BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT:

Algebra - SwapX

Fee Plugin - Update Audit

July 2024

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Algebra Fee Plugin - Update Audit (differential)
Website	algebra.finance
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/cryptoalgebra/Algebra/tree/3a25f11a6b9fc871a2958d19748402180d855b9c
Resolution 1	https://github.com/cryptoalgebra/Algebra/tree/3513a9c97c7766d738d89646160fdfff8eb1f3be/src/periphery/contracts
Resolution 2	https://github.com/cryptoalgebra/Algebra/blob/e1f358c1c2c9f6cda455c6f0934b333af624e964/src/periphery/contracts/NonfungiblePositionManager.sol
Resolution 3	https://github.com/cryptoalgebra/Algebra/tree/0a15a8e807a424cba2d0cd06fca84976297f6424/src

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	1			1
Medium	2			2
Low	6	2		4
Informational	3	1		2
Governance	1			1
Total	13	3		10

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

Bailsec was tasked with a differential audit of Algebras NonfungiblePositionManager/Plugin architecture:

Old Commit:
https://github.com/cryptoalgebra/Algebra/tree/0fe96d86be5ae446901d3abb244b96be7600adeb

New Commit:
https://github.com/cryptoalgebra/Algebra/tree/3a25f11a6b9fc871a2958d19748402180d855b9c

Files in Scope:

1. NonfungiblePositionManager.sol (https://www.diffchecker.com/WzNG7bxX/)
2. BasePluginV1Factory.sol (https://www.diffchecker.com/L0XXu83W/)
3. AlgebraBasePluginV1.sol (https://www.diffchecker.com/xo9C2ghV/)

Primary Update Overview:

The key update in this differential audit is about the introduction of the withdrawalFee and the incorporation in the Plugin and the PluginFactory

Security Considerations:

a) Incorrect Fee Calculation: The main problem could be the incorrect calculation of the fee. This must be carefully investigated and all possible edge-cases must be considered.

b) Fee Circumvention: In certain scenarios or edge-cases it may be possible that users can circumvent paying the fee. This must be investigated

c) "Breaking" Changes: Changes, especially fee calculations can often introduce issues which result in a revert. This may be some arithmetic reverts or simply mistakes in the calculation that request a too large fee. Furthermore, due to the adjustments in the BasePluginV1Factory and AlgebraBasePluginV1, another potential scenario for accidental reverts is introduced.

d) Impact on Farming Module: Due to the fact that a fee is applied on the liquidity, this will have an impact on the Farming Module. It must be checked that this is reflected properly.

e) Impact on Rewards: Rewards are always based on the provided liquidity. It must be checked that these rewards are still calculated correctly.

NonfungiblePositionManager

The NonfungiblePositionManager is the main entry contract that allows users to interact with the Core and Farming modules.

Users can provide one or both tokens and mint LP tokens. These LP tokens are then wrapped into a NFT which represents the position. Specifically, the following interactions are possible:

a) Mint: Allows users to create a new tokenId by providing token0/token1 to the liquidity pool.

b) IncreaseLiquidity: Allows users to increase the position size on an already existing tokenId.

c) DecreaseLiquidity: Allows users to decrease the position size on an already existing tokenId.

d) Collect: Allows users to accrue fees from a position. Accrued fees are calculated since the last position update.

Furthermore, once a position/tokenId has been created, this position can be used to farm rewards within the Farming module.

A novel feature is the implementation of a withdrawal fee. This allows governance the introduction of withdrawal fees whenever a position is decreased and it works as follows:

1) A pool can be assigned with a withdrawal fee. This includes APR for token0 and token1 and a withdrawal fee.

2) Whenever a position is adjusted (increased/decreased), the withdrawal fee is calculated based on the apr settings and the withdrawal fee. This fee is then converted to the corresponding liquidity and stored in the withdrawalFeeLiquidity variable for the specific tokenId

3) Whenever a position is decreased, this fee will be deducted from a position's liquidity.

