

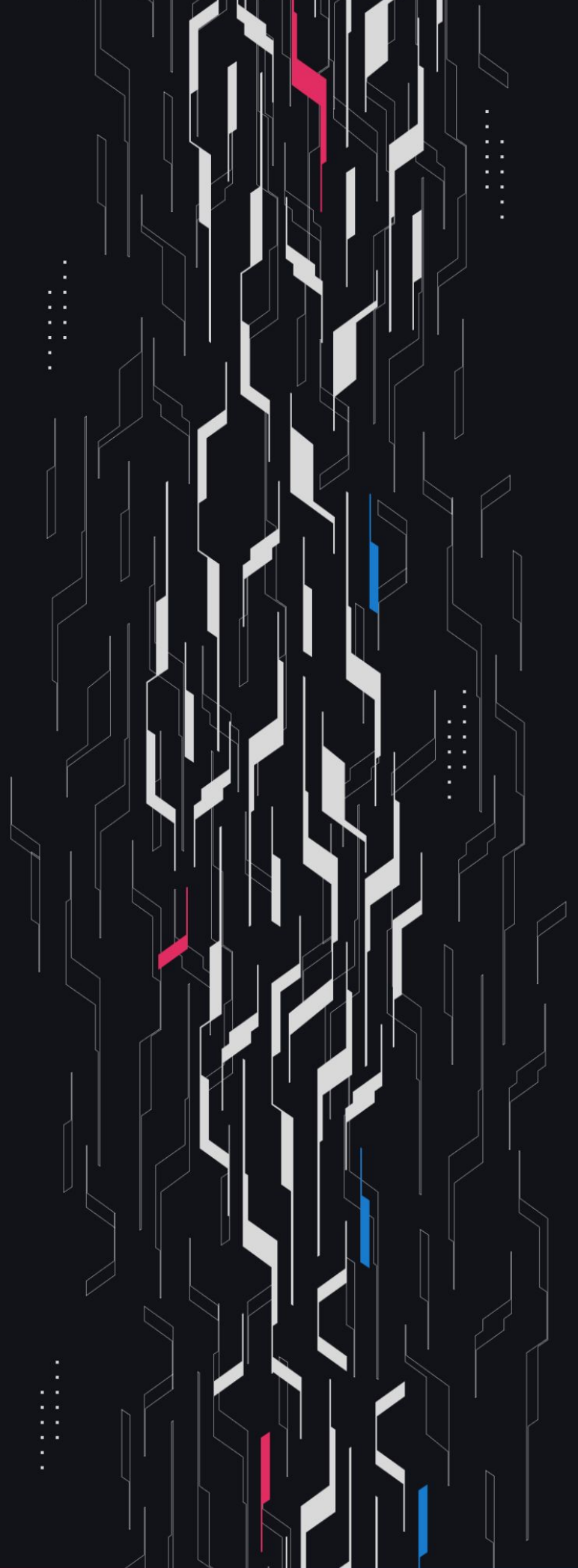
GA GUARDIAN

Gamma

UniV4 Limit Orders

Security Assessment

April 14th, 2025



Summary

Audit Firm Guardian

Prepared By Owen Thurm, Daniel Gelfand, Django, Ilchovski, Smbv, Vladimir Zotov

Client Firm Gamma

Final Report Date April 14, 2025

Audit Summary

Gamma engaged Guardian to review the security of their limit order system using UniswapV4 hooks. From the 5th of March to the 17th of March, a team of 6 auditors reviewed the source code in scope. All findings have been recorded in the following report.

Issues Detected Throughout the engagement 12 High/Critical issues were uncovered and promptly remediated by the Gamma team.

Security Recommendation Given the number of High and Critical issues detected as well as additional code changes made after the main review, Guardian recommends that an independent security review of the protocol at a finalized frozen commit is conducted before deployment.

For a detailed understanding of risk severity, source code vulnerability, and potential attack vectors, refer to the complete audit report below.

✓ Verify the authenticity of this report on Guardian's GitHub: <https://github.com/guardianaudits>

📊 Code coverage & PoC test suite: <https://github.com/GuardianOrg/univ4-limit-order-hook-fuzz>

Table of Contents

Project Information

Project Overview 4

Audit Scope & Methodology 5

Smart Contract Risk Assessment

Invariants Assessed 7

Findings & Resolutions 10

Addendum

Disclaimer 57

About Guardian Audits 58

Project Overview

Project Summary

Project Name	Gamma
Language	Solidity
Codebase	https://github.com/GammaStrategies/univ4-limit-order-hook
Commit(s)	Initial commit: b9e157b1952193edef4824219f6f3f91a59514e6 Final commit: 5f3d853106b2f5808c2b5d2773d7114532520b30

Audit Summary

Delivery Date	April 14, 2025
Audit Methodology	Static Analysis, Manual Review, Test Suite, Contract Fuzzing

Vulnerability Summary

Vulnerability Level	Total	Pending	Declined	Acknowledged	Partially Resolved	Resolved
● Critical	7	0	0	0	0	7
● High	5	0	0	0	0	5
● Medium	3	0	0	2	0	1
● Low	28	0	0	3	0	25

Audit Scope & Methodology

Vulnerability Classifications

Severity	Impact: <i>High</i>	Impact: <i>Medium</i>	Impact: <i>Low</i>
Likelihood: <i>High</i>	● Critical	● High	● Medium
Likelihood: <i>Medium</i>	● High	● Medium	● Low
Likelihood: <i>Low</i>	● Medium	● Low	● Low

Impact

- High**

Significant loss of assets in the protocol, significant harm to a group of users, or a core functionality of the protocol is disrupted.
- Medium**

A small amount of funds can be lost or ancillary functionality of the protocol is affected. The user or protocol may experience reduced or delayed receipt of intended funds.
- Low**

Can lead to any unexpected behavior with some of the protocol's functionalities that is notable but does not meet the criteria for a higher severity.

Likelihood

- High**

The attack is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount gained or the disruption to the protocol.
- Medium**

An attack vector that is only possible in uncommon cases or requires a large amount of capital to exercise relative to the amount gained or the disruption to the protocol.
- Low**

Unlikely to ever occur in production.

