BAILSEC.IO
OFFICE@BAILSEC.IO
X: @BAILSECURITY
TG: @HELLOATBAILSEC

FINAL REPORT

Algebra Core

Update Audit (differential)

August 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

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

Project	Algebra Core - Update Audit (differential)
Website	algebra.finance
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/cryptoalgebra/Algebra/tree/3681a498219e44b1e8e6776166ac8ea2aeea289f/src/core/contracts
Resolution 1	https://github.com/cryptoalgebra/Algebra/tree/2df017183bc3d326284ba85c649ddfade9686115/src/core

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	2	2		
Medium				
Low	4	2		2
Informational	1			1
Governance	1			1
Total	8	4		4

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 Algebra’s Core.

Latest audited commit:
https://github.com/cryptoalgebra/Algebra/tree/28f5b33a29a2b81352e84652a9782e7e4b750f bc/src/core/contracts

New Commit:
https://github.com/cryptoalgebra/Algebra/tree/3681a498219e44b1e8e6776166ac8ea2aeea289 f/src/core/contracts

Files in Scope:

1. /base/AlgebraPoolBase.sol
https://www.diffchecker.com/5hRJaV0u/

2. /base/ReservesManager.sol
https://www.diffchecker.com/ZQcJA9wn/

3. /base/SwapCalculation.sol
https://www.diffchecker.com/h1LJfxEw/

4. /AlgebraPool.sol
https://www.diffchecker.com/AL6mnlhn/

5. /AlgebraFactory.sol
https://www.diffchecker.com/TFtESGDE/

6. /libraries/Constants.sol
https://www.diffchecker.com/caRX99up/

Primary Update Overview:

The key update from the previous iteration is an additional plugin fee on top of the community fee. This fee is determined within the plugin itself and will be applied on two occasions:

a) During swaps, following the same mechanics as the communityFee. The liquidity provider fee will be simply decreased by the communityFee and now additionally also by the pluginFee.

b) During LP removal/burns. The fee will be simply deducted from the received amount.

Security Considerations: The main change in the architecture is the implementation of the mentioned, additional fee. This will expose the following risk.

a) Correctness of fee calculation: There may be issues with taking an excess/insufficient fee. In the scenario of excess fee, this could result in a loss of funds or DoS. It is important to ensure that fees align with how much was actually taken from the base fee.

b) Correctness of fee distribution/allocation: After the fee has been correctly calculated, it must be ensured that the distribution or allocation is correct.

c) Side-effects on the overall architecture: All possible side-effects which could happen due to this fundamental change must be properly checked.

d) Side-effects due to incorrect assembly usage within the updateFeeAmounts function.

Below we will highlight all important changes in-depth:

Audit Changes

Fee Calculation Old:

Previously, the nominal fee was calculated within the PriceMovementMath._movePriceTowards target function based on the dynamic fee share and the input amount.
Once the nominal fee was then determined, a share for the community was taken and the leftover nominal fee was determined:

```
        uint256 delta = (step.feeAmount.mul(cache.communityFee)) /
Constants.COMMUNITY_FEE_DENOMINATOR;
        step.feeAmount -= delta;
        communityFeeAmount += delta;
```

Fee Calculation New:

Similar as in the old methodology, the nominal fee is calculated within the PriceMovementMath._movePriceTowardsTarget function. However, this is not done using the dynamic fee but rather using the static overrideFee value. This overrideFee is mandatory to be set when the pluginFee is activated.

After the nominal fee is calculated, a share for the plugin **and** community part is taken:

```
        if (cache.pluginFee > 0 && cache.fee > 0) {
        uint256 delta = FullMath.mulDiv(step.feeAmount, cache.pluginFee,
cache.fee);
        step.feeAmount -= delta;
        fees.pluginFeeAmount += delta;
        }

        if (cache.communityFee > 0) {
        uint256 delta = (step.feeAmount.mul(cache.communityFee)) /
Constants.COMMUNITY_FEE_DENOMINATOR;
        step.feeAmount -= delta;
```

```
fees.communityFeeAmount += delta;
}
```

Reserve Change Old:

The _changeReserves function was invoked upon the following interactions:

- mint
- collect
- swap
- swapWithPaymentInAdvance
- flash

This function simply adjusts the reserves based on the corresponding amounts for the operation with the delta to any pending community fee.

