PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Algebra

12 February 2024

paladinsec.co

info@paladinsec.co

Table of Contents

Disclaimer

Paladin Blockchain Security ("Paladin") has conducted an independent audit to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the codes that were provided for the scope of this audit. This audit report does not constitute agreement, acceptance or advocation for the Project that was audited, and users relying on this audit report should not consider this as having any merit for financial advice in any shape, form or nature. The contracts audited do not account for any economic developments that may be pursued by the Project in question, and that the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are completely free of exploits, bugs, vulnerabilities or deprecation of technologies. Further, this audit report shall not be disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Paladin.

All information provided in this report does not constitute financial or investment advice, nor should it be used to signal that any persons reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report. Information is provided 'as is', and Paladin is under no covenant to the completeness, accuracy or solidity of the contracts audited. In no event will Paladin or its partners, employees, agents or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this audit report.

Cryptocurrencies and any technologies by extension directly or indirectly related to cryptocurrencies are highly volatile and speculative by nature. All reasonable due diligence and safeguards may yet be insufficient, and users should exercise considerable caution when participating in any shape or form in this nascent industry.

The audit report has made all reasonable attempts to provide clear and articulate recommendations to the Project team with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts provided. It is the sole responsibility of the Project team to sufficiently test and perform checks, ensuring that the contracts are functioning as intended, specifically that the functions therein contained within said contracts have the desired intended effects, functionalities and outcomes of the Project team.

Paladin retains the right to re-use any and all knowledge and expertise gained during the audit process, including, but not limited to, vulnerabilities, bugs, or new attack vectors. Paladin is therefore allowed and expected to use this knowledge in subsequent audits and to inform any third party, who may or may not be our past or current clients, whose projects have similar vulnerabilities. Paladin is furthermore allowed to claim bug bounties from third-parties while doing so.

1 Overview

This report has been prepared for Algebra Integral on the Ethereum network. Paladin provides a user-centred examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective.

The audit for the Algebra Integral project incorporates a time-boxed audit for the following smart contract audits, with a focus on the core module. Throughout the report the following declarations can happen:

- `token0` -> `tokenX`
- `token1` -> `tokenY`

1.1 Summary

Project Name	Algebra Integral
URL	https://algebra.finance/
Platform	Ethereum
Language	Solidity
Preliminary	https://github.com/cryptoalgebra/Algebra/tree/2be46e835c444072f968ee88c307699f1a755d86
Resolution	https://github.com/cryptoalgebra/Algebra/commit/eea3a9bda1b623dfc3b6f606a8ef08dec76f18b2

1.2 Contracts Assessed

The team will deploy the contracts at a later date and has requested for Paladin to publish the report first. Users should check that the code of the deployed contracts match the audited contracts.

Contract	Address	Live Match
AlgebraPoolBase	UNDEPLOYED	N/A
Positions	UNDEPLOYED	N/A
ReentrancyGuard	UNDEPLOYED	N/A
ReservesManager	UNDEPLOYED	N/A
SwapCalculation	UNDEPLOYED	N/A
TickStructure	UNDEPLOYED	N/A
AlgebraPool	UNDEPLOYED	N/A
AlgebraCommunityVault	UNDEPLOYED	N/A
AlgebraFactory	UNDEPLOYED	N/A
AlgebraPoolDeployer	UNDEPLOYED	N/A
LiquidityMath	UNDEPLOYED	N/A
Plugins	UNDEPLOYED	N/A
PriceMovementMath	UNDEPLOYED	N/A
SafeTransfer	UNDEPLOYED	N/A
TickManagement	UNDEPLOYED	N/A
TickTree	UNDEPLOYED	N/A
TokenDeltaMath	UNDEPLOYED	N/A
IncentiveKey	UNDEPLOYED	N/A

VirtualTickStructure	UNDEPLOYED	N/A
AlgebraEternalFarming	UNDEPLOYED	N/A
EternalVirtualPool	UNDEPLOYED	N/A
FarmingCenter	UNDEPLOYED	N/A
ERC721Permit	UNDEPLOYED	N/A
LiquidityManagement	UNDEPLOYED	N/A
Multicall	UNDEPLOYED	N/A
PeripheryImmutableState	UNDEPLOYED	N/A
PeripheryPayments	UNDEPLOYED	N/A
PeripheryPaymentsWithFee	UNDEPLOYED	N/A
PoolInitializer	UNDEPLOYED	N/A
SelfPermit	UNDEPLOYED	N/A
NonfungiblePositionManager	UNDEPLOYED	N/A
CallbackValidation	UNDEPLOYED	N/A
LiquidityAmounts	UNDEPLOYED	N/A
PoolAddress	UNDEPLOYED	N/A
PoolInteraction	UNDEPLOYED	N/A
PositionKey	UNDEPLOYED	N/A
AlgebraFeeConfiguration	UNDEPLOYED	N/A
AlgebraBasePluginV1	UNDEPLOYED	N/A
BasePluginV1Factory	UNDEPLOYED	N/A
AlgebraOracleV1TWAP	UNDEPLOYED	N/A
OracleLibrary	UNDEPLOYED	N/A

AdaptiveFee	UNDEPLOYED	N/A
VolatilityOracle	UNDEPLOYED	N/A

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change)
Governance	2	-	-	2
High	1	-	-	1
Medium	6	1	1	4
Low	11	4	-	7
Informational	6	3	-	3
Total	**26**	**8**	**1**	**17**

Classification of Issues

Severity	Description
Governance	Issues under this category are where the governance or owners of the protocol have certain privileges that users need to be aware of, some of which can result in the loss of user funds if the governance's private keys are lost or if they turn malicious, for example.
High	Exploits, vulnerabilities or errors that will certainly or probabilistically lead towards loss of funds, control, or impairment of the contract and its functions. Issues under this classification are recommended to be fixed with utmost urgency.
Medium	Bugs or issues with that may be subject to exploit, though their impact is somewhat limited. Issues under this classification are recommended to be fixed as soon as possible.
Low	Effects are minimal in isolation and do not pose a significant danger to the project or its users. Issues under this classification are recommended to be fixed nonetheless.
Informational	Consistency, syntax or style best practices. Generally pose a negligible level of risk, if any.

1.3.1 AlgebraPoolBase

No issues found.

1.3.2 Positions

No issues found.

1.3.3 ReentrancyGuard

No issues found.

1.3.4 ReservesManager

No issues found.

1.3.5 SwapCalculation

No issues found.

1.3.6 TickStructure

No issues found.

1.3.7 AlgebraPool

ID	Severity	Summary	Status
01	GOV	Governance: Introduction of hooks	ACKNOWLEDGED
02	HIGH	Flashloan fees and reflection tokens are distributed unfairly	ACKNOWLEDGED
03	MEDIUM	`flash` function induces faulty state during `_flashCallback`	ACKNOWLEDGED
04	LOW	Lack of refund for sent tokens if amounts are zero	ACKNOWLEDGED
05	LOW	`communityFee` should never be zero	ACKNOWLEDGED
06	LOW	Transfer out happens before reserve change	ACKNOWLEDGED
07	LOW	`swapWithPaymentInAdvance` does not update reserves during beginning	ACKNOWLEDGED

1.3.8 AlgebraCommunityVault

No issues found.

1.3.9 AlgebraFactory

No issues found.

1.3.10 AlgebraPoolDeployer

No issues found.

1.3.11 LiquidityMath

No issues found.

1.3.12 Plugins

No issues found.

1.3.13 PriceMovementMath

No issues found.

1.3.14 SafeTransfer

No issues found.

1.3.15 TickManagement

No issues found.

1.3.16 TickTree

ID	Severity	Summary	Status
08	INFO	Gas optimizations	✓ RESOLVED

1.3.17 TokenDeltaMath

No issues found.

1.3.18 IncentiveKey

No issues found.

1.3.19 VirtualTickStructure

No issues found.

1.3.20 AlgebraEternalFarming

ID	Severity	Summary	Status
09	GOV	Governance: Change of various parameters can result in DoS and loss of rewards	ACKNOWLEDGED
10	MEDIUM	`emergencyWithdraw` can result in manipulated farming	PARTIAL
11	MEDIUM	`setRates` lacks proper input validation which allows `INCENTIVE_MAKER` to steal all rewards	ACKNOWLEDGED
12	MEDIUM	Loss of rewards upon `exitFarming` during `emergencyWithdraw`	ACKNOWLEDGED
13	LOW	`POOLS_ADMINISTRATOR_ROLE` can frontrun `createEternalFarming`	✓ RESOLVED
14	LOW	Malicious user might abuse disconnected state	ACKNOWLEDGED
15	LOW	Unnecessary tick determination during `_updatePosition`	ACKNOWLEDGED
16	LOW	Lack of validation for key throughout the codebase	✓ RESOLVED

1.3.21 EternalVirtualPool

ID	Severity	Summary	Status
17	LOW	Risk of overflow for reward calculation	ACKNOWLEDGED
18	INFO	Typographical issues	✓ RESOLVED
19	INFO	Gas optimizations	ACKNOWLEDGED

1.3.22 FarmingCenter

ID	Severity	Summary	Status
20	LOW	Invalid return value for `claimReward`	✓ RESOLVED

1.3.23 ERC721Permit

No issues found.

1.3.24 LiquidityManagement

No issues found.

1.3.25 Multicall

ID	Severity	Summary	Status
21	LOW	Errors during a revert might not be handled correctly	✓ RESOLVED

1.3.26 PeripheryImmutableState

No issues found.

1.3.27 PeripheryPayments

ID	Severity	Summary	Status
22	MEDIUM	The pay function can result in full loss of `msg.value` in certain scenarios	✓ RESOLVED

1.3.28 PeripheryPaymentsWithFee

ID	Severity	Summary	Status
23	INFO	Functions do not serve any use-case	ACKNOWLEDGED

1.3.29 PoolInitializer

No issues found.

1.3.30 SelfPermit

No issues found.

1.3.31 NonfungiblePositionManager

ID	Severity	Summary	Status
24	MEDIUM	Advanced exploit including frontrunning allows for stealing native tokens from users upon minting / increase of liquidity if multicall is not used	ACKNOWLEDGED

1.3.32 CallbackValidation

No issues found.

1.3.33 LiquidityAmounts

No issues found.

1.3.34 PoolAddress

No issues found.

1.3.35 PoolInteraction

ID	Severity	Summary	Status
25	INFO	Unused import	✓ RESOLVED

1.3.36 PositionKey

No issues found.

1.3.37 AlgebraFeeConfiguration

No issues found.

1.3.38 AlgebraBasePluginV1

ID	Severity	Summary	Status
26	INFO	Gas optimizations	ACKNOWLEDGED

1.3.39 BasePluginV1Factory

No issues found.

1.3.40 AlgebraOracleV1TWAP

No issues found.

1.3.41 OracleLibrary

No issues found.

1.3.42 AdaptiveFee

No issues found.

1.3.42 VolatilityOracle

No issues found.

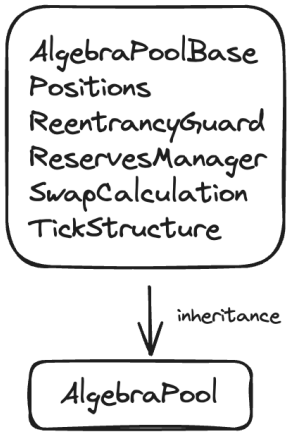
2 Findings

2.1 Core/Base

The contracts within the Base section form the Pool Base together, which includes several functionalities including but not limited to:

- taking care of the overall pool/pair state
- handling the position logic
- providing a reentrancy guard
- handling the reserves
- calculating swaps

This Base is inherited by the AlgebraPool contract, and together they form the core architecture for a liquidity pool.



2.2 Core/Base/AlgebraPoolBase

`AlgebraPoolBase` acts as the base contract for a pool. It keeps track of the global state and several other fundamental state variables. Additionally, it includes internal callback functions.