Audit Scope & Methodology

Methodology

Guardian is the ultimate standard for Smart Contract security. An engagement with Guardian entails the following:

- Two competing teams of Guardian security researchers performing an independent review.
- A dedicated fuzzing engineer to construct a comprehensive stateful fuzzing suite for the project.
- An engagement lead security researcher coordinating the 2 teams, performing their own analysis, relaying findings to the client, and orchestrating the testing/verification efforts.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross-referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.
Comprehensive written tests as a part of a code coverage testing suite.
- Contract fuzzing for increased attack resilience.

Invariants Assessed

During Guardian’s review of Gamma, fuzz-testing was performed on the protocol’s main functionalities. Given the dynamic interactions and the potential for unforeseen edge cases in the protocol, fuzz-testing was imperative to verify the integrity of several system invariants.




























Throughout the engagement the following invariants were assessed for a total of 20,000,000+ runs with a prepared fuzzing suite.

ID	Description	Tested	Passed	Remediation	Run Count
GLOB-01	positionTickRangeList ranges have to be sorted in ascending order of bottom tick	✓	✓	✓	20M+
GLOB-02	The sum of user fees for a position never exceeds the total fees	✓	✓	✓	20M+
GLOB-03	All position ticks are within the valid range	✓	✓	✓	20M+
GLOB-04	At all times, calling _findOverlappingPositions should always return all of the positions which are executable by the tickAfterSwap	✓	✗	✓	20M+
GLOB-05	In all positions, bottomTick must be strictly less than topTick	✓	✓	✓	20M+
GLOB-06	In all positions, bottomTick must be greater than or equal to minUsableTick(tickSpacing)	✓	✓	✓	20M+
GLOB-07	In al positions, topTick must be less than or equal to maxUsableTick(tickSpacing)	✓	✓	✓	20M+
GLOB-08	Top and bottom ticks must be rounded according to tickSpacing	✓	✓	✓	20M+
GLOB-09	getUserFees should accurately return each user's fair share of fees	✓	✗	✓	20M+

Invariants Assessed

ID	Description	Tested	Passed	Remediation	Run Count
GLOB-10	Position's total liquidity equals the sum of all user liquidities	✓	✗	✓	20M+
GLOB-11	Order can be executed only when the order is sold completely	✓	✗	✓	20M+
GLOB-12	A position cannot be simultaneously active and have claimable principal	✓	✓	✓	20M+
GLOB-13	After execution, a position should be removed from `positionTickRangeList`	✓	✓	✓	20M+
GLOB-14	Token0 positions (isToken0=true) should only have token1 principal and vice versa	✓	✓	✓	20M+
GLOB-15	When the last contributor to a position removes their liquidity, the position is removed from positionTickRangeList	✓	✓	✓	20M+
GLOB-16	The positionContributors set ensures each user is counted only once per position	✓	✓	✓	20M+
GLOB-17	An order that is executed should not be able to be executed again.	✓	✓	✓	20M+
GLOB-18	An order that is executed should always have ≥ 1 positionContributors	✓	✓	✓	20M+
CREATE-01	No limit order is created with an amount less than the minimum amount	✓	✓	✓	20M+
CREATE-02	T0: currentTick must be strictly less than targetTick	✓	✓	✓	20M+

Invariants Assessed

ID	Description	Tested	Passed	Remediation	Run Count
CREATE-03	T0: The difference between ticks must be at least tickSpacing				20M+
CREATE-04	T1: currentTick must be strictly greater than targetTick				20M+
CREATE-05	T1: The difference between ticks must be at least tickSpacing				20M+
CREATE-06	T0: bottomTick must be strictly less than upperTick				20M+
CREATE-07	T0: The difference between ticks must be at least tickSpacing				20M+
CREATE-08	T1: upperTick must be strictly greater than bottomTick				20M+
CREATE-09	T1: The difference between ticks must be at least tickSpacing				20M+
CANCEL-01	Token0 and Token1 balances of User after claim and cancel should not differ				20M+
CANCEL-02	After cancellation, a position must either have claimable principal or be fully claimed				20M+

Findings & Resolutions

ID	Title	Category	Severity	Status
C-01	Anyone Can Trigger Limit Orders	Access Control	● Critical	Resolved
C-02	Position Liquidity Unaltered	Logical Error	● Critical	Resolved
C-03	Range Orders Arbitraged	Validation	● Critical	Resolved
C-04	Limit Orders Missed Due	Logical Error	● Critical	Resolved
C-05	Wrongful Liquidity Burn	Logical Error	● Critical	Resolved
C-06	No Access Control	Access Control	● Critical	Resolved
C-07	User Remains Position Contributor	Logical Error	● Critical	Resolved
H-01	Pools With Low SqrtPrices Are Unusable	Rounding	● High	Resolved
H-02	Incorrect Positions Removed	Logical Error	● High	Resolved
H-03	Execution Of Orders Can Be Skipped	Logical Error	● High	Resolved
H-04	Order Creation And Execution Can Be DoSed	DoS	● High	Resolved
H-05	Batch Cancels Cause Users To Cancel Incorrectly	Logical Error	● High	Resolved
M-01	Keeper Frontrunning	Frontrunning	● Medium	Acknowledged

Findings & Resolutions

ID	Title	Category	Severity	Status
M-02	Permissionless Use Of Pools	Validation	● Medium	Resolved
M-03	Updating Fee Values Affects Fees	Logical Error	● Medium	Acknowledged
L-01	Insufficient Event Data	Events	● Low	Resolved
L-02	Lacking Keeper Execution Validation	Validation	● Low	Acknowledged
L-03	Scale Order Validation Does Not Take Place	Logical Error	● Low	Resolved
L-04	Lacking safeTransferFrom Usage	Best Practices	● Low	Resolved
L-05	Lacking Zero Address Validations	Validation	● Low	Resolved
L-06	Unnecessary Position State	Superfluous Code	● Low	Resolved
L-07	Lacking setExecutablePositionsLimit Validation	Validation	● Low	Resolved
L-08	Unnecessary Repeated Minting	Gas Optimization	● Low	Resolved
L-09	Orders Can Be Made For Unconnected Pools	Validation	● Low	Resolved
L-10	Refunds Missed For Non-Native Currencies	Warning	● Low	Resolved
L-11	Unused Errors	Best Practices	● Low	Resolved