If for example a mint function results in 100e18 tokenX and 100e18 tokenY being added to the pair while the pair has an outstanding fee of 10 tokenX and 10 tokenY, the reserves will be increased by 90e18 tokenX and 90e18 tokenY.

Additionally, the pending fee is transferred out which then further decreases the reserve delta.

Reserve Change New:

The _changeReserves function is invoked upon the following interactions:

- mint
- collect
- swap
- swapWithPaymentInAdvance
- flash

Contrary to the previous implementation, there are 6 parameters instead of only 4 parameters:

- deltaR0
- deltaR1
- communityFee0
- communityFee1
- **pluginFee0**
- **pluginFee1**

Furthermore, a step in between, namely a call to the **updateFeeAmounts** function has been implemented. This function does the following:

> fetches current pending fees
> optionally increases pending fees by new additional fees
> optionally transfers fees out
> optionally decrease deltaR0/1 by these fees
> returns adjusted deltaR0/1

This is more or less similar to what was done in the previous iteration with the following additions:

-> Usage of assembly for fetching variables and storing new state
-> Implementation of additional pluginFee
-> Callback to the plugin after successful plugin fee transfer

Burn Function Old:

The previous burn function was trivial and worked follows:

> lock the contract
> update reserves
> update position/ticks/fees based on liquidity to remove
> returns received amount0/amount1
> save new position.fees

Burn Function New:

The new burn function incorporates an additional fee which is applied on the received amount0/1, it works as follows:

> fetch pluginFee
> lock the contract
> update reserves
> update position/ticks/fees based on liquidity to remove
> returns received amount0/amount1
> calculate fee based on pluginFee and amount0/1
    > decrease amount0/1
    > increase pluginFeePending0/1
> unlock the contract

Resolution 1 Changes

During the resolution, the following changes have been implemented:

ReservesManager:

The flow for changing reserves, updating and transferring pending fees has been completely refactored. A new _accrueAndTransferFees function has been developed which takes care of accruing and transferring fees. The following callpaths are exposed:

Call Path 1: Transfer fees and update storage

Conditions:

-> fee0 or fee1 is non-zero

-> frequency is reached or uint104 is exceeded

Actions:

-> Load feePending0 and feePending1 from storage using assembly

-> Add fee0 to feePending0 and fee1 to feePending1

-> Load recipient from storage using assembly

-> Call _transferFees(feePending0, feePending1, recipient) to transfer the fees

-> Update the storage slot specified by feePendingSlot with zeros using assembly

-> Return (0, 0, feeSent0, feeSent1)

-> decrease delta0/1 by feeSent0/1

-> update lastFeeTransferTimestamp

Call Path 2: Update pending fees and return without transferring

Conditions:

-> fee0 or fee1 is non-zero

-> frequency is not reached and uint104 is not exceeded

Actions:

-> Load feePending0 and feePending1 from storage using assembly

-> Add fee0 to feePending0 and fee1 to feePending1

-> Return (uint104(feePending0), uint104(feePending1), 0, 0)

-> Update storage to reflect increased pending fees

—---

Call Path 3: Transfer pending fees and update storage

Conditions:

-> fee0 and fee1 are both zero

-> frequency has been reached

-> feePending0 or feePending1 is non-zero

Actions:

-> Load feePending0 and feePending1 from storage using assembly

-> Load recipient from storage using assembly

-> Call _transferFees(feePending0, feePending1, recipient) to transfer the pending fees

-> Update the storage slot specified by feePendingSlot with zeros using assembly

-> Return (0, 0, feeSent0, feeSent1)

-> decrease delta0/1 by feeSent0/1

-> update lastFeeTransferTimestamp

—---

Call Path 4: Return without transferring or updating pending fees

Conditions:

-> fee0 and fee1 are both zero

-> Either frequency not passed or both pending fees are zero.