2.2.1 Issues & Recommendations

No issues found.

2.3 Core/Base/Positions

`Positions` is an abstract contract which is part of the Base for Core. It handles the logic for determining and updating positions. Each position has a unique key which is derived from the `owner`, `bottomTick` and `topTick`.

Whenever a position is created or removed, the corresponding bottom and top tick states (`liquidityDelta`, `feeGrowth` and `totalLiquidity`) are updated and the `innerFeeGrowth` for the position is calculated, which is automatically used to update the positions `innerFeeGrowth` and eventually adds claimable fees.

Moreover, if a tick is newly initialized, which means that the tick is used for the first time for a position, it is added to the `TickTree`. Similarly, it is removed when there is no `totalLiquidity` left.

2.3.1 Issues & Recommendations

No issues found.

2.4 Core/Base/ReentrancyGuard

`ReentrancyGuard` is a simple lock modifier which prevents re-entrance into certain functions.

2.4.1 Issues & Recommendations

No issues found.

2.5 Core/Base/ReservesManager

`ReservesManager` is an abstract contract that manages the accounting for the reserves of the pool. In short, there are two different accounting use-cases:

a) Updating the reserves due to external balance changes, such as direct token transfers to the pool.

b) Updating the reserves due standard business logic operations, such as minting, burning, fee collection, swapping.

It is important to mention that the maximum value for the reserves can be `uint128`. Everything which exceeds this value will be sent to the community vault.

2.5.1 Issues & Recommendations

No issues found.

2.6 Core/Base/SwapCalculation

`SwapCalculation` is the heart of the swap mechanism and handles the accounting for a swap execution. Additionally, it also updates the active liquidity and corresponding ticks which are eventually crossed. At the end of the function, the previous and next ticks are updated to correctly reflect the pool's state change and prime it for the next swap.

The main math logic lies within the `PriceMovementMath` and is described within this section.

Generally speaking, a swap can be executed in four different scenarios, namely:

a) 0 -> 1 ; explicit input

b) 1 -> 0; explicit input

c) 0 -> 1; explicit output

d) 1 -> 0; explicit output

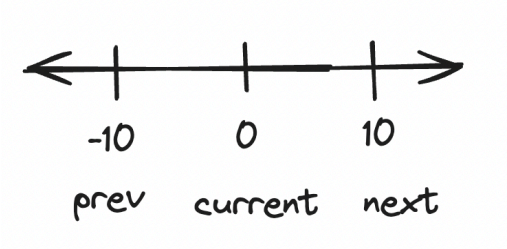
2.6.1 Issues & Recommendations

No issues found.

2.7 Core/Base/TickStructure

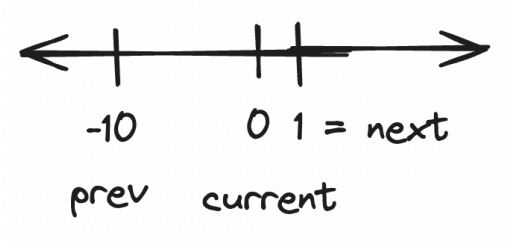
`TickStructure` is responsible for correctly determining the previous and next ticks for a newly added or removed tick. It then returns and updates any potential changes for `prevInitializedTick` and `nextInitializedTick` which might have occurred due to the removal or addition of a new tick.

As an example, if an added tick is in between `prevTickGlobal` and `nextTIckGlobal`, one of these values must be replaced by the newly added tick to keep the structure correct. It is illustrated as follows:



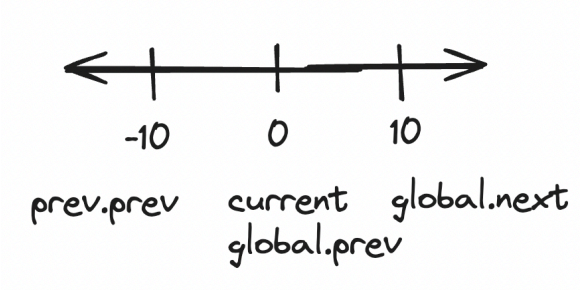
Next swaps would be as follows:
X -> Y = [0;-10]
Y -> X = [10; 0]

Now, tick 1 is initialized above the current tick:



Next swaps would now be as follows:
X -> Y = [0; -10]
Y -> X = [0; 1]

A tick can be initialized below or even equal to the current tick, which would then adjust `prevTick`:



Next swaps would be as follows:
X -> Y = [0;0]
Y -> X = [0;10]

For removal of a tick, only one of the global variables is updated if the removed tick occupied one of these. In this case, it is ensured that the correct previous or next tick is used.

2.7.1 Issues & Recommendations

No issues found.

2.8 Core/AlgebraPool

`AlgebraPool` is a pivotal component, providing a robust platform for investors to interact with a liquidity pool. This contract includes a range of essential functionalities that are fundamental to the fluid operation of the liquidity pool, including minting, burning, collecting, swapping, advanced payment swapping, and facilitating flash loans.

The `mint` and `burn` functions allow investors to respectively add to and withdraw from the liquidity pool, enabling a dynamic adjustment of their participation in the pool.

The `collect` function is designed to retrieve accumulated fees or rewards, a critical feature for liquidity providers to realize their earnings.

Swapping, a core functionality, enables users to exchange one asset for another within the pool, providing the essential mechanism for traders and arbitrageurs. `swapWithPaymentInAdvance` extends this capability, offering more complex trade structures, which expects the input token to be transferred before the swap calculation logic.

Lastly, the flash loan feature offers an advanced tool for users requiring temporary liquidity without relinquishing asset ownership. A notable aspect of the `AlgebraPool` contract is its integration with a plugin system. This system introduces a layer of customization and extensibility to the pool's operations. The plugin can usually be invoked at two key points: before and after the execution of the primary functions (`mint`, `burn`, `swap`, `flash` etc.). This design allows additional logic to be executed, which can range from custom fee structures to additional security checks, or other business rules as defined by the pool's governance.

The contract is initialized with a default `pluginConfig`, which includes flags such as `AFTER_INIT`, `BEFORE_SWAP`, and `DYNAMIC_FEE`.

In essence, `AlgebraPool` is designed as a comprehensive gateway for liquidity pool interactions, backed by a flexible plugin system. This structure not only caters to the fundamental needs of liquidity provision and asset exchange but also opens avenues for sophisticated financial operations and customizable business logic, including potential compliance necessities such as KYC checks.

It is important to note that this audit was only conducted with reference to the `AlgebraBasePluginV1`.

2.8.1 Issues & Recommendations

Issue #01	Governance: Introduction of hooks
Severity	GOVERNANCE
Description	All interactions excluded from the `collect` function have the potential to invoke hooks which call the plugin. This methodology opens up the allows for various governance privileges. Theoretically it would be possible for a privileged address to change the plugin and the plugin config and execute any arbitrary logic on the hook, which would not result in a loss of funds but can result in a DoS for all interactions. Additionally, the risk of reentrancy is also given.
Recommendation	Consider incorporating a Gnosis multi-signature contract as privileged address and ensuring that the Gnosis participants are trusted entities.
Resolution	ACKNOWLEDGED The client stated the following: "Strict access control is required. To improve security, it is possible to use multisign and special contracts with more granular access rights."

Issue #02	Flashloan fees and reflection tokens are distributed unfairly
Severity	HIGH SEVERITY
Description	Any excess of token from the reserves is caught by the `updateReserves` function and distributed to only active liquidity positions, i.e, those that contain the active tick. However, the flashloan might use more tokens than the one containing the active tick which is an unfair distribution as other positions will not receive any fees. Furthermore, as liquidity is used, a position with twice as much tokens but twice the number of ticks as well will receive the same amount of token as the narrowed position. This gets worse for pairs that contain reflected tokens, such as `stEth`. The yield will only be distributed to active liquidity positions and not according to the actual balances of the reflection token as it will only be based on the liquidity of the positions. Additionally, if a flashloan or the reflection distribution was to be frontrun, the malicious user could move to add liquidity where he is the main/only liquidity provider to steal the majority of, if not all, flashloan fees or the tokens from the reflection.
Recommendation	Consider fixing the distribution issue if this poses an issue. For reflection tokens, the solution might simply be to not support them and expect users to use a wrapped version of the staked token that would take care of the distribution. In any case, this should be communicated clearly in the Algebra documentation and in comments.
Resolution	ACKNOWLEDGED The client stated the following: "Works as intended, this is not a loss of user funds, but a feature of the pool's operation. We are not convinced that the described issue meets the definition of high severity from the description of the audit methodology. It is necessary to indicate in more detail in the documentation the nature of the distribution of such 'excess' pool profits."

Issue #03	`flash` function induces faulty state during `_flashCallback`
Severity	MEDIUM SEVERITY
Description	Within the `flash` function, tokens are sent out to the caller to accommodate the requested flashloan. However, during the callback function, the vault is within an un-updated state as the transferred tokens are still considered within the reserves, which of course does not reflect the actual state. While we could not identify an exploit, incidents in the past have shown that it is of utmost importance to reflect an updated state at all times, specifically since this transaction can result in a very large deviation from the real state.
Recommendation	Consider updating the reserves before the callback function in an effort to reflect the real state of the pool.
Resolution	ACKNOWLEDGED The client stated the following: "These tokens are still considered part of the pool reserve as they will be returned. External contracts should in any case check the reentrancy lock when reading the pool state to avoid any manipulation (read-only reentrancy)."

Issue #04	Lack of refund for sent tokens if amounts are zero
Severity	LOW SEVERITY
Description	The following assumption is made if both or one of both amounts is zero: `receivedAmount0 = amount0 == 0 ? 0 : _balanceToken0() - receivedAmount0;` `receivedAmount1 = amount1 == 0 ? 0 : _balanceToken1() - receivedAmount1;` If a user accidentally or intentionally still sends one of both tokens within the callback function, these tokens will not be refunded, since a refund is only done if the amount assignment is > 0: `if (amount0 > 0) {` `  if (receivedAmount0 > amount0) _transfer(token0, leftoversRecipient, receivedAmount0 - amount0);` `  else assert(receivedAmount0 == amount0); // must always be true` `}` `if (amount1 > 0) {` `  if (receivedAmount1 > amount1) _transfer(token1, leftoversRecipient, receivedAmount1 - amount1);` `  else assert(receivedAmount1 == amount1); // must always be true` `}` In such a scenario, the tokens will not be refunded but rather included in the reserves within the next update. This issue is only rated as low severity since this error is completely attributed towards the user.
Recommendation	Consider if it makes sense to refactor this logic to also account for such scenarios. If yes, consider removing the ternary operator for the first highlighted spot and refund in any scenario where `receivedAmount > amount`.
Resolution	ACKNOWLEDGED The client stated that the current approach provides good gas savings for the vast majority of cases.

Issue #05	**communityFee should never be zero**
Severity	LOW SEVERITY
Description	Since privileged addresses can change the `communityFee` including setting it to zero, this could open up the scenario for users to take out flash loans without paying an actual fee. In a scenario where a user is the only liquidity provider for the current actual range or holds the majority liquidity for that range, this user would be the major/sole recipient of the flashloan fee. This comes with several risks, not only to the protocol but also to the overall DeFi landscape. Of course, even if the `communityFee` is > 0, a user can become the largest stakeholder in the community vault. However, this would at least further limit the practicality of such a scenario.
Recommendation	Consider ensuring that the `communityFee` is always > 0.
Resolution	ACKNOWLEDGED The client stated the following: "Works as intended. Sometimes this fee should be disabled, sometimes enabled. The described potential manipulation scenarios do not justify fixing the `communityFee` value; this contradicts the logic of the protocol."

