MixBytes()

Algebra Limit Order Plugin Security Audit Report

JUNE 25, 2025

Table of Contents

1. Introduction

1.1 Disclaimer

The audit makes no statements or warranties regarding the utility, safety, or security of the code, the suitability of the business model, investment advice, endorsement of the platform or its products, the regulatory regime for the business model, or any other claims about the fitness of the contracts for a particular purpose or their bug-free status.

1.2 Executive Summary

The Limit Order Plugin is a project built on top of the Algebra DEX that enables users to create limit orders for token swaps by leveraging concentrated liquidity and discrete price ticks. These orders are implemented as liquidity positions placed at specific ticks and are automatically executed when the pool's price crosses the predefined levels.

A team of 3 auditors conducted the audit over 2 days, performing a detailed manual review and analysis via automated tools.

During the audit, in addition to verifying well-known attack vectors and items from our internal checklist, we thoroughly investigated the following areas:

Concurrent Usage and Liquidity Isolation
The plugin is designed for concurrent use by multiple users, who can create orders in different pools with various token pairs and across different price ranges. We have verified that the liquidity provided by each user is exclusively available to them and cannot be unjustly captured by others, whether through accidental or intentional actions.

Limit-Order Execution and Cleanup
The plugin implements limit-order logic, meaning that the user intends to swap their entire supplied amount of one token for the corresponding amount of the second token. Once executed, the order is considered fulfilled and removed from the market, and subsequent events should no longer affect it. We have confirmed that this logic is implemented correctly and that no liquidity remains in the market after order execution.

Order and Epoch Abstractions
The project employs the abstractions of "order" and "epoch" (multiple orders sharing similar parameters). We have examined how the lifecycle of orders and epochs is implemented, ensuring that no violations occur that could lead to unexpected behavior endangering user funds.

Epoch Identifier Collision Prevention
When multiple orders with identical parameters are active simultaneously, the system groups them into so-called epochs, each identified by a unique number. We have verified that no collisions of these identifiers occur within a reasonable timeframe.

Fairness of Concurrent Order Conditions
We have confirmed that the conditions applied to users placing orders concurrently are fair, and that no user—whether accidentally or through deliberate manipulation—can gain an unjustified advantage over another.

Hook Function Security (afterSwap & mintCallback)
To handle swap events in the pool, the `afterSwap` hook function is used. We have verified that it is protected against unauthorized invocations and cannot be called by anyone other than the pool itself. We have also reviewed the `mintCallback` protection to ensure that unauthorized parties cannot exploit it or misuse user approvals for undesired actions, including the misappropriation of other users' funds.

Tick-Spacing Parameter Synchronization
A critical aspect of the system is the tick-spacing parameter. We have examined how the plugin addresses the possibility of this parameter being changed during the pool's operation, how synchronization of this parameter between the pool and the plugin is performed, and what issues may arise for orders created before the change.

Reentrancy Protection
We have confirmed that all user-facing functions are safeguarded against reentrancy attacks, ensuring that no user can gain control over execution during any inconsistent contract state.

Post-Execution Asset Distribution
When multiple orders share identical parameters, the challenge becomes fairly distributing assets among users after execution. We have verified that the final distribution is accurate, corresponds to the users' original deposits, and does not leave significant funds stranded in the contract or the pool.

Sentinel Technique Verification
Special attention was given to the sentinel technique, whereby the token0 liquidity in an epoch is virtually increased by one. We have reviewed this implementation to ensure that it does not enable the misappropriation of user funds or cause any unintended locking of assets within the contract.

We also note that if tokens are accidentally sent to the `LimitOrderManager` contract, they could be claimed by any user who creates and then kills a position, since the `claimTo` function transfers the contract's entire token balance. However, this has no impact on the protocol or its users, as all protocol-user transfers are atomic, and the `LimitOrderManager` should not hold excess tokens unless they were mistakenly sent directly to it.

The codebase demonstrates high quality. No critical security issues were discovered during the audit. However, we have outlined several areas where refinements could enhance the protocol's overall robustness, clarity, and maintainability. These points are presented in detail in the **Findings Report** section below.