Below we will summarize all changes since the last commit:

- PositionWithdrawalFee struct. This struct exposes lastUpdateTimestamp and withdrawalFeeLiquidity and will be corresponding to each existing tokenId.
- FEE_DENOMINATOR: This variable is used for fee and apr calculation purposes and is denominated in 1e3.
- withdrawalFeesVault: This address will receive all withdrawal fees.
- withdrawalFeePoolParams[pool]: This mapping is corresponding to each pool, it exposes apr0, apr1 and withdrawalFee
- positionsWithdrawalFee: This view-only function shows the current fee state for a tokenId
- mint: The mint function was adjusted to set the initial fee state for a tokenId
- _calculateWithdrawalFees: This is the most fundamental function and calculates the fee for a tokenId since the last update. The fee is determined as liquidity.
- increaseLiquidity: The increaseLiquidity function was adjusted to update and store the fee since the last update time.
- decreaseLiquidity: The decreaseLiquidity function was adjusted to update and store the fee since the last update time as well as deducting the fee from the position. It was implemented to adjust for several edge-cases.
- Different setters: Setter function for aprs, withdrawalFee and withdrawalFeesVault were implemented.

Privileged Functions:

- setFarmingCenter
- setTokenAPR
- setWithdrawalFee
- setVaultAddress

Issue_01	Governance Privilege: Owner can set exorbitant fee
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. This includes setting a large APR/withdrawalFee.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_02	Liquidity fetching approach is incorrect
Severity	**High**
Description	Fee calculation flow explained: 1. Period since last update is determined 2. TWAT since last update is fetched 3. amount0/amount1 for position based on liquidity and TWAT are fetched 4. Fee on one or both tokens is calculated based on time passed, apr and withdrawalFee 5. Fee amount(s) from token0/1 is converted to the corresponding liquidity After the fee in token0 and token1 has been determined, the corresponding liquidity for these amounts is fetched (step 5). This is done by invoking the getLiquidityForAmount0/1 functions. The problem: These functions can only be used when the position is one-sided, otherwise they will return an incorrect value. *We will*

demonstrate this with a PoC below.

Now one would think it may be possible to use the getLiquidityForAmounts function, which is possible if the ratio between both tokens is exactly the same and if the fee is the same. If this is not the case, only the smaller of both liquidities will be returned:

https://github.com/cryptoalgebra/Algebra/blob/3a25f11a6b9fc871a2958d19748402180d855b9c/src/core/contracts/test/LiquidityAmounts.sol#L71

Illustrated incorrect version:

- 100000e6 liquidity
- currentTick = 0 (79228162514264337593543950336)
- lowerTick = - 10000 (48055510970269007215549348797)
- upperTick = 10000 (130621891405341611593710811006)

This will correspond to:

amount0 = 39345417784
amount1 = 39345417784

If we now consider a 10% fee on both amounts this will result in the following fee:

fee0 = 3934541778
fee1 = 3934541778

Now following the amount to liquidity conversion using getLiquidityForAmount0/1:

liquidityForFee0 = 3775465434
liquidityForFee1 = 3775465434

	Summarized, the following liquidity will be taken as fee: liquidityTakenTotal = 7550930868 As one can clearly see, this is not 10% of 1000000e6. We can also use the same example and only apply a 10% fee on tokenX or tokenY (not on both), this will then result in a liquidity fee of 3775465434, which does not represent 10%/2. Furthermore, if we test with different position size: tickLower: -1, tickHigher: 1, tickAvg: 0 liquidity: 1e18, amount0: 4.9996e13, amount1: 4.9996e13 feeLiq: 9.9997e16, fee0: 4.9995e12, fee1: 4.9995e12 tickLower: -100, tickHigher: 100, tickAvg: 0 liquidity: 1e18, amount0: 4.9872e15, amount1: 4.9872e15 feeLiq: 9.9750e16, fee0: 4.9748e14, fee1: 4.9748e14 tickLower: -10000, tickHigher: 10000, tickAvg: 0 liquidity: 1e18, amount0: 3.9345e17, amount1: 3.9345e17 feeLiq: 7.5509e16, fee0: 2.9709e16, fee1: 2.9709e16 tickLower: -500000, tickHigher: 500000, tickAvg: 0 liquidity: 1e18, amount0: 9.9999e17, amount1: 9.9999e17 feeLiq: 2.7810e6, fee0: 2.7810e6, fee1: 2.7810e6 We can see that the bigger the position, the lower the fees. This could be used by users to pay 0 fees and is inconsistent between positions that are highly concentrated and those that are very wide. Note that this is almost not the case for positions that are single sided. Such positions will almost all have the same fee.
Recommendations	**The problem: Algebra/Uniswap was never developed with the need to fetch liquidity which is corresponding to a single token in**