Issue #06	Transfer out happens before reserve change
Severity	LOW SEVERITY
Description	The `mint` function transfers excess tokens out and then only afterwards changes the reserves, reflecting the deposited tokens. This can result in a read-only reentrancy state upon receipt of ERC777 tokens. Since there is no harm with changing the reserves with the corresponding amounts for the liquidity before the transfer of the excess, this issue can be easily fixed. This issue is also present for the `collect` function.
Recommendation	Consider changing the reserves before the transfer of the excess tokens.
Resolution	ACKNOWLEDGED The client stated the following: "If done as proposed, there will still be a potential risk of incorrect data, since one token will be sent earlier than the second. Accordingly, at some point the pool balance will not match the reserve. External contracts should in any case check the reentrancy lock when reading the pool state to avoid any manipulation (read-only-reentrancy)."

Issue #07	**`swapWithPaymentInAdvance` does not update reserves during beginning**
Severity	LOW SEVERITY
Description	Throughout the whole codebase, reserves are always updated at the beginning of the function. However, this is not the case for the `swapWithPaymentInAdvance` function. Upon our inspection, there is no specific reason for not keeping consistent with the reserve update upon beginning of the function
Recommendation	Consider updating the reserves at the beginning of the function.
Resolution	ACKNOWLEDGED The client stated the following: "We prefer to update reserves (and distribute excessive tokens) after "beforeSwap" hook. This hook, in turn, needs to know the exact number of tokens being exchanged. Therefore, updating the reserves and calling the hook occurs after the callback to receive tokens."

2.9 Core/AlgebraCommunityVault

`AlgebraCommunityVault` is used to receive community fees from Algebra protocol's pair. `algebraFeeManager` and any address that has the role `COMMUNITY_FEE_WITHDRAWER_ROLE` can call the `withdraw` function sends an amount of token to the desired address and also sends a share of that amount to the `algebraFeeReceiver`.

The `AlgebraFeeManager` can propose a new Algebra fee that the administrators (any address that have the `COMMUNITY_FEE_VAULT_ADMINISTRATOR` role) can accept.

This contract is used to share the protocol fees with Algebra following the agreements between the two teams.

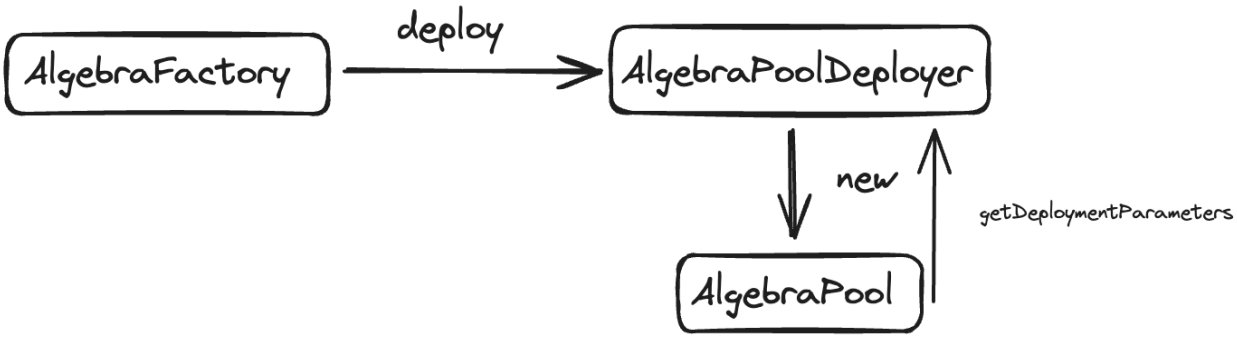
2.9.1 Privileged Functions

- `withdraw [algebraFeeManager or COMMUNITY_FEE_WITHDRAWER_ROLE]`
- `withdrawTokens [algebraFeeManager or COMMUNITY_FEE_WITHDRAWER_ROLE]`
- `acceptAlgebraFeeChangeProposal [COMMUNITY_FEE_VAULT_ADMINISTRATOR]`
- `changeCommunityFeeReceiver [COMMUNITY_FEE_VAULT_ADMINISTRATOR]`
- `transferAlgebraFeeManagerRole [algebraFeeManager]`
- `acceptAlgebraFeeManagerRole [pendingAlgebraFeeManager]`
- `proposeAlgebraFeeChange [algebraFeeManager]`
- `cancelAlgebraFeeChangeProposal [algebraFeeManager]`
- `changeAlgebraFeeReceiver [algebraFeeManager]`

2.9.2 Issues & Recommendations

No issues found.

2.10 AlgebraFactory



`AlgebraFactory` is used to create new Algebra pairs. It will check that the pair has not already been created and index it in the `poolByPair` mapping.

This contract is also used to define the default parameters for pairs, such as `defaultFee`, `defaultCommunityFee` and `defaultPluginFactory`.

2.10.1 Privileged Functions

- `setDefaultCommunityFee`
- `setDefaultFee`
- `setDefaultTickspacing`
- `setDefaultPluginFactory`
- `startRenounceOwnership`
- `stopRenounceOwnership`
- `renounceOwnership`

2.10.2 Issues & Recommendations

No issues found.

2.11 AlgebraPoolDeployer

`AlgebraPoolDeployer` is a helper contract used by by the pair contract to get its parameters during creation. The factory sets the parameter for the pair to fetch them.

2.11.1 Privileged Functions

- `deploy [factory]`

2.11.2 Issues & Recommendations

No issues found.

2.12 Core/Libraries/LiquidityMath

`LiquidityMath` is used to calculate the amounts for a given position/range given a specific liquidity and price. This logic is used within `mint` and during the update of a position during a mint/burn. In addition to the `amount` calculations, this library also determines the potential `globalLiquidityDelta` for the scenario where liquidity is added to / removed from the active tick . This library leverages the `TokenDeltaMath` library for these calculations.

2.12.1 Issues & Recommendations

No issues found.

2.13 Core/Libraries/Plugins

`Plugins` is a simple helper contract which checks the enabled flags within `pluginConfig`. The following flags are possible:

```
BEFORE_SWAP_FLAG = 1 (0000 0001)
AFTER_SWAP_FLAG = 1<<1 (0000 0010)
BEFORE_POSITION_MODIFY_FLAG = 1<<2 (0000 0100)
AFTER_POSITION_MODIFY_FLAG = 1<<3 (0000 1000)
BEFORE_FLASH_FLAG = 1<<4 (0001 0000)
AFTER_FLASH_FLAG = 1<<5 (0010 0000)
AFTER_INIT_FLAG = 1<<6 (0100 0000)
DYNAMIC_FEE_FLAG = 1<<7 (1000 0000)
```

In the brackets, the binary denomination of the different flags are represented. To create a valid `pluginConfig`, one can simply `OR` the different binary representations and will then receive a valid binary expression. Converted to hexadecimal, this will represent a valid `pluginConfig`.

To check if a flag exists within a `pluginConfig`, one can simply `AND` the `pluginConfig` and the flag denomination.

2.13.2 Issues & Recommendations

No issues found.

2.14 Core/Libraries/PriceMovementMath

`PriceMovementMath` is the fundamental contract for calculating swap steps. In simple terms, it handles the amount of tradeable tokens within the current active liquidity. There are two possible scenarios:

1. Swap crosses a range

The desired swap value cannot be covered by the current active range, which means that the active range cannot completely facilitate the incoming swap. In that scenario, the input amount will be simply the full amount the range can accommodate:

```
input = getInputTokenAmount(resultPrice, currentPrice, liquidity);
```

The output amount will be simply the corresponding amount of the output token within the range:

```
output = (zeroToOne ? getOutputTokenDelta01 : getOutputTokenDelta10)(resultPrice, currentPrice, liquidity)
```

It is important to note that for security reasons, these calculations always round against the favor of the user. The input amount will round up, and the output amount will round down.

Moreover, a fee is taken for this step, which is decreased from the input amount before the swap in the next range.

2. Swap does not cross a range

Unlike the first example, the current range in this scenario can completely facilitate the swap, which means nothing other than sufficient liquidity is available to cover the swap needs. In that scenario, `resultPrice` is calculated, which is simply the post-swap price:

```
resultPrice = getNewPriceAfterInput(currentPrice, liquidity, amountAvailableAfterFee, zeroToOne);
```

The price calculation is the most complex portion within the swap process, but in simple terms it can be explained as to which price the swap pushes the current tick. The mathematical explanation is as follows:

X -> Y

From the UniswapV3 whitepaper, we have the following mathematical equation:

$$\Delta x = \Delta \frac{1}{\sqrt{P}} \cdot L \qquad (6.16)$$

Deviating this formula:

$$\sqrt{x} = \left(\frac{1}{\sqrt{P_{target}}} - \frac{1}{\sqrt{P_{current}}} \right) * L$$

$$\frac{\sqrt{x}}{L} + \frac{1}{\sqrt{P_{current}}} = \frac{1}{\sqrt{P_{target}}}$$

$$\frac{\sqrt{x} * \sqrt{P_{current}} + L}{\sqrt{P_{current}} * L} = \frac{1}{\sqrt{P_{target}}}$$

$$\frac{\sqrt{P_{current}} * L}{\sqrt{x} * \sqrt{P_{current}} + L} = \sqrt{P_{target}}$$

Which is exactly how the new price is calculated for a X -> Y swap.

Y -> X

Given the UniswapV3 whitepaper, we have the following mathematical equation:

$$L = \frac{\Delta Y}{\Delta \sqrt{P}} \qquad (6.7)$$

Deviating this formula results in

$$\Delta \sqrt{P} = \frac{\Delta Y}{L}$$

Which is exactly the way the new price is calculated for a Y -> X swap, given the input amount liquidity and current price:

$$\sqrt{TargetPrice} = \sqrt{CurrentPrice} - \sqrt{\Delta Price}$$

After this price is determined, the same approach as before is used, where the input amount is calculated on the new determined range (using the new price) and liquidity. The swapper will then receive the output amount in this range for the corresponding liquidity and deposit the input amount, both rounded against the favor of the user.

Finally, it can be said that the swap calculation is completely based on liquidity. On a meta-level explanation, it is just the swapper taking out the current liquidity, moving the price to the target and adding liquidity again, all while the rounding is against the favor of the user.

2.14.1 Issues & Recommendations

No issues found.

2.15 Core/Libraries/SafeTransfer

`SafeTransfer` is derived from Solmate's `safeTransferLib` and solely includes the `safeTransfer` function. It is important to mention that this function does not revert for calls to an EOA.

2.15.1 Issues & Recommendations

No issues found.

2.16 Core/Libraries/TickManagement

`TickManagement` handles all interactions related to ticks, such as initializing the tick state with the minimum and maximum tick during deployment. Ticks are also added or removed depending on whether a tick was toggled and liquidity was added or removed.

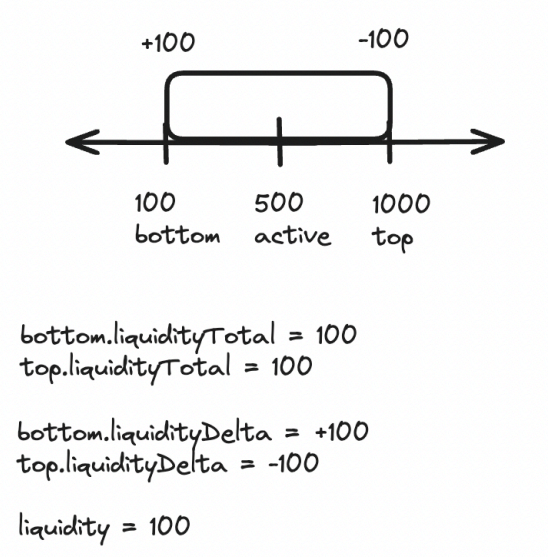
In simple terms, when a tick is added, the contract handles the correct determination of the next and previous tick for the newly added ticks and also adjusts the `nextTick` from the `previousTick` and the `previousTick` from the `nextTick`, such that the newly added tick is perfectly inserted. During removal, the tick will be completely removed and the corresponding previous and next ticks will be interconnected with each other, such that the whole due to the removed tick is perfectly filled.

