CANTINA

Gamma: Uni v4 Limit Orders

Competition

May 23, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Gamma is a protocol designed for non-custodial, automated, and active management of concentrated liquidity.

From Apr 28th to May 5th Cantina hosted a competition based on gamma-univ4-limit-orders. The participants identified a total of **10** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 1
- Low Risk: 3
- Gas Optimizations: 0
- Informational: 5

3 Findings

3.1 High Risk

3.1.1 Execution of positions will be skipped when the tick spacing is 1

Submitted by ZoA, also found by Ollam

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: During the execution of orders after a swap happens, it finds the eligible ticks to be executed in `_findOverlappingPositions` function of `LimitOrderManager` contract. However, its logic for fetching next tick is flawed, so it causes issues when the tick spacing is 1.

Finding Description: The `nextInitializedTickWithinOneWord` function of `TickBitmap` library is used to find the next initialized tick in Uniswap. As you can see in the code below, it starts searching for next tick from the current tick when `lte` is `true` and from the next tick when `lte` is `false`.

```
function nextInitializedTickWithinOneWord(
    mapping(int16 => uint256) storage self,
    int24 tick,
    int24 tickSpacing,
    bool lte
) internal view returns (int24 next, bool initialized) {
    unchecked {
        int24 compressed = compress(tick, tickSpacing);

        if (lte) {
            (int16 wordPos, uint8 bitPos) = position(compressed); // <<<
            // ...
        } else {
            // start from the word of the next tick, since the current tick state doesn't matter
            (int16 wordPos, uint8 bitPos) = position(++compressed); // <<<
            // ...
        }
    }
}
```


So that's why Uniswap's logic is implemented as follows, when `zeroForOne` is true, the next tick is `step.tickNext - 1` and when `zeroForOne` is false, the next tick is `step.tickNext`.

```
unchecked {
    result.tick = zeroForOne ? step.tickNext - 1 : step.tickNext;
}
```

However, in the `_findOverlappingPositions` function of `LimitOrderManager`, it sets the next tick as `nextInitializedTick - 1` when `zeroForOne` is true and `nextInitializedTick + 1` when `zeroForOne` is false.

```
// Move to position just beyond this tick (exactly like Uniswap does)
tick = zeroForOne ?
    nextInitializedTick - 1 : // When going down, we need to go to the position before
    nextInitializedTick + 1;  // When going up, we need to go to the position after
```

This usually has no problem except when the tick spacing is 1. Because, when `zeroForOne` is false, the next tick will be `nextInitializedTick + 1` and for next search, it will start from the next tick, so effectively the `nextInitializedTick + 1` is skipped.

Impact Explanation: Users positions will be skipped from being executed.

Likelihood Explanation: The issue only happens when the tick spacing is 1, so it is rare.

Proof of Concept:

```
function test_audit_skip_orders_tickspacing_one() public {
    int24 startTick = 10;
    uint256 orderCount = 10;

    // Create 10 orders
    for (uint256 i = 0; i < orderCount; i++) {
        int24 targetTick = startTick + int24(uint24(i));
        limitOrderManager.createLimitOrder(true, targetTick, 1 ether, poolKey);
```

```
    }

    // Swap to move the tick to the end of the order
    swapRouter.swap(
        poolKey,
        IPoolManager.SwapParams({
            zeroForOne: false,
            amountSpecified: 20 ether,
            sqrtPriceLimitX96: TickMath.getSqrtPriceAtTick(startTick + 13)
        }),
        PoolSwapTest.TestSettings({
            takeClaims: false,
            settleUsingBurn: false
        }),
        ""
    );

    // Check user positions
    ILimitOrderManager.PositionInfo[] memory positions = limitOrderManager.getUserPositions(address(this),
    ↪  poolKey.toId());
    assertEq(positions.length, orderCount, "Incorrect number of positions");

    uint256 activePositions = 0;

    // Check the tick of each position
    for (uint256 i = 0; i < orderCount; i++) {
        (,, bool isActive,,) = limitOrderManager.positionState(poolKey.toId(), positions[i].positionKey);
        if (isActive) activePositions++;
    }

    // Verify half positions not executed yet
    assertEq(activePositions, orderCount / 2);
}
```

Recommendation: Use the correct logic for finding the next tick.

```
  tick = zeroForOne ?
      nextInitializedTick - 1 : // When going down, we need to go to the position before
-     nextInitializedTick + 1;  // When going up, we need to go to the position after
+     nextInitializedTick;  // When going up, we need to go to the position after
```

3.2 Medium Risk

3.2.1 Leftover positions can lose fees

Submitted by ZoA

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: When limit orders are created at `[T, T + tickSpacing]`, corresponding amount of tokens are added as liquidity to the Uniswap pool which accrues the fees generated by existing liquidity. However, the liquidity is not distinguished between `token0` and `token1`, so when the liquidity for `token0` is added, the fees for `token1` are claimed and causes loss.

Description: Here's a diagram to illustrate the issue:

Here's the scenario that describes the illustration above:

1. Execution limit is set to 2, and there are 4 orders created below current tick.
2. A swap happens that moves the tick below the lowest order, which makes 2 orders executable(red colored) and the other 2 are left over(blue colored), waiting for keeper.
3. A new user creates orders at the same tick of orders waiting for keeper(green colored).
4. The keeper executes 2 leftover positions, but they do not accrue fees from swaps.

Now, let's take a look why this issue happens based on the codebase. When an order is created, the liquidity is added to the Uniswap pool where it accrues the fees generated by existing liquidity, and then

the fees are distributed to the existing liquidity that contributed to swaps.

```
function _createOrder(
    ILimitOrderManager.OrderInfo[] memory orders,
    bool isToken0,
    uint256 totalAmount,
    uint256 totalOrders,
    uint256 sizeSkew,
    PoolKey calldata key
) internal whenNotPaused returns (CreateOrderResult[] memory results) {
    // ...

    BalanceDelta[] memory feeDeltas = abi.decode(
        poolManager.unlock(abi.encode(
            UnlockCallbackData({
                callbackType: CallbackType.CREATE_ORDERS,
                data: abi.encode(CreateOrdersCallbackData({key: key, orders: orders, isToken0: isToken0,
                ↪  orderCreator: msg.sender}))
            })
        )),
        (BalanceDelta[])
    );

    bytes32 positionKey;
    OrderInfo memory order;
    for (uint256 i; i < orders.length; i++) {
        order = orders[i];
        (, positionKey) = PositionManagement.getPositionKeys(currentNonce, poolId, order.bottomTick,
        ↪  order.topTick, isToken0);
        require(!positionState[poolId][positionKey].isWaitingKeeper);
        _retrackPositionFee(poolId, positionKey, feeDeltas[i]); // <<<

        // ...
    }

    return results;
}
```


As shown in the code snippet, the fees will be distributed in `_retrackPositionFee` function, which is implemented as follows:

```
function _retrackPositionFee(
    PoolId poolId,
    bytes32 positionKey,
    BalanceDelta feeDelta
) internal {
    PositionState storage posState = positionState[poolId][positionKey];
    if (posState.totalLiquidity == 0) return; // <<<

    if(feeDelta == ZERO_DELTA) return;

    posState.feePerLiquidity = posState.feePerLiquidity +
        PositionManagement.calculateScaledFeePerLiquidity(feeDelta, posState.totalLiquidity);
}
```

So, when the Step 3 happens, the fees are accrued from Uniswap V3 pool because the liquidity for `token1` exists and contributed to swaps. However, when distributing the fees, the `totalLiquidity` is 0 because it's total liquidity of `token0`. As a result, the fees are not distributed but locked in the manager contract.

Impact Explanation: Users lose their fees which can be considered as loss of funds.

Likelihood Explanation: The issue can happen whenever there are leftover positions, or a malicious user can intentionally create left over positions by applying a large swap.

Proof of Concept:

```
// test/BasicOrderTest.t.sol
// NOTE: import SqrtPriceMath, `import {SqrtPriceMath} from "v4-core/libraries/SqrtPriceMath.sol";`

function test_audit_leftover_positions_lose_fee() public {
    limitOrderManager.setExecutablePositionsLimit(1);
    limitOrderManager.setKeeper(address(this), true);
    orderManager.setHookFeePercentage(0); // set to 0 for easy calculation below

    int24 t = -360;
```

```
    int24 ts = poolKey.tickSpacing;

    // Create 2 orders, isToken0 = false, [-360, -300], [-300, -240]
    limitOrderManager.createLimitOrder(false, t, 1 ether, poolKey);
    limitOrderManager.createLimitOrder(false, t + ts, 1 ether, poolKey);

    // Simulate swaps to move the tick below -360
    swapRouter.swap(
        poolKey,
        IPoolManager.SwapParams({
            zeroForOne: true,
            amountSpecified: -4 ether,
            sqrtPriceLimitX96: TickMath.getSqrtPriceAtTick(t - ts - 10)
        }),
        PoolSwapTest.TestSettings({
            takeClaims: false,
            settleUsingBurn: false
        }),
        ""
    );

    // executablePositionsLimit = 1, so 1 position should be waiting for keeper, which is the one at [-360,
    ↪  -300]
    PoolId poolId = poolKey.toId();
    ILimitOrderManager.PositionInfo[] memory positions = limitOrderManager.getUserPositions(address(this),
    ↪  poolId);
    uint256 waitingForKeeper = 0;
    for (uint256 i = 0; i < positions.length; i++) {
        (,,, bool isWaitingKeeper,) = limitOrderManager.positionState(poolId, positions[i].positionKey);
        if (isWaitingKeeper) waitingForKeeper++;
    }
    assertEq(positions.length, 2, "Should have 2 positions");
    assertEq(waitingForKeeper, 1, "Should have 1 positions waiting for keeper");

    // Create 1 order, isToken0 = true, [-360, -300]
    limitOrderManager.createLimitOrder(true, t + ts, 1 ether, poolKey);

    // Keeper executes leftover positions, which is the one at [-360, -300]
    ILimitOrderManager.PositionTickRange[] memory leftoverPositions = new
    ↪  ILimitOrderManager.PositionTickRange[](1);
    leftoverPositions[0] = ILimitOrderManager.PositionTickRange({
        bottomTick: t,
        topTick: t + ts,
        isToken0: false
    });
    limitOrderManager.executeOrderByKeeper(poolKey, leftoverPositions);

    // Claim the order at [-360, -300]
    uint256 token0BalanceBefore = IERC20Minimal(Currency.unwrap(poolKey.currency0)).balanceOf(address(this));
    limitOrderManager.claimOrder(poolKey, getPositionKey(t, t + ts, false, 0), address(this));
    uint256 token0BalanceAfter = IERC20Minimal(Currency.unwrap(poolKey.currency0)).balanceOf(address(this));

    uint256 token0Claimed = token0BalanceAfter - token0BalanceBefore; // Actual claimed amount
    uint256 calculatedToken0Amount = SqrtPriceMath.getAmount0Delta(
        TickMath.getSqrtPriceAtTick(t),
        TickMath.getSqrtPriceAtTick(t + ts),
        positions[0].liquidity,
        false
    ); // The token0 amount between [-360, -300]

    // Claim the order at [-300, -240], this is to remove all remaining assets from the order manager
    limitOrderManager.claimOrder(poolKey, getPositionKey(t + ts, t + 2 * ts, false, 0), address(this));

    // The position should have claimed the fees, but no fees accrued
    assertApproxEqAbs(token0Claimed, calculatedToken0Amount, 1e3);

    // Instead, the fees are locked in the order manager contract
    uint256 orderManagerBalance = manager.balanceOf(address(orderManager),
    ↪  uint256(uint160(Currency.unwrap(poolKey.currency0))));
    assertApproxEqAbs(orderManagerBalance, calculatedToken0Amount * 3 / 997, 1e3);
}
```


