BAIL security

Algebra Fee Architecture Update (differential)

FINAL REPORT

February ‘2025

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 Architecture Update (differential)
Website	algebra.finance
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/cryptoalgebra/Algebra/tree/95b2a8de87d5ced321fb1a880ee70e2b24add619/src
Resolution 1	

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed resolution
High					
Medium					
Low	5			5	
Informational					
Governance					
Total	5			5	

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 by Algebra:

Old Commit:

https://github.com/cryptoalgebra/Algebra/tree/2df017183bc3d326284ba85c649ddfade9686115/src/core

New Commit:

https://github.com/cryptoalgebra/Algebra/tree/95b2a8de87d5ced321fb1a880ee70e2b24add619/src

Files in Scope:

1. SwapCalculation.sol: https://www.diffchecker.com/od6DTQD0/
2. AlgebraPool.sol: https://diffchecker.com/mPVQTFAk

Primary Update Overview:

The key update in this differential audit involves the following change

1) **Change in how the fee within _calculateSwap is applied:**
    a) The communityFee is applied before the pluginFee which increases the nominal fee amount towards the community. This has the effect that more value will be assigned to the communityFee.

2) **Change within _beforeSwap which implements a strict revert condition in case the plugin is wrongly configured**
    a) It is prohibited to have an existing overrideFee or pluginFee if the plugin disabled the DYNAMIC_FEE flag. This has the effect that a plugin is not accidentally marked without the DYNAMIC_FEE flag while still overriding the static fee.

3) **Change within setFee which handles the access control for said function**
    a) It is prohibited for the Plugin to call the setFee function. Since the setFee function is in this iteration only used to set the static fee and not the dynamic fee, the privilege of the static fee change solely remains towards governance.

Security Considerations: All changes are straightforward in themselves. The main security consideration for this audit is any unexpected side-effect from the new fee determination flow. To address this and find potential bugs, we will examine the full fee flow in the old version versus the full fee flow in the new version.

SwapCalculation

The SwapCalculation contract is responsible for calculating the outcome of a swap via the _calculateSwap function. It allows for overriding the previous fee and/or adding a pluginFee towards it.

Previous flow:

Scenario 1: Zero overrideFee

The function fetches the fee from the contract's state and optionally adds the pluginFee on top. Based on this fee, the nominal fee amount is calculated.

Afterwards the pluginFee is deducted which increases the leftover nominal fee and after that, the communityFee is deducted which also decreases the leftover nominal fee.

The leftover amount is then used to calculate the liquidity provider fee.

Scenario 2: Non-zero overrideFee

The function fetches the fee from the contract's state and simply overrides it with the overrideFee. The pluginFee is then optionally applied on top of the overrideFee. Afterwards it follows the same pattern as in scenario 1: Calculating pluginFee amount, decreasing the leftover fee and calculating the communityFeeAmount based on the decreased leftover fee and again decreasing the leftover fee. The final leftover fee is then used to calculate the liquidity provider fee.

New flow:

Scenario 1: Zero overrideFee

The function fetches the fee from the contract's state and optionally adds the pluginFee on top. Based on this fee, the nominal fee amount is calculated during the swap.
Afterwards the communityFee is calculated which decreases the leftover nominal fee:

> communityFee = nominalFee * communityFee / COMMUNITY_FEE_DENOMINATOR

and after that the pluginFee is calculated which also decreases the leftover nominal fee.

> pluginFee = nominalFee * pluginFee / fee

The leftover amount is then used to calculate the liquidity provider fee.

> feeAmount / currentLiquidity

Using this approach, the communityFee is calculated based on the initial nominal fee and the pluginFee is calculated based on the leftover fee after the communityFee. This has the effect that more value is assigned towards the communityFee.

Scenario 2: Non-zero overrideFee

The function fetches the fee from the contract's state and overrides it. The new fee now consists out of the overrideFee plus the pluginFee:

> fee = overrideFee + pluginFee

Based on this fee, the nominal fee amount is calculated during the swap.

Afterwards the communityFee is calculated based on the nominal which decreases the leftover nominal fee:

> communityFee = nominalFee * communityFee / COMMUNITY_FEE_DENOMINATOR

and after that the pluginFee is calculated based on the leftover nominal fee which also decreases the leftover nominal fee.

> pluginFee = nominalFee * pluginFee / fee

The leftover amount is then used to calculate the liquidity provider fee.

> feeAmount / currentLiquidity

Using this approach, the communityFee is calculated based on the initial nominal fee and the pluginFee is calculated by the nominal fee after the communityFee. This has the effect that more value is assigned towards the communityFee.