Additionally, the update and cross of a tick is being handled, the first takes care of the correct total and delta liquidity (liquidity net) and the initial setting of `outerFeeGrowthToken`. The latter takes care of the correct updated of `outerFeeGrowthToken`.

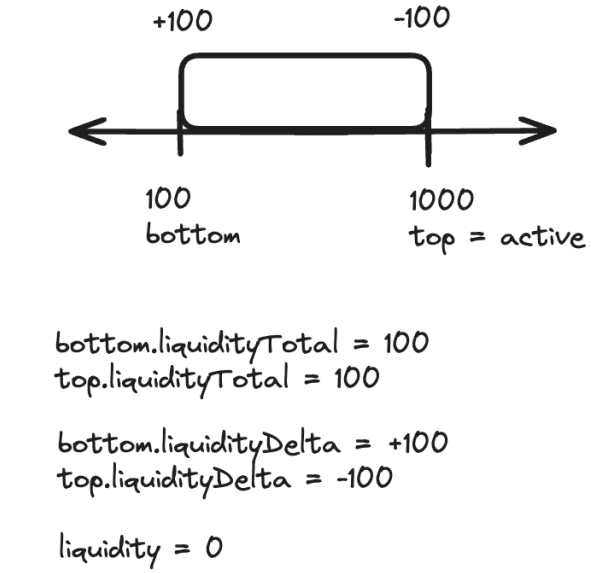
The logic behind the liquidity net ensures that only the liquidity for the current active ranges is displayed as liquidity (active liquidity), the logic behind this can be illustrated with the following example.

Initial liquidity addition across the active tick

- Add 100 LIQ on [100; 1000]; activeTick = 500

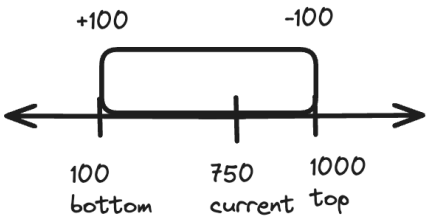


- Swap Y -> X; cross top (from left to right):



As can be seen, the liquidity (active liquidity) is zero, since the cross of the top tick resulted in a liquidity decrease.

- Swap X -> Y; cross top (from right to left):




```
bottom.liquidityTotal = 100
top.liquidityTotal = 100

bottom.liquidityDelta = +100
top.liquidityDelta = -100

liquidity = 100
```

As seen above, the liquidity (active liquidity) was increased again. The following rule applies for the swap directions:

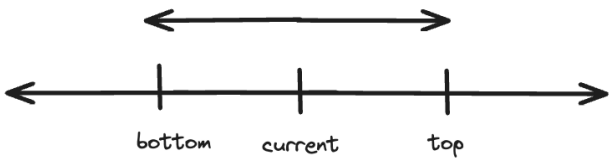
- Cross tick from X -> Y: Add negative `liquidityDelta`

- Cross tick from Y -> X: Add positive `liquidityDelta`

This logic is universally applicable and works for all different state transitions.

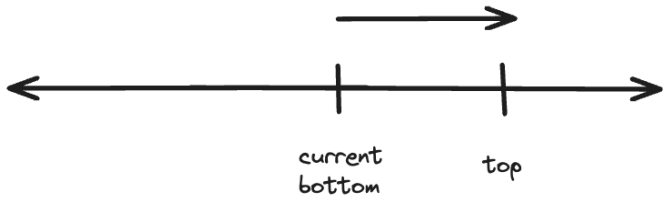
When it comes to the fee calculation logic, there are five different possible scenarios to calculate the inner growth inside a given range:

1.



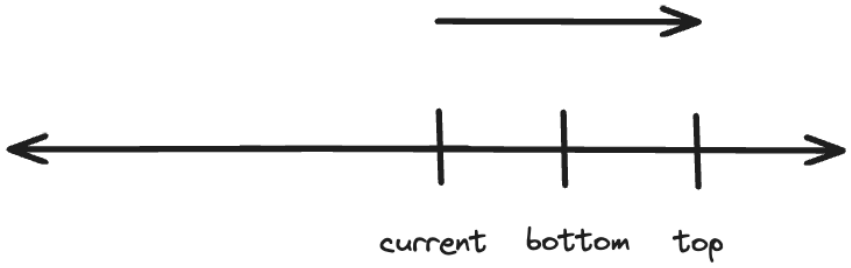

```
innerFeeGrowthToken = globalFeeGrowthToken - lower.outerFeeGrowthToken - upper.outerFeeGrowthToken
```

2.



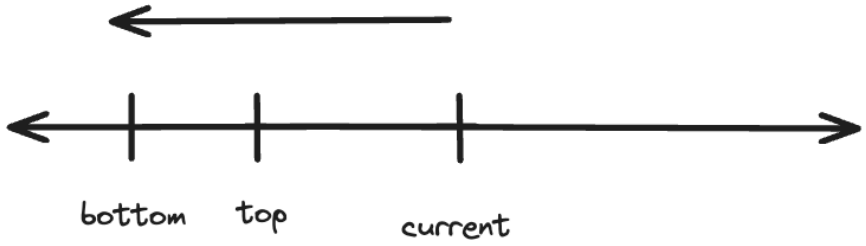
innerFeeGrowthToken = globalFeeGrowthToken - lower.outerFeeGrowthToken - upper.outerFeeGrowthToken

3.



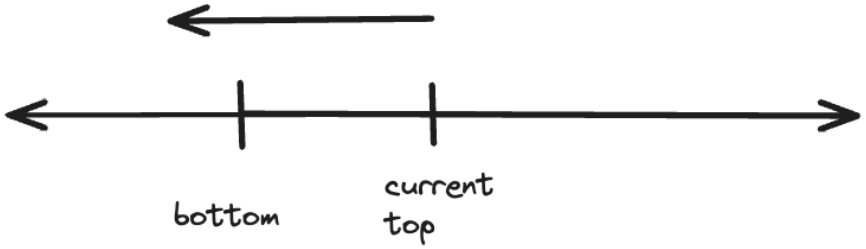
innerFeeGrowthToken = lower.outerFeeGrowthToken - upper.outerFeeGrowthToken

4.



innerFeeGrowthToken = upper.outerFeeGrowthToken - lower.outerFeeGrowthToken

5.



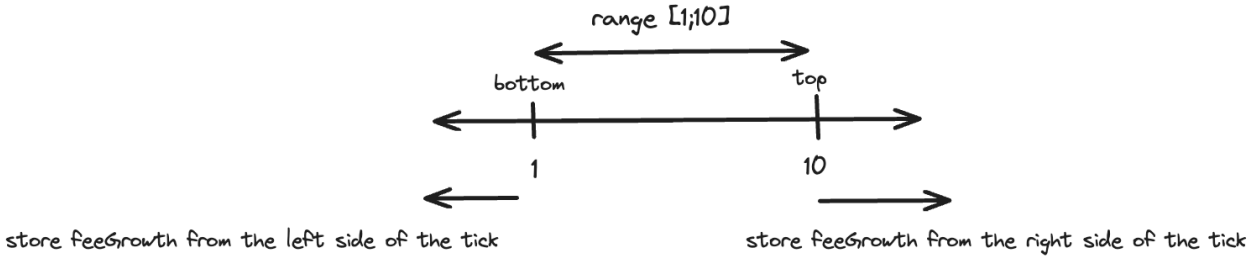
innerFeeGrowthToken = upper.outerFeeGrowthToken - lower.outerFeeGrowthToken

The fee determination for a position is simply based on the delta of how much fee was made between the upper and lower boundaries of said position and vice-versa (if the active tick is not inside the range). If the active tick is inside the range, `totalFeeGrowth` is included in the calculation.

The correctness of the result depends on the underlying logic when a tick is crossed, as it will be always stored how much fee was made on the left/right side of the crossed tick, depending on the swap direction.

If a tick was crossed from left to right (Y to X), the fee growth on the left side of that tick is stored, and respectively a cross from right to left (X to Y) will store the fee growth from the right side. It is important to note that the fee calculation logic also incorporates the special case where the `bottomTick` is crossed and the new `activeTick` is `bottomTick -1`.

To simplify, it can be illustrated as follows:



1. active tick between [1;10], can be equal 1:
-> innerFeeGrowth = totalFeeGrowth - bottom.feeGrowth - upper.feeGrowth

2. active tick below 1:
-> bottom.feeGrowth - upper.feeGrowth

3. active tick >= 10:
-> upper.feeGrowth - bottom.feeGrowth

As already mentioned above, at first glance, it might seem odd that scenario 1 includes the first tick being 1. However, this is for the specific scenario where the bottom tick was not crossed but the active tick became the bottom tick due to the special logic within the `swapCalculation` for the `zeroToOne` direction.

2.16.1 Issues & Recommendations

No issues found.

2.17 Core/Libraries/TickTree

`TickTree` is an optimized tree used to track initialized ticks. It helps to find the next initialized tick in over 1M6 ticks in less than five reads.

The tree is used by the Tick themselves to store the previous and next tick to optimize the swap iterations.

2.17.1 Issues & Recommendations

Issue #08	Gas optimizations
Severity	INFORMATIONAL
Description	L68 `tick++;` This line can be changed to `++tick` to save some gas L135 `nextBitIndex = bitIndex + int24(uint24(255 - bitIndexInWord));` This line can be updated to: `nextBitIndex = bitIndex | 255;`
Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	RESOLVED

2.18 Core/Libraries/TokenDeltaMath

`TokenDeltaMath` is responsible for the correct mathematical calculation of tokens for a given range and liquidity. The math is based on the following.

Via mathematical derivation, equation 2.2 in Uniswap:

$$(x + \frac{L}{\sqrt{p_b}})(y + L\sqrt{p_a}) = L^2 \qquad (2.2)$$

Following the logical flow to either provide `token0` or `token1` and setting the other value to zero can yield into the following formulas for liquidity calculation.

Liquidity for `token0`:

```
L = amount0 * (sqrt(upper) * sqrt(lower)) / (sqrt(upper) - sqrt(lower))
```

Liquidity for `token1`:

```
L = amount1 / (sqrt(upper) - sqrt(lower))
```

Via equivalent forming, we receive the corresponding formula for the liquidity -> amounts calculation.

Amount of `token0`:

```
amount0 =  ((sqrt(upper)-sqrt(lower)) * L / sqrt(upper)) / sqrt(lower)
```

Amount of `token1`:

```
amount1 = L * (sqrt(upper) - sqrt(lower))
```

The corresponding functions will round against the favor of the user: when adding liquidity, it will round up, taking more tokens from the user. During burn, it will round down, resulting in less tokens for the user.

2.18.1 Issues & Recommendations

No issues found.

2.19 Farming

The Farming Module is an architecture that accommodates farming with liquidity positions, leveraging the `NonfungiblePositionManager` logic.

There are three main contracts involved:

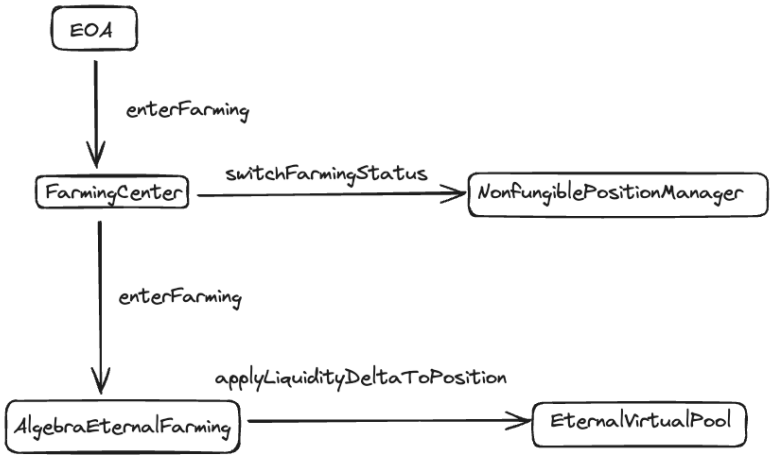
- `FarmingCenter`: Allows users to manually enter and exit positions using their corresponding NFT. It is invoked upon position manipulation (increase or decrease of liquidity) by the `NonfungiblePositionManager`.