1.3 Project Overview

Summary

Title	Description
Client Name	Algebra
Project Name	Limit Order Plugin
Type	Solidity
Platform	EVM
Timeline	12.06.2025 - 23.06.2025

Scope of Audit

File	Link
packages/limit-order/contracts/ LimitOrderManager.sol	LimitOrderManager.sol
packages/limit-order/contracts/ LimitOrderPlugin.sol	LimitOrderPlugin.sol
packages/limit-order/contracts/ libraries/EpochLibrary.sol	EpochLibrary.sol
packages/limit-order/contracts/base/ LimitOrderPayments.sol	LimitOrderPayments.sol

Versions Log

Date	Commit Hash	Note
12.06.2025	ac412688390d20ad2f29793c1e5ba9585fb3047b	Initial Commit
19.06.2025	b1adba18438bebd5e0267cebcd39039aada125cb	Re-audit Commit

Date	Commit Hash	Note
23.06.2025	998a1d51753c21105b33e6941f3e1352cd2fd603	Re-audit Commit

Mainnet Deployments

File	Address	Blockchain
LimitOrderManager.sol	0xF05bd4...0E4EdEE1	Base

1.4 Security Assessment Methodology

Project Flow

Stage	Scope of Work
Interim audit	**Project Architecture Review:** · Review project documentation · Conduct a general code review · Perform reverse engineering to analyze the project's architecture based solely on the source code · Develop an independent perspective on the project's architecture · Identify any logical flaws in the design OBJECTIVE: UNDERSTAND THE OVERALL STRUCTURE OF THE PROJECT AND IDENTIFY POTENTIAL SECURITY RISKS.
	Code Review with a Hacker Mindset: · Each team member independently conducts a manual code review, focusing on identifying unique vulnerabilities. · Perform collaborative audits (pair auditing) of the most complex code sections, supervised by the Team Lead. · Develop Proof-of-Concepts (PoCs) and conduct fuzzing tests using tools like Foundry, Hardhat, and BOA to uncover intricate logical flaws. · Review test cases and in-code comments to identify potential weaknesses. OBJECTIVE: IDENTIFY AND ELIMINATE THE MAJORITY OF VULNERABILITIES, INCLUDING THOSE UNIQUE TO THE INDUSTRY.
	Code Review with a Nerd Mindset: · Conduct a manual code review using an internally maintained checklist, regularly updated with insights from past hacks, research, and client audits. · Utilize static analysis tools (e.g., Slither, Mythril) and vulnerability databases (e.g., Solodit) to uncover potential undetected attack vectors. OBJECTIVE: ENSURE COMPREHENSIVE COVERAGE OF ALL KNOWN ATTACK VECTORS DURING THE REVIEW PROCESS.

Stage	Scope of Work
	Consolidation of Auditors' Reports: · Cross-check findings among auditors · Discuss identified issues · Issue an interim audit report for client review OBJECTIVE: COMBINE INTERIM REPORTS FROM ALL AUDITORS INTO A SINGLE COMPREHENSIVE DOCUMENT.
Re-audit	**Bug Fixing & Re-Audit:** · The client addresses the identified issues and provides feedback · Auditors verify the fixes and update their statuses with supporting evidence · A re-audit report is generated and shared with the client OBJECTIVE: VALIDATE THE FIXES AND REASSESS THE CODE TO ENSURE ALL VULNERABILITIES ARE RESOLVED AND NO NEW VULNERABILITIES ARE ADDED.
Final audit	**Final Code Verification & Public Audit Report:** · Verify the final code version against recommendations and their statuses · Check deployed contracts for correct initialization parameters · Confirm that the deployed code matches the audited version · Issue a public audit report, published on our official GitHub repository · Announce the successful audit on our official X account OBJECTIVE: PERFORM A FINAL REVIEW AND ISSUE A PUBLIC REPORT DOCUMENTING THE AUDIT.

1.5 Risk Classification

Severity Level Matrix

Severity	Impact: High	Impact: Medium	Impact: Low
Likehood: High	Critical	High	Medium
Likehood: Medium	High	Medium	Low
Likehood: Low	Medium	Low	Low

Impact

- **High** – Theft from 0.5% OR partial/full blocking of funds (>0.5%) on the contract without the possibility of withdrawal OR loss of user funds (>1%) who interacted with the protocol.
- **Medium** – Contract lock that can only be fixed through a contract upgrade OR one-time theft of rewards or an amount up to 0.5% of the protocol's TVL OR funds lock with the possibility of withdrawal by an admin.
- **Low** – One-time contract lock that can be fixed by the administrator without a contract upgrade.

Likelihood

- **High** – The event has a 50–60% probability of occurring within a year and can be triggered by any actor (e.g., due to a likely market condition that the actor cannot influence).
- **Medium** – An unlikely event (10–20% probability of occurring) that can be triggered by a trusted actor.
- **Low** – A highly unlikely event that can only be triggered by the owner.

Action Required

- **Critical** – Must be fixed as soon as possible.
- **High** – Strongly advised to be fixed to minimize potential risks.
- **Medium** – Recommended to be fixed to enhance security and stability.
- **Low** – Recommended to be fixed to improve overall robustness and effectiveness.