Appendix: Invariants for changes

INV 1: communityFee must always be deducted first and calculated based on initial step.feeAmount

INV 2: pluginFee must be calculated on fraction of step.feeAmount after communityFee has been deducted

INV 3: After deducting both fees, the remainder must be >= 0

Privileged Functions

- N/A for differential-audits

Issue_01	No pluginFee if communityFee is set to 100%
Severity	**Low**
Description	In the previous iteration, it was not possible that pluginFee is set to 100% due to the following check: cache.fee += pluginFee; if (cache.fee >= 1e6) revert incorrectPluginFee(); It was theoretically possible that the fee is set to 0.1% and the pluginFee to 99.9% but in the standard scenario where the fee is a reasonable value, the pluginFee cannot be near 100%. This means, it was guaranteed that under all circumstances, a communityFee is taken (unless the actual fee itself was zero or near zero) In the current implementation, it is possible that the communityFee is set to 100% (even if the fee itself is a reasonable value), which can then result in the whole fee being taken as communityFee: *if (cache.communityFee > 0) {* *uint256 delta = (step.feeAmount.mul(cache.communityFee)) / Constants.COMMUNITY_FEE_DENOMINATOR;* *step.feeAmount -= delta;* *fees.communityFeeAmount += delta;* *}* Notably, if pluginFee is > 0, it means that the communityFee is taken based on overrideFee + pluginFee, which means even the full pluginFee share is taken towards the communityFee. If communityFee is 100%, step.feeAmount becomes zero and essentially no pluginFee is taken, even if pluginFee > 0
Recommendations	Consider keeping this in mind and setting always a reasonable communityFee.

Comments / Resolution	Acknowledged: This is desired behaviour. If communityFee is 100% then *everything*: LPfee + pluginFee goes to the DEX.

Issue_02	The swap fee can never be overridden to 0
Severity	**Low**
Description	*if (overrideFee != 0) {* *cache.fee = overrideFee + pluginFee;* *if (cache.fee >= 1e6) revert incorrectPluginFee();* *} else {* *if (pluginFee != 0) {* *cache.fee += pluginFee;* *if (cache.fee >= 1e6) revert incorrectPluginFee();* *}* *}* Currently the overrideFee which is given by the plugin only replaces globalState.lastFee if it is greater than 0. There might be use cases where it is desired that the swap fee is set to 0. In the previous iteration, this was simply possible by addressing this via the plugin by calling setFee. In the current setup, this would need to be done by governance via the setFee function. However, this is not allowed if DYNAMIC_FEE flag is true. Additionally, the contract is forced to use lastFee if the overrideFee is zero, which can result in unexpected side-effects.
Recommendations	Consider allowing governance to call the setFee function even if DYNAMIC_FEE flag is true, with the goal to allow using the lastFee value if overrideFee is zero.
Comments / Resolution	Acknowledged: Algebra is aware of this. However, can be set to 0 with .setFee() called manually

AlgebraPool

The AlgebraPool contract includes all functionalities for users to create and modify liquidity positions. Within the scope of this differential audit lies the fee adjustment mechanism, related to the dynamic fee approach.

Previous flow:
In the last audited version, it was already possible that an overrideFee and/or pluginFee is determined by the plugin within the _beforeSwap function. However, in the AlgebraBasePluginV1 itself, it was always mandatory to write the fee to the AlgebraPool contract via the setFee function.
Additionally to that, it was possible to override this newly set fee and/or add the pluginFee on top of it.

Summarized, the dynamic fee was possible via two scenarios:

a) Overriding the existing fee via the overrideFee
b) Writing the dynamic fee to the AlgebraPool storage via the setFee function

New flow:
The current version follows a different approach than the previous flow. While in the previous flow the dynamic fee was (as explained) possible via:

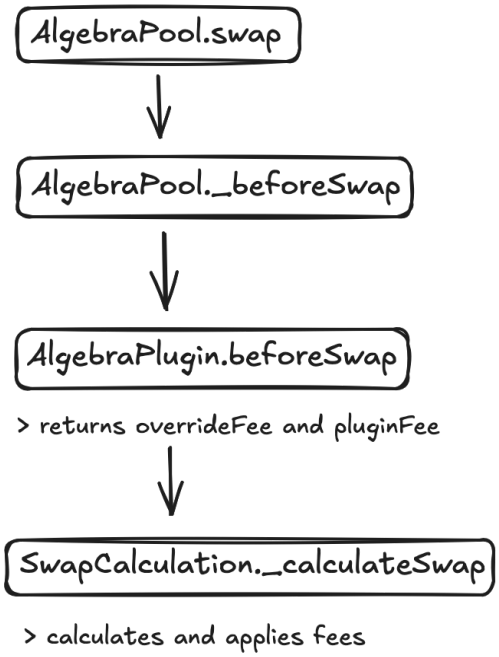
a) Overriding the existing fee the overrideFee
b) Writing the dynamic fee to the AlgebraPool storage via the setFee function

This new approach solely relies on a) and removes the previous ability of overriding lastFee with the new dynamic fee in storage. This means that the lastFee variable within the AlgebraPool storage will never be altered unless governance changes it - even if the dynamic fee model is applied.