- `AlgebraEternalFarming`: Allows the `IncentiveMaker` to create various pools. Connects the `FarmingCenter` with the virtual pool. Keeps track of the storage for the different positions and handles the reward distribution.

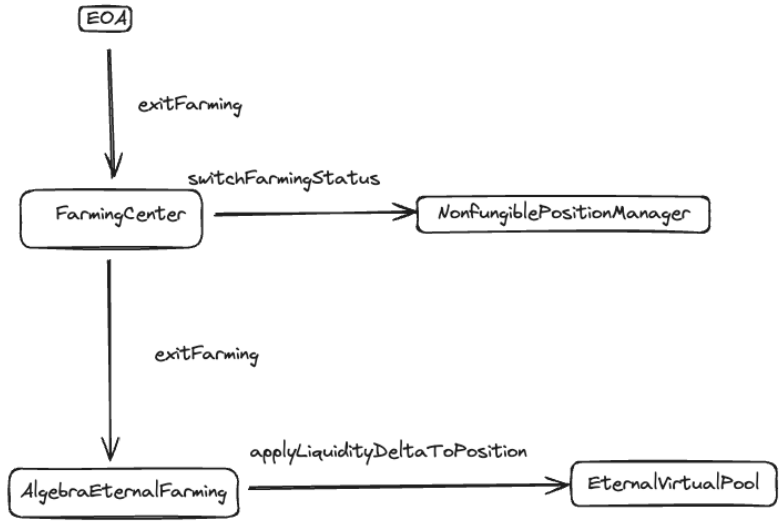
- `EternalVirtualPool`: Inherently connected to the previous parts of the module, keeps track of the correct mimic from the real pool during entering and exiting of positions (tick update, `currentLiquidity` update, prev and next tick update). Handles inherent challenge of the virtual pool having less positions than the real pool due the nature of not every position being included in farming.

Users can enter/exit/modify their farming position on the corresponding callpaths.

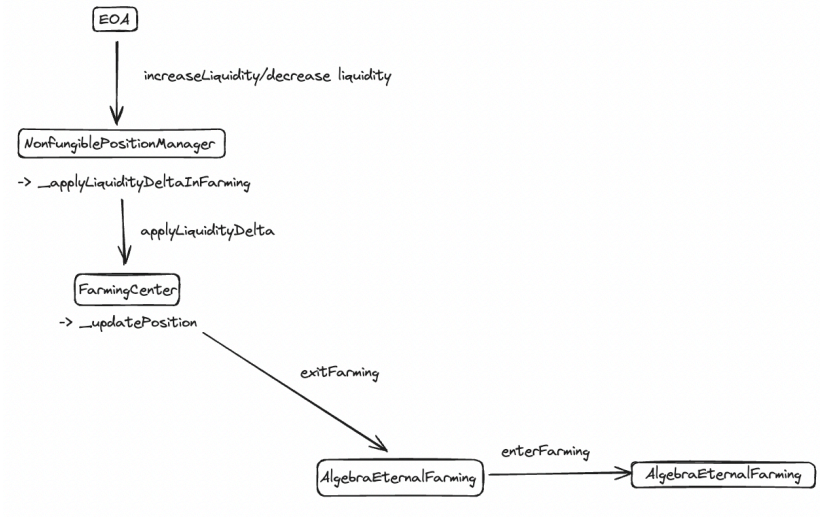
1. Enter Farming: This allows users to enter the farming module with a pre-existing position.



2. Exit Farming: This allows users to exit the farming module fully.



3. Increasing/decreasing a position: Users can increase or decrease a position which will have inherent effects if this position is used in the farming module:


EOA
increaseLiquidity/decrease liquidity
NonfungiblePositionManager
-> _applyLiquidityDeltaInFarming
applyLiquidityDelta
FarmingCenter
-> _updatePosition
exitFarming
enterFarming
AlgebraEternalFarming
AlgebraEternalFarming

2.20 Farming/IncentiveKey

`IncentiveKey` is a simple helper contract which stores the data for the IncentiveKey struct.

2.20.1 Issues & Recommendations

No issues found.

2.21 Farming/VirtualTickStructure

`VirtualTickStructure` contains exactly the same logic as the `TickStructure` contract and is invoked upon the update of a pool position in the virtual pool.

2.21.1 Issues & Recommendations

No issues found.

2.22 Farming/AlgebraEternalFarming

AlgebraEternalFarming is the main contract for the farming module.

Incentive creation

First of all, any address with the `INCENTIVE_MAKER_ROLE` can call the `createEternalFarming` function, which creates a new farming opportunity (incentive) and deploys a corresponding virtual pool, which is only possible if there is currently no incentive plugged and then automatically plugs the created incentive to the corresponding pool.

Upon the creation, the caller can determine a reward and bonus token and deposit these tokens into the contract. Additionally, an arbitrary reward rate can be set for both the main and bonus token.

At this point, it is important to note that there is no duration attached to this call, as the rewards are naturally only distributed until the provided rewards, marked as `_rewardReserve`, is depleted. The top-up of rewards is granted to any address, even non-privileged, via the `addRewards` function.

Each created incentive is uniquely determined by the hash of `rewardToken`, `bonusToken`, `pool` and nonce with an ever-increasing nonce upon creation to prevent any collision. Any created incentive can be deactivated and disconnected from the plugin by any address with the `INCENTIVE_MAKER_ROLE`. Once an incentive is disconnected, it cannot be connected again, forming the path for the necessity to create a new incentive.

Business logic

Besides the privileged interactions with this contract to create, deactivate or manipulate an incentive, this contract accommodates the `enter` and `exit` farming calls which are executed by the `FarmingCenter` and forwards these calls to `EternalVirtualPool` to ensure the correctness of the tick structure and virtual pool state. Additionally, correct reward accumulation is ensured.

To further increase the module safety, an `emergencyWithdraw` functionality has been implemented which can be turned on and off by the admin. In the case where `emergencyWithdraw` is activated, users cannot enter the farming and exiting will not update the virtual pool state.

2.22.1 Privileged Functions

- `createEternalFarming (IncentiveMaker)`
- `deactivateIncentive (IncentiveMaker)`
- `setRates (IncentiveMaker)`
- `decreaseRewardsAmount (Administrator)`
- `setFarmingCenterAddress (Administrator)`
- `setEmergencyWithdrawStatus (Administrator)`
- `enterFarming (FarmingCenter)`
- `exitFarming (FarmingCenter)`
- `claimRewardFrom (FarmingCenter)`
- `collectRewards (FarmingCenter)`

2.22.2 Issues & Recommendations

Issue #09	Governance: Change of various parameters can result in DoS and loss of rewards
Severity	GOVERNANCE
Description	The governance structure allows for changing sensitive parameters like `farmingCenter`, which essentially allow the governance to DoS the proper functionality of an incentive as well as claiming rewards on users behalf.
Recommendation	Consider having the governance under a KYCed multi-signature contract that can be trusted to not change these parameters.
Resolution	ACKNOWLEDGED The client stated the following: "Strict access control is required. To improve security, it is possible to use multisign and special contracts with more granular access rights."

Issue #10	`emergencyWithdraw` can result in manipulated farming
Severity	MEDIUM SEVERITY
Description	The contract implements an `emergencyWithdraw` mode which can be turned on and off by governance. If turned on, further deposits are not permitted and exits will not update the virtual pool. To understand what could go wrong, we need to take a look at the `_updatePosition` function in `FarmingCenter`: `if (liquidity == 0 ‖ virtualPoolAddresses[address(key.pool)] == address(0)) { _exitFarming(key, tokenId, tokenOwner); } else { IAlgebraEternalFarming(eternalFarming).exitFarming(key, tokenId, tokenOwner);` `if (IAlgebraEternalFarming(eternalFarming).isIncentiveDeactivated(IncentiveId.compute(key))) {` `// exit completely if the incentive has stopped (manually or automatically)` `_switchFarmingStatusOff(tokenId);` `} else {` `// reenter with new liquidity value` `IAlgebraEternalFarming(eternalFarming).enterFarming(key, tokenId);` `} }` As highlighted, the farming status for `tokenId` is only switched off if the liquidity is zero, i.e., if a user decreased the liquidity completely. But if a user attempts to increase the liquidity such that 1 wei is left, it will enter the else clause:

```
} else {
    IAlgebraEternalFarming(eternalFarming).exitFarming(key,
tokenId, tokenOwner);
    if
(IAlgebraEternalFarming(eternalFarming).isIncentiveDeactivat
ed(IncentiveId.compute(key))) {
        _switchFarmingStatusOff(tokenId);
    } else {
        // reenter with new liquidity value

IAlgebraEternalFarming(eternalFarming).enterFarming(key,
tokenId);
    }
}
```

Which then first exits the farming (the virtual pool is not updated due to the special scenario) and deletion of the farm storage. It is important to mention that incentive is NOT DEACTIVATED in this scenario. Which means an attempt is made to enter the farming again with the new liquidity value, which is 1 in that scenario. However, the `enterFarming` call will directly revert in the first line:

```
if (isEmergencyWithdrawActivated) revert
emergencyActivated();
```

This results in the sub-call ultimately reverting, which then of course also reverses the storage transitions which have been made in the exit call. Additionally, due to the try-catch logic within the `NonfungiblePositionManager`, the liquidity decrease was successfully finalized.

The result of this scenario is that a user has withdrawn the majority of their position while the farming for this position is still active. This means that a user will now still receive rewards for the initial value of the position, while in fact the position has only a size of 1 wei.

Additionally, it is also notable to mention that the normal exit of a position (no decrease) will not change the virtual pool. In any scenario where the `emergencyWithdraw` is deactivated again, this will result in an inflated active liquidity.

	We understand that `emergencyWithdraw` is only invoked upon unexpected events and that most users will therefore exit farming fully. However, if there are rewards left, very sophisticated users can use this exploit to withdraw their position (with leaving 1 wei) while still receiving rewards for their full position.
Recommendation	Since the main issue with this logic is that users will still have the full existing position after they have decreased their position, it might make sense to not allow the increase/decrease of a position during an emergency situation but rather force the user to exit the farming with the full position. Furthermore, it must be noted that if users are limited to exit farming, they will not receive their updated rewards due to the lack of the virtual pool update.
Resolution	PARTIALLY RESOLVED The team stated the following: "It is not possible to manipulate, the try-catch will not catch such a revert (custom error), so the transaction will be reverted. However, we prefer to handle this scenario explicitly, so we added a check for the emergency status."

Issue #11	**`setRates` lacks proper input validation which allows `INCENTIVE_MAKER` to steal all rewards**
Severity	MEDIUM SEVERITY
Description	First of all, it needs to be mentioned that the code is explicitly structured in such a way that only the administrator can withdraw rewards via the `decreaseRewardsAmount` function. This means that the `INCENTIVE_MAKER` is likely a less trusted entity which should not have this privilege. Unfortunately, `INCENTIVE_MAKER` can still withdraw rewards by abusing the `setRates` function: 1. Add liquidity to a very distant range. 2. Swap `tokenX` to `tokenY` to bring the pool in this range. 3. Call `setRewards` with astronomically high rates such that during the next block, all outstanding rewards will be accumulated to the `currentLiquidity` which is just the liquidity in the distant range. 4. Collect and claim rewards.
Recommendation	Consider setting a reasonable limit for the `setRewards` function.
Resolution	ACKNOWLEDGED The team stated the following: "This role must be assigned to a trusted contract or EOA. To reduce risks, it is worth using a multisig or contract with more granular access rights."