Recommendation: This issue can be fixed by distinguishing the liquidity between `token0` and `token1`, which can be accomplished by setting different `salt` for `token0` and `token1`, in `handleCreateOrdersCallback` function. For example, 0 for `token0` and 1 for `token1`.

```
function handleCreateOrdersCallback(
    CallbackState storage state,
    ILimitOrderManager.CreateOrdersCallbackData memory callbackData
) internal returns(bytes memory) {
    // BalanceDelta[] memory deltas = new BalanceDelta[](callbackData.orders.length);
    BalanceDelta[] memory feeDeltas = new BalanceDelta[](callbackData.orders.length);
    BalanceDelta accumulatedMintFees;
    for (uint256 i = 0; i < callbackData.orders.length; i++) {
        ILimitOrderManager.OrderInfo memory order = callbackData.orders[i];
        callbackData.isToken0 ?
            callbackData.key.currency0.settle(state.poolManager, address(this), order.amount, false) :
            callbackData.key.currency1.settle(state.poolManager, address(this), order.amount, false);
        (, BalanceDelta feeDelta) = state.poolManager.modifyLiquidity(
            callbackData.key,
            IPoolManager.ModifyLiquidityParams({
                tickLower: order.bottomTick,
                tickUpper: order.topTick,
                liquidityDelta: int256(uint256(order.liquidity)),
                salt: bytes32(0) // <<<
            }),
            ""
        );
        // deltas[i] = delta;
        feeDeltas[i] = feeDelta;
        // Accumulate fee details if greater than zero
        if (feeDelta != ZERO_DELTA) {
            accumulatedMintFees = accumulatedMintFees + feeDelta;
        }
    }
    _mintFeesToHook(state, callbackData.key, accumulatedMintFees);

    // Clear any remaining dust amounts
    _clearExactDelta(state, callbackData.key.currency0);
    _clearExactDelta(state, callbackData.key.currency1);

    return abi.encode(feeDeltas);
}
```


3.3 Low Risk

3.3.1 An attacker can abuse the isWaitingKeeper flag to keep other users from creating positions

Submitted by Ollam

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Summary: When too many limit orders are crossed in a single transaction, the remaining orders are marked for keeper execution. The isWaitingKeeper flag keeps users from creating a new position on that tick. In pools with thin liquidity, an attacker could make a series of small limit orders on every tick, slam the price in both directions, and then lock out any other user from making limit orders on those ticks.

Finding Description: In _handleLeftoverPositions(), the isWaitingKeeper() flag is set to true. This flag allows the keeper to execute the limit order in the event that a trade was so big that some limit orders had to be skipped for gas savings purposes.

```
function _handleLeftoverPositions(

    PoolId poolId,

    int24[] memory executableTicks,

    uint256 executableCount,

    bool zeroForOne

) internal {

    uint256 leftoverCount = executableTicks.length - executableCount;


    // Mark remaining positions for keeper execution

    for(uint256 i = executableCount; i < executableTicks.length; i++) {

        int24 tick = executableTicks[i];

        bytes32 posKey = zeroForOne ?

            token1PositionAtTick[poolId][tick] :

            token0PositionAtTick[poolId][tick];


          positionState[poolId][posKey].isWaitingKeeper = true;

    }
```

However, this flag prevents users from creating positions on any of those ticks in _createOrder:

```
bytes32 positionKey;

OrderInfo memory order;

for (uint256 i; i < orders.length; i++) {

    order = orders[i];

    (, positionKey) = PositionManagement.getPositionKeys(currentNonce, poolId, order.bottomTick,
    ↪  order.topTick, isToken0);

    require(!positionState[poolId][positionKey].isWaitingKeeper); // <<<

    _retrackPositionFee(poolId, positionKey, feeDeltas[i]);
```

This allows an attacker to DOS a chunk of a pool, keeping legitimate users from placing orders. If liquidity is thin enough it would cost the attacker very little to pull off. Additionally, because the attacker can essentially freeze a huge chunk of the order book, he can make it such that the only possible place to buy a token is at a far a way price that is nowhere close to fair value. In order to pull this off, the attacker must place a ton limit orders covering every tick, buy a huge amount, then sell a huge amount.

This could also occur without an intentional attack happening at all. If there is enough volatility before the keeper can operate or if there is a sandwich attack of some sort, the protocol could still be left with DOS'd ticks.

Impact Explanation: I have chosen a high impact for this finding because it allows an attacker to DOS a range of ticks from having orders placed in them. Until the market moves through those ticks, they will remain entirely unusable to users, as the keeper will be unable to execute those orders unless price goes back above them.

Likelihood Explanation: I chose a high likelihood for this finding because although certain circumstances need to be true for this to be a worthwhile attack, those circumstances will likely not be all that rare. Namely, the liquidity in the pool needs to be thin over a range. Given that this protocol will have an entirely separate liquidity pool from standard uniswap pools, it is very likely that little if any liquidity will exist in the pool outside of the actual limit orders made by the protocol. This means that after any large

move in price, the liquidity in the other direction will be almost entirely vacant, allowing an attacker to DOS the pool gas and a small amount of fees, incurring very little slippage. This attack is also possible immediately after pool initialization. When a pool is first added to the protocol, there will be no limit orders at all, which would allow an attacker to lock down a huge portion of the pool for very little.

This could also occur through normal, albeit somewhat rare usage of the protocol.

Proof of Concept: Please paste this proof of concept into `BasicOrderTest.t.sol`:

```
function test_unusable_ticks() public {
    int24 currentTick;
    limitOrderManager.setExecutablePositionsLimit(5);

    // Initialize pool with tick spacing of 2
    PoolKey memory newPool = PoolKey(currency0, currency1, 2, 2, hook);
    manager.initialize(newPool, SQRT_PRICE_1_1);
    orderManager.setWhitelistedPool(newPool.toId(), true);

    // Approve tokens to manager
    IERC20Minimal(Currency.unwrap(currency0)).approve(address(limitOrderManager), type(uint256).max);
    IERC20Minimal(Currency.unwrap(currency1)).approve(address(limitOrderManager), type(uint256).max);

    // Add initial liquidity for testing
    modifyLiquidityRouter.modifyLiquidity(
        newPool,
        IPoolManager.ModifyLiquidityParams({
            tickLower: -887220,
            tickUpper: 887220,
            liquidityDelta: 100 ether,
            salt: bytes32(0)
        }),
        ""
    );

    // create 100 orders from ticks 2 to 202
    // 100 more after that
    // 100 more after that, all sequential ticks
    limitOrderManager.createScaleOrders(true, 2, 202, 1e10, 100, 1e18, newPool);
    limitOrderManager.createScaleOrders(true, 202, 402, 1e10, 100, 1e18, newPool);
    limitOrderManager.createScaleOrders(true, 402, 602, 1e10, 100, 1e18, newPool);

    // swap above tick 602
    swap(newPool, false, 3e18, "");
    (, currentTick,,) = StateLibrary.getSlot0(manager, newPool.toId());
    console.log(currentTick);

    // swap below tick 0
    swap(newPool, true, 4e18, "");
    (, currentTick,,) = StateLibrary.getSlot0(manager, newPool.toId());
    console.log(currentTick);

    // if a user attempts to add to the position after a quick flash pump they will be unable to
    vm.expectRevert();
    limitOrderManager.createScaleOrders(true, 2, 202, 1e18, 10, 1e18, newPool);
}
```

Recommendation: This can be fixed by removing the requirement that a new order can't be placed on a tick marked for keeper execution. Removing LimitOrderManager.sol#L211 would fix the DOS but would add some complexity into the system as long positions and short positions would need to be tracked separately to avoid collisions. This could be done by giving token0 and token1 swaps different salts and adding a direction bool to `UserPosition` and `PositionState`.