This approach is cleaner and more gas-efficient as it simply leverages the overriding possibility to introduce the dynamic fee instead of writing to storage before every swap (if the fee has changed).

Moreover, it is now explicitly prohibited for the plugin to call the setFee function and the overrideFee and pluginFee must be zero if the DYNAMIC_FEE flag is false. It is also not allowed to call the setFee function if the DYNAMIC_FEE flag is set to true.

The flow for the new model is as follows:



Summarized, there are now two distinct fee settings:

a) Static/Non-Dynamic:

This fee is determined by the lastFee variable within the AlgebraPool storage. It is required that the DYNAMIC_FEE flag is set to false and additionally requires that the overrideFee and pluginFee return values from the plugin are both zero. If one or both values are non-zero, the swap will revert. Governance can determine this fee.

b) Dynamic:

This fee is solely determined by the return value of the plugin. It is expected that the plugin returns an overrideFee != 0 and optionally a pluginFee while the DYNAMIC_FEE flag is set to true. While it is not inherently enforced that the overrideFee is non-zero, it could still theoretically be possible that a swap uses the lastFee variable. However, this is usually not what is expected.

Appendix: Invariants for changes

INV 1: If DYNAMIC_FLAG is disabled, overrideFee and pluginFee must be zero

INV 2: Only governance can call setFee

INV 3: globalState.lastFee must only be updated via setFee and represents the static fee

INV 4: Dynamic fee application must never override globalState.lastFee
INV 5: overrideFee must be non-zero if DYNAMIC_FEE flag is set to true

Privileged Functions

- N/A for differential-audits

Issue_03	Old AlgebraBasePluginV1 can never be used with new implementation
Severity	**Low**
Description	The old AlgebraBasePluginV1 enforced the call to setFee in the scenario where the new dynamic fee is different from lastFee: `if (newFee != fee) {` `IAlgebraPool(pool).setFee(newFee);` `}` This flow will not work anymore because the setFee function is now only callable by governance.
Recommendations	Consider simply using a plugin which uses the overrideFee for dynamic fee purposes.
Comments / Resolution	Acknowledged, only the new plugin will be used.

Issue_04	lastFee variable cannot be used to determine dynamic fee
Severity	**Low**
Description	In the previous implementation, lastFee was updated by the plugin and set to the current dynamic fee (once per block; if necessary): *if (_lastTimepointTimestamp == currentTimestamp) return;* *(, int24 tick, uint16 fee,) = _getPoolState();* *(uint16 newLastIndex, uint16 newOldestIndex) = timepoints.write(_lastIndex, currentTimestamp, tick);* *timepointIndex = newLastIndex;* *lastTimepointTimestamp = currentTimestamp;* *uint16 newFee = _getFeeAtLastTimepoint(newLastIndex, newOldestIndex, tick, feeConfig_);* *if (newFee != fee) {* *IAlgebraPool(pool).setFee(newFee);* *}* This means, lastFee always reflected the latest dynamic fee and was thus a reliable view-only indicator of the current fee. In the current implementation, as described, lastFee only reflects the static fee and is at no point a reliable indication of the current fee.
Recommendations	Consider using a different mechanism to display the swap fee. This should be communicated with all partners that are integrating this version, since some may simply rely on the lastFee variable.
Comments / Resolution	Acknowledged, Algebra has provided a new algorithm of fee simulation.

Issue_05	pluginFee not allowed in case of static fee
Severity	**Low**
Description	If a plugin is exposed but the DYNAMIC_FEE flag is set to false, it is expected that the lastFee variable is used and overrideFee as well as pluginFee is zero . However, generally speaking, it could theoretically be desired that pluginFee is still applied, even in a static fee scenario. As long as the pluginFee is consistent, it does not violate the static fee nature. This is currently due to the following limitation not possible: *if (!pluginConfig.hasFlag(Plugins.DYNAMIC_FEE) && (overrideFee > 0 || pluginFee > 0)) revert dynamicFeeDisabled();*
Recommendations	Consider if it is desired to allow a pluginFee also for the static fee scenario. If yes, the above highlighted snippet must be slightly refactored for pluginFee to be non-zero. It must be additionally ensured that the pluginFee is consistent.
Comments / Resolution	Acknowledged, Non-zero plugin fee automatically implies that fee is not static. Static - means that the swap fee is entirely defined by`lastFee`

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.