Findings & Resolutions

ID	Title	Category	Severity	Status
L-12	Incorrect getUserClaimableBalances Result	Logical Error	● Low	Resolved
L-13	Sync DoS Attack	DoS	● Low	Resolved
L-14	Follow Import Best Practices	Best Practices	● Low	Pending
L-15	Follow CEI Pattern When Claiming Positions	Best Practices	● Low	Resolved
L-16	Random Users Can Spam LimitOrderClaimed Events	Validation	● Low	Resolved
L-17	Unnecessary Reads From Storage	Gas Optimization	● Low	Acknowledged
L-18	Integrators May Receive Unexpected Token	Validation	● Low	Acknowledged
L-19	maxOrderLimit Behaviour Differs From Comment	Informational	● Low	Resolved
L-20	Variable Denoted As Constant	Best Practices	● Low	Resolved
L-21	Keeper Execution DoS'd	Logical Error	● Low	Resolved
L-22	Unchecked Msg.value In Token Transfer	Validation	● Low	Resolved
L-23	Max Limit Can Prevent Scale Orders	Validation	● Low	Pending
L-24	Return Value Of transferFrom() Not Checked	Best Practices	● Low	Resolved

Findings & Resolutions

ID	Title	Category	Severity	Status
L-25	_executePosition Does Not Reset isWaitingKeeper	Unexpected Behavior	● Low	Resolved
L-26	Min/Max Tick Validation In Scale Orders	Validation	● Low	Resolved
L-27	Use nonReentrant() Modifier	Best Practices	● Low	Resolved
L-28	Incorrect Current Nonce On Position	Logical Error	● Low	Resolved

C-01 | Anyone Can Trigger Limit Orders

Category	Severity	Location	Status
Access Control	● Critical	LimitOrderManager.sol: 394	Resolved

Description

The `executeOrder` function is external with no validation performed on the `tickBeforeSwap` and `tickAfterSwap` values to ensure that they align with the pool state.

As a result a malicious actor can trigger limit orders when they are not actually filled causing accounting issues within the system and of course defeating the purpose of a limit order.

Recommendation

Add access controls to the `executeOrder` function such that it is only callable by the `LimitOrderHook` contract.

Resolution

Gamma Team: The issue was resolved in [LimitOrderManager.sol#L550](#).

C-02 | Position Liquidity Unaltered

Category	Severity	Location	Status
Logical Error	● Critical	LimitOrderManager.sol: 255	Resolved

Description [PoC](#)

In the `cancelOrder` function there is no logic to reduce the `totalLiquidity` amount stored on the position such as in the `_updateUserPosition` function.

As a result, as soon as one user cancels an order in a given range, all other users will be unable to be executed and swaps in the pool will be DoS'd past that tick as the callback reverts due to attempting to burn more liquidity than exists in the position.

Recommendation

Introduce logic into the `cancelOrder` function such that the `totalLiquidity` of the position is reduced in line with the amount that was burnt for the user.

Resolution

Gamma Team: The issue was resolved in [LimitOrderManager.sol#L415](#).

C-03 | Range Orders Arbitraged

Category	Severity	Location	Status
Validation	● Critical	LimitOrderManager.sol	Resolved

Description

The creation of range orders starts from the current pool price using a rounded tick. However the current pool rounded tick at the time in which the user decides to send their range may be significantly different then the current pool price when their order creation transaction is recorded in a block.

The Uniswap V4 pool that is used for Gamma limit orders is distinct from other Uniswap V4 liquidity sources, therefore it is entirely possible that there is little or no liquidity below the current market price when a limit order creation transaction is submitted to the mempool by a user.

A malicious actor can therefore swap with 0 input amount and a `sqrtPriceLimit` set to an extremely low price to move the pool price very low directly before a user’s range is created.

After the user’s orders are created, liquidity will now be available in the pool at a very low price for the provided token. The malicious actor can now buy up this provided liquidity from the user at a very advantageous price to net a profit.

Recommendation

Consider allowing users to specify a minimum and maximum tick which they will accept the range, similar to a slippage tolerance.

Resolution

Gamma Team: The issue was resolved in commit [2e003c8](#).

C-04 | Limit Orders Missed Due

Category	Severity	Location	Status
Logical Error	● Critical	LimitOrderManager.sol: 580	Resolved

Description [PoC](#)

In the `_findOverlappingPositions` function the search for executable orders is halted as soon as a position is encountered which has an end tick after the resulting pool price and is therefore unexecutable.

However, for `zeroForOne` limit orders, since the `positionTickRangeList` is sorted by the bottom tick this results in orders often being skipped when they are in fact executable.

Consider the following sorted `positionTickRangeList`.

0: Position(lower: 60, 300)

1: Position(lower: 120, 360)

2: Position(lower: 180, 300)

When the pool price swaps to tick 320 the loop in the `_findOverlappingPositions` function will break because the order in the first index has a top tick which is above the resulting pool price. However the subsequent order in index 2 should have been included in the `positionsId` list and executed.

Recommendation

Consider implementing a combination of a bitmap to show ticks which are the end tick for at least one order in the Gamma system and a mapping from tick to a set of positions which end at that tick.

Resolution

Gamma Team: The issue was resolved in [LimitOrderManager.sol#L696](#).

C-05 | Wrongful Liquidity Burn

Category	Severity	Location	Status
Logical Error	● Critical	LimitOrderManager.sol: 266C9-270C10	Resolved

Description [PoC](#)

When a position is executed, all of the liquidity at that position is burned. A user can also cancel their order after a position has been executed, which also burns the user's share of liquidity.

Therefore, the following scenario can occur which will result in permanent loss of funds:

- User1 creates limit order at (60,120)
- Swap pushes to tick 180, limit order is executed. Liquidity is burned.
- Swap pushes the price back down to tick 0.
- User2 creates limit order at (60,120)
- User1 can cancel their original limit order that's already been executed.
- User2's liquidity has been burned and they cannot cancel their order. Orders at this position will revert when executed, also causing system-wide DOS for swaps around these ticks.