3.3.2 Incorrect use of `slot0.tick` for limit order validation results in valid orders being impossible to create

Submitted by 0xrafaelnicolau

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

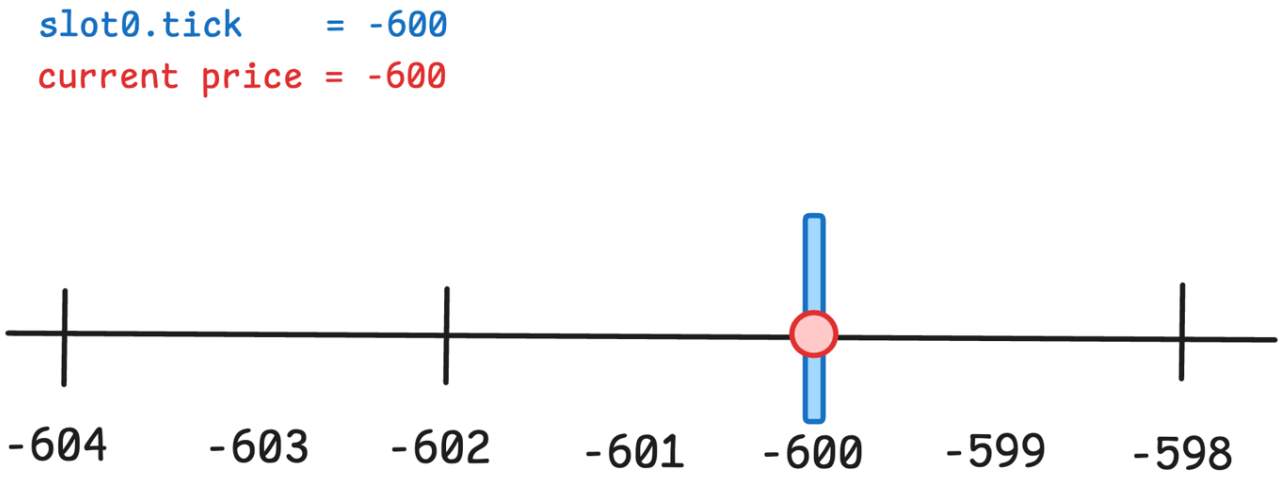
Summary: The `LimitOrderManager.createLimitOrder()` and `LimitOrderManager.createScaleOrders()` functions incorrectly use `slot0.tick` as the current tick instead of the tick derived from the current price

`sqrtPriceX96` when validating new limit orders. This discrepancy breaks an important invariant assumption about minimum tick spacing differences between the `currentRoundedTick` and the `targetRoundedTick`.

Finding Description: When creating an order through the `LimitOrderManager.createLimitOrder()` function the `currentTick` is read directly from `slot0`:

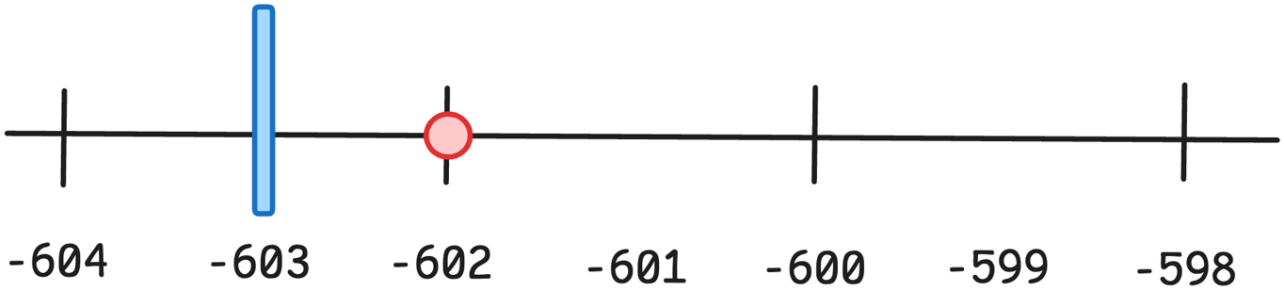
```
function createLimitOrder(
    bool isToken0,
    int24 targetTick,
    uint256 amount,
    PoolKey calldata key
) external payable override returns (CreateOrderResult memory) {
    // Get current tick for validation
    PoolId poolId = key.toId();

    (, int24 currentTick, , ) = StateLibrary.getSlot0(poolManager, poolId); // <<<

    // For limit orders, target tick is either lower or upper based on direction
    (int24 bottomTick, int24 topTick) = TickLibrary.getValidTickRange(
        currentTick,
        targetTick,
        key.tickSpacing,
        isToken0
    );

    ILimitOrderManager.OrderInfo[] memory orders = new ILimitOrderManager.OrderInfo[](1);
    orders[0] = ILimitOrderManager.OrderInfo({
        bottomTick: bottomTick,
        topTick: topTick,
        amount: 0,
        liquidity: 0
    });

    CreateOrderResult[] memory results = _createOrder(orders, isToken0, amount, 1, 0, key);
    return results[0]; // Return just the first result since it's a single order
}
```

Which is not accurate if the current price is near a border. In the following example the current tick stored at `slot0` and the current tick derived from the current price are both at `-600`.



However, by executing a swap where `zeroForOne = true` up to tick `-602` left border, the resulting tick stored at `slot0` will be `-603` while the current tick derived from the current price will be at `-602`.

```
slot0.tick    = -603
current price = -602
```



Take a look at the following code snippet from Uniswap V4 pool contract to understand why this occurs:

```
// Shift tick if we reached the next price, and preemptively decrement for zeroForOne swaps to tickNext - 1.
// If the swap doesn't continue (if amountRemaining == 0 or sqrtPriceLimit is met), slot0.tick will be 1 less
// than getTickAtSqrtPrice(slot0.sqrtPrice). This doesn't affect swaps, but donation calls should verify both
// price and tick to reward the correct LPs.
if (result.sqrtPriceX96 == step.sqrtPriceNextX96) {
    // if the tick is initialized, run the tick transition
    if (step.initialized) {
        (uint256 feeGrowthGlobal0X128, uint256 feeGrowthGlobal1X128) = zeroForOne
            ? (step.feeGrowthGlobalX128, self.feeGrowthGlobal1X128)
            : (self.feeGrowthGlobal0X128, step.feeGrowthGlobalX128);
        int128 liquidityNet =
            Pool.crossTick(self, step.tickNext, feeGrowthGlobal0X128, feeGrowthGlobal1X128);
        // if we're moving leftward, we interpret liquidityNet as the opposite sign
        // safe because liquidityNet cannot be type(int128).min
        unchecked {
            if (zeroForOne) liquidityNet = -liquidityNet;
        }

        result.liquidity = LiquidityMath.addDelta(result.liquidity, liquidityNet);
    }

    unchecked {
        result.tick = zeroForOne ? step.tickNext - 1 : step.tickNext; // <<<
    }
} else if (result.sqrtPriceX96 != step.sqrtPriceStartX96) {
    // recompute unless we're on a lower tick boundary (i.e. already transitioned ticks), and haven't moved
    result.tick = TickMath.getTickAtSqrtPrice(result.sqrtPriceX96);
}
```

This allows for the following invariant `The difference between ticks must be at least tickSpacing` to be broken. This invariant can be broken due to the discrepancy between the current tick stored at `slot0` and the current tick derived from the current price, which renders the following validation invalid:

```
if(tickDiff < tickSpacing)
    revert RoundedTicksTooClose(currentTick, roundedCurrentTick, targetTick, roundedTargetTick, isToken0);
```

Impact Explanation: Medium. The vulnerability prevents users from creating valid limit orders in specific edge cases where the current price is near a tick boundary, restricting certain trading opportunities.

Likelihood Explanation: Medium. This issue will consistently manifest whenever a swap causes the current price to cross a left tick boundary during a `zeroForOne` swap, a common occurrence in active trading pools.

Proof of Concept:

1. Create a file inside the `test/` folder called `Poc.t.sol`.

2. Paste the following test in the `Poc.t.sol` file.
3. Run the following test with `forge test --mt test_POC -vvv`.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "forge-std/console.sol";
import {Deployers} from "@uniswap/v4-core/test/utils/Deployers.sol";
import {PoolSwapTest} from "v4-core/test/PoolSwapTest.sol";
import {IPoolManager} from "v4-core/interfaces/IPoolManager.sol";
import {PoolKey} from "v4-core/types/PoolKey.sol";
import {PoolId, PoolIdLibrary} from "v4-core/types/PoolId.sol";
import {Currency, CurrencyLibrary} from "v4-core/types/Currency.sol";
import {StateLibrary} from "v4-core/libraries/StateLibrary.sol";
import {TickMath} from "v4-core/libraries/TickMath.sol";
import {LimitOrderHook} from "src/LimitOrderHook.sol";
import {LimitOrderManager} from "src/LimitOrderManager.sol";
import {Hooks} from "v4-core/libraries/Hooks.sol";
import {IERC20Minimal} from "v4-core/interfaces/external/IERC20Minimal.sol";
import {BalanceDelta} from "v4-core/types/BalanceDelta.sol";
import {LiquidityAmounts} from "@uniswap/v4-core/test/utils/LiquidityAmounts.sol";
import {ILimitOrderManager} from "src/ILimitOrderManager.sol";
import {LimitOrderLens} from "src/LimitOrderLens.sol";
import "../src/TickLibrary.sol";

contract Poc is Test, Deployers {
    using PoolIdLibrary for PoolKey;
    using CurrencyLibrary for Currency;

    address public immutable TREASURY = makeAddr("treasury");

    LimitOrderHook hook;
    ILimitOrderManager limitOrderManager;
    LimitOrderManager orderManager;
    LimitOrderLens lens;
    PoolKey poolKey;

    function setUp() public {
        deployFreshManagerAndRouters();
        (currency0, currency1) = deployMintAndApprove2Currencies();

        orderManager = new LimitOrderManager(address(manager), TREASURY, address(this));

        uint160 flags = uint160(Hooks.BEFORE_SWAP_FLAG | Hooks.AFTER_SWAP_FLAG);
        address hookAddress = address(flags);

        deployCodeTo(
            "LimitOrderHook.sol", abi.encode(address(manager), address(orderManager), address(this)),
            ↪  hookAddress
        );
        hook = LimitOrderHook(hookAddress);
        limitOrderManager = ILimitOrderManager(address(orderManager));
        lens = new LimitOrderLens(address(orderManager), address(this));
        limitOrderManager.setExecutablePositionsLimit(5);
        limitOrderManager.setHook(address(hook));
        orderManager.setWhitelistedPool(poolKey.toId(), true);
    }

    function test_POC() public {
        // 1. Initialize the pool with fee tier = 100 (tick spacing = 2)
        (poolKey,) = initPool(currency0, currency1, hook, 100, TickMath.getSqrtPriceAtTick(-600));
        orderManager.setWhitelistedPool(poolKey.toId(), true);