	a position which is two-sided. (For a one-sided position the used concept would actually work) **Such a state would be invalid because a two-sided position always consists of tokenX/tokenY. Hence, such a functionality does not exist in the architecture.** It goes without saying that such a development is non-trivial and a lot of resources will flow into auditing such a novel implementation. It will require the following: a) Mathematical derivation using UniswapV2's Whitepaper b) Extensive fuzzing
Comments / Resolution	Acknowledged, the team decided to accept this risk and added invariant tests which ensure that the fee can never become higher than the anticipated percentage. This ensures users will never bear a loss but can trick the system to pay less fee than expected. It goes without saying that this issue was not fixed and invariant tests are no guarantee that certain scenarios won't happen, however, due to the circumstances this solution seemed as the path of the least resistance. Eventual side-effects cannot be excluded.

Issue_03	Change of APR / withdrawalFee will alter fee in hindsight for all positions
Severity	**Medium**
Description	Currently, the setTokenAPR and setWithdrawalFee functions allow to alter the fee which is corresponding to a pool and its tokens. If this value is changed, it will alter all fee calculations in hindsight. **Illustrated:** a) Alice has a large position and token0 has currently an APR of 10% and a withdrawalFee of 10%. The staked at the beginning of the year and has since then not manipulated her position. b) Governance changes apr for token0 to 100% and withdrawalFee to 100% in the second half of the year. c) Alice will effectively pay this fee for the first half of the year as well.
Recommendations	There is no trivial code-based solution for this problem. A possibility would be to implement a function that loops over all tokenIds and updates their fee state. This would need to be carefully audited to ensure it doesn't have negative side-effects. However, currently we are just recommending to inform the community such that they can update their positions beforehand.
Comments / Resolution	Acknowledged.

Issue_04	APR for Vault Token needs adjustment compared to the underlying asset
Severity	**Low**
Description	The earnings from staking are currently calculated using the formula: (tokenAPR * period * liquidityAmount) / (FEE_DENOMINATOR * 365 days) For vault tokens (e.g., wstETH), this calculation is incorrect. For example, if the APR is 5% and a user stakes 1 wstETH for a year, the amountEarnedFromStake is calculated as 0.05 wstETH. However, due to the compounding nature of wstETH, the actual earnings would be closer to 0.0476 wstETH. Therefore, the APR for vault tokens needs to be adjusted to account for compounding.
Recommendations	Consider carefully adjusting the APR calculation for vault tokens to avoid overestimating the earnings. In our example, the APR should have been set to `0.05 / 1.05 = 0.0476`
Comments / Resolution	Acknowledged.

Issue_05	Lack of validation for withdrawalFeesVault upon contract deployment
Severity	**Low**
Description	The withdrawalFeesVault is set upon contract deployment but lacks an address(0) check. In such a scenario where it is set to address(0) and remains address(0), it will break fee collection and therefore decreaseLiquidity will result in a DoS.
Recommendations	Consider validating it accordingly within the constructor.
Comments / Resolution	Resolved.

Issue_06	Fee deduction will not impact farming position/position size until decreaseLiquidity function is invoked
Severity	**Low**
Description	Currently, the fee is accumulated during the increaseLiquidity and decreaseLiquidity functions. However, it is only applied during the decreaseLiquidity function. Therefore, even if users theoretically have to pay a fee, this fee is not realized until a position is decreased. Thus users can benefit from their full position size for liquidity and farming rewards.
Recommendations	A solution would be to decrease the liquidity also when the position is increased. However, we do not recommend this change as it incorporates complexity and increased gas costs. We simply recommend acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_07	Uncollected fees can still be taken if fee exceeds position liquidity
Severity	**Informational**
Description	In the event where the withdrawalFee exceeds the position size, the fee will be limited to the position size: https://github.com/cryptoalgebra/Algebra/blob/3a25f11a6b9fc871a2958d19748402180d855b9c/src/periphery/contracts/NonfungiblePositionManager.sol#L444 This will however not incorporate any accrued rewards. Therefore, even if users may be solvent due to accrued rewards, none of these rewards would be taken to pay the excess fee.
Recommendations	This is likely a design issue and incorporating the unclaimed rewards would mean that the contract must be rewritten. Therefore this issue can be acknowledged if this is in fact desired per design.
Comments / Resolution	Acknowledged.