Recommendation

The early return `userPositions[poolId][positionKey][user].claimablePrincipal != ZERO_DELTA` needs to be amended to also include if the position is not active.

This if clause can never be reached because `claimablePrinciple` is never non-zero at this point. The only check should be for `position.isActive` to indicate whether the position has been executed or not.

Resolution

Gamma Team: The issue was resolved in [LimitOrderManager.sol#L391](#).

C-06 | No Access Control

Category	Severity	Location	Status
Access Control	● Critical	LimitOrderHook.sol	Resolved

Description

The vulnerability in the `LimitOrderHook` contract arises due to the lack of access control in the `beforeSwap()` and `afterSwap()` hook callback functions, which are critical to the proper execution of limit orders during swaps.

Specifically, the `beforeSwap()` function stores the tick value before a swap, while the `afterSwap()` function processes limit orders based on the state change between the previous and new ticks. However, these functions do not have proper access control to restrict who can invoke them.

Without mechanisms like the `onlyByPoolManager()` modifier, unauthorized users can call these functions directly, including malicious actors who may manipulate swap behavior.

This could lead to serious consequences such as unauthorized order executions, where users could trigger limit orders to be processed without them actually being filled or meet the conditions of the swap.

For example, the `executeOrder()` function, which is designed to be executed only by the `LimitOrderHook` contract after a legitimate swap, can currently be called by any user.

This allows for orders to be executed or cleared arbitrarily, completely undermining the protocol's limit order mechanism and breaking its core functionality.

Given that these hooks are integral to the protocol's operation, this access control vulnerability has the potential to severely disrupt the system, leading to manipulation of the limit order functionality.

Recommendation

Add `onlyByPoolManager()` of `SafeCallback.sol` in all hooks for access control.

Resolution

Gamma Team: The issue was resolved in [LimitOrderHook.sol#L77](#).

C-07 | User Remains Position Contributor

Category	Severity	Location	Status
Logical Error	● Critical	LimitOrderManager.sol	Resolved

Description [PoC](#)

Users are never removed from `positionContributors` set upon cancellation of their order. This leads to the following scenario:

- (1) User creates an order, position count is 1.
- (2) User cancels the order, position count is 0 after `userPositionKeys` is cleared, but user remains a position contributor for that position key in mapping `positionContributors`.
- (3) User creates the same order. Because the user is already seen as a position contributor, the position key is not added to `userPositionKeys` and the position is not registered for the user, hence their position count remains 0. This impacts most view functions and `cancelBatchOrders` which directly rely on `userPositionKeys`.
- (4) If the user were to attempt to cancel their "unregistered" order to retrieve their funds, there would be a revert when attempting to remove the tick range, as it does not exist.

This is because the "ghost" position was never cleared, hence a new tick range was not inserted upon order creation in step (3). Specifically, an underflow revert on `uint256 right = positionTickRangeList[poolId].length - 1;` within function `findPositionTickRangeldx` will occur. Ultimately, a user who creates the same order after cancelling a previous order loses their funds.

Recommendation

Ensure upon cancellation that a user is removed as a position contributor as necessary and that the position key is marked as inactive.

Resolution

Gamma Team: The issue was resolved in commit [bf63099](#).

H-01 | Pools With Low SqrtPrices Are Unusable

Category	Severity	Location	Status
Rounding	● High	Global	Resolved

Description

Because of precision loss that occurs when converting the user’s specified amount to the order’s associated liquidity, pools for quote assets with small prices relative to the base token are unusable.

This is because the creation of an order does not clear any remaining amount delta from the `Uniswap PoolManager`.

As a result non zero balance deltas are left in the delta for the user’s input currency and therefore the `NonzeroDeltaCount` is not decremented to zero upon the `modifyLiquidity` call.

This applies to tokens with a `sqrtPrice` in the range of $1e27$ (\$0.0001) and below, but also for token pairs where one token has larger decimals than the other, simulating a very low price.

Recommendation

At the end of the `CREATE_ORDERS` callback, be sure to settle any dust that was left for the user by either minting it or clearing it from the `PoolManager` contract. Be aware that if it is cleared then this amount is lost.

Resolution

Gamma Team: The issue was resolved in commit [ec5654f](#).

Guardian Team: Dust still remains in the `accountDelta` within the `PoolManager` when creating orders with low prices, hence the issue remains. Trigger `PositionManager.clear` to clear the leftover delta on the account and decrement the `NonzeroDeltaCount`. Note that clearing will lock the funds in the `PoolManager` permanently.

H-02 | Incorrect Positions Removed

Category	Severity	Location	Status
Logical Error	● High	LimitOrderManager.sol: 463	Pending

Description

In the `executeOrderByKeeper` function the `executablePositionIds` list is populated with the indexes of each position tick range before any modifications have been made to the `positionTickRangeList`.

However in the `_handleKeeperExecuteCallback` function the ids are removed sequentially, thereby adjusting the indexes of some positions after each removal and invalidating the provided `executablePositionIds`.

This will end up in the wrong positions being removed from the `executablePositionIds` list. This will only be avoided if the keeper provides a carefully ordered list to the `executeOrderByKeeper` function which is in such an order that the removal of each iterative position does not affect the index of each subsequent one.

Recommendation

Consider reading the most up to date index of each position directly before removing it in the `_handleKeeperExecuteCallback` for loop.

Resolution

Gamma Team: The issue was resolved in commit [cfc3067](#).

H-03 | Execution Of Orders Can Be Skipped

Category	Severity	Location	Status
Logical Error	● High	LimitOrderManager.sol: 567	Resolved

Description [PoC](#)

Executable orders can get skipped in certain cases. Let's say we are at tick 0 and we have a limit order at 120-180 range `isToken0: true`. If we make a swap `zeroForOne: false` then we will move the price up. If the price happens to land somewhere between 120-180 the limit orders will not execute.

The issue appears when when we make another such swap then `beforeSwapTick` is going to be between 120-180 and the `afterSwapTick` is going to be after 180. The limit orders will not execute again.