        // 2. Add liquidity across multiple tick boundaries
        deal(Currency.unwrap(currency0), address(this), 5e18);
        deal(Currency.unwrap(currency1), address(this), 5e18);
        IERC20Minimal(Currency.unwrap(currency0)).approve(address(limitOrderManager), 5e18);
        IERC20Minimal(Currency.unwrap(currency1)).approve(address(limitOrderManager), 5e18);
        modifyLiquidityRouter.modifyLiquidity(
            poolKey,
            IPoolManager.ModifyLiquidityParams({
                tickLower: -604,
                tickUpper: -596,
                liquidityDelta: 5e18,
                salt: bytes32(0)
```

```
            }),
            ""
        );

        // 3. Assert that the pool was initialized and its current tick is -600.
        (uint160 sqrtPriceX96, int24 tick,,) = StateLibrary.getSlot0(manager, poolKey.toId());
        console.log("current tick (slot0)           : ", tick);
        console.log("current tick (price)           : ", TickMath.getTickAtSqrtPrice(sqrtPriceX96));
        assertEq(tick, -600);
        assertEq(TickMath.getTickAtSqrtPrice(sqrtPriceX96), -600);

        // 4. Create an order with bottom tick -602 and top tick at -600 to swap token1 for token0.
        LimitOrderManager.CreateOrderResult memory order =
            limitOrderManager.createLimitOrder(false, -602, 1e18, poolKey);
        assertEq(order.bottomTick, -602);
        assertEq(order.topTick, -600);

        // 5. Swap token0 for token1 to cross the left boundary of the -600 tick.
        deal(Currency.unwrap(currency0), address(this), 100e18);
        swapRouter.swap(
            poolKey,
            IPoolManager.SwapParams({
                zeroForOne: true,
                amountSpecified: -100e18,
                sqrtPriceLimitX96: TickMath.getSqrtPriceAtTick(-602)
            }),
            PoolSwapTest.TestSettings({takeClaims: false, settleUsingBurn: false}),
            ""
        );

        // 6. Assert that the current tick at slot0 is -603 but the tick based on sqrtPriceX96 is -602.
        (sqrtPriceX96, tick,,) = StateLibrary.getSlot0(manager, poolKey.toId());
        console.log("current tick (slot0)           : ", tick);
        console.log("current tick (price)           : ", TickMath.getTickAtSqrtPrice(sqrtPriceX96));
        assertEq(tick, -603);
        assertEq(TickMath.getTickAtSqrtPrice(sqrtPriceX96), -602);

        // 7. Assert that if a order was to be created at tick -603, and tick -602 the current rounded
        // tick will result in two different values.
        console.log("rounded current tick (slot0)   : ", TickLibrary.getRoundedCurrentTick(-603, true, 2));
        console.log("rounded current tick (price)   : ", TickLibrary.getRoundedCurrentTick(-602, true, 2));
        assertEq(TickLibrary.getRoundedCurrentTick(-603, true, 2), -602);
        assertEq(TickLibrary.getRoundedCurrentTick(-602, true, 2), -600);

        // 8. Assert that the order can not be created because the contract computes the current tick
        ↪  incorrectly.
        // -> Actual:
        // current tick : -603
        // target tick  : -604
        // tick diff    : 1    // REVERT
        // -> Expected:
        // current tick : -602
        // target tick  : -604
        // tick diff    : 2    // NO REVERT
        vm.expectRevert(
            abi.encodeWithSelector(TickLibrary.RoundedTicksTooClose.selector, -603, -604, -604, -604, false)
        );
        order = limitOrderManager.createLimitOrder(false, -604, 1e18, poolKey);
    }
}
```

Recommendation: Derive the `currentTick` from the current `sqrtPriceX96` and use it for validation ineffective.

```
  function createLimitOrder(
      bool isToken0,
      int24 targetTick,
      uint256 amount,
      PoolKey calldata key
  ) external payable override returns (CreateOrderResult memory) {
      // Get current tick for validation
      PoolId poolId = key.toId();

-     (, int24 currentTick, , ) = StateLibrary.getSlot0(poolManager, poolId);
+     (uint160 sqrtPriceX96,,,) = StateLibrary.getSlot0(poolManager, poolId);
+     int24 currentTick = TickMath.getTickAtSqrtPrice(sqrtPriceX96);
```

```
        // For limit orders, target tick is either lower or upper based on direction
        (int24 bottomTick, int24 topTick) = TickLibrary.getValidTickRange(
            currentTick,
            targetTick,
            key.tickSpacing,
            isToken0
        );

        ILimitOrderManager.OrderInfo[] memory orders = new ILimitOrderManager.OrderInfo[](1);
        orders[0] = ILimitOrderManager.OrderInfo({
            bottomTick: bottomTick,
            topTick: topTick,
            amount: 0,
            liquidity: 0
        });

        CreateOrderResult[] memory results = _createOrder(orders, isToken0, amount, 1, 0, key);
        return results[0]; // Return just the first result since it's a single order
    }

    function createScaleOrders(
        bool isToken0,
        int24 bottomTick,
        int24 topTick,
        uint256 totalAmount,
        uint256 totalOrders,
        uint256 sizeSkew,
        PoolKey calldata key
    ) external payable returns (CreateOrderResult[] memory results) {
        // Get current tick for validation
        PoolId poolId = key.toId();

-       (, int24 currentTick, , ) = StateLibrary.getSlot0(poolManager, poolId);
+       (uint160 sqrtPriceX96,,,) = StateLibrary.getSlot0(poolManager, poolId);
+       int24 currentTick = TickMath.getTickAtSqrtPrice(sqrtPriceX96);

        require(totalOrders <= maxOrderLimit);
        ILimitOrderManager.OrderInfo[] memory orders =
            TickLibrary.validateAndPrepareScaleOrders(bottomTick, topTick, currentTick, isToken0, totalOrders,
            ↪  sizeSkew, key.tickSpacing);

        results = _createOrder(orders, isToken0, totalAmount, totalOrders, sizeSkew, key);
    }
```


3.3.3 In pools with no broad liquidity positions, new limit orders can be DOS'd by 1 wei swaps

Submitted by Ollam

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Summary: If there is no broad liquidity position (minTick to maxTick or something similar) costing traders slippage every time they trade, an attacker can frontrun new limit orders with a very small swap and DOS them. This only applies to the range between the highest bid and the lowest ask, but any new orders in that range are susceptible to this.

Finding Description: When creating a new position, there are safeguards in place to make sure that the new position doesn't overlap with the current tick. This check occurs in TickLibrary.sol in getValidTick-Range():

```
function getValidTickRange(
    int24 currentTick,
    int24 targetTick,
    int24 tickSpacing,
    bool isToken0
) public pure returns (int24 bottomTick, int24 topTick) {
    if(isToken0 && currentTick >= targetTick)
        revert WrongTargetTick(currentTick, targetTick, true);
    if(!isToken0 && currentTick <= targetTick)
        revert WrongTargetTick(currentTick, targetTick, false);

    int24 roundedTargetTick = getRoundedTargetTick(targetTick, isToken0, tickSpacing);
    int24 roundedCurrentTick = getRoundedCurrentTick(currentTick, isToken0, tickSpacing);
```

However, given that the liquidity of these pools is likely to be nonexistent or incredibly thin other than the limit orders themselves, that allows an attacker to push the price around almost for free, which can dos any new positions within a certain range if the attacker is able to frontrun legitimate users. This will lead to a significantly degraded user experience.

Impact Explanation: I chose a medium impact for this because it is a temporary DOS that will last until the next swap in the next direction. It also doesn't impact any orders out of the range of the highest bid and lowest ask, so it definitely does not warrant a high.

Likelihood Explanation: I chose a medium likelihood for this given that it costs almost nothing for an attacker to DOS individual limit orders, but does require front running capability. There is a potential profit motive if the attacker owns the positions at the limit and is trying to get fills at distant prices, but that isn't guaranteed by any means.

Proof of Concept: Please place this in `BasicOrderTest.t.sol`:

```
function test_tiny_swap_dos() public {

    int24 currentTick;
    limitOrderManager.setExecutablePositionsLimit(5);

    // Initialize pool with tick spacing of 1
    PoolKey memory newPool = PoolKey(currency0, currency1, 2, 2, hook);
    manager.initialize(newPool, SQRT_PRICE_1_1);
    orderManager.setWhitelistedPool(newPool.toId(), true);

    // Approve tokens to manager
    IERC20Minimal(Currency.unwrap(currency0)).approve(address(limitOrderManager), type(uint256).max);
    IERC20Minimal(Currency.unwrap(currency1)).approve(address(limitOrderManager), type(uint256).max);

    // no initial liquidty
    // let's make two positions , one around tick 1000 one around tick -1000
    limitOrderManager.createLimitOrder(true, 1000, 1e18, newPool);
    limitOrderManager.createLimitOrder(false, -1000, 1e18, newPool);
    // current tick is zero
    (, currentTick,,) = StateLibrary.getSlot0(manager, newPool.toId());
    console.log(currentTick);

    // swap all the way down for 1 wei, frontrun legitamate user who was trying to create a position
    swap(newPool, true, 1, "");
    (, currentTick,,) = StateLibrary.getSlot0(manager, newPool.toId());
    console.log(currentTick);

    // legit user was trying to make a swap that would have worked, got blocked by 1 wei
    vm.expectRevert();
    limitOrderManager.createLimitOrder(false, -1000, 1e18, newPool);

    // swap all the way up for 3 wei, frontrun legitamate user who was trying to create a position
    swap(newPool, false, 3, "");
    (, currentTick,,) = StateLibrary.getSlot0(manager, newPool.toId());
    console.log(currentTick);