Issue_08	DecreaseLiquidity event emits incorrect liquidity parameter
Severity	**Informational**

Description	The DecreaseLiquidity event has the following parameters:
	
	event DecreaseLiquidity(
	    uint256 indexed tokenId,
	    uint128 liquidity,
	    uint128 withdrawalFeeliquidity,
	    uint256 amount0,
	    uint256 amount1
	);
	
	The liquidity parameter represents how much was de facto withdrawn.
	
	See actual implementation:
	
	emit DecreaseLiquidity(params.tokenId, params.liquidity, positionWithdrawalFeeLiquidity, amount0, amount1);
	
	In the scenario where params.liquidity > liquidityDeltaWithoutFee, this event emission is incorrect.

Recommendations	Consider using liquidityDeltaWithoutFee as 2nd parameter.
Comments / Resolution	Resolved.

BasePluginV1Factory

The BasePluginV1Factory contract is the main Factory contract for Plugins. It exposes functionality that is invoked before a Pool creation and will deploy the BasePlugin which is then plugged onto the created pool. Furthermore it keeps track of the newly developed plugin.

The only change which was made is the introduction of the modifyLiquidityEntryPoint variable and its corresponding setter function. This variable is solely fetched upon the AlgebraBasePluginV1 deployment to set the corresponding entry point.

Privileged Functions:

- setDefaultFeeConfiguration
- setFarmingAddress
- setModifyLiquidityEntrypoint

No issues found.

AlgebraBasePluginV1

The AlgebraBasePluginV1 contract is the base plugin which is used for pool deployments. It incorporates all base hook functions and handles the dynamic fee.

The only change to the previous version is the addition of the BEFORE_MODIFY_POSITION_FLAG and the modifyLiquidityEntryPoint. Whenever a position is modified (mint/burn), the beforeModifyPosition hook is called. This hook ensures that the caller is in fact the modifyLiquidityEntryPoint (NonfungiblePositionManager).

Privileged Functions

- setModifyLiquidityEntrypoint
- setIncentive
- changeFeeConfiguration

Issue_09	Inability to invoke setModifyLiquidityEntrypoint via factory
Severity	**Low**
Description	The setModifyLiquidityEntrypoint function can either be invoked by the factory owner or the pluginFactory. The BasePluginV1Factory does however not expose such a cross-contract call to deployed plugins.
Recommendations	Consider removing the ability for the pluginFactory to invoke this function.
Comments / Resolution	Resolved.

NonfungiblePositionManager

After the audit, an update has been implemented on the NonfungiblePositionManager which introduces the distribution of withdrawal fees to different addresses based on the pool.
In the previous iterations, all fees were withdrawn to one single address. Now it is allowed to determine multiple different withdrawal addresses for each pool.

This will result in the following diff-check:

https://www.diffchecker.com/X3csW02v/

The following changes are explicit:

a) Introduction of multi-distribution for withdrawal fee to different vaults with different percentages
b) Introduction of setVaultsForPool function which allows to set different vaults with corresponding fee for each pool
c) Addition of feeVaults array in the WithdrawalFeePoolParams struct

Disclaimer: This audit involves only the changes provided by the corresponding diffchecker files. Please be advised that for issues which are reported outside of the diffchecker scope, an additional resolution must be scheduled. A differential audit is always a constrained task because not the full codebase is re-audited. This will have inherent consequences if intrusive changes have side-effects on parts of a codebase/module, which is not part of the audit scope.

Issue_10	Position withdrawal in case of blacklisted vault will result in DoS
Severity	**Low**
Description	The decreaseLiquidity function transfers out fees either to one default address or to multiple different vaults. If one of these vaults is blacklisted for the fee token, this will result in a revert of the transfer and thus in a DoS of the decreaseLiquidity function until the vault is removed.
Recommendations	Consider removing the vault in such a scenario.
Comments / Resolution	Acknowledged.