This limit orders will get executed if the `beforeSwapTick` is before 120 and the `afterSwapTick` is more than 180. In reality the price have already passed the limit order range and should have been executed already.

In the current implementation and example the bottom tick gets compared to the `beforeSwapTick` during the binary search and this causes the orders to get skipped.

Recommendation

Consider modifying `_findOverlappingPositions` to select positions that the price range have already passed, despite the `beforeSwapTick` value.

Resolution

Gamma Team: The issue was resolved in commit [6d9eff8](#).

H-04 | Order Creation And Execution Can Be DoSed

Category	Severity	Location	Status
DoS	● High	PositionManagement.sol: 336-346	Resolved

Description [PoC](#)

New positions are added to the `positionTickRangeList` array. This array can be artificially bloated, causing a DOS to new order creation or existing order execution. Upon creating an order for a new position or executing an existing position, the `positionTickRangeList` must be reordered, i.e. the array indexes must be moved around.

A malicious user can create many scale orders or range limit orders to inflate the size of the array which will cause out-of-gas reverts when a legitimate user attempts to create an order at a new position or a swap attempts to execute an order. Essentially, all swaps through the pool will revert. The attacker is able to recoup all of the funds used to set up the positions by cancelling them.

Furthermore, an oversight in the position removal logic allows these spam positions to stay permanently without possibility of removal.

Note: Through testing, it was determined that scale orders could be spammed to create enough positions so that a single order execution would cost ~15,000,000 gas, while the estimated block gas limit is 30,000,000. Range limit orders could be used to fill up the array more to achieve the full OOG, though this is not performed in the POC.

Recommendation

Make sure when the attacker cancels a limit order, this call to also remove the created position if his order was the only one for this position. This would make the attack more expensive but still possible since he will not be able to retrieve his funds and hold the DOS at the same time.

To prevent the issue from happening involves a design decision from the team either to limit the amount of positions that can be opened at the same time or to seek a different method of storing and identifying executable positions.

Resolution

Gamma Team: The issue was resolved in [LimitOrderManager.sol#L447](#).

Guardian Team: Function `removePositionTickRange` remains costly and order execution may be entirely prevented due to OOG reverts. In testing, it was found that swapping across just 5 tick spacings consumes nearly 36 million gas with about 5000 orders in the tick range list. Besides creating numerous orders, users can use arbitrary orders to make emergency cancellation extremely inefficient.

H-05 | Batch Cancels Cause Users To Cancel Incorrectly

Category	Severity	Location	Status
Logical Error	● High	LimitOrderManager.sol: 222	Resolved

Description

Imagine you have 10 active limit orders and you want to cancel the first 3 orders. User position keys in storage = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. We call `cancelBatchOrders(poolKey, 0, 3)`; The loop inside `cancelBatchOrders()` on the first iteration selects user position key at index 0.

Then code executes `_cancelOrder() > _claimOrder()` [here](#).

The user position keys in storage get changed from [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] to [9, 1, 2, 3, 4, 5, 6, 7, 8]. Then on the next iteration code will select again the first element of the array because i is going to be 1 but `canceledCount` will be 1 as well. When subtracted we will get the 0th index [again](#).

At the end of the looping instead of removing user positions 0, 1 and 2, `cancelBatchOrders()` will remove 0, 9, 8 - this could be unexpected by the user and cancel orders by mistake. User positions will change in the following way: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] > [7, 1, 2, 3, 4, 5, 6]

Recommendation

Consider removing positions that are right after the user specified offset.

Resolution

Gamma Team: The issue was resolved in [LimitOrderManager.sol#L288](#).

M-01 | Keeper Frontrunning

Category	Severity	Location	Status
Frontrunning	● Medium	LimitOrderManager.sol	Acknowledged

Description

The `executeOrderByKeeper` function allows the keeper to execute orders which are technically fulfillable based on the pools price but that were not executed in the swap which crossed their end tick.

However if the pool price is back within their position end tick at the time of keeper execution, then the position is marked as no longer executable by the keeper and is not executed.

A bad faith actor could frontrun the keeper’s call to the `executeOrderByKeeper` function and force the pool price to go below the position’s execution range and thus prevent these positions from being executed.

Recommendation

Be aware of this risk and consider using MEV protection such as flashbots for the keeper role.

Resolution

Gamma Team: Acknowledged.

M-02 | Permissionless Use Of Pools

Category	Severity	Location	Status
Validation	● Medium	LimitOrderManager.sol: 89-94	Resolved

Description

Gamma allows for orders to be created/cancelled/executed for arbitrary pools. This may allow a malicious token pair to take advantage of user's locked up funds after creating orders, to prevent token transfer when claiming/cancelling and lead to loss of user funds.

A malicious token can be used to create a pool which orders are created for on Gamma. The malicious token could intentionally allow an execution to occur of limit orders, with an overflow of orders being assigned as keeper executable.

Before the keeper's execution of the orders the malicious token could be updated to expend a significant amount of gas and even store this gas in a canonical "gas token" to extract value from the keeper.

Recommendation

Consider whitelisting the pools that are allowed to be used with the Gamma limit system to avoid the risk of malicious tokens in arbitrary pools.

Resolution

Gamma Team: Resolved.

M-03 | Updating Fee Values Affects Fees

Category	Severity	Location	Status
Logical Error	● Medium	LimitOrderManager.sol: 966-971	Acknowledged

Description

When `setHookFeePercentage()` is called, the hook fee percentage will be updated (increased or decreased). An increase in this percentage will cause the user to receive less fees than expected from their already accumulated fees.

A user might be checking that they should receive \$100 in fees based on swaps already executed, then the admin increases the fee percentage, now the user will only receive\$ 50.

Recommendation

Implement a fee change functionality that operates based off of the original fee percentage until `_retrackPositionFee()` is called. This will ensure that the proper percentage is used until the pool is updated.

Further, potentially create another function that performs fee checkpointing for multiple positions in a batch.

Resolution

Gamma Team: Acknowledged.

L-01 | Insufficient Event Data

Category	Severity	Location	Status
Events	● Low	LimitOrderManager.sol	Resolved

Description

In the `_claimOrder` function the `LimitOrderClaimed` event only emits the `poolId` and `user` as information, however the position tick ranges are also useful to discern which position has been claimed from the pool by the user.