    // once again legitimate user blocked
    vm.expectRevert();
    limitOrderManager.createLimitOrder(true, 1000, 1e18, newPool);
}
```

Recommendation: Consider making a small broad liquidity position when making a new pool to make

this sort of attack infeasible due to slippage.

3.4 Informational

3.4.1 Missing Whitelist Validation in Keeper Functionality

Submitted by Maze

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: The `executeOrderByKeeper()` function does not validate that the pool is whitelisted, allowing keepers to execute orders on pools that have been deliberately de-whitelisted, bypassing an important security control enforced in other functions.

Description: The LimitOrderManager contract implements a whitelist mechanism for pools via the `whitelistedPool[poolId]` mapping. This acts as a security control to restrict which pools can be interacted with. Most functions in the contract, including order creation functions like `_createOrder()`, properly enforce this security check by verifying that `whitelistedPool[poolId]` is true before proceeding.

However, the `executeOrderByKeeper()` function (around line 1140) does not include this check. This oversight allows keepers to execute orders on pools that have been deliberately de-whitelisted by the contract owner, bypassing the security mechanism.

The inconsistency is evident when comparing the implementations:

```
// _createOrder() function properly checks the whitelist
function _createOrder(...) internal whenNotPaused returns (CreateOrderResult[] memory results) {
    // ...
    PoolId poolId = key.toId();
    if (!whitelistedPool[poolId]) revert NotWhitelistedPool();
    // ...
}

// executeOrderByKeeper() lacks the whitelist check
function executeOrderByKeeper(PoolKey calldata key, PositionTickRange[] memory waitingPositions) external {
    require(isKeeper[msg.sender]);
    if (waitingPositions.length == 0) return;
    PoolId poolId = key.toId();
    // Missing check: if (!whitelistedPool[poolId]) revert NotWhitelistedPool();
    // ...
}
```


This inconsistency creates a loophole in the system's security model.

Impact: Medium. The vulnerability allows keepers (who are trusted roles with special permissions) to bypass an important security control and interact with pools that have been deliberately de-whitelisted.

If a pool has been de-whitelisted due to a security concern (such as a vulnerability in the pool's implementation), keepers could still execute orders on that pool, potentially putting user funds at risk. While this requires a malicious or compromised keeper, it undermines the purpose of the whitelist security control.

Likelihood: Medium. The likelihood depends on the trust model of keepers and the frequency of pool de-whitelisting. Since keepers are designated as trusted roles in the system (as stated in the `README.md`), the likelihood of malicious exploitation is reduced. However, this security check inconsistency represents a significant flaw in the system's security architecture that could be exploited if a keeper is compromised or acts maliciously.

Proof of Concept: I've created a proof of concept test that demonstrates this vulnerability:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "forge-std/console.sol";
import {Deployers} from "@uniswap/v4-core/test/utils/Deployers.sol";
import {PoolKey} from "v4-core/types/PoolKey.sol";
import {PoolId, PoolIdLibrary} from "v4-core/types/PoolId.sol";
import {Currency, CurrencyLibrary} from "v4-core/types/Currency.sol";
import {TickMath} from "v4-core/libraries/TickMath.sol";
```

```
import {IHooks} from "v4-core/interfaces/IHooks.sol";
import {ILimitOrderManager} from "src/ILimitOrderManager.sol";
import {LimitOrderManager} from "src/LimitOrderManager.sol";
import {LimitOrderHook} from "src/LimitOrderHook.sol";
import {IERC20Minimal} from "v4-core/interfaces/external/IERC20Minimal.sol";

contract MissingWhitelistCheckTest is Test, Deployers {
    using PoolIdLibrary for PoolKey;
    using CurrencyLibrary for Currency;

    LimitOrderHook hook;
    ILimitOrderManager limitOrderManager;
    LimitOrderManager orderManager;
    address public treasury;

    address public orderCreator = address(0x1);
    address public keeper = address(0x2);

    PoolKey poolKey;
    PoolId poolId;
    uint24 fee = 3000;
    int24 tickSpacing = 60;

    error NotWhitelistedPool();

    function setUp() public {
        deployFreshManagerAndRouters();
        (currency0, currency1) = deployMintAndApprove2Currencies();

        treasury = makeAddr("treasury");

        orderManager = new LimitOrderManager(
            address(manager),
            treasury,
            address(this)
        );

        uint160 flags = uint160(
            Hooks.BEFORE_SWAP_FLAG |
            Hooks.AFTER_SWAP_FLAG
        );
        address hookAddress = address(flags);

        deployCodeTo(
            "LimitOrderHook.sol",
            abi.encode(address(manager), address(orderManager), address(this)),
            hookAddress
        );
        hook = LimitOrderHook(hookAddress);

        limitOrderManager = ILimitOrderManager(address(orderManager));
        limitOrderManager.setExecutablePositionsLimit(5);
        limitOrderManager.setHook(address(hook));

        (poolKey,) = initPool(
            currency0,
            currency1,
            IHooks(address(hook)),
            fee,
            tickSpacing,
            SQRT_PRICE_1_1
        );
        poolId = poolKey.toId();

        orderManager.setWhitelistedPool(poolId, true);
        orderManager.setKeeper(keeper, true);

        deal(Currency.unwrap(currency0), orderCreator, 10 ether);
        deal(Currency.unwrap(currency1), orderCreator, 10 ether);

        vm.startPrank(orderCreator);
        IERC20Minimal(Currency.unwrap(currency0)).approve(address(limitOrderManager), 10 ether);
        IERC20Minimal(Currency.unwrap(currency1)).approve(address(limitOrderManager), 10 ether);
        vm.stopPrank();
    }
```

```
    function test_missing_whitelist_check_in_keeper_execute() public {
        // 1. Create orders on whitelisted pool
        vm.startPrank(orderCreator);
        limitOrderManager.createScaleOrders(
            true,
            60,
            180,
            5 ether,
            2,
            1 ether,
            poolKey
        );
        vm.stopPrank();

        // Verify orders were created successfully
        assertTrue(limitOrderManager.getUserPositions(orderCreator, poolId).length > 0);

        // 2. De-whitelist the pool
        orderManager.setWhitelistedPool(poolId, false);
        assertFalse(orderManager.whitelistedPool(poolId));

        // 3. Normal order creation fails on de-whitelisted pool
        vm.startPrank(orderCreator);
        vm.expectRevert(NotWhitelistedPool.selector);
        limitOrderManager.createScaleOrders(
            true,
            60,
            180,
            1 ether,
            2,
            1 ether,
            poolKey
        );
        vm.stopPrank();

        // 4. Create a control pool (never whitelisted)
        PoolKey memory controlPoolKey;
        uint24 differentFee = 500;
        (controlPoolKey,) = initPool(
            currency0,
            currency1,
            IHooks(address(hook)),
            differentFee,
            tickSpacing,
            SQRT_PRICE_1_1
        );
        PoolId controlPoolId = controlPoolKey.toId();
        assertFalse(orderManager.whitelistedPool(controlPoolId));

        // 5. Setup keeper execution test
        ILimitOrderManager.PositionTickRange[] memory waitingPositions = new
        ↪  ILimitOrderManager.PositionTickRange[](2);
        waitingPositions[0] = ILimitOrderManager.PositionTickRange({
            bottomTick: 60,
            topTick: 120,
            isToken0: true
        });
        waitingPositions[1] = ILimitOrderManager.PositionTickRange({
            bottomTick: 120,
            topTick: 180,
            isToken0: true
        });

        // 6. Demonstrate vulnerability: execute on never-whitelisted pool
        vm.startPrank(keeper);
        // This should revert but doesn't because there's no whitelist check
        limitOrderManager.executeOrderByKeeper(controlPoolKey, waitingPositions);
        vm.stopPrank();

        // 7. Demonstrate vulnerability: execute on de-whitelisted pool
        vm.startPrank(keeper);
        // This should also revert but doesn't, demonstrating the vulnerability
        limitOrderManager.executeOrderByKeeper(poolKey, waitingPositions);
        vm.stopPrank();
    }
}
```

This test demonstrates that:

1. Order creation functions properly enforce the whitelist check.
2. The `executeOrderByKeeper()` function allows execution on both:
    - A pool that was never whitelisted.
    - A pool that was previously whitelisted but has been de-whitelisted.

Recommendation: Add the missing whitelist validation to the `executeOrderByKeeper()` function to maintain consistent security controls across all functions:

```
function executeOrderByKeeper(
    PoolKey calldata key,
    PositionTickRange[] memory waitingPositions
) external {
    require(isKeeper[msg.sender]);
    if (waitingPositions.length == 0) return;
    PoolId poolId = key.toId();

    // Add this missing check
    if (!whitelistedPool[poolId]) revert NotWhitelistedPool();

    // Rest of the function remains unchanged
    // ...
}
```

3.4.2 Keeper Functions Still Usable When Contract is Paused

Submitted by Maze

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: The `executeOrderByKeeper` function in the `LimitOrderManager` contract lacks the `whenNotPaused` modifier that is present on other critical functions. This allows keeper operations to continue even when the contract is paused, bypassing the emergency pause mechanism.

Description: The `LimitOrderManager` contract inherits from OpenZeppelin's `Pausable` contract to implement an emergency pause mechanism. When paused, most operations like creating orders and canceling orders are halted because they include the `whenNotPaused` modifier.

However, the `executeOrderByKeeper` function is missing this modifier. This oversight allows keepers to continue executing orders when the contract is paused, which violates the security assumption that all contract functionality should be suspended during a pause.

The pause functionality is a critical security mechanism intended to halt all operations in case of emergencies such as detected vulnerabilities, market anomalies, or other unexpected behaviors. By allowing keeper operations to continue during a pause, the contract fails to fully protect users from potential issues.

Impact: This vulnerability undermines the emergency pause functionality in the following ways:

1. During a security incident where the contract is paused to prevent further exploitation, keepers could still interact with the contract and execute orders, potentially worsening the situation.
2. If a critical vulnerability related to order execution is discovered, pausing the contract would not fully protect users as keeper-executed orders would still process.
3. If the contract is paused due to market anomalies (e.g., extreme volatility), keepers could still execute orders against users' interests.

The impact is not considered high because:

1. Keepers are typically trusted entities selected by the contract owner (via `setKeeper` function).
2. The vulnerability doesn't directly lead to fund loss but rather bypasses a security control.

Likelihood: The likelihood of this vulnerability being exploited is medium because:

1. It requires a keeper to intentionally or unknowingly execute orders during a pause event.

2. Pause events are relatively rare and typically only occur during emergency situations.
3. Keepers are often trusted entities, which reduces the chance of malicious exploitation.