Actions:

-> Return (0, 0, 0, 0)

SwapCalculation:

Whenever the overrideFee is defined and the pluginFee is defined, the following overall fee will be taken:

> overrideFee + pluginFee

Whenever the overrideFee is not defined but the pluginFee is defined, the following overall fee will be taken:

> globalState.lastFee + pluginFee

Whenever the overrideFee is defined and the pluginFee is not defined, the following overall fee will be taken:

> overrideFee

Whenever the overrideFee is not defined and the pluginFee is not defined, the following overall fee will be taken:

> globalState.lastFee

AlgebraPool

The burn function has been adjusted as per our recommendation to fix the issue:

“Uint104 implementation for pluginFeePending can result in loss of funds and DoS of the burn call”

The _changeReserves function is now invoked with the plugin fees as parameter which will then either increase pluginFeePending0/1 and write the increased value to storage or simply transfers out the pending fee including the newly accrued fee and sets pluginFeePending0/1 to zero afterwards.

Furthermore, it is not ensured anymore that overrideFee is larger/equal pluginFee. This necessity became redundant due to the change within the SwapCalculation contract.

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_01	Governance: Plugin owner can prevent several actions
Severity	**Governance**
Description	Currently, several actions are dependent on correct callbacks towards the plugin. If the plugin address can be changed or is a proxy, this could then result in a DoS of several functionalities, below is just one example for illustration purposes: The updateFeeAmounts function is invoked on the following occasions: swap/swapWithPaymentInAdvance/flash This has the background that only during these function a fee applies and therefore this is the only scenario where the if-clause is triggered: *if (communityFee0 > 0 || communityFee1 > 0 || pluginFee0 > 0 || pluginFee1 > 0) {* ... *updateFeeAmounts* ... *}* Whenever there is now a pluginFee being transferred to the plugin, the handlePluginFee function within the plugin is triggered. This is a vulnerable point as it is possible that this function results in a revert which then effectively results in a DoS of all aforementioned functions. Another example could be a simple blacklisting scenario for token0 or token1 which would then result in a revert of the transfer.
Recommendations	Consider implementing a strong governance body. For the blacklist scenario, it might be worth a consideration to adjust the transfer and

	wrap it into a try/catch call or switch to a pull based model. However, since blacklist scenarios were no issues in the past for the communityVault (which follows the exact same scheme), we are of the opinion that this is likely fine in the future as well.
Comments / Resolution	Acknowledged.

Issue_02	Malicious user can temporarily and permanently DoS pluginFee transferrals
Severity	**High**
Description	To understand how this exploit works, we need first to reiterate the logic of how and when fees are actually being transferred out. Fees are being transferred out whenever the _changeReserves function is called and two points are satisfied: **a) The actual fee (not pending) is non-zero:** *if (communityFee0 | communityFee1 != 0)* **b) The interval frequency has been reached:** *if (_blockTimestamp() - lastTimestamp >= Constants.FEE_TRANSFER_FREQUENCY* There is also a third point which is whenever the value of the pending fee is above uint104. However, that is a special edge-case and will not happen for most tokens. Now after we understand that a) and b) must be satisfied in order to transfer the fee out, we can successfully craft an exploit which always updates the interval while only transferring the communityFee out. This is the scenario whenever the flash function is invoked because this function will call _changeReserves with the following parameters: *_changeReserves(int256(communityFee0), int256(communityFee1), communityFee0, communityFee1, 0, 0);* As one can see, pluginFee0 and pluginFee1 are zero, therefore, even if there is a pending plugin fee, this fee will never be transferred out.

	A malicious user can now inspect the interval threshold, e.g. every 8 hours and then call the flash function and donate 1 wei (or any small amount) to trigger the transfer of the community fee and reset the interval. If that is done repetitively whenever the interval threshold is met (in the first block e.g.), this will then permanently prevent the plugin fee from being transferred out because the interval threshold is always only met with a valid community fee and on all other _changeReserve calls, the interval threshold will be never met. This then has the impact that pluginFeePending0/1 is ever increasing but will never be transferred out (unless it becomes larger than uint104, which is unlikely). The plugin fee remains therefore **permanently stuck** in the contract.
Recommendations	Consider implementing a function which allows manually transferring out the plugin fee. It must be ensured that this correctly decreases the reserves in the exact same manner as if the updateFeeAmounts function is called.
Comments / Resolution	Resolved, the logic within the ReserveManager contract was refactored and an _accrueAndTransferFees function was crafted. This function is invoked whenever the _changeReserves function is invoked and follows a similar logic as the updateFeeAmounts flow but it exposes additional flexibility which counters this exploit. Below we will describe all different callpaths: Call Path 1: Transfer fees and update storage Conditions: -> fee0 or fee1 is non-zero -> frequency is reached or uint104 is exceeded Actions:

	-> Load feePending0 and feePending1 from storage using assembly -> Add fee0 to feePending0 and fee1 to feePending1 -> Load recipient from storage using assembly -> Call _transferFees(feePending0, feePending1, recipient) to transfer the fees -> Update the storage slot specified by feePendingSlot with zeros using assembly -> Return (0, 0, feeSent0, feeSent1) -> update lastFeeTransferTimestamp with the current block.timestamp —————————————————————————— Call Path 2: Update pending fees and return without transferring Conditions: -> fee0 or fee1 is non-zero -> frequency is not reached and uint104 is not exceeded Actions: -> Load feePending0 and feePending1 from storage using assembly -> Add fee0 to feePending0 and fee1 to feePending1 -> Return (uint104(feePending0), uint104(feePending1), 0, 0) -> Update storage to reflect updated fees —————————————————————————— Call Path 3: Transfer pending fees and update storage

	Conditions: -> fee0 and fee1 are both zero -> frequency has been reached -> feePending0 or feePending1 is non-zero Actions: -> Load feePending0 and feePending1 from storage using assembly -> Load recipient from storage using assembly -> Call _transferFees(feePending0, feePending1, recipient) to transfer the pending fees -> Update the storage slot specified by feePendingSlot with zeros using assembly -> Return (0, 0, feeSent0, feeSent1) —--- Call Path 4: Return without transferring or updating pending fees Conditions: -> fee0 and fee1 are both zero -> Either frequency not passed or both pending fees are zero. Actions: -> Return (0, 0, 0, 0) —--- The implementation of the 3rd condition fixes the mentioned exploit, as now, even if pluginFee0/1 is zero, the pending fee is transferred out once the frequency has been reached.

Issue_03	Uint104 implementation for pluginFeePending can result in loss of funds and DoS of the burn call
Severity	**High**
Description	First of all we need to understand that any pending fee, such as communityFeePending0/1 and pluginFeePending0/1 can be only from type uint104 at max. This is explicitly handled in the definition of these variables: *uint104 internal communityFeePending0;* *uint104 internal communityFeePending1;* *uint104 internal pluginFeePending0;* *uint104 internal pluginFeePending1;* Now we need to understand that it is theoretically still possible for these "pending fees" to become larger than uint104. The Algebra Development Team has implemented special logic to incorporate this. First of all, during the updateFeeAmounts function, these pending fees are temporarily cached as uint256 instead of uint104 in an effort to be able to "add" the new incoming fee to the overall pending fee. This was done as follows: *uint256 feePending0;* *uint256 feePending1;* *assembly {* *// Load the storage slot specified by the slot argument* *let sl := sload(slot)* *// Extract the uint104 value* *feePending0 := and(sl, 0xFFFFFFFFFFFFFFFFFFFFFFFFFF)* *// Shift right by 104 bits and extract the uint104 value* *feePending1 := and(shr(104, sl),*

```
0xFFFFFFFFFFFFFFFFFFFFFFFFFF)
        }

        feePending0 += fee0;
        feePending1 += fee1;
```

This allows for the pending fees to become temporarily larger than uint104, which can naturally occur during standard business operations. In an effort to then properly handle these pending fees, they are transferred out in the scenario where it exceeds uint104. The idea behind this is to not accidentally cache a value which is larger than uint104 to the pending fee in storage:

```
if (_blockTimestamp() - lastTimestamp >=
Constants.FEE_TRANSFER_FREQUENCY || feePending0 >
type(uint104).max || feePending1 > type(uint104).max)
```

This solution in itself is perfect and covers this exact issue.

However, during the *burn* function, this edge-case is unfortunately not considered.