Finding Status

- **Fixed** – The recommended fixes have been implemented in the project code and no longer impact its security.
- **Partially Fixed** – The recommended fixes have been partially implemented, reducing the impact of the finding, but it has not been fully resolved.
- **Acknowledged** – The recommended fixes have not yet been implemented, and the finding remains unresolved or does not require code changes.

1.6 Summary of Findings

Findings Count

Severity	Count
Critical	0
High	0
Medium	3
Low	11

Findings Statuses

ID	Finding	Severity	Status
M-1	Missed Fills When Tick-Spacing Changes	Medium	Fixed
M-2	Locked Surplus ETH and Potential Double Charge	Medium	Fixed
M-3	Early ETH Refund Breaks Mixed-Token Payment	Medium	Fixed
L-1	Centralized Control of Tick-Spacing	Low	Acknowledged
L-2	Missing Event on Tick-Spacing Update	Low	Fixed
L-3	Unused ZERO_BYTES Constant	Low	Fixed
L-4	Redundant unchecked Block in place()	Low	Fixed
L-5	Redundant getTickLower() Call After Swap	Low	Fixed
L-6	Gas-Heavy Epoch Lookup for Uninitialized Ticks	Low	Acknowledged
L-7	Potential Underflow in token0Total due to Sentinel-Value Subtraction	Low	Fixed
L-8	Epoch Remains Active After Full Empty by kill()	Low	Acknowledged
L-9	Missing Existence Check for Epoch in kill()	Low	Acknowledged
L-10	Premature Use of Unverified Epoch Data in withdraw()	Low	Fixed

L-11	Redundant `afterInitialize()` Function	Low	Fixed

2. Findings Report

2.1 Critical

Not Found

2.2 High

Not Found

2.3 Medium

M-1	Missed Fills When Tick-Spacing Changes		
Severity	Medium	**Status**	Fixed in b1adba18

Description

`LimitOrderManager._getCrossedTicks()` relies on `tickLowerLasts[pool]`, a value recorded under the previous tick-spacing. If governance changes a pool's tick-spacing (e.g., from 10 to 20), the stored `tickLowerLast` no longer aligns with the new grid. When the next swap occurs, the function calculates `lower` and `upper` from mismatched bases, causing any open positions between the old and new boundaries to be skipped and never filled.
This directly affects user funds and trading logic, warranting **Medium** severity.

Recommendation

We recommend resetting `tickLowerLasts[pool]` to `getTickLower(getTick(pool), newTickSpacing)` on every tick-spacing change so the correct boundary is retrieved after an update.

Client's Commentary:

Commit: b1adba18. Tickspacing changes are rare (usually before limit order plugin is used) and do not pose a risk to user funds. However, we believe that dexes using this plugin should notify users to close their positions when the tickspacing changes

M-2	Locked Surplus ETH and Potential Double Charge		
Severity	Medium	**Status**	Fixed in b1adba18

Description

When users call `LimitOrderManager.place()` with excess `msg.value`, the extra ETH remains in the `LimitOrderManager` balance instead of being forwarded or refunded. When `algebraMintCallback()` is called, that balance is wrapped into `WNativeToken` and sent to the pool, but any surplus never returns to the user. Conversely, if `msg.value` is lower than the amount of `WNativeToken` required for mint and the user previously gave `WNativeToken` allowance to `LimitOrderManager`, `LimitOrderPayments._pay()` pulls ERC-20 tokens in addition to consuming the partial ETH, resulting in a double charge.

Because this directly risks user funds by locking or over-charging assets, it is classified as **Medium** severity.

Recommendation

We recommend including the original `msg.value` in `MintCallbackData`, and then calculating how much ETH the `_pay` operation actually consumed and refunding any remainder back to the caller.

Client's Commentary:

Commit: 16f2df49. Since the `LimitOrderManager` contract is not expected to hold any native currency, we can refund the user the entire remaining contract balance.

M-3	Early ETH Refund Breaks Mixed-Token Payment		
Severity	Medium	**Status**	Fixed in 998a1d51

Description

In `LimitOrderManager.algebraMintCallback()` the debts to the pool are paid in order: first `token0`, then `token1`.

When `token0` is not `wNativeToken` but `token1` is, and the user supplies ETH with the call:

1. `_pay(token0, …)` executes the ERC-20 branch, transferring `token0` from the user.
2. The final `if (address(this).balance > 0)` line refunds all ETH held by the contract back to the payer, even though it will be needed momentarily.
3. `_pay(token1, …)` now finds the contract's ETH balance at zero, cannot wrap ETH into `WNativeToken`, and reverts.