However, automated keeper systems might continue running during a pause event if not properly notified, making unintentional exploitation possible.

Proof of Concept: I've created a test file that demonstrates how keeper operations can bypass the pause mechanism:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import {Deployers} from "@uniswap/v4-core/test/utils/Deployers.sol";
import {PoolSwapTest} from "v4-core/test/PoolSwapTest.sol";
import {IPoolManager} from "v4-core/interfaces/IPoolManager.sol";
import {PoolKey} from "v4-core/types/PoolKey.sol";
import {PoolId, PoolIdLibrary} from "v4-core/types/PoolId.sol";
import {Currency, CurrencyLibrary} from "v4-core/types/Currency.sol";
import {StateLibrary} from "v4-core/libraries/StateLibrary.sol";
import {TickMath} from "v4-core/libraries/TickMath.sol";
import {LimitOrderHook} from "src/LimitOrderHook.sol";
import {LimitOrderManager} from "src/LimitOrderManager.sol";
import {Hooks} from "v4-core/libraries/Hooks.sol";
import {IERC20Minimal} from "v4-core/interfaces/external/IERC20Minimal.sol";
import {BalanceDelta} from "v4-core/types/BalanceDelta.sol";
import {ILimitOrderManager} from "src/ILimitOrderManager.sol";
import {Pausable} from "@openzeppelin/contracts/utils/Pausable.sol";

/**
 * @title KeeperPauseVulnerabilityTest
 * @notice Test demonstrating the vulnerability where keeper functions can be called when contract is paused
 * @dev Shows that normal operations are blocked by pause but keeper operations still work
 */
contract KeeperPauseVulnerabilityTest is Test, Deployers {
    using PoolIdLibrary for PoolKey;
    using CurrencyLibrary for Currency;

    LimitOrderHook hook;
    ILimitOrderManager limitOrderManager;
    LimitOrderManager orderManager;
    address public treasury;
    PoolKey poolKey;
    address keeper;

    function setUp() public {
        deployFreshManagerAndRouters();
        (currency0, currency1) = deployMintAndApprove2Currencies();

        // Set up treasury address
        treasury = makeAddr("treasury");

        // First deploy the LimitOrderManager
        orderManager = new LimitOrderManager(
            address(manager), // The pool manager from Deployers
            treasury,
            address(this) // This test contract as owner
        );

        // Deploy hook with proper flags
        uint160 flags = uint160(
            Hooks.BEFORE_SWAP_FLAG |
            Hooks.AFTER_SWAP_FLAG
        );
        address hookAddress = address(flags);

        // Deploy the hook with the LimitOrderManager address
        deployCodeTo(
            "LimitOrderHook.sol",
            abi.encode(address(manager), address(orderManager), address(this)),
            hookAddress
        );
        hook = LimitOrderHook(hookAddress);

        // Set up manager interface and hook
        limitOrderManager = ILimitOrderManager(address(orderManager));
```

```
        limitOrderManager.setExecutablePositionsLimit(3); // Only execute 3 positions max
        limitOrderManager.setHook(address(hook));

        // Initialize pool with 1:1 price
        (poolKey,) = initPool(currency0, currency1, hook, 3000, SQRT_PRICE_1_1);

        orderManager.setWhitelistedPool(poolKey.toId(), true);

        // Approve tokens to manager
        IERC20Minimal(Currency.unwrap(currency0)).approve(address(limitOrderManager), type(uint256).max);
        IERC20Minimal(Currency.unwrap(currency1)).approve(address(limitOrderManager), type(uint256).max);

        // Add initial liquidity for testing
        modifyLiquidityRouter.modifyLiquidity(
            poolKey,
            IPoolManager.ModifyLiquidityParams({
                tickLower: -120,
                tickUpper: 120,
                liquidityDelta: 100 ether,
                salt: bytes32(0)
            }),
            ""
        );

        // Setup keeper
        keeper = makeAddr("keeper");
        limitOrderManager.setKeeper(keeper, true);

        // Setup initial amounts
        deal(Currency.unwrap(currency0), address(this), 100 ether);
        deal(Currency.unwrap(currency1), address(this), 100 ether);
    }

    /// @notice This test demonstrates the vulnerability where keeper can execute orders after contract is
    ↪  paused
    function test_keeper_execution_when_paused() public {
        // === STEP 1: Create orders and trigger a price change to make some positions wait for keeper ===
        console.log("1. Creating orders and triggering a price change");

        (, int24 currentTick,,) = StateLibrary.getSlot0(hook.poolManager(), poolKey.toId());

        // Create 5 orders at different price levels
        for(uint i = 0; i < 5; i++) {
            int24 tickOffset = int24(int256(120 * (i + 2))); // Using 120 and starting from 2
            limitOrderManager.createLimitOrder(true, (currentTick + tickOffset), 1 ether, poolKey);
        }

        // Perform swap that crosses prices and should mark positions for keeper execution
        swapRouter.swap(
            poolKey,
            IPoolManager.SwapParams({
                zeroForOne: false,
                amountSpecified: -10 ether,
                sqrtPriceLimitX96: TickMath.getSqrtPriceAtTick(currentTick + 1000) // Beyond all positions
            }),
            PoolSwapTest.TestSettings({
                takeClaims: false,
                settleUsingBurn: false
            }),
            ZERO_BYTES
        );

        // === STEP 2: Verify some positions are set for keeper execution ===
        PoolId poolId = poolKey.toId();
        ILimitOrderManager.PositionInfo[] memory positions = limitOrderManager.getUserPositions(address(this),
        ↪  poolId);

        // Count positions waiting for keeper
        uint256 waitingForKeeper = 0;
        for(uint i = 0; i < positions.length; i++) {
            (,, bool isActive, bool isWaitingKeeper,) = limitOrderManager.positionState(poolId,
            ↪  positions[i].positionKey);
            if(isWaitingKeeper) waitingForKeeper++;
        }

        // Should have positions waiting for keeper
```

```
        assertTrue(waitingForKeeper > 0, "Should have positions waiting for keeper");
        console.log("Positions waiting for keeper:", waitingForKeeper);

        // === STEP 3: Pause the contract ===
        console.log("2. Pausing the contract");
        orderManager.pause();
        assertTrue(orderManager.paused(), "Contract should be paused");

        // === STEP 4: Demonstrate that normal operations fail when paused ===
        console.log("3. Demonstrating that normal operations fail when paused");

        // Try to create new order - should fail when paused
        vm.expectRevert();
        limitOrderManager.createLimitOrder(true, (currentTick + 1200), 1 ether, poolKey);
        console.log("   Creating orders is blocked when paused");

        // === STEP 5: Demonstrate the vulnerability - keeper can still execute orders ===
        console.log("4. VULNERABILITY: Demonstrating that keeper can still execute orders when paused");
        ILimitOrderManager.PositionTickRange[] memory waitingPositions = getLeftoverPositions(poolId);

        // Execute as keeper - should work despite contract being paused!
        vm.prank(keeper);
        limitOrderManager.executeOrderByKeeper(poolKey, waitingPositions);

        // === STEP 6: Verify that keeper operations worked despite pause ===
        // Count positions waiting for keeper after keeper execution
        uint256 waitingAfterExecution = 0;
        for(uint i = 0; i < positions.length; i++) {
            (,, bool isActive, bool isWaitingKeeper,) = limitOrderManager.positionState(poolId,
            ↪  positions[i].positionKey);
            if(isWaitingKeeper) waitingAfterExecution++;
        }

        // Should have fewer positions waiting for keeper
        assertLt(waitingAfterExecution, waitingForKeeper, "Keeper execution should have reduced waiting
        ↪  positions");
        console.log("Positions waiting for keeper after execution:", waitingAfterExecution);

        console.log("\n======= VULNERABILITY SUMMARY =======");
        console.log("The executeOrderByKeeper() function lacks the whenNotPaused modifier");
        console.log("This allows keepers to execute orders even when the contract is paused");
        console.log("Severity: Medium - This bypasses an important security control");
    }

    /// Helper function to get positions waiting for keeper execution
    function getLeftoverPositions(PoolId poolId) internal view returns (ILimitOrderManager.PositionTickRange[]
    ↪  memory) {
        // Get user position keys
        ILimitOrderManager.PositionInfo[] memory positions = limitOrderManager.getUserPositions(address(this),
        ↪  poolId);

        // First count how many positions are waiting for keeper
        uint256 waitingCount = 0;
        for(uint i = 0; i < positions.length; i++) {
            (,, bool isActive, bool isWaitingKeeper,) = limitOrderManager.positionState(poolId,
            ↪  positions[i].positionKey);
            if(isWaitingKeeper) {
                waitingCount++;
            }
        }

        // Create array of correct size
        ILimitOrderManager.PositionTickRange[] memory leftoverPositions = new
        ↪  ILimitOrderManager.PositionTickRange[](waitingCount);

        // Fill array only with positions that are waiting for keeper
        uint256 posIndex = 0;
        for(uint i = 0; i < positions.length; i++) {
            (,, bool isActive, bool isWaitingKeeper,) = limitOrderManager.positionState(poolId,
            ↪  positions[i].positionKey);
            if(isWaitingKeeper) {
                (int24 bottomTick, int24 topTick, bool isToken0,) =
                ↪  _decodePositionKey(positions[i].positionKey);
                leftoverPositions[posIndex] = ILimitOrderManager.PositionTickRange({
                    bottomTick: bottomTick,
                    topTick: topTick,
```