There are two problems with this implementation:

a) Unsafe casting to uint104 from deltaFeePluginPending0/1:

```
pluginFeePending0 += uint104(deltaPluginFeePending0);
```

This can result in a loss of funds in case of silent overflow

b) Aggregation of deltaPluginFeePending0/1 to pluginFeePending0/1. This is the same operation as highlighted above but not with the extra step of caching it as uint256.

This will simply result in a revert of the burn call.

	Illustrated example for a): - Charles wants to burn liquidity and receives 1e36 of token0 - Plugin fee is 0.1% - 1e36 * 0.001 = 1e34 - 1e34 is casted as uint104 -> silent overflow
Recommendations	There are two possible solutions: a) Simply setting pluginFeePending0/1 to uint256 instead of uint104 and removing the unsafe cast. However, This of course includes a careful check and adjustment of the assembly usage and the removal of the unnecessary timestamp sstore operation. b) Invoke the _changeReserves function with the actual fee and all other parameters being zero. This would then ensure that this problem does not happen because in the scenario where the value becomes larger than uint104, it will simply be transferred out immediately and reserves will be adjusted accordingly.
Comments / Resolution	Resolved, the _changeReserves function is now invoked with the actual fee. This will ensure that the aggregated pending fee is transferred out in case it becomes larger than uint104.max.

Issue_04	Advanced griefing attack allows users to lock plugin fee by abusing uint104.max scenario in specific pairs
Severity	**Low**
Description	The _accrueAndTransfer fees function has several different call-paths based on conditions. For this issue it is important to understand the first call-path which has the following condition *-> fee0 OR fee1 is non-zero* *-> frequency is reached OR uint104.max is exceeded from either feePending0 or feePending1* *In that scenario:* *-> fees are transferred out* *-> pending fee storage is reset* *-> lastFeeTransferTimestamp is set to the current block.timestamp* This condition exposes a sophisticated exploit method which is based on the uint104.max edge-case and will only work for pairs that have one of both tokens with a very large supply. While it is quite uncommon to have a large supply, it is not something we can exclude. The Algebra DEX is compatible with tokens that do have a totalSupply() of up to uint128. This is already acknowledged and specifically handled throughout the architecture. uint128.max corresponds to 3.40282e38 uint104.max corresponds to 2.02824e31

	This means that uint104.max is approximately 0.000595965% of uint128.max. This value is later important. Due to the fact that the lastFeeTransferTimestamp is set to the current block.timestamp in the scenario where one of both tokens exceeds uint104.max, we can craft a token donation exploit which allows to grief fee transfers in a similar fashion as the "*Malicious user can temporarily and permanently DoS pluginFee transferrals*" issue. Illustrated: a) A TOKEN-USDC pair is created from a new famous project, the TOKEN supply is within the uint128.max range. Therefore this pair is supported by Algebra. b) A malicious user holds 0.05% of the TOKEN supply and is the founder of an Algebra competitor c) The TOKEN-USDC pair is traded on the Algebra DEX and constantly accrues plugin as well as community fees. d) Always shortly before the frequency duration of 8 hours has passed, the malicious user invokes the flash function and donates uint104.max +1 of TOKEN. This will now always trigger _changeReserves with communityFees0 = uint104.max +1 e) This will then in turn trigger _accrueAndTransferFees with the first callpath. Notably, the frequency has not yet been reached. Due to the fact that uint104.max is exceeded, it will still transfer out the community fee and set lastFeeTransferTimestamp to block.timestamp. f) Since the pending plugin fee has not exceeded uint104.max and the frequency has not yet been reached, the plugin fee is not being transferred out (it will now following callpath #2). Moreover, since the frequency is reset (lastFeeTransferTimestamp = block.timestamp), the

	plugin fee will not be transferred out for another 8 hours. g) One block before the next frequency period would be reached, the malicious user repeats this step, transferring the community fee out and updating lastFeeTransferTimestamp. This will effectively withhold the plugin fee until it eventually exceeds uint104.max, which is likely possible at some point due to the high supply. However, this of course depends on the trading volume and the total supply of the token, which can also be lower than uint128.max which then in turn increases the capital needed by the exploiter. The burn function logic can be abused in a similar manner to block the community fee.
Recommendations	A solution would be to implement two different timestamps. However, since that would be another intrusive change to the contract logic, it may also be an option to acknowledge this issue because it can only happen in very rare occasions.
Comments / Resolution	Acknowledged.