Issue #12	Loss of rewards upon `exitFarming` during `emergencyWithdraw`
Severity	MEDIUM SEVERITY
Description	Whenever users exit their full position via `exitFarming`, the virtual pool is not updated (during `emergencyWithdraw`). While this logic is in fact necessary since the core logic of `emergencyWithdraw` is to ignore the virtual pool state, it will also result in users losing their rightful rewards since the last update. While we acknowledge that it is possible to still call `collectRewards` in the `emergencyWithdraw` scenario, this is most probably not a scenario which will be invoked by users. So in fact we have two issues here: 1. Global rewards are not updated. 2. Users will not get any assigned rewards. This results in users irreversibly losing their accumulated rewards. The same issue will apply if users decrease their liquidity to zero.
Recommendation	Consider applying the following changes: 1. Distribute rewards upon calling `emergencyWithdraw` to ensure an updated reward state. 2. Change the logic in `exitFarming` such that users still get their accumulated rewards assigned to the `rewards[owner]` mapping.
Resolution	ACKNOWLEDGED The team stated the following: "We believe that the emergency withdraw mode should be activated only in extreme cases in which the loss of part of the rewards is an acceptable loss."

Issue #13	**POOLS_ADMINISTRATOR_ROLE can frontrun `createEternalFarming`**
Severity	LOW SEVERITY
Description	When a new farm is created for a pool, `createEternalFarming` is called. This relies on the pool's `plugin()` value to retrieve the plugin that is used to create a new `EternalVirtualPool`. This plugin is responsible for calling the `crossTo` function which reports the ticks used to determine the rewards distribution. If a malicious administrator frontruns this creation and updates the plugin with a malicious one, then it can lead to various issues: - Will not be possible to create that farm - Will not be possible to disconnect the farm from the plugin - The plugin manipulates the price movement inside the virtual pool by providing invalid data.
Recommendation	Consider passing the plugin as well as a parameter when calling `createEternalFarming` which then can be checked with the actual `plugin()` value from the pool.
Resolution	RESOLVED

Issue #14	Malicious user might abuse disconnected state
Severity	LOW SEVERITY
Description	The governance body has the privilege of disconnecting an incentive from the corresponding pool. This can either be done by changing the plugin config in the core or by calling deactivatingIncentive. In the latter scenario, the rates are automatically set to zero, which is perfectly fine. However, in the first scenario, the rates are not reset and the incentive is not deactivated. This opens the possibility for users to still enter the farming and take any potential leftover rewards, while enjoying the benefit that the activeLiquidity range is not changed anymore due to the disconnection from the real pool.
Recommendation	Consider keeping this scenario in mind and acting accordingly, which means that a config change should also include a disconnection.
Resolution	ACKNOWLEDGED

Issue #15	Unnecessary tick determination during `_updatePosition`
Severity	LOW SEVERITY
Description	`_updatePosition` determines the tick depending on the deactivation status: `int24 tick = _isIncentiveDeactivated(incentive) ? virtualPool.globalTick() : _getTickInPoolAndCheckLock(key.pool);` A deactivation scenario is either true if manually deactivated OR if the virtual pool is detached. In such a scenario, the tick is determined as the global tick from the current pool and is used to update the virtual pool: `_updatePositionInVirtualPool(address(virtualPool), farm.tickLower, farm.tickUpper, liquidityDelta, tick);` However, the crux here is that in such a scenario where the deactivation happened manually or automatically in the virtual pool, the position is never updated as it returns early: `if (_deactivated) {` `  // early return if virtual pool is deactivated` `  return;` `}` This in itself is not an issue, however, this logic makes the previous determination of the tick unnecessary as the tick variable will not have an effect in that scenario.
Recommendation	Consider removing the unnecessary determination.
Resolution	ACKNOWLEDGED The team stated the following: "We prefer to leave double checking to prevent potential problems when changing the logic."

Issue #16	Lack of validation for key throughout the codebase
Severity	LOW SEVERITY
Description	Certain functionalities compute the incentiveId from the key without explicitly checking if such an incentive exists. While we acknowledge that this is not exploitable, we are still of the opinion that a regular and general validation, such as calling `_getIncentiveByKey` should be present.
Recommendation	Consider calling `_getIncentiveByKey` instead of computing the `incentiveId` from the key directly.
Resolution	RESOLVED

2.23 Farming/EternalVirtualPool

`EternalVirtualPool` contains logic necessary to mimic the real pool. Inherently, it comes with the challenge that the virtual pool does not represent the real pool 1:1, since naturally not all positions are used for farming purposes. This will result in an inconsistency where the virtual pool will contain fewer positions, corresponding initialized ticks. This can result in discrepancies where the real pool executes swaps to its next/prevTick which is not covered by the virtual pool. The contract was developed in such a manner to automatically use its own prev/nextTicks, which then ensures the correct adjusted update of the active liquidity.

Additionally, this contract handles all reward related calculations, such as increasing/decreasing the `rewardReserves`, calculating the inner growth for a position and distributing the rewards, which simply increases the accumulator values (`totalRewardGrowth0/1`).

2.23.1 Issues & Recommendations

Issue #17	Risk of overflow for reward calculation
Severity	LOW SEVERITY
Description	Within `_distributeRewards`, the following calculation is executed: `(uint256 reward0, uint256 reward1) = (rewardRate0 * timeDelta, rewardRate1 * timeDelta);` Since the `rewardRate` variables do not have an upper safeguard, this can overflow, which will then not revert due to the unchecked bracket.
Recommendation	Consider ensuring a reasonable upper limit which will not overflow even with a large `timeDelta`.
Resolution	ACKNOWLEDGED The team stated the following: "`timeDelta` does not exceed `uint32`, `rewardRate` does not exceed `uint128`, their product cannot overflow `uint256`."

Issue #18	Typographical issues
Severity	INFORMATIONAL
Description	`IAlgebraPool` import is not used.
Recommendation	Consider fixing the typographical issues.
Resolution	RESOLVED

Issue #19	Gas optimizations
Severity	INFORMATIONAL
Description	The `crossTo` function returns early if the pool is deactivated. — L126-134 ``` uint128 _currentLiquidity = currentLiquidity; int24 _globalTick = globalTick ; uint32 _prevTimestamp = prevTimestamp; bool _deactivated = deactivated; int24 previousTick = globalPrevInitializedTick; int24 nextTick = globalNextInitializedTick; if (_deactivated) return false; // early return if virtual pool is deactivated ``` The variables that are retrieved from the storage before the return can be moved after. — `_currentLiquidity` and `_prevTimestamp` can be moved to after the `if` on line 144.
Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	ACKNOWLEDGED The team stated the following: "All these values are written into one storage slot, inheritance rules lead to this. Therefore, the change described will only make things worse."

2.24 Farming/FarmingCenter

`FarmingCenter` acts as the bridge between the `NonfungiblePositionManager` and the incentive. First of all, users can interact directly with the `FarmingCenter` and entering and exiting positions, which is only possible with the full position value. Since the `NonfungiblePositionManager` includes not only the minting of a position but also functions which allow for the manipulation of the positions, namely `increaseLiquidity` and `decreaseLiquidity`, there must be functionality to adjust the positions in the incentive to the correct updated value. This is exactly where the `FarmingCenter` exposes its aforementioned bridge to the incentive, as such a position change will be handled by the `FarmingCenter` and first fully exits the position and then enters the position with the new value.

Additionally, `FarmingCenter` includes functionality to claim and collect rewards and connect/disconnect the virtual pool to/from the plugin.

2.24.1 Privileged Functions

- `applyLiquidityDelta (NonfungiblePositionManager)`
- `connectVirtualPoolToPlugin (EternalFarming)`
- `disconnectVirtualPoolToPlugin (EternalFarming)`

2.24.1 Issues & Recommendations

Issue #20	Invalid return value for `claimReward`
Severity	LOW SEVERITY
Description	`claimReward` returns the reward variable which should be the amount which was effectively claimed. However, the return value is in fact the outstanding reward before the claim. This results in issues if third-parties building on top of algebra relies on the correctness of the return value. Furthermore, the addition of the reward is unnecessary, the reward should simply be assigned to the reward value. `reward += eternalFarming.claimRewardFrom(rewardToken, msg.sender, to, amountRequested)`
Recommendation	Consider adjusting the return value to reflect how much was in fact claimed and consider using a direct assignment instead of an addition.
Resolution	RESOLVED

2.25 Periphery/ERC721Permit

`ERC721Permit` is an abstract contract that extends the functionality of OpenZeppelin's `ERC721Enumerable`. It introduces a permit system to the ERC721 token, allowing token approvals via signatures rather than transactions. This is particularly useful for approving transactions without paying gas fees.

2.25.1 Issues & Recommendations

No issues found.

2.26 Periphery/LiquidityManagement

`LiquidityManagement` is an abstract contract which is inherited by the NonFungiblePositionManager and implements logic for adding liquidity, which is invoked upon the `mint` and `increaseLiquidity` functions.

2.26.1 Issues & Recommendations

No issues found.

2.27 Periphery/Multicall

`Multicall` is inherited by `NonFungiblePositionManager` and `SwapRouter` and facilitates the execution of multiple transactions in the same call.

2.27.1 Issues & Recommendations

Issue #21	Errors during a revert might not be handled correctly
Severity	LOW SEVERITY
Description	L13 - 22 `(bool success, bytes memory result) = address(this).delegatecall(data[i]);` `if (!success) {` `// Next 5 lines from https://ethereum.stackexchange.com/a/83577` `if (result.length < 68) revert();` `assembly {` `result := add(result, 0x04)` `}` `revert(abi.decode(result, (string)));` `}` Errors that are smaller than 68 will revert without any message and custom errors that are bigger than this number will revert when trying to be decoded to a string.
Recommendation	Consider bubbling up every errors following Open Zeppelin implementation: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/Address.sol#L146-L158
Resolution	RESOLVED

2.28 Periphery/PeripheryImmutableState

`PeripheryImmutableState` is inherited by the `PeripheryPaymentsWithFee` contract and handles ERC20 as well as native token transfers.

2.28.1 Issues & Recommendations

No issues found.

2.29 Periphery/PeripheryPayments

`PeripheryPayments` inherited by `PeripheryPaymentsWithFee` and handles `ERC20` as well as native token transfers.

2.29.1 Issues & Recommendations

Issue #22	**The pay function can result in full loss of `msg.value` in certain scenarios**
Severity	MEDIUM SEVERITY
Description	The pay function only uses the native token if in fact the balance of the contract is larger than the necessary amount: `if (token == WNativeToken && address(this).balance >= value)` This will not work for a scenario where a user attempts to mint/increase liquidity or calls the `exactOutputSingle` function on the `SwapRouter`. Essentially this issue relies on the provided `msg.value` to be lower than what is expected — this is partially prevented by the input parameters for `mint` and `increaseLiquidity`. However, for the swap logic consider the following scenario: 1. User calls `exactOutputSingle` and wants to buy 2550 USDC with 1 ETH, the provided msg.value is 1 ETH and the `amountInMaximum` variable is 2575.5, which indicates a 1% acceptable slippage. 2. A malicious user frontruns this transaction, increasing the price to 2562,75 (0.5% increase), which is perfectly fine with the user's provided slippage. 3. However, the pair then attempts to callback with a value of 1.005 ETH: `_swapCallback(amount0, amount1, data); // callback to get tokens from the msg.sender` `else {` `  amountInCached = amountToPay;` `  tokenIn = tokenOut; // swap in/out because exact output swaps are reversed` `  pay(tokenIn, data.payer, msg.sender, amountToPay);` `}`

4. The pay function gets now invoked with 1.005 ETH:

```
function pay(address token, address payer, address
recipient, uint256 value) internal {
    if (token == WNativeToken && address(this).balance >=
value) {
        // pay with WNativeToken
        IWNativeToken(WNativeToken).deposit{value: value}
(); // wrap only what is needed to pay
        IWNativeToken(WNativeToken).transfer(recipient,
value);
    } else if (payer == address(this)) {
        // pay with tokens already in the contract (for the
exact input multihop case)
        TransferHelper.safeTransfer(token, recipient,
value);
    } else {
        // pull payment
        TransferHelper.safeTransferFrom(token, payer,
recipient, value);
}
```

Since 1.005ETH is more than the provided 1 ETH, the first-condition will not be fulfilled. Thus, it automatically enters in the third condition and transfers WETH from the caller. If the caller has WETH in the wallet which is approved, the swap will succeed but `msg.value` will be stuck in the contract. The attacker can now sweep it.

The issue can be avoided if `multicall` is used alongside `refundNativeToken` which will refund the leftovers. Due to this, we decided to mark this issue as medium instead of high severity.

Recommendation

Consider explicitly refunding any balance in the contract at the end of the swap function.

OR

Ensure `multicall` with `refundNativeToken` is used always by the frontend, and consider documenting this issue mentioning that the users should always use the multicall method.

Resolution

RESOLVED

A comment has been added to let users know they must use multicall.

2.30 Periphery/PeripheryPaymentsWithFee

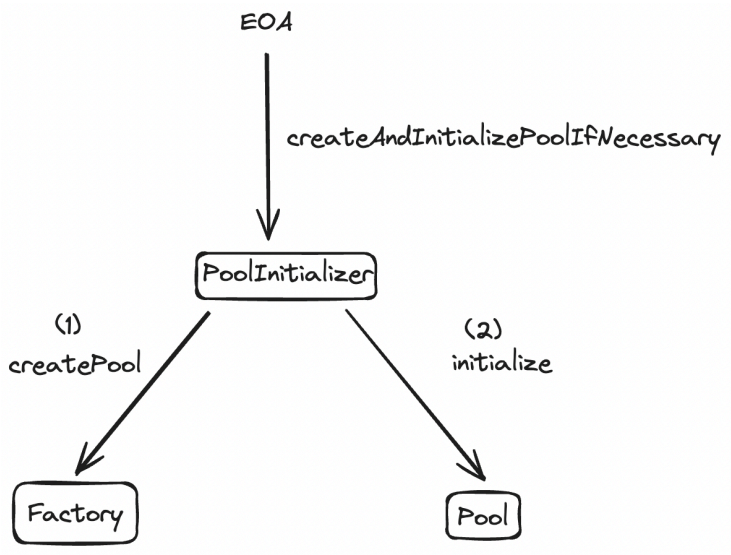
`PeripheryPaymentsWithFee` handles the sweeping and unwrapping of the native token with a fee. It is inherited by `SwapRouter`.

2.30.1 Issues & Recommendations

Issue #23	Functions do not serve any use-case
Severity	INFORMATIONAL
Description	While we acknowledge that certain functions might be used with the `multicall` feature, `unwrapNativeTokenWithFee` and `sweepTokenWithFee` do not seem to have any use-case besides enforcing a potential fee via frontend configuration. At Paladin we always aim to guide contract development away from frontend reliance.
Recommendation	Consider either strictly adding `NATSPEC` to these functions or simply removing them from the architecture.
Resolution	ACKNOWLEDGED

2.31 Periphery/PoolInitializer

`PoolInitializer` is a simple helper contract that facilitates the creation and initialization of pools. It is inherited by the `NonfungiblePositionManager`.



2.31.1 Issues & Recommendations

No issues found.

2.32 Periphery/SelfPermit

`SelfPermit` facilitates the `permit` calls to different tokens — this functionality is usually used within multicalls. This contract is inherited by `SwapRouter` and `NonfungiblePositionManager`.

2.32.1 Issues & Recommendations

No issues found.

2.33 Periphery/NonfungiblePositionManager

`NonfungiblePositionManager` facilitates the sweeping and unwrapping of the native token with a fee. It is inherited by the SwapRouter.

2.33.1 Issues & Recommendations

Issue #24	**Advanced exploit including frontrunning allows for stealing native tokens from users upon minting / increase of liquidity if multicall is not used**
Severity	MEDIUM SEVERITY
Description	First of all, it must be mentioned that several functions including: - `decreaseLiquidity` - `collect` - `burn` - `approveForFarming` are unnecessarily marked as `payable`. Calling these functions with a `msg.value` will result in the balance being MEV'd in the next block due to the `refundNativeToken` function in the `PeripheryPayments` contract. However, it gets interesting if we take a look at the `mint` and `increaseLiquidity` functions, where it is in fact possible to provide a `msg.value`, which then enters in the following clause within the pay function: `if (token == WNativeToken && address(this).balance >= value)` `{` `// pay with WNativeToken` `IWNativeToken(WNativeToken).deposit{value: value}(); // wrap only what is needed to pay` `IWNativeToken(WNativeToken).transfer(recipient, value);` `}` The problem here lies within the provided value. During a `mint` interaction, the `LiquidityAmounts` library calculates the liquidity for the desired amounts: `liquidity = LiquidityAmounts.getLiquidityForAmounts(` `sqrtPriceX96,` `sqrtRatioAX96,` `sqrtRatioBX96,` `params.amount0Desired,` `params.amount1Desired` `);`

In the scenario where `tokenX` and `tokenY` are provided, the liquidity for each provided amount is calculated:

```
    } else if (sqrtRatioX96 < sqrtRatioBX96) {
        uint128 liquidity0 =
getLiquidityForAmount0(sqrtRatioX96, sqrtRatioBX96,
amount0);
        uint128 liquidity1 =
getLiquidityForAmount1(sqrtRatioAX96, sqrtRatioX96,
amount1);

        liquidity = liquidity0 < liquidity1 ? liquidity0 :
liquidity1;
```

But the lower liquidity is taken if a user provided `msg.value` which exactly corresponds to the `desiredAmount` but the liquidity calculation for the other token yields in a lower liquidity, meaning that the provided `msg.value` is excessive. Normally, within the pool, such an excessive provided value is naturally refunded:

```
unchecked {
    // return leftovers
    if (amount0 > 0) {
        if (receivedAmount0 > amount0) _transfer(token0,
leftoversRecipient, receivedAmount0 - amount0);
        else assert(receivedAmount0 == amount0); // must
always be true
    }
    if (amount1 > 0) {
        if (receivedAmount1 > amount1) _transfer(token1,
leftoversRecipient, receivedAmount1 - amount1);
        else assert(receivedAmount1 == amount1); // must
always be true
    }
}
```

The problem is that the pay function within `PeripheryPayments` does NOT forward the whole `msg.value` but only the necessary amount which is determined by the pool:

```
_mintCallback(amount0, amount1, data);
function algebraMintCallback(uint256 amount0Owed, uint256
amount1Owed, bytes calldata data) external override {
    MintCallbackData memory decoded = abi.decode(data,
(MintCallbackData));
    CallbackValidation.verifyCallback(poolDeployer,
decoded.poolKey);

    if (amount0Owed > 0) pay(decoded.poolKey.token0,
decoded.payer, msg.sender, amount0Owed);
    if (amount1Owed > 0) pay(decoded.poolKey.token1,
decoded.payer, msg.sender, amount1Owed);
]}

function pay(address token, address payer, address
recipient, uint256 value) internal {
    if (token == WNativeToken && address(this).balance >=
value) {
        // pay with WNativeToken
        IWNativeToken(WNativeToken).deposit{value: value}();
// wrap only what is needed to pay
        IWNativeToken(WNativeToken).transfer(recipient,
value);
    } else if (payer == address(this)) {
        // pay with tokens already in the contract (for the
exact input multihop case)
        TransferHelper.safeTransfer(token, recipient,
value);
    } else {
        // pull payment
        TransferHelper.safeTransferFrom(token, payer,
recipient, value);
    }
}
```

While in our scenario, `msg.value` is larger than `value`. This is in fact a situation which can occur during the normal business logic. Even if the frontend calculates the correct amounts, a shift of the pool state can already cause such a situation where the provided `msg.value` is excessive.

A refund by the `NonfungiblePositionManager` is however non-existent. A malicious user can therefore scan any `mint` / `increaseLiquidity` call with provided `msg.values` in the mempool, frontrun this transaction and manipulate the pool state such that the provided `msg.value` is excessive and then afterwards sweep the excess native token from the contract.

We acknowledge that slippage parameters are provided as well, however, often there is a small default slippage which is then drained by the attacker.

The root-cause of this problem is that the whole architecture is designed in such a way that the pair always invokes the callback with the exact necessary amounts to execute the transaction, and for standard ERC20 tokens, there will be no excess since they are not provided with the transaction but rather transferred in upon the callback. However, for the native token, that is not the case. Generally speaking Paladin auditors are always extra careful if there is a native token execution involved.

It is important to mention that this can also happen during the normal business logic, without the impact of a malicious user.

A similar issue can be applied to `exactOutputSingle` in the `SwapRouter`.

The issue can be avoided if `multicall` is used alongside `refundNativeToken` which will refund the leftovers. Due to this, we decided to mark this issue as medium instead of high severity.

Recommendation	Consider refunding any excess `msg.value` OR Simply forwarding the full `msg.value` during the pay call, which then automatically results in the pool refunding the excess to the user. However, it is important to check for any potential impact other callbacks such as during the swap. As the pair does not automatically refund excess amounts during the swap, this can result in potential issues. Therefore, the easiest solution might in fact be refunding the leftovers. OR Ensure the `multicall` with `refundNativeToken` is used always by the frontend, and also consider documenting this issue mentioning that the users should always use the `multicall` method.
Resolution	ACKNOWLEDGED The client stated the following: "1. `are unnecessarily marked as `payable` - it is required for correct multicall usage. 2. Partially repeats #22. Users must already use multicall with a refund here, because the exact input amount is not known in advance."

2.34 Libraries/CallbackValidation

`CallbackValidation` verifies the callbacks from the Algebra Pools, mostly used to verify if the caller is a valid algebra pool.

2.34.1 Issues & Recommendations

No issues found.

2.35 Libraries/LiquidityAmounts

`LiquidityAmounts` is responsible for calculating the liquidity for amounts and vice-versa. It is exclusively used within the `LiquidityManager` contract in the `addLiquidity` function.

Via mathematical derivation, the equation 2.2 in Uniswap:

$$(x + \frac{L}{\sqrt{p_b}})(y + L\sqrt{p_a}) = L^2 \qquad (2.2)$$

Following the logical flow to either provide `token0` or `token1` and setting the other value to zero can yield into the following formulas for liquidity calculation.

Liquidity for `token0`:

```
L = amount0 * (sqrt(upper) * sqrt(lower)) / (sqrt(upper) - sqrt(lower))
```

Liquidity for `token1`:

```
L = amount1 / (sqrt(upper) - sqrt(lower))
```

Via equivalent forming, we receive the corresponding formula for the liquidity -> amounts calculation.

Amount of `token0`:

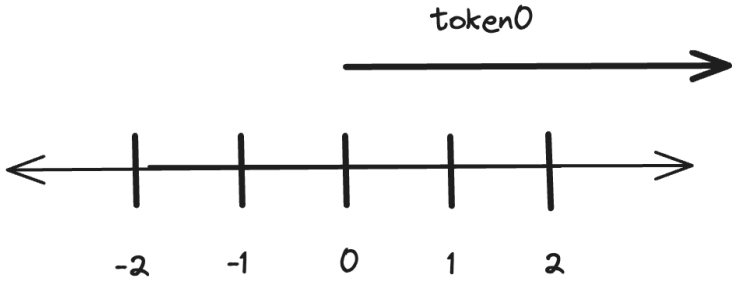
```
amount0 =  ((sqrt(upper)-sqrt(lower)) * L / sqrt(upper)) / sqrt(lower)
```

Amount of `token1`:

```
amount1 = L * (sqrt(upper) - sqrt(lower))
```

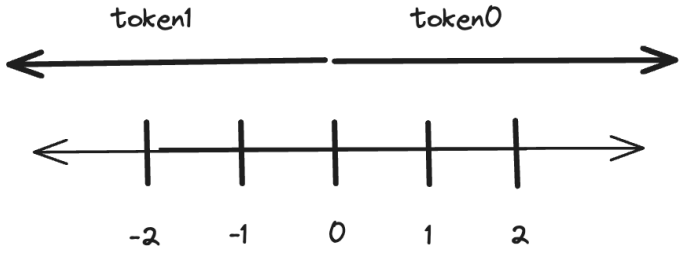
There are three scenarios in how liquidity can be added, in all examples the current tick is 0:

Scenario 1: Add the range starting above or equal current tick



amount0 * (sqrt(upper) * sqrt(lower)) / (sqrt(upper) - sqrt(lower))

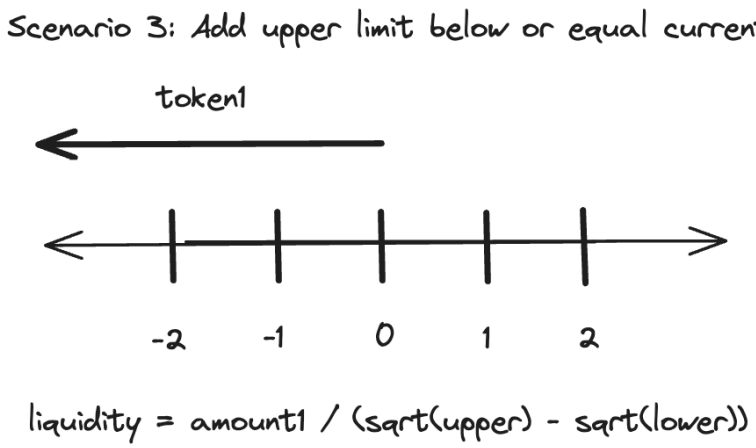
Scenario 2: Add the range crossing the current tick
(lower < current && upper > current)



liquidity = amount0 * (sqrt(upper) * sqrt(lower)) / (sqrt(upper) - sqrt(lower))

liquidity = amount1 / (sqrt(upper) - sqrt(lower))

use lower liquidity



2.35.1 Issues & Recommendations

No issues found.

2.36 Libraries/PoolAddress

`PoolAddress` is responsible for deterministically computing a pool address from the pool deployer and the tokens.

2.36.1 Issues & Recommendations

No issues found.

2.37 Libraries/PoolInteraction

`PoolInteraction` includes commonly used interactions with an Algebra Pool.

2.37.1 Issues & Recommendations

Issue #25	Unused import
Severity	INFORMATIONAL
Description	The `IAlgebraFactory` import is not used.
Recommendation	Consider removing this import.
Resolution	RESOLVED

2.38 Libraries/PositionKey

`PositionKey` is used to compute a position in the pool by using the owner, top tick and bottom tick.The position is encoded in a `bytes32` variable.

2.38.1 Issues & Recommendations

No issues found.

2.39 Plugin

The Plugin module is used to enhance the somewhat limited functionality of an AlgebraPool. Generally speaking, a plugin consists of a suit of hooks that can be called at certain actions:

- Before swap
- After spap
- Before position modification
- After position modification
- Before flash loan
- After flash loan
- Before initialization of the pool
- After initialization of the pool
- If Dynamic fee is enabled

A pool owner can customize one plugin per pool. By default, all pools will use the Default Algebra Plugin if the default plugin factory is set within the `AlgebraFactory`.

This default plugin handles the following actions:

- Dynamic fee calculation
- Updating positions in farming
- TWAP oracle updates

2.40 Plugin/Base/AlgebraFeeConfiguration

`AlgebraFeeConfiguration` contains a struct designed to stores the parameters for the adaptive fee calculation.

2.40.1 Issues & Recommendations

No issues found.

2.41 Plugin/Base/AlgebraBasePluginV1

`AlgebraBasePluginV1` is the default Algebra plugin that was created by the team to handle:

- Dynamic fee calculation
- TWAP oracle values
- Farming updates

This plugin implements all the callback functions that can be used by a pool and should be used as a template to extend its functionality if a pool owner wants to extend functionality of the pool.

We must note that this contract has actual logic only in the following callbacks:

- After initialization
- Before swap
- Dynamic fee

If the pool configuration is changed to trigger any other callbacks, this plugin automatically resets the configuration to the default one that enables the use of the aforementioned functionality.

2.41.1 Privileged Functions

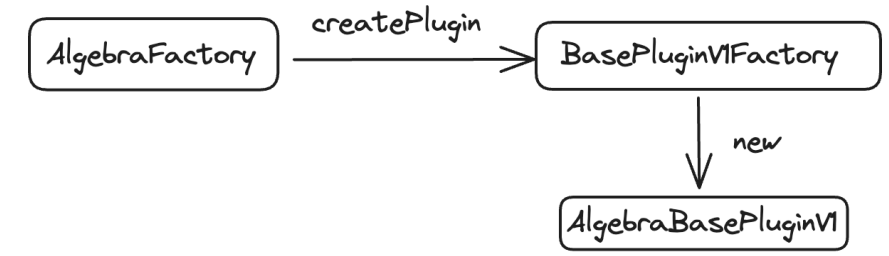
- setIncentive (farming address or last incentive owner)
- beforeInitialize (Pool)
- afterInitialize (Pool)
- beforeModifyPosition (Pool)
- afterModifyPosition (Pool)
- beforeSwap (Pool)
- afterSwap (Pool)
- beforeFlash (Pool)
- afterFlash (Pool)
- changeFeeConfiguration (Plugin Factory or ALGEBRA_BASE_PLUGIN_MANAGER)

2.41.2 Issues & Recommendations

Issue #26	Gas optimizations
Severity	INFORMATIONAL
Description	L188 - 191 `// we allow the one who connected the incentive to disconnect it,` `// even if he no longer has the rights to connect incentives` `if (_lastIncentiveOwner != address(0)) accessAllowed = msg.sender == _lastIncentiveOwner;` `if (!accessAllowed) accessAllowed = msg.sender == IBasePluginV1Factory(pluginFactory).farmingAddress();` `accessAllowed` can be simplified to `accessAllowed = msg.sender == _lastIncentiveOwner || msg.sender == IBasePluginV1Factory(pluginFactory).farmingAddress();`
Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	ACKNOWLEDGED The client stated the following: "The proposed optimization provides small savings of gas on logical branching. However, we prefer a more explicit and strict syntax here since we do not expect to use this function very often."

2.42 Plugin/Deployment/BasePluginV1Factory

`BasePluginV1Factory` is the default factory used by the pool factory to deploy new plugins for each pool that is created.



2.42.1 Privileged Functions

- `createPlugin (Algebra Pool Factory)`
- `createPluginForExistingPool (POOLS_ADMINISTRATOR_ROLE)`
- `setDefaultFeeConfiguration (Algebra Pool Factory Owner or ALGEBRA_BASE_PLUGIN_FACTORY_ADMINISTRATOR)`
- `setFarmingAddress (Algebra Pool Factory Owner or ALGEBRA_BASE_PLUGIN_FACTORY_ADMINISTRATOR)`

2.42.2 Issues & Recommendations

No issues found.

2.43 Plugin/Lens/AlgebraOracleV1TWAP

`AlgebraOracleV1TWAP` is the default factory used by the pool factory to deploy new plugins for each pool that is created.

2.43.1 Issues & Recommendations

No issues found.

2.44 Plugin/Libraries/OracleLibrary

`OracleLibrary` contains logic that is used in `AlgebraOracleV1TWAP`.

2.44.1 Issues & Recommendations

No issues found.

2.45 Plugin/Libraries/AdaptiveFee

`AdaptiveFee` is a library used to calculate the adaptive fee as explained in the Algebra Tech Paper https://algebra.finance/static/Algerbra%20Tech%20Paper-15411d15f8653a81d5f7f574bfe655ad.pdf.

2.45.1 Issues & Recommendations

No issues found.

2.46 Plugin/Libraries/VolatilityOracle

`VolatilityOracle` is a library that is used to provide the price and volatility data that is stored in "timepoints". One timepoint is created before each swap (once per block, in case multiple swaps are happening in the same block) and stores the following data:

- `initalized`: If the timepoint has been initialized
- `blockTimestamp`: The timestamp of the block this timepoint was created
- `tickCumulative`: The tick accumulator which means the tick * time elapsed since the pool was first initialized
- `volatilityCumulative`: The volatility accumulator
- `tick`: Current tick at that block timestamp
- `averageTick`: Aaverage tick at this timestamp
- `windowStartIndex`: The closest timepoint lower or equal with the last timepoint - WINDOW

2.46.1 Issues & Recommendations

No issues found.

3 Appendix: Impermanent Loss

Impermanent loss refers to the temporary loss of funds experienced by liquidity providers in an Automated Market Maker (AMM) like Uniswap/Algebra due to volatility in the price of assets in their liquidity pool. It occurs when the price of assets in a liquidity pool diverges from their price at the time of deposit. The loss is 'impermanent' because it could be recovered if the asset prices return to their original ratio.

Unfortunately, specifically concentrated liquidity protocols such as Algebra can result in a large impermanent loss for users if positions are not managed properly. During our research, we realized that there is no widely available calculation plan for impermanent loss in concentrated liquidity protocols. Therefore, we will provide a simple guide such that anyone can calculate their (potential) impermanent loss.

When a position is currently active, the current price is within the boundaries of this position. In a total equilibrium where the price is exactly in the middle, both tokens are worth the same denominated in each other's asset. Note that that the codebase itself uses the `sqrtPriceX96` instead of clear prices (square root of the price in 96 bits format).

Example: Price = 2000, range [1904.7, 2100], ETH = 10; USDC = 20 000

The range calculation for equal ratios can be done by:

```
sqrtPriceCurrentX96 ^2 / sqrtPriceUpperX96
```

respectively dividing by `sqrtPriceLowerX96`,

`sqrtPriceX96(1904) = 345780049422742027739788815516 3`

`sqrtPriceX96(2000) = 354319114228591420592203432321 4`

`sqrtPriceX96(2100) = 363069051893879129126778256292 2`

1. Calculating the liquidity using the mathematical expressions

```
Lx = amountX * (sqrtPriceX(lowerPrice) * sqrtPriceX(upperPrice)) / (sqrtPriceX(upperPrice) - sqrtPriceX(lowerPrice))

Ly = amountY * (1<<96) / (sqrtPriceX(upperPrice) - sqrtPriceX(lowerPrice))
```

`Lx = 18556636895410796738094`
`Ly = 18556636895410796738094`

Both liquidities are equal since the range and amounts perfectly align in the ratio.

2. Calculating the new composition of the range

Now we can trivially determine the impermanent loss which occurs due to a shift of the current tick, using the following formula to calculate the amounts for a change in the current tick:

```
amountX = (liquidity <<96) * ((sqrtRatioUpperX96 - sqrtRatioLowerX96) / sqrtRatioUpperX96) / sqrtRatioLowerX96

amountY = liquidity * (sqrtRatioUpperX96 - sqrtRatioLowerX96) / 1<<96
```

As we know, the initial range was `[1904,7 ; 2100]` with `activeSqrtPriceX96 = 2000`.

If the price of ETH increases by 2.5% to 2050, the corresponding amounts can be calculated as follows:

`sqrtRatioX96(2050) = 358720762676802631235172110736 2`

New range for ETH : [2050; 2100]

```
amountX = (18556636895410796738094 << 96) *
(3630690518938791291267782562922- 358720762676802631235172110736 2) /
3630690518938791291267782562922 / 358720762676802631235172110736 2
-> 4.90e18
```

New range for USDC: [1904,7; 2050]

```
amountY = 18556636895410796738094 * (358720762676802631235172110736 2 -
345780049422742027739788815516 3) / (1<<96)
-> 30304e18
```

3. Comparing the pre and post values

Total USD value if ETH was held separately:

From ETH: 10 * 2050 = 20500
From USDC: 20000
= 40500

Total USD value in liquidity position:

From ETH: 4.90 * 2050 = 10045
From USDC: 30304
= 40349

Total IL: 0.374%

The impermanent loss must always be calculated manually using this formula, as liquidity ranges for different depositors always change.


PALADIN
BLOCKCHAIN SECURITY