```
                    isToken0: isToken0
                });
                posIndex++;
            }
        }

        return leftoverPositions;
    }

    /// Helper function to decode position key
    function _decodePositionKey(bytes32 key) internal pure returns (
        int24 bottomTick,
        int24 topTick,
        bool isToken0,
        uint256 nonce
    ) {
        uint256 value = uint256(key);
        return (
            int24(uint24(value >> 232)),          // bottomTick
            int24(uint24(value >> 208)),          // topTick
            (value & 1) == 1,                     // isToken0
            (value >> 8) & ((1 << 200) - 1)       // nonce (200 bits)
        );
    }
}
```

To run the proof of concept:

1. Save this file as `KeeperPauseVulnerability.t.sol` in the project's test directory.
2. Run the test with: `forge test --match-path "test/KeeperPauseVulnerability.t.sol" -v`.

The test successfully proves that `executeOrderByKeeper` can be called even when the contract is paused, demonstrating the vulnerability.

Recommendation: Add the `whenNotPaused` modifier to the `executeOrderByKeeper` function to ensure it respects the contract's.

3.4.3 Non Inclusion of actual target tick in calculated tick range causes spread on limit orders

Submitted by timefliez, also found by Hosam, Egbe, 0x15, 0xrafaelnicolau, 0xDeoGratias, kalyanSingh and Maaly

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: During the calculation of the `bottomTick` and `topTick` in `LimitOrderManager::createLimitOrder` the actual `targetTick` is not included inside the range of `[bottomTick; topTick]` causing "spread" during the execution of such orders:

Finding Description: `TickLibrary::getValidTickRange`:

```
function getValidTickRange(
    int24 currentTick,
    int24 targetTick,
    int24 tickSpacing,
    bool isToken0
) public pure returns (int24 bottomTick, int24 topTick) {

    ////////////////////////////////////////////////
    ////////////////// SNIP ////////////////////////
    ////////////////////////////////////////////////

    if(isToken0) {
        if(roundedCurrentTick >= roundedTargetTick)
        revert RoundedTargetTickLessThanRoundedCurrentTick(currentTick, roundedCurrentTick, targetTick,
        ↪  roundedTargetTick);
        topTick = roundedTargetTick; // <<< 1
        bottomTick = topTick - tickSpacing; // <<< 2
    } else {
        if(roundedCurrentTick <= roundedTargetTick)
        revert RoundedTargetTickGreaterThanRoundedCurrentTick(currentTick, roundedCurrentTick, targetTick,
        ↪  roundedTargetTick);
        bottomTick = roundedTargetTick;
        topTick = bottomTick + tickSpacing;
    }
    ////////////////////////////////////////////////
    ////////////////// SNIP ////////////////////////
    ////////////////////////////////////////////////
}
```

`TickLibrary::getRoundedTargetTick`:

```
function getRoundedTargetTick(
    int24 targetTick,
    bool isToken0,
    int24 tickSpacing
) internal pure returns(int24 roundedTargetTick) {
    if (isToken0) {
        roundedTargetTick = targetTick >= 0 ?
        (targetTick / tickSpacing) * tickSpacing : // <<< 3
        ((targetTick % tickSpacing == 0) ? targetTick : ((targetTick / tickSpacing) - 1) * tickSpacing);
    } else {
        roundedTargetTick = targetTick < 0 ?
        (targetTick / tickSpacing) * tickSpacing :
        ((targetTick % tickSpacing == 0) ? targetTick : ((targetTick / tickSpacing) + 1) * tickSpacing);
    }
}
```

As you can see under 3. in the provided code above, we calculate the rounded target tick by dividing the `targetTick` with the `tickSpacing` which will produce the lower end of the tick boundary of the target tick.

However, considering now 1. and 2. in provided code above, this result is treated as the upper bound of the valid tick range, effectively excluding the user selected `targetTick` from the tick range the order will be executed in.

Consider the following scenario (provided in the PoC below):: A user wants to swap 20e18 tokens in currency 0 in a 1:1 initiated pool, tick spacing 60. He sets a price limit of `1.2e18` per token0. If the current tick is expected to be Tick 0, this would result in his target tick being 1799. Following above quoted codes math:

```
1799 / 60 = 29 (in solidity)
29 * 60   = 1740 <= rounded target tick
```

Now in 1. and 2. in above code we will now deduct the tick spacing to calculate the valid range, reulsting in a tick range of `[1680;1740]` excluding the users target price and target tick. The Result is that the user will receive less amounts of token1 than expected and set in the limit order.

Impact Explanation: The spread showcased in the PoC below might not seem like a lot, however, since we are talking about a Limit Order system, not an AMM, it can not be neglected. Users of limit orders actually do so to avoid the spreads of market buy/sell orders, therefore it is crucial that orders will actually by executed on the `topTick` within the range, which does include the selected `targetTick`. A Medium Impact seems justified considering a relatively low spread.

Likelihood Explanation: The likelihood is High, since no preconditions need to be met and this issue occurs every time a limit order is created as long as `targetTick % tickSpacing != 0`.

Proof of Concept: Please copy the following code into a file `PoC.t.sol` within the `./test` directory and execute:

```
forge test --mt test_targetTickNotIncludedInRange -vv
```

```
//SPDX-License-Identifier: MIT

pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "forge-std/console.sol";
import {Deployers} from "@uniswap/v4-core/test/utils/Deployers.sol";
import {PoolSwapTest} from "v4-core/test/PoolSwapTest.sol";
import {IPoolManager} from "v4-core/interfaces/IPoolManager.sol";
import {PoolKey} from "v4-core/types/PoolKey.sol";
import {PoolId, PoolIdLibrary} from "v4-core/types/PoolId.sol";
import {Currency, CurrencyLibrary} from "v4-core/types/Currency.sol";
import {StateLibrary} from "v4-core/libraries/StateLibrary.sol";
import {TickMath} from "v4-core/libraries/TickMath.sol";
import {LimitOrderHook} from "src/LimitOrderHook.sol";
import {LimitOrderManager} from "src/LimitOrderManager.sol";
import {Hooks} from "v4-core/libraries/Hooks.sol";
import {IERC20Minimal} from "v4-core/interfaces/external/IERC20Minimal.sol";
import {BalanceDelta} from "v4-core/types/BalanceDelta.sol";
import {LiquidityAmounts} from "@uniswap/v4-core/test/utils/LiquidityAmounts.sol";
import {ILimitOrderManager} from "src/ILimitOrderManager.sol";
import {LimitOrderLens} from "src/LimitOrderLens.sol";
import "../src/TickLibrary.sol";
import {ERC20Mock} from "@openzeppelin/contracts/mocks/token/ERC20Mock.sol";

contract PoC is Test, Deployers {

    using PoolIdLibrary for PoolKey;
    using CurrencyLibrary for Currency;
    uint256 public HOOK_FEE_PERCENTAGE = 50000;
    uint256 public constant FEE_DENOMINATOR = 100000;
    uint256 internal constant Q128 = 1 << 128;
    LimitOrderHook hook;
    ILimitOrderManager limitOrderManager;
    LimitOrderManager orderManager;
    LimitOrderLens lens;
    address public treasury;
    PoolKey poolKey;

    function setUp() public {
        deployFreshManagerAndRouters();
        (currency0, currency1) = deployMintAndApprove2Currencies();
        treasury = makeAddr("treasury");
        orderManager = new LimitOrderManager(
            address(manager),
            treasury,
            address(this)
        );
        uint160 flags = uint160(
            Hooks.BEFORE_SWAP_FLAG |
            Hooks.AFTER_SWAP_FLAG
        );
        address hookAddress = address(flags);
        deployCodeTo(
            "LimitOrderHook.sol",
            abi.encode(address(manager), address(orderManager), address(this)),
            hookAddress
        );
        hook = LimitOrderHook(hookAddress);
        limitOrderManager = ILimitOrderManager(address(orderManager));
        lens = new LimitOrderLens(
            address(orderManager),
            address(this)
        );
        limitOrderManager.setExecutablePositionsLimit(5);
        limitOrderManager.setHook(address(hook));
        (poolKey,) = initPool(currency0, currency1, hook, 3000, SQRT_PRICE_1_1);
        orderManager.setWhitelistedPool(poolKey.toId(), true);
```

```
        IERC20Minimal(Currency.unwrap(currency0)).approve(address(limitOrderManager), type(uint256).max);
        IERC20Minimal(Currency.unwrap(currency1)).approve(address(limitOrderManager), type(uint256).max);
    modifyLiquidityRouter.modifyLiquidity(
        poolKey,
        IPoolManager.ModifyLiquidityParams({
            tickLower: -887220,
            tickUpper: 887220,
            liquidityDelta: 100 ether,
            salt: bytes32(0)
        }),
        ""
        );
    }

    function test_targetTickNotIncludedInRange() public {
        address user = makeAddr("user");
        ERC20Mock token0 = ERC20Mock(Currency.unwrap(currency0));
        ERC20Mock token1 = ERC20Mock(Currency.unwrap(currency1));
        token0.mint(user, 20e18);
        assertEq(20e18, token0.balanceOf(user));
        assertEq(0, token1.balanceOf(user));
        uint256 price = 1.2e18;
        uint256 amount = 20e18;
        uint256 roundedPrice = TickLibrary.getRoundedPrice(price, poolKey, true);
        int24 targetTick = TickMath.getTickAtSqrtPrice(TickLibrary.getSqrtPriceFromPrice(roundedPrice));
        vm.startPrank(user);
        token0.approve(address(limitOrderManager), 20e18);
        LimitOrderManager.CreateOrderResult memory result = limitOrderManager.createLimitOrder(true,
        ↪  targetTick, amount, poolKey);
        vm.stopPrank();
        console.log("=============================================================================================
        ↪  =");
        console.log("Target Tick found corresponding to price: ", targetTick);
        console.log("Calculated Bottom Tick for Limit Order: ", result.bottomTick);
        console.log("Calculated Top Tick for Limit Order: ", result.topTick);
        console.log("=============================================================================================
        ↪  =");

        swapRouter.swap(
        poolKey,
        IPoolManager.SwapParams({
        zeroForOne: false,
        amountSpecified: -40 ether,
        sqrtPriceLimitX96: TickMath.getSqrtPriceAtTick(result.topTick + 100)
        }),
        PoolSwapTest.TestSettings({
        takeClaims: false,
        settleUsingBurn: false
        }),
        ""
        );
        LimitOrderManager.PositionInfo[] memory position = limitOrderManager.getUserPositions(user,
        ↪  poolKey.toId());
        vm.prank(user);
        limitOrderManager.claimOrder(poolKey, position[0].positionKey, user);
        console.log("=============================================================================================
        ↪  =");
        console.log("Expected Amount out in currency 1 for set limit order: ", (amount * price) / 1e18);
        console.log("Actual Amount out received in currency 1 for set limit order: ", token1.balanceOf(user));
        console.log("Collected Treasury Fees for limit order execution: ", token1.balanceOf(treasury));
        console.log("Sanity Check to showcase its not the fees building the difference: ", 24e18 -
        ↪  token1.balanceOf(user) - token1.balanceOf(treasury));
        console.log("=============================================================================================
        ↪  =");
    }
}
```


Executing above code will produce the following log:

```
Ran 1 test for test/PoC.t.sol:PoC
[PASS] test_targetTickNotIncludedInRange() (gas: 830839)
Logs:
  ================================================================================================
  Target Tick found corresponding to price:  1799
  Calculated Bottom Tick for Limit Order:  1680
  Calculated Top Tick for Limit Order:  1740
  ================================================================================================
  ================================================================================================
  Expected Amount out in currency 1 for set limit order:  24000000000000000000
  Actual Amount out received in currency 1 for set limit order:  23765313627633905976
  Collected Treasury Fees for limit order execution:  35701522725537307
  Sanity Check to showcase its not the fees building the difference:  198984849640556717
  ================================================================================================

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.47ms (862.21µs CPU time)

Ran 1 test suite in 93.81ms (3.47ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Showcasing that indeed in this scenario a >1% spread (even after fee deduction) occured.