Issue_05	Pending fee will never be distributed if fees are suddenly set to zero
Severity	**Low**
Description	Both the pending community fee as well as the pending plugin fee are only transferred out during the updateFeeAmounts function whenever the interval threshold has been met. There is a special edge-case in the scenario where both or one of both fees is being set to zero, in that scenario it would not trigger the updateFeeAmounts function because that specific transaction has not captured any fee (because the fee is zero): *if (communityFee0 | communityFee1 != 0) {* *..* *updateFeeAmounts* *...* *}* This then further means that any previously pending fee will be never transferred out in such a scenario, until the fee is then being set back to non-zero.
Recommendations	Consider transferring any pending fee out in the scenario where the pluginFee or communityFee is being set to zero. Since it is out of scope for this audit, we have not checked all scenarios where the fee can be set to zero. Optionally, this issue can be acknowledged.
Comments / Resolution	Acknowledged.

Issue_06	lastTimestamp will be stored in unused part of bits
Severity	**Low**
Description	The updateFeeAmounts function is called two times within the _changeReserves function. The first time it is called with slot = communityFeePending0.slot and the second time it is called with pluginFeePending0.slot. These slots are then used to cache the pending fee and are also manipulated in the storage afterwards: *assembly {* *sstore(slot, or(or(feePending0, shl(104, feePending1)), shl(208, lastTimestamp)))* *}* A problem arises in the latter scenario where pluginFeePending0.slot is used because this will then set the unused part of the bits in this storage slot to lastTimestamp. Fortunately due to the fact that these bits are unused there is no harm during this operation, however, this is something which should be prevented.
Recommendations	Consider simply removing the storage of the lastTimestamp in these unused bits.
Comments / Resolution	Resolved.

Issue_07	Redundant lastFeeTransferTimestamp setting within communityFeePending slot
Severity	**Low**
Description	In the previous issue: “lastTimestamp will be set to unused part of bits”, we have already elaborated the erroneous setting of the lastFeeTransferTimestamp during the updateFeeAmounts function call with pluginFeePending0.slot. While this was already incorrect, during the updateFeeAmounts function call with communityFeePending0.slot, the lastFeeTransferTimestamp is updated but incorrect: *assembly {* *sstore(slot, or(or(feePending0, shl(104, feePending1)), shl(208, lastTimestamp)))* *}* Within this assembly snippet, lastFeeTransferTimestamp is just set to lastTimestamp. However, lastTimestamp has previously been cached as lastFeeTransferTimestamp: *uint32 lastTimestamp = lastFeeTransferTimestamp;* making this whole operation redundant because these are essentially the same values.
Recommendations	Consider simply removing the setting of the following bits because lastFeeTransferTimestamp will be anyways correctly set at the end of the _changeReserves function:

	if (currentTimestamp - lastTimestamp >= Constants.FEE_TRANSFER_FREQUENCY) lastFeeTransferTimestamp = currentTimestamp;
Comments / Resolution	Resolved.

Issue_08	Violation of checks-effects-interactions pattern
Severity	**Informational**
Description	Within the updateFeeAmounts function, tokens are eventually being transferred from the AlgebraPool to the communityVault and to the plugin: *if (feePending0 > 0) {* *_transfer(token0, feesRecipient, feePending0);* *deltaR0 = deltaR0 - feePending0.toInt256();* *feeSent0 = feePending0;* *feePending0 = 0;* *}* This will result in an un updated state at the time of the transfer because the feePending0 value and its corresponding storage slot is not yet updated. This does not actually expose a real risk because as from our knowledge and analysis, there is no way to reenter because Algebra has valid reentrancy-checks throughout the protocol. However, this will expose a known read-only vulnerability. Moreover, we need to highlight that this is no full protocol audit and thus we have not analyzed the full codebase for any potential reentrancy doors.

Recommendations	While we ideally would like to see this issue fixed, we understand that a similar issue was already part of the previous iteration with regards to the community fee. Therefore, we lean towards recommending to acknowledge this issue with an additional emphasis on cross-checking and additional testing.
Comments / Resolution	Acknowledged.