqrtPriceStartX96 = state.sqrtPriceX96;
    ...
```

`step.sqrtPriceStartX96` is never used after this assignment, and removing it would save gas and reduce confusion.

Recommendations:

Remove this line.

Monday.trade: Resolved with `@c47c8dd38b ...`

Zenith: Verified.

[I-3] Mismatch between comment and code in enableFeeAmount

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- MondayFactory.sol

Description:

The function `enableFeeAmount` contains a comment referring to a `tickSpacing` cap of `16384`:

- MondayFactory.sol#L93

```
function enableFeeAmount(uint24 fee, int24 tickSpacing) public override {
    require(msg.sender == owner);
    require(fee < 1000000);
▷   // tick spacing is capped at 16384 to prevent the situation where
    tickSpacing is so large that
    // TickBitmap#nextInitializedTickWithinOneWord overflows int24 container
    from a valid tick
    // 16384 ticks represents a >5x price change with ticks of 1 bips
▷   require(tickSpacing > 0 && tickSpacing <= 256); // for innerbitmap
    require(feeAmountTickSpacing[fee] == 0);

    feeAmountTickSpacing[fee] = tickSpacing;
    emit FeeAmountEnabled(fee, tickSpacing);
}
```

But the actual code caps `tickSpacing` at `256`, not `16384`.

Recommendations:

Update comment.

Monday.trade: Resolved with `@edc8437702 ...`

Zenith: Verified.

[I-4] Tick.initialized comment is inaccurate - should include limit orders

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- Tick.sol

Description:

Current comment claims `initialized` is `true` if `liquidityGross ≠ 0`. That is incorrect: a `tick` can also be considered initialized when it has active limit `orders` (e.g. `orderRecord.totalSize ≠ 0`). The `initialized` flag covers both `liquidity positions` and `limit orders`.

The comment is misleading and may confuse auditors/readers.

Recommendations:

Update comments for `initialzied` value correctly.

Monday.trade: Resolved with `@4272a55f7f ...`

Zenith: Verified.

[I-5] SwapRouter does not support batchPlace

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- SwapRouter.sol

Description:

The `pool`'s `batchPlace` calls `place` multiple times, and each `place` triggers `uniswapV3MintCallback`. However, `SwapRouter` only implements the `single-order` `place` function and its corresponding `uniswapV3MintCallback` flow. `SwapRouter` does not provide any `batchPlace()` entrypoint. This means:

- Users cannot call `batchPlace` through the `SwapRouter`.
- Anyone calling `batchPlace` directly must implement `uniswapV3MintCallback()` themselves.

Recommendations:

Consider adding `batchPlace` implementation in the `SwapRouter`.

Monday.trade: Resolved with `@3f4ae363df ...`

Zenith: Verified.

[I-6] Redundant condition in crossRangeTickOrder

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- LibOrder.sol

Description:

In `crossRangeTickOrder`, the final condition includes `ticks[tick].orderRecord.totalSize == 0`, which is unnecessary because `totalSize` is already reset to `0` at the start of the function.

- LibOrder.sol#L321

```
function crossRangeTickOrder(
    mapping(int24 ⇒ Tick.Info) storage ticks,
    mapping(int16 ⇒ uint256) storage tickBitmap,
    int24 tick,
    bool zeroForOne,
    int24 tickSpacing,
    int24 orderTickSpacing
) internal {
▷   ticks[tick].orderRecord.totalSize = 0;
    ticks[tick].orderRecord.totalTaken = 0;
    if (!zeroForOne) {
        ticks[tick].orderRecord.innerBitmap0 = 0;
        ticks[tick].orderRecord.totalNonce0 = _maxNonceInRange(ticks, tick,
    tickSpacing, orderTickSpacing, true) + 1;
    } else {
        ticks[tick].orderRecord.innerBitmap1 = 0;
        ticks[tick].orderRecord.totalNonce1 = _maxNonceInRange(ticks, tick,
    tickSpacing, orderTickSpacing, false) + 1;
    }

▷   if (ticks[tick].orderRecord.totalSize == 0 && ticks[tick].liquidityGross
    == 0) {
        tickBitmap.flipTick(tick, tickSpacing);
        ticks[tick].initialized = false;
```

```
    }
}
```

Removing this redundant check would simplify the code and slightly reduce gas usage.

Recommendations:

Remove this redundant condition check.

Monday.trade: Resolved with `@4c832bf60dbc ...`

Zenith: Verified.