Recommendation: The issue could potentially be fixed in 2 places.

1. `TickLibrary::getValidTickRange`.
2. `TickLibrary::getRoundedTargetTick`.

However, fixing it within `getRoundedTargetTick` could have implications on other parts within the protocol, therefore I will focus on `getValidTickRange`:

```
  function getValidTickRange(
      int24 currentTick,
      int24 targetTick,
      int24 tickSpacing,
      bool isToken0
  ) public pure returns (int24 bottomTick, int24 topTick) {
      if(isToken0 && currentTick >= targetTick)
      revert WrongTargetTick(currentTick, targetTick, true);
      if(!isToken0 && currentTick <= targetTick)
      revert WrongTargetTick(currentTick, targetTick, false);

      int24 roundedTargetTick = getRoundedTargetTick(targetTick, isToken0, tickSpacing);
      int24 roundedCurrentTick = getRoundedCurrentTick(currentTick, isToken0, tickSpacing);
      int24 tickDiff = roundedCurrentTick > roundedTargetTick ?
      roundedCurrentTick - roundedTargetTick :
  roundedTargetTick - roundedCurrentTick;
      if(tickDiff < tickSpacing)
      revert RoundedTicksTooClose(currentTick, roundedCurrentTick, targetTick, roundedTargetTick, isToken0);
      if(isToken0) {
          if(roundedCurrentTick >= roundedTargetTick)
          revert RoundedTargetTickLessThanRoundedCurrentTick(currentTick, roundedCurrentTick, targetTick,
          ↪  roundedTargetTick);
-         topTick = roundedTargetTick;
-         bottomTick = topTick - tickSpacing;
+         bottomTick = roundedTargetTick;
+         topTick = roundedTargetTick + tickSpacing;
      } else {
          if(roundedCurrentTick <= roundedTargetTick)
          revert RoundedTargetTickGreaterThanRoundedCurrentTick(currentTick, roundedCurrentTick, targetTick,
          ↪  roundedTargetTick);
          bottomTick = roundedTargetTick;
          topTick = bottomTick + tickSpacing;
      }
      if(bottomTick >= topTick)
      revert SingleTickWrongTickRange(bottomTick, topTick, currentTick, targetTick, isToken0);
      if (bottomTick < minUsableTick(tickSpacing) || topTick > maxUsableTick(tickSpacing))
      revert TickOutOfBounds(targetTick);
  }
```

This fix will ensure orders will be executed more towards the users expectations, reducing the spread and boosting protocol revenue.

3.4.4 Proportional Tick Distribution Conflicts with Strict Order Density Check

Submitted by willycode20

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: The `validateAndPrepareScaleOrders` function attempts to distribute multiple limit orders across a tick range using a proportional allocation mechanism. However, a flawed validation check on the number of allowable orders may incorrectly revert valid inputs due to a mismatch between the proportional spacing logic and the enforced maximum density constraint.

Finding Description: The function distributes `totalOrders` across the `effectiveRange = topTick - bottomTick - tickSpacing` using a proportional strategy. The logic approximates each order's position using:

```
bottomTick + int24(uint24((i * uint256(uint24(effectiveRange))) / (totalOrders - 1)))
```

To prevent excessive density, the function includes this check:

```
if (totalOrders > uint256(uint24((topTick - bottomTick) / tickSpacing)))
    revert MaxOrdersExceeded();
```

However, this validation assumes linear (non-overlapping) spacing. In practice, the rounding of each proportional position to the nearest multiple of `tickSpacing` can lead to overlapping. Therefore, valid configurations that technically fit within the tick range may be rejected unnecessarily.

Impact: Users may be prevented from placing valid sets of limit orders within a specified tick range. This restricts functionality, particularly for strategies requiring dense order placement or when operating within narrow tick ranges. In some cases, the failure to pass this check can render certain strategies non-functional.

Likelihood: Medium — The issue will surface frequently when:

1. The tick range is relatively narrow.
2. The `tickSpacing` is large (e.g., wide-range pools).
3. The `totalOrders` is moderately high (e.g., >5), or...
4. Users attempt to perform overlapping order placement.

Proof Of Concept: Copy and paste this test function in the `ScaleOrders.t.sol`:

```
function test_createScaleOrders_vulnerability() public {
    deal(Currency.unwrap(currency0), address(this), 100 ether);
    deal(Currency.unwrap(currency1), address(this), 100 ether);

    // Get current tick
    (, int24 currentTick,,) = StateLibrary.getSlot0(hook.poolManager(), poolKey.toId());
    console.log("Current tick:", currentTick);

    // Define parameters for the test
    int24 bottomTick = currentTick + 60; // Bottom tick just above current tick
    int24 topTick = currentTick + 300;  // Top tick within a valid range
    uint256 totalOrders = 5;            // Number of orders
    uint256 totalAmount = 10 ether;     // Total amount to distribute
    uint256 sizeSkew = 1e18;            // No skew

    // Attempt to create scale orders
    vm.expectRevert(TickLibrary.MaxOrdersExceeded.selector);
    limitOrderManager.createScaleOrders(
        true,        // isToken0
        bottomTick,  // Bottom tick
        topTick,     // Top tick
        totalAmount, // Total amount
        totalOrders, // Total orders
        sizeSkew,    // Size skew
        poolKey      // Pool key
    );

    console.log("Test failed due to MaxOrdersExceeded, even though tick distribution can accommodate the
    ↪  orders.");
}
```

The function will revert with `MaxOrdersExceeded`, but upon implementing the suggested modification this order passes successfully.

Recommendation: Update the density validation check to account for the actual proportional spacing strategy. A preferably fix is:

```
if (totalOrders - 1 > uint256(uint24((topTick - bottomTick) / tickSpacing)))
    revert MaxOrdersExceeded();
```

Alternatively, if orders must be non-overlapping. Modify below. `orderBottomTick = bottomTick + int24(i) * tickSpacing;`.

3.4.5 Keeper only emergency cancel orders

Submitted by 0xAlexSR

Severity: Informational

Context: LimitOrderManager.sol#L300-L314

Summary: The `emergencyCancelOrders` function in `LimitOrderManager` is restricted to addresses marked as "keepers" and cannot be called by the contract owner, even in emergencies. This design flaw prevents the owner from intervening in critical situations and gives excessive power to keepers which isn't a trusted role following the OOS description.

Finding Description: The `emergencyCancelOrders` function is intended as an emergency tool to allow for the cancellation of user orders in exceptional circumstances. However, the function is protected by the following check:

```
require(isKeeper[msg.sender]);
```

This means only addresses explicitly set as keepers can call this function. The contract owner, even though they have ultimate authority over the contract, cannot call this function unless they are also set as a keeper.

This design breaks the expected security model where the owner (or a governance multisig) should have the ability to perform this emergency action. In this case, the owner is powerless to intervene. And, if a malicious keeper is set, they have unilateral power to cancel any user's orders, while the owner cannot act at all.

Impact Explanation:

- The owner cannot directly perform emergency order cancelation.
- Malicious keepers can cancel any user orders.

Likelihood Explanation:

- The issue will always occur unless the owner is also set as a keeper.
- It is an emergency function that will probably rarely be called.

Proof of Concept: Add the following test in `BasicOrderTest.t.sol`:

```
function test_owner_cannot_emergency_cancel_orders() public {
    // setup
    deal(Currency.unwrap(currency0), address(this), 100 ether);
    deal(Currency.unwrap(currency1), address(this), 100 ether);
    uint256 roundedPrice = TickLibrary.getRoundedPrice(1.02e18, poolKey, true);
    int24 targetTick = TickMath.getTickAtSqrtPrice(TickLibrary.getSqrtPriceFromPrice(roundedPrice));
    LimitOrderManager.CreateOrderResult memory result = limitOrderManager.createLimitOrder(true, targetTick, 1
    ↪  ether, poolKey);
    bytes32 baseKey = getBasePositionKey(result.bottomTick, result.topTick, result.isToken0);
    PoolId poolId = poolKey.toId();
    uint256 nonce = limitOrderManager.currentNonce(poolId, baseKey);
    bytes32 positionKey = getPositionKey(result.bottomTick, result.topTick, result.isToken0, nonce);

    // owner is address(this), prank isn't needed
    assertEq(LimitOrderManager(address(limitOrderManager)).owner(), address(this));

    // fails if owner calls emergencyCancelOrders
    vm.expectRevert();
    LimitOrderManager(address(limitOrderManager)).emergencyCancelOrders(poolKey, address(this), 0, 2);

    // keeper can call emergencyCancelOrders
    limitOrderManager.setKeeper(address(this), true);
    LimitOrderManager(address(limitOrderManager)).emergencyCancelOrders(poolKey, address(this), 0, 2);
}
```

Recommendation: Remove the keeper check and use `onlyOwner` decorator.