Issue_11	Truncation during fee collection for multiple vaults
Severity	**Informational**

Description	The multi fee distribution is calculated as follows:
	
	`for (uint i = 0; i < vaults.length; i++) {`
	`uint16 feePart = vaults[i].fee;`
	`pool.collect(`
	`vaults[i].feeVault,`
	`tickLower,`
	`tickUpper,`
	`uint128((amount0 * feePart) / FEE_DENOMINATOR),`
	`uint128((amount1 * feePart) / FEE_DENOMINATOR)`
	`);`
	`}`
	
	The practice of percentage proportional calculations will always result in truncation for solidity operation.

Recommendations	This issue can be acknowledged because it will only accumulate dust and no noticeable amounts.
Comments / Resolution	Acknowledged.

[UPDATE] Entry Point Whitelisting

After the initial update and the above mentioned update, a 2nd update has been introduced which is reflected as resolution 3 under the following commit:

https://github.com/cryptoalgebra/Algebra/tree/0a15a8e807a424cba2d0cd06fca84976297f6424/src

This update modifies the following contracts:

a. NonfungiblePositionManager
b. BasePluginV1Factory
c. AlgebraBasePlugin

The main change which has been introduced is the shift from one entry point to multiple entry points. The rationale behind this change is to not only allow the NonfungiblePositionManager to modify a position but also potentially other third party contracts. The side-effect of that is that every party which is whitelisted can interact directly with the AlgebraPool contract, bypassing the LST fee.

The following changes were made explicit:

a. NonfungiblePositionManager: https://www.diffchecker.com/TbiByIVb/
   - Implementation of calculateLatestWithdrawalFeesLiquidity: This function calculates and returns the full pending withdrawalFeeLiquidity for a positionId, up to the current block.timestamp.
   - Implementation of calculatePendingWithdrawalFeesLiquidity: This function calculates the current pending fee from lastUpdateTimestamp to block.timestamp without the existing withdrawalFeeLiquidity.

b. BasePluginV1Factory: https://www.diffchecker.com/s5rk3da4/
   - Implementation of multiple entry points. Instead of as in the previous iteration where the contract had only one entry point: modifyLiquidityEntrypoint, the contract now exposes a mapping: modifyLiquidityEntryPointsStatuses, which can be set by governance and now allows for the addition of multiple entry points.
   - Functions for adding and removing entry points were created.

- Function selector change of beforeCreatePoolHook to createPlugin which is now standardized to Algebra V1.0

c. AlgebraBasePluginV1: https://www.diffchecker.com/SnXhyIvU/
- Logic for single entry point was removed
- beforeModifyPosition function was adjusted to now consult the BasePluginV1Factory.modifyLiquidityEntryPointStatuses mapping

Issue_12	View-only calculateLatestWithdrawalFeesLiquidity may return incorrect data in case of hindsight update
Severity	**Medium**
Description	Within the main review, we have already raised the issue that if the APR of withdrawalFee is changed, this will skew the fee calculation since lastUpdateTime. In the scenario where a third-party provider uses the calculateLatestWithdrawalFeesLiquidity function uses and relies on the correctness of it, this issue is even amplified as such a hindsight update will essentially manipulate the assumption which can result in critical issues depending on the implementation.
Recommendations	There is no trivial fix besides removing the view-only nature of the function and automatically updating the fee and lastUpdateTime. However, such a change would mean a 4th iteration of this audit is needed which will significantly increase the risk. Thus we recommend executing a dusting attack on all existing positions to update their state before changing the fee.
Comments / Resolution	Acknowledged. We advise users interacting with the calculateLatestWithdrawalFeesLiquidity onchain to make the call in the same transaction in which the position is interacted with

Issue_13	Isolated pool entry points are impossible
Severity	**Low**
Description	In the previous iteration, the entry point was set during the initialization to the state variable within the factory. However, the setModifyLiquidityEntrypoint function allowed governance to change the entry point which allowed each pool to have its own entry point without requiring other pools to follow the same. This flexibility is now removed as the factory is considered directly within the beforeModifyPosition function.
Recommendations	Consider if this limited flexibility is acceptable, if not a different approach is required.
Comments / Resolution	Acknowledged. This limited flexibility is acceptable to us.