BAIL security

BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT

Gamma Vaults

February 2025

Disclaimer:

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

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

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

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

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

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

1. Project Details

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

Project	Gamma Vaults
Website	gamma.xyz
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/GammaStrategies/AlgebraIntegralHypervisor/tree/1d74699324ca5d96c227608c14aa1f3929963548
Resolution 1	https://github.com/GammaStrategies/AlgebraIntegralHypervisor/tree/b757a3bcbaf6948d3276355776c6c2b5b7ef73d8/contracts

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	14	4	1	6
Medium	17	2	1	14
Low	16	3		13
Informational	16	5		11
Governance	3			3
Total	66	14	2	47

2.1 Detection Definitions

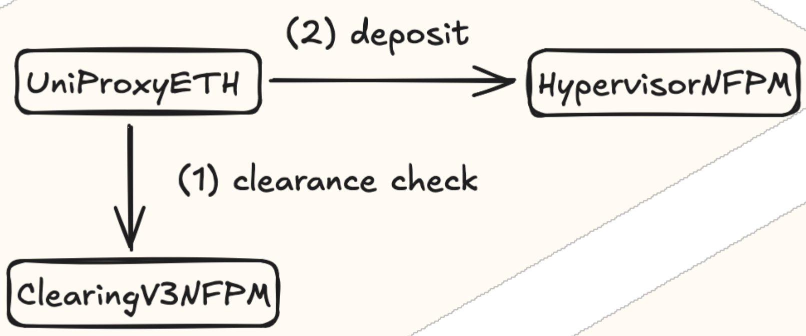
Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
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

Global

Issue_01	Contract does not work with transfer-tax tokens
Severity	**Informational**
Description	This contract is not compatible with transfer-tax tokens. If these token types are used for any purpose within the contract, this will result in down-stream issues and inherently break the accounting.
Recommendations	Consider not using these tokens.
Comments / Resolution	Acknowledged.

Hypervisor Module



PositionValue

The PositionValue contract is a simple library which is used within the HypervisorNFPM contract for the following purposes:

- Fetch amounts for [liquidity, range, price]
- Fetch liquidity for [amounts, range, price]
- Fetch position information for a specific tokenId
- Fetch liquidity amount for corresponding share amount
- Calculation of unclaimed fees based on [feeGrowthInside; feeGrowthInsideLast]
- Calculation of feeGrowthInside based on [totalFeeGrowth; upper.outerFeeGrowth; inner.outerFeeGrowth

Appendix: feeGrowthInside calculation

In Algebra, pools accumulate fees during swaps. If a swap from X -> Y is executed, the pool accumulates Y, if the swap is vice-versa, the pool accumulates X. Since v3 usually has different positions over different ranges, it is not possible to simply accrue these fees over the full range. Instead, fees must be accrued by these positions which are currently in range.

To accommodate that, a totalFeeGrowthToken0/1 variable was implemented for tokenX/tokenY. This variable tracks the fee per liquidity, scaled by 1x128.

> totalFeeGrowthToken += fees * (1<<128) / liquidity

Example:

A swap has happened and accrued 1e18 fees in tokenX while the active liquidity is 1100e18:

totalFeeGrowthToken += 1e18 * (1<<128) / 1100e18
totalFeeGrowthToken = 3.039e35

Now another swap happens which accrues 10e18 fees in tokenX while the activeLiquidity is 1000e18

totalFeeGrowthToken = 3.039e35
3.039e35 += 10e18 (*1<<128) / 1000e18
3.039e35 += 3.402e36
totalFeeGrowthToken = 3.705e36

Now since we are aware that totalFeeGrowthToken aggregates the fee per unit of liquidity, the following calculation allows us to determine how much fees a specific position has accrued:

> (innerFeeGrowthToken - innerFeeGrowthTokenLast) * liquidity

Now as we can see, the variable which is used includes "inner". We need to understand that, as already explained, each v3 position incorporates its very own storage mechanism.

Here are a all important properties:

a) Whenever a position is newly added, it consists of upper and lower tick

   i) If the tick had liquidity before, outerFeeGrowth is not adjusted
   ii) If the tick had no liquidity before and the tick is <= currentTick, outerFeeGrowth is set to totalFeeGrowth
   iii) If the tick had no liquidity before and the tick is >= currentTick, outerFeeGrowth is not set

b) Whenever a position is newly added, feeGrowthInside will be set to the current feeGrowthInside for the range

c) Whenever a swap happens and a tick for a position is crossed, the outerFeeGrowth of the tick will be set to [feeGrowthTotal - outerFeeGrowth]

d) outerFeeGrowth always corresponds to the feeGrowth outside of the position to either the left or the right side (depending on lower/upper)