The order fails despite the user providing sufficient ETH, interrupting liquidity provision and forcing a retry.

This finding has been classified as **Medium** severity because legitimate actions are being blocked.

Recommendation

We recommend refunding the surplus ETH after both payments are processed.

Client's Commentary:

Fixed. Commit: 417edc77

2.4 Low

L-1	Centralized Control of Tick-Spacing		
Severity	Low	**Status**	Acknowledged

Description

`LimitOrderManager.setTickSpacing()` allows any address with the `ALGEBRA_BASE_PLUGIN_MANAGER` role to change a pool's tick-spacing. The value governs the entire price grid, so an accidental or malicious change can disrupt order placement and filling across all users.

Recommendation

Remove this function altogether, or add a check ensuring that the new tick-spacing cleanly divides the pool's native tick-spacing without remainder.

Client's Commentary:

`ALGEBRA_BASE_PLUGIN_MANAGER` role should be granted to trusted parties only

L-2	Missing Event on Tick-Spacing Update		
Severity	Low	**Status**	Fixed in b1adba18

Description

`LimitOrderManager.setTickSpacing()` changes the `tickSpacings` mapping but emits no event, leaving off-chain indexers unaware of configuration changes.

Recommendation

We recommend emitting an event whenever tick-spacing is updated.

Client's Commentary:

Commit: e877bc12

L-3	Unused ZERO_BYTES Constant		
Severity	Low	**Status**	Fixed in b1adba18

Description

Although the LimitOrderManager contract defines bytes internal constant ZERO_BYTES = bytes('');, three calls to IAlgebraPool.burn() still pass the literal empty string '' instead of the constant, reducing consistency and readability.

Recommendation

We recommend replacing the string literals with ZERO_BYTES in all relevant calls.

Client's Commentary:

Commit: ddcf5daa

L-4	Redundant `unchecked` Block in `place()`		
Severity	Low	**Status**	Fixed in `b1adba18`

Description

`LimitOrderManager.place()` wraps `epochNext = epoch.unsafeIncrement;` in an `unchecked {}` block even though `unsafeIncrement` already performs its arithmetic unchecked.

Recommendation

We recommend removing the outer `unchecked {}` wrapper.

Client's Commentary:

Commit: 5c910795

L-5	Redundant `getTickLower()` Call After Swap		
Severity	Low	**Status**	Fixed in `b1adba18`

Description

`LimitOrderManager.afterSwap()` sets `tickLowerLasts[pool] = getTickLower(tickLower, tickSpacing);`, but `tickLower` is already the output of `getTickLower()`. The extra call performs the same calculation again without changing the result.

Recommendation

We recommend assigning `tickLowerLasts[pool] = tickLower;` directly.

Client's Commentary:

Commit: ba45eea5

L-6	Gas-Heavy Epoch Lookup for Uninitialized Ticks		
Severity	Low	**Status**	Acknowledged

Description

In `LimitOrderManager._fillEpoch()`, the contract computes `Epoch epoch = getEpoch(pool, lower, upper, zeroForOne);` for every crossed tick. When no orders exist at a tick (the common case), this still incurs a `keccak256` hash and `SLOAD` before confirming that epoch equals the default value. Repeating this across many empty ticks during volatile trading periods wastes gas.

Recommendation

Maintain a bitmap tree structure of initialized ticks inside `LimitOrderManager`, so `_fillEpoch()` can skip uninitialized ranges without hashing and storage reads, similar to the tick bitmap used in concentrated-liquidity AMMs.

Client's Commentary:

Acknowledged

L-7	Potential Underflow in token0Total due to Sentinel-Value Subtraction		
Severity	Low	**Status**	Fixed in b1adba18

Description

Under normal operation, token0Total is always initialized to at least **1** so that proportional calculations work correctly. However, after a full LimitOrderManager.kill() of an epoch (which zeroes out all liquidity), token0Total can legitimately become **0**. Since the code in LimitOrderManager._fillEpoch still performs:

```
epochInfo.token0Total += uint128(amount0) - 1;
```

It allows an underflow when amount0 == 0, wrapping token0Total to type(uint128).max. In practice, once an epoch's token0Total has been corrupted this way, no further operations—kill or withdraw—can occur because its liquidity is zero, and _fillEpoch will not run again since the epoch is already marked as filled. However, reading token0Total thereafter will yield incorrect values, potentially confusing off-chain clients. Additionally, the issue could be exacerbated as this flawed logic is extended or refactored in the future.

Recommendation

We recommend removing this sentinel-value hack entirely, or replacing it with a safer implementation.

Client's Commentary:

Commit: 4abf6e29

L-8	Epoch Remains Active After Full Empty by `kill()`		
Severity	Low	**Status**	Acknowledged

Description

When `LimitOrderManager.kill()` drains an epoch's liquidity to zero, the epoch remains marked as "unfilled". Consequently, future swaps that cross its ticks will still invoke `LimitOrderManager._fillEpoch()` on an epoch with no liquidity, wasting gas.

Recommendation

We recommend marking the epoch as filled immediately after its liquidity reaches zero in the `kill()` function.

Client's Commentary:

Acknowledged

L-9	Missing Existence Check for Epoch in kill()		
Severity	Low	**Status**	Acknowledged

Description

In LimitOrderManager.kill(), the code does:

```
EpochInfo storage epochInfo = epochInfos[epoch];
```

without verifying that epoch was ever initialized. If epoch is unset, this returns an empty storage slot and the function proceeds as if that epoch existed. Operating on such a "phantom" epoch can potentially produce silent misaccounting and leave the contract state in an inconsistent or confusing state, especially if later logic is extended or reordered.

Recommendation

We recommend checking that the epoch is set (for example, epoch != EPOCH_DEFAULT), and reverting with a clear error (e.g., EpochNotFound()) if it is not.

Client's Commentary:

The implicit check to determine whether the epoch has been initialized is sufficient

L-10	Premature Use of Unverified Epoch Data in withdraw()		
Severity	Low	**Status**	Fixed in b1adba18

Description

In the LimitOrderManager.withdraw() function, the contract loads epochInfo and immediately uses its fields—such as deployer, token0, and token1—to reconstruct the pool key and compute the pool address. Only after these operations does it check:

```
if (!epochInfo.filled) revert NotFilled();
```

If the epoch isn't actually filled, all of that preparatory work (and gas) is wasted, and constructing a pool address from uninitialized or zeroed fields may produce misleading or invalid values before the revert.

Recommendation

We recommend moving the `if (!epochInfo.filled) revert NotFilled();` check to immediately after loading epochInfo—before any use of its other fields—so that unfilled epochs are rejected at the earliest possible point.

Client's Commentary:

Commit: 7800fba9

L-11	Redundant `afterInitialize()` Function		
Severity	Low	**Status**	Fixed in `b1adba18`

Description

The `LimitOrderManager.afterInitialize` function exposed to plugins duplicates the exact same initialization that `LimitOrderManager.place()` already performs for any uninitialized pool. Since `place()` calls `_initialize()` when `initialized[pool] == false`, there is no scenario in which a pool remains uninitialized by the time a limit order is placed. Keeping `afterInitialize` merely expands the public API surface and increases maintenance complexity without providing any real benefit.

Recommendation

We recommend removing the `afterInitialize` endpoint and relying exclusively on the in-place initialization logic within `place()`.

Client's Commentary:

Commit: f8103fde

3. About MixBytes

MixBytes is a leading provider of smart contract audit and research services, helping blockchain projects enhance security and reliability. Since its inception, MixBytes has been committed to safeguarding the Web3 ecosystem by delivering rigorous security assessments and cutting-edge research tailored to DeFi projects.

Our team comprises highly skilled engineers, security experts, and blockchain researchers with deep expertise in formal verification, smart contract auditing, and protocol research. With proven experience in Web3, MixBytes combines in-depth technical knowledge with a proactive security-first approach.

Why MixBytes

- **Proven Track Record:** Trusted by top-tier blockchain projects like Lido, Aave, Curve, and others, MixBytes has successfully audited and secured billions in digital assets.
- **Technical Expertise:** Our auditors and researchers hold advanced degrees in cryptography, cybersecurity, and distributed systems.
- **Innovative Research:** Our team actively contributes to blockchain security research, sharing knowledge with the community.

Our Services

- **Smart Contract Audits:** A meticulous security assessment of DeFi protocols to prevent vulnerabilities before deployment.
- **Blockchain Research:** In-depth technical research and security modeling for Web3 projects.
- **Custom Security Solutions:** Tailored security frameworks for complex decentralized applications and blockchain ecosystems.

MixBytes is dedicated to securing the future of blockchain technology by delivering unparalleled security expertise and research-driven solutions. Whether you are launching a DeFi protocol or developing an innovative dApp, we are your trusted security partner.

Contact Information

https://mixbytes.io/

https://github.com/mixbytes/audits_public

hello@mixbytes.io

https://x.com/mixbytes