Recommendation

Consider including the lower and upper ticks in the `LimitOrderClaimed` event.

Resolution

Gamma Team: Resolved.

L-02 | Lacking Keeper Execution Validation

Category	Severity	Location	Status
Validation	● Low	LimitOrderManager.sol	Acknowledged

Description

The `executeOrderByKeeper` function allows the keeper to pass a list of `waitingPositions` however no validation is performed to ensure that the `positionKey` values are not duplicated in this list.

There are however no consequential impacts of this as the `_handleKeeperExecuteCallback` function will continue for positions that have already been processed.

Recommendation

Consider adding validation against duplicated `positionKey` entries to avoid any unexpected behavior and limit keeper error in the `executeOrderByKeeper` function.

Resolution

Gamma Team: Acknowledged.

L-03 | Scale Order Validation Does Not Take Place

Category	Severity	Location	Status
Logical Error	● Low	LimitOrderManager.sol: 166	Resolved

Description

The `_createOrder` function calls the `validateScaleOrderSizes` but does not assert that the returned boolean is true.

The most important validations in the `validateScaleOrderSizes` revert upon failure, however the zero length and `totalAmount` validations choose to return false rather than reverting. These less important sanity checks are not enforced by the invocation in `_createOrder`.

Recommendation

Standardize the enforcement of all validations in the `validateScaleOrderSizes` to all either return false or revert on failure and be sure to correctly assert that the returned boolean is true if standardizing on a boolean return value.

Resolution

Gamma Team: The issue was resolved in [PositionManagement.sol#L99](#).

L-04 | Lacking safeTransferFrom Usage

Category	Severity	Location	Status
Best Practices	● Low	LimitOrderManager.sol: 823, 827	Resolved

Description

In the `_handleTokenTransfer` function the `transferFrom` function is used to transfer tokens from the user. However `safeTransferFrom` should be used in this case since arbitrary tokens may be used within this context.

Recommendation

Use `safeTransferFrom` in the `_handleTokenTransfer` function.

Resolution

Gamma Team: The issue was resolved in commit [1c24bd1](#).

L-05 | Lacking Zero Address Validations

Category	Severity	Location	Status
Validation	● Low	LimitOrderManager.sol: 63	Resolved

Description

In the constructor of the LimitOrderManager contract the _treasury address is validated to be nonzero, however the _poolManager address is not validated to be nonzero.

Recommendation

Consider also validating the _poolManager address to verify that it is not errantly assigned as the zero address.

Resolution

Gamma Team: The issue was resolved in commit [abafc71](#).

L-06 | Unnecessary Position State

Category	Severity	Location	Status
Superfluous Code	● Low	Global	Resolved

Description

In the `LimitOrderManager` contract the `ps` storage variable is unnecessary and can be removed. In addition to this the `PositionStorage` struct defined in the `PositionManagement` library contains duplicate entries of the important storage values in the `LimitOrderManger` and therefore can also be removed.

Recommendation

Consider removing the `ps` variable and the `LimitOrderManager` struct.

Resolution

Gamma Team: Resolved.

L-07 | Lacking setExecutablePositionsLimit Validation

Category	Severity	Location	Status
Validation	● Low	LimitOrderManager.sol	Resolved

Description

The `executeOrder` function reverts when the `executablePositionsLimit` value is assigned to 0. However the `setExecutablePositionsLimit` function does not prevent the owner address from assigning 0 as a value and therefore preventing execution of limit orders.

Recommendation

Consider including validation in the `setExecutablePositionsLimit` function to prevent the owner from causing a DoS either accidentally or maliciously.

Resolution

Gamma Team: The issue was resolved in commit [5145377](#).

L-08 | Unnecessary Repeated Minting

Category	Severity	Location	Status
Gas Optimization	● Low	LimitOrderManager.sol	Resolved

Description

In the `handleCreateOrdersCallback` function the `_mintFeesToHook` function is called repeatedly in the loop for each order creation. However because the fees are all claimed for the same pool, the `_mintFeesToHook` function can be called only once with the aggregate fees to mint to save gas.

Recommendation

Consider calling the `_mintFeesToHook` with the aggregated fee amount after all orders have been created.

Resolution

Gamma Team: The issue was resolved in commit [c67bbfe](#).

L-09 | Orders Can Be Made For Unconnected Pools

Category	Severity	Location	Status
Validation	● Low	LimitOrderManager.sol	Resolved

Description

In the `_createOrder` function there is no validation that the orders being created belong to the hook address associated with the Gamma limit system.

As a result positions can be created that are not executable by the hook or keeper. No other impacts have been identified other than unexpected behavior for users.

However it would be prudent to remove this capability to limit the potential attack surface area of the contracts and prevent users from creating orders for incorrect pools.

Recommendation

Consider validating that the limit hook is a part of the pool key in the `_createOrder`.

Resolution

Gamma Team: The issue was resolved in commit [bfc3845](#).

L-10 | Refunds Missed For Non-Native Currencies

Category	Severity	Location	Status
Warning	● Low	LimitOrderManager.sol: 805	Resolved

Description

The `_handleTokenTransfer` function does not perform any validation on the `msg.value` in the case where the currency specified is not native. As a result if the native currency is not provided and the `msg.value` is nonzero this ether amount can become stuck in the contract.

Recommendation

Consider adding validation to prevent users from errantly sending `msg.value` when the native currency is not being used.

Resolution

Gamma Team: Resolved.

L-11 | Unused Errors

Category	Severity	Location	Status
Best Practices	● Low	TickLibrary.sol	Resolved

Description

In the TickLibrary library, the MinimumAmountNotMet and InvalidScaleParameters errors are defined and yet never used in that library.

Recommendation

Consider removing these errors.

Resolution

Gamma Team: The issue was resolved in commit [67cfe17](#).

L-12 | Incorrect getUserClaimableBalances Result

Category	Severity	Location	Status
Logical Error	● Low	LimitOrderManager.sol	Resolved

Description

In the `getUserClaimableBalances` function the `getPositionBalances` function reports an amount which includes the pending unclaimed fees for the current Gamma system position deployed at that range.

This amount is reported as being claimable by the user even if the user had a previously executed limit order in the same range and thus should not be receiving fees from the new position in the same range which is now present in the pool.

Notice that the `getUserProportionateFees` function does not early return when a user’s position has already been executed and yet to be claimed because the `posState.totalLiquidity` is not reset to 0 upon execution of a position and rather the position nonce is incremented.

Recommendation

For positions that are no longer active use the `_getUserFees` function to compute the actual fees of the user which have already been claimed by Gamma and are no longer accruing in the Uniswap pool.

Resolution

Gamma Team: The issue was resolved in commit [4695dbe](#).

L-13 | Sync DoS Attack

Category	Severity	Location	Status
DoS	● Low	CurrencySettler.sol	Resolved

Description

The `CurrencySettler.settle` function neglects to call the `sync` function for native currency payments. The `sync` function can be called by a malicious actor to sync an unexpected currency at any time.

Therefore actions made with the `LimitOrderManager` in the Gamma system can be DoS'd by a simple call to sync a non-native currency by a malicious actor.

Recommendation

Include a call to `sync` for the native currency in the `settle` function.

Resolution

Gamma Team: The issue was resolved in commit [d9d7f54](#).

L-14 | Follow Import Best Practices

Category	Severity	Location	Status
Best Practices	● Low	LimitOrderManager.sol: 13	Resolved

Description

The LimitOrderManager.sol contract utilizes the CurrencySettler.sol library in order to sync and transfer tokens to the PoolManager contract. However, CurrencySettler is imported from Uniswap's test suite.

import {CurrencySettler} from "@uniswap/v4-core/test/utils/CurrencySettler.sol"; It is possible that Uniswap may alter the implementation of this library without consideration of potential integrators.

Recommendation

It is recommended to recreate the library within this repository to ensure consistency.

Resolution

Gamma Team: The issue was resolved in commit [eda5f41](#).

L-15 | Follow CEI Pattern When Claiming Positions

Category	Severity	Location	Status
Best Practices	● Low	LimitOrderManager.sol: 367-386	Resolved

Description

Check-Effects-Interaction pattern should be followed when claiming an order. Firstly, the pool is unlocked to transfer tokens and after the `userPositions` mapping is deleted.

No reentrancy can occur due to the pool manager's inherent reentrancy protection via `PoolManager.unlock()`. However, it would be a good idea to update storage prior to making the external calls.

Recommendation

Move the logic to delete the user positions and removing the user position key prior to the call to `PoolManager.unlock()`.

Resolution

Gamma Team: The issue was resolved in commit [5f3d853](#).

L-16 | Random Users Can Spam LimitOrderClaimed Events

Category	Severity	Location	Status
Validation	● Low	LimitOrderManager.sol: 386	Resolved

Description

Users without active limit orders can call `claimOrder()` which will complete successfully and emit `LimitOrderClaimed` events. This may cause confusion or affect backend processes that index such logs.

Recommendation

Validate that the user has the specified position in the `userPositions` mapping.

Resolution

Gamma Team: The issue was resolved in commit [e67b6e7](#).

L-17 | Unnecessary Reads From Storage

Category	Severity	Location	Status
Gas Optimization	● Low	LimitOrderManager.sol: 770	Acknowledged

Description

A collection of storage reads that can be optimized: [LimitOrderManager.sol:770](#)

Recommendation

Convert from storage to memory to reduce reads from storage.

Resolution

Gamma Team: Acknowledged.

L-18 | Integrators May Receive Unexpected Token

Category	Severity	Location	Status
Validation	● Low	LimitOrderManager.sol: 267-270	Acknowledged

Description

Calling `cancelOrder()` claims the order if it has already been executed. This feature may cause problems with downstream integrations. An integrating protocol may create an order with USDC to be swapped for ETH and then cancels the order expecting to receive USDC back.

However, if the position is executed prior to the call to `cancelOrder()`, the integrating contract will receive back ETH. There is no guarantee on which token (or native ETH) will be received when calling `cancelOrder()`.

Recommendation

Consider adding an `expectedOutputToken` parameter to verify the token the caller expects to receive.

Resolution

Gamma Team: Acknowledged.

L-19 | maxOrderLimit Behaviour Differs From Comment

Category	Severity	Location	Status
Informational	● Low	LimitOrderManager.sol: 973	Resolved

Description

maxOrderLimit limits the number of orders that be created in createScaleOrders() at once and not the total number of limit orders that can be active per pool like the comments above setMaxOrderLimit() state.

Recommendation

Adjust the documentation.

Resolution

Gamma Team: The issue was resolved in commit [cf1c46d](#).

L-20 | Variable Denoted As Constant

Category	Severity	Location	Status
Best Practices	● Low	LimitOrderManager.sol: 50	Resolved

Description

HOOK_FEE_PERCENTAGE is denoted in all capitals, signifying a constant. However, this value can be changed by the admin.

Recommendation

Consider renaming to hook_fee_percentage to avoid confusion.

Resolution

Gamma Team: The issue was resolved in commit [3221d25](#).

L-21 | Keeper Execution DoS'd

Category	Severity	Location	Status
Logical Error	● Low	LimitOrderManager.sol	Resolved

Description

A user can frontrun a keeper's transaction to `executeOrderByKeeper` to cancel their order. Even though the position has been removed, `executeOrderByKeeper` still attempts to clear the position since `isWaitingKeeper` is still set to true after cancellation.

Consequently, `executeOrderByKeeper` will trigger a Uniswap revert and fail to execute all other `waitingPositions` because liquidity is attempted to be removed from ticks that have already had their liquidity removed.

Recommendation

Upon cancellation, ensure the position's `isWaitingKeeper` status is set to false.

Resolution

Gamma Team: The issue was resolved in commit [e9a5807](#).

L-22 | Unchecked Msg.value In Token Transfer

Category	Severity	Location	Status
Validation	● Low	LimitOrderManager.sol	Resolved

Description

The vulnerability arises in the `_handleTokenTransfer` function of the smart contract, which is responsible for handling token transfers based on the `isToken0` flag.

When `isToken0` is false, the function is designed to transfer `token1` (a non-native token) using the `transferFrom` method of the `IERC20Minimal` interface. However, the function fails to validate whether `msg.value` is zero in this scenario.

If a user mistakenly sends Ether while calling this function with `isToken0` set to false, the Ether will be accepted by the contract but not utilized in any way. This occurs because the function does not enforce a check to ensure that `msg.value` is zero when dealing with non-native tokens.

As a result, the Ether remains trapped in the contract, with no mechanism for the user to recover it.

Recommendation

To mitigate this vulnerability, the `_handleTokenTransfer` function should include a validation step to ensure that `msg.value` is zero when `isToken0` is false.

Resolution

Gamma Team: Resolved.

L-23 | Max Limit Can Prevent Scale Orders

Category	Severity	Location	Status
Validation	● Low	LimitOrderManager.sol	Resolved

Description

The owner is able to set the `maxOrderLimit` with function `setMaxOrderLimit`. If the `maxOrderLimit = 1`, all scale orders will be prevented because scale orders require a minimum of 2 orders: if `(totalOrders < 2)` revert `MinimumTwoOrders()`;

Recommendation

Be aware of this behavior or add validation within `setMaxOrderLimit` that the `_limit` is greater than 1.

Resolution

Gamma Team: The issue was resolved in commit [90bb9d0](#).

L-24 | Return Value Of transferFrom() Not Checked

Category	Severity	Location	Status
Best Practices	● Low	LimitOrderManager.sol	Resolved

Description

Not all ERC20 implementations `revert()` when there's a failure in `transfer()/transferFrom()`. The function signature has a boolean return value and they indicate errors that way instead.

By not checking the return value, operations that should have marked as failed, may potentially go through without actually making a payment.

```
IERC20Minimal(Currency.unwrap(key.currency0)).transferFrom(msg.sender, address(this), amount);
IERC20Minimal(Currency.unwrap(key.currency1)).transferFrom(msg.sender, address(this), amount);
```

Recommendation

Consider using the `SafeTransfer` library for transferring ERC20 tokens.

Resolution

Gamma Team: The issue was resolved in commit [669e488](#).

L-25 | `_executePosition` Does Not Reset `isWaitingKeeper`

Category	Severity	Location	Status
Unexpected Behavior	● Low	LimitOrderManager.sol	Resolved

Description

The `_findOverlappingPositions` function does not skip positions which have been marked with a `true` value for `isWaitingKeeper`. As a result these positions can be executed normally when price re-crosses their upper boundary.

When this scenario is hit the position is executed, however the `positionState` entry is left with a `true` value for `isWaitingKeeper`.

The nonce of the position is incremented however, so the impact is limited to consumers of the `LimitOrderManager` state which may be confused by the false reporting of `isWaitingKeeper`.

Recommendation

Consider if the `isWaitingKeeper` positions should be skipped in the `_findOverlappingPositions` function. If not, then ensure that the `isWaitingKeeper` value is set to `false` after the execution of a position that was previously flagged for keeper execution.

Resolution

Gamma Team: The issue was resolved in commit [983c00e](#).

L-26 | Min/Max Tick Validation In Scale Orders

Category	Severity	Location	Status
Validation	● Low	LimitOrderManager.sol	Resolved

Description

The `minUsableTick` and `maxUsableTick` values are defined as limits beyond which orders should not be placed. This validation is implemented in the `createLimitOrder()` function, but it is absent in the `createScaleOrders()` function.

As a result, scale orders can be placed beyond the acceptable `min/max` tick range, which is not intended. This oversight could lead to invalid order placements, potentially disrupting the expected functionality and behavior.

Recommendation

A check for `minUsableTick` and `maxUsableTick` should be added in the `createScaleOrders()` function, similar to the implementation in the `createLimitOrder()` function. This will ensure that scale orders are only placed within the defined tick range.

Resolution

Gamma Team: The issue was resolved in commit [7d342cb](#).

L-27 | Use nonReentrant() Modifier

Category	Severity	Location	Status
Best Practices	● Low	LimitOrderManager.sol	Resolved

Description

In the `_handleTokenTransfer` function, specifically when `isToken0` is true and the native token (Ether) is being handled. In this scenario, the function checks if the sent Ether (`msg.value`) is sufficient to cover the required amount.

If excess Ether is sent, the function refunds the difference to the sender using a low-level call. While the current implementation does not expose an immediate reentrancy risk , the use of call without a reentrancy guard violates best practices.

If the contract's logic is modified in the future or if the call is replaced with a more complex operation, it could create a reentrancy vulnerability.

Recommendation

To mitigate this, `nonReentrant` modifier from OpenZeppelin's `ReentrancyGuard` contract should be used.

Resolution

Gamma Team: The issue was resolved in commit [a5dd43f](#).

L-28 | Incorrect Current Nonce On Position

Category	Severity	Location	Status
Logical Error	● Low	LimitOrderManager.sol	Resolved

Description

function `positionState` aims to return all `PositionState` information, including the position's `currentNonce` used. However, `currentNonce` is never set in the `positionState` mapping upon order creation, hence it will always be 0 regardless of the actual nonce used for the position.

Although `PositionState.currentNonce` is not utilized within the contract, it can potentially impact external systems reading the position's incorrect nonce and confuse users.

For example. a system may display an executed, inactive order with nonce 0, and simultaneously have an unexecuted order over the same tick range also with nonce 0.

Recommendation

Set the position's current nonce upon order creation.

Resolution

Gamma Team: The issue was resolved in commit [de3c848](#).

Disclaimer

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Guardian to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Guardian’s position is that each company and individual are responsible for their own due diligence and continuous security. Guardian’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by Guardian is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract’s safety and security. Therefore, Guardian does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

About Guardian Audits

Founded in 2022 by DeFi experts, Guardian Audits is a leading audit firm in the DeFi smart contract space. With every audit report, Guardian Audits upholds best-in-class security while achieving our mission to relentlessly secure DeFi.

To learn more, visit <https://guardianaudits.com>

To view our audit portfolio, visit <https://github.com/guardianaudits>

To book an audit, message <https://t.me/guardianaudits>