e) Depending on the currentTick and the range, the following calculations are used to calculate feeGrowthInside:

    i) **currentTick < lowerTick** (lowerTick was crossed; X > Y):
    > innerFeeGrowth = lower.outerFeeGrowth - upper.outerFeeGrowth

    ii) **currentTick >= upperTick** (upperTick was crossed; Y > X):
    > innerFeeGrowth = upper.outerFeeGrowth - lower.outerFeeGrowth

    iii) **lower < current = upperTick** (upperTick was crossed; Y > X)
    > innerFeeGrowth = upper.outerFeeGrowth - lower.outerFeeGrowth

    iv) **lowerTick < currentTick < upperTick** (no tick was crossed):
    > innerFeeGrowth = totalFeeGrowth - lower.outerFeeGrowth - upper.outerFeeGrowth

    v) **lowerTick = currentTick** < upperTick (lowerTick was not crossed):
    > innerFeeGrowth = totalFeeGrowth - lower.outerFeeGrowth - upper.outerFeeGrowth

f) The last scenario: lowerTick = currentTick < upperTick is a special scenario, as one would assume the calculation must be as follows:

> innerFeeGrowth = lower.outerFeeGrowth - upper.outerFeeGrowth.

The reason is because one would assume that is that the scenario:

lowerTick = currentTick

resulted in a cross of the tick and thus lower.outerFeeGrowth = [feeGrowthTotal - outerFeeGrowth]. This is the only thing that would make sense.
However, in fact, in that scenario, the tick has not crossed. This is because of the edge-case within SwapCalculation._calculateSwap for the zeroToOne scenario where currentTick = nextTick - 1. Consider the following example: lowerTick = 0; currentTick = 10. X -> Y swap is being executed nextTick = 1, which means the lowerTick is not crossed. However, currentTick will be set to nextTick - 1 which means now that the condition: currentTick = lowerTick is met without the tick being crossed and thus tick = 0 still remains the same outerFeeGrowth as before and the calculation must be executed as if the currentTick is inside the range.

g) The calculation explicitly assumes underflows/overflows which is derived from the "unchecked" setting. There are specific scenarios where this is necessary to keep the integrity of the fee calculation mechanism. For example, in the following situation, this becomes relevant: https://github.com/Uniswap/v3-core/issues/573.

More specifically, the following conditions must be true:
a) currentTick is lower than lowerTick
b) lowerTick of the position was not yet initialized
c) upperTick was initialized. If now a position is added based on these properties, innerFeeGrowth will be calculated as follows:

> innerFeeGrowth = lower.outerFeeGrowth - upper.outerFeeGrowth
This means it becomes negative.

Example:

a) currentTick = 0
b) lowerTick = 1
c) upperTick = 10
d) lowerTick.outerFeeGrowth = 0
e) upperTick.outerFeeGrowth = 100

f) totalFeeGrowth = 100

This will result in: innerFeeGrowth = 0 - 100

Core Invariants:

INV 1: calculatePositionFee must include tokenOwed0/1 and unclaimed fee b/w [feeGrowthInsideLast; feeGrowthInside]

INV 2: feeGrowthInside0Lastx128/1 must represent latest updated state

INV 3: calculatePositionFee must always return the exact same value as NonfungiblePositionManager.collect

Privileged Functions

- none

Issue_02	Lack of underflow possibility within calculatePositionFee and getFeeGrowthInside
Severity	**High**
Description	Currently, there is no unchecked block within both aforementioned functions which allows for the correct calculation of the fee. Instead, it will not allow for underflow and thus the fee calculation can become flawed / revert in that specific edge-case as described within the appendix.
Recommendations	Consider completely removing calculatePositionFee and getFeeGrowthInside and instead, calling zeroBurn before the ratio mandate calculation which ensures that the ratio is in fact calculated based on the correct aum amount of the vault and not some calculated values which may derive due to rounding, whatsoever.
Comments / Resolution	Failed resolution, while an unchecked block was added within the getFeeGrowthInside function, the calculatePositionFee function still follows the wrong pattern. The correct way to do it can be found within [Positions._recalculatePosition]()

Issue_03	getFeeGrowthInside is not 1:1 taken from Algebra
Severity	**Informational**
Description	Algebra's feeGrowthInside calculation is as follows: *if (currentTick < topTick) {* *if (currentTick >= bottomTick) {* *innerFeeGrowth0Token = totalFeeGrowth0Token - lower.outerFeeGrowth0Token;* *innerFeeGrowth1Token = totalFeeGrowth1Token - lower.outerFeeGrowth1Token;* *} else {* *innerFeeGrowth0Token = lower.outerFeeGrowth0Token;* *innerFeeGrowth1Token = lower.outerFeeGrowth1Token;* *}* *innerFeeGrowth0Token -= upper.outerFeeGrowth0Token;* *innerFeeGrowth1Token -= upper.outerFeeGrowth1Token;* *} else {* *innerFeeGrowth0Token = upper.outerFeeGrowth0Token - lower.outerFeeGrowth0Token;* *innerFeeGrowth1Token = upper.outerFeeGrowth1Token - lower.outerFeeGrowth1Token;* *}* Which is in fact, from observation the same as the refactored code. However, we still always recommend following the original code instead of introducing redundant risk.
Recommendations	Consider following the exact same style as Algebra.
Comments / Resolution	Acknowledged.

PositionManagementLibrary

A new function computeAndDirectDeposit has been added. This function was not recommended by Bailsec and its use can have potential critical consequences.
This has not been audited by Bailsec. However, the client ensured that the directDeposit feature will never be used.

The PositionManagementLibrary contract is a simple library contract which is used by the HypervisorNFPM contract and handles interactions with the Algebra NFPM contract.

Specifically, the following interactions are executed:

a) Minting new tokenIds
b) Increasing liquidity towards tokenIds
c) Withdrawing from tokenIds

Privileged Functions

- none

Issue_04	Deadlines are unenforced
Severity	**Low**
Description	In the PositionManagementLibrary, we have several functions which interact with the NonfungiblePositionManager. Some of these functions include a deadline parameter, beyond which execution will revert. This is useful in the case that a transaction gets stuck in the mempool for a long time before being processed. In case the transaction gets stuck for long enough, the market may change significantly, leading to the previously desired amounts to receive no longer being desirable. In each of: mintLiquidity, increaseLiquidity, and decreaseLiquidity, we provide a deadline of block.timestamp. When the transaction is processed and the deadline is validated, block.timestamp will always be the current timestamp. As a result, the check that deadline <= block.timestamp will always succeed since it's effectively just block.timestamp <= block.timestamp.
Recommendations	We do not recommend a change since slippage parameters are mostly sufficient. However, this should be kept in mind.
Comments / Resolution	Acknowledged.

Issue_05	directDeposit = true can often be bypassed
Severity	**Low**
Description	The computeAndIncreaseLiquidity function is invoked upon a user deposit while directDeposit = true as well as during compounds. If there is currently no pair and idle tokens only single sided, a user can trivially prevent the directDeposit during his own deposit by swapping the pair to such a price which doesn't allow the addition of single sided tokenX (as example) which then results in zero liquidity and thus no liquidity addition is executed.
Recommendations	Generally speaking, we recommend simply deactivating directDeposit.
Comments / Resolution	Acknowledged.

Issue_06	Liquidity is minted for pools with deployer = address(0)
Severity	**Informational**
Description	Currently, liquidity is only added for pools with deployer = address(0), which is correct in itself as this means liquidity is only added for default pools. However, in the scenario where liquidity should be added to custom pools, it will never work.
Recommendations	We do not recommend a change. However, this should be kept in mind for future iterations.
Comments / Resolution	Acknowledged.

UniProxyETH

A slippage check for the deposit function has been incorporated which calculates min and max deposit based on the provided slippage parameter and the initial deposit0/1 parameters. This was not recommended by Bailsec and not audited

The UniProxyETH contract forms the entry contract for users to deposit into Gamma's vault architecture.

It interacts with the ClearingV3NFPM contract to ensure users can only deposit in the exact same ratio as the vault's underlying assets. It furthermore offers flexibility via switching out the ClearingV3NFPM contract in the future which allows for more or less strict deposit control.

Deposits can be made either as simply deposit via the deposit function or as a deposit via depositAndStake which then further stakes the received ERC20 tokens in the MultiFeeDistribution contract in an effort to receive rewards.

Core Invariants:

INV 1: Deposits can only be made in the current ratio of the vault's aggregated tokenX/tokenY balances

INV 2: Deposits can only be made if the contract is not paused

INV 3: actualDeposit0 can never be larger than deposit0 and actualDeposit1 can never be larger than deposit1

INV 4: cleared must be true before a deposit is being executed

Privileged Functions

- transferClearance
- transferOwnership
- pause
- rescueERC20

Issue_07	Governance Privilege: IClearing can be switched out
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For instance, the IClearing address can be switched out which then would allow bypassing the clearing check, potentially resulting in a critical vulnerability
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_08	General lack of slippage for share amount
Severity	**Medium**
Description	Currently, there is no slippage check against the received share amount, while at the same time there are multiple different scenarios where users could receive less shares than expected.
Recommendations	Consider implementing a slippage check for the share amount.
Comments / Resolution	Acknowledged, instead of a slippage check for the actual share return amount, a slippage check for deposit0/deposit1 amount has been implemented.

Issue_09	_safeTransferETH function is unused
Severity	**Informational**
Description	Functions which are unused will unnecessarily increase the contract size for no reason and will confuse third-party reviewers: *function safeTransferETH(address to, uint256 value) internal {* *(bool success,) = to.call{value: value}("");* *require(success, "ETH transfer failed");* *}*
Recommendations	Consider removing this function.
Comments / Resolution	Acknowledged.

Issue_10	WETH is unnecessarily determined
Severity	**Informational**
Description	Currently, the WETH variable is determined which then forces the fallback function to be only callably by WETH. This is redundant as there is essentially no usage for that.
Recommendations	Consider simply removing the corresponding parts.
Comments / Resolution	Acknowledged.

ClearingV3NFPM

The ClearingV3NFPM contract is used as a validation contract between the UniProxyETH and HypervisorNFPM contract. The most essential function, clearDeposit is considered upon deposits which ensures that the provided token amounts match the ratio of tokenX/tokenY in the system.

The contract owner can manually add new Hypervisor contracts where users can then deposit tokens as well as add addresses which are whitelisted and dont need to follow the ratio mandate.

Appendix: Ratio Mandate Enforcement in clearDeposit

The core principle behind clearDeposit is to ensure that any deposit of tokenX/tokenY maintains the **same ratio** as the vault's current underlying ratio of tokenX to tokenY. In other words, the function checks how many tokensX and tokensY the vault has in total (including unclaimed fees), then **forces** each deposit to mirror that ratio.

Below is a **step-by-step** explanation with numerical examples:

Example A

- Vault ratio: total0=143, total1=89 => ratio = 89/143 ≈ 0.622
- Caller proposes: deposit0=14.3, deposit1=8.9.

expectedToken1 = (14.3 * 89) / 143 = 8.9

expectedToken0 = (8.9 * 143) / 89 = 14.3

Because expectedToken1=8.9 <= deposit1=8.9, we do:

actualDeposit0=14.3, actualDeposit1=8.9

Example B

- Vault ratio: total0=200, total1=100 => ratio = 100/200=0.5
- Caller proposes: deposit0=50, deposit1=50.

expectedToken1 = (50 * 100)/200 = 25

expectedToken0 = (50 * 200)/100 = 100

expectedToken1=25 is < deposit1=50, so the code picks:

actualDeposit0=50, actualDeposit1=25

We basically scale down the user's Y deposit from 50 to 25 to maintain the 2:1 ratio in X:Y.

Core Invariants:

INV 1: actualDeposit0/actualDeposit1 must never be larger than deposit0/deposit1

INV 2: Addresses which are not whitelisted must follow the system ratio

INV 3: Deposits are not allowed if contract is paused

INV 4: Deposits can only happen to added pools

INV 5: Within ratio mandate enforcement, always the smaller deposit side is picked to enforce the ratio.

Privileged Functions

- appendList
- removeListed
- pause
- transferOwnership
- addPosition

Issue_11	Governance: Whitelisted from address can bypass mandate
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For example, any “from” address which is whitelisted can bypass the ratio mandate and thus drain the Hypervisor.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_12	total0/total1 check can be trivially bypassed
Severity	**Medium**
Description	The clearDeposit function incorporates the following check: *require(total0 > 0 && total1 > 0, "cannot determine ratio");* This check basically ensures that tokenX and tokenY are existent in the system. For example, in case there is no idle balance, it would prevent scenarios where a swap is being executed before a deposit such that the range of the pair is crossed and liquidity is only single sided. This is an important security feature to limit user flexibility and to prevent division by zero in subsequent operations. However, this check can be trivially bypassed by simply transferring dust into the contract. Furthermore, in the scenario of a zeroBurn (which will be implemented), it will almost always be bypassed b/c fees are claimed.
Recommendations	Consider simply executing a price deviation check which ensures that the price does not cross the position ranges. Optionally, it is also possible to consider tokenX/tokenY from the basePosition. This furthermore means deposits are not only prohibited in case the price is manipulated but also in case the basePosition itself is out of range, which seems like a reasonable additional safeguard.
Comments / Resolution	Partially resolved, the following condition is enforced: baseLower <= currentTick < baseUpper The goal of this check is to ensure that in fact both tokenY and tokenX are existent.

	However, in the scenario where baseLower = currentTick < baseUpper and currentPriceSqrt = baseLowerSqrt Only tokenX would be existent. A better solution would be to directly check for sqrtPrice which then ensures no side-effect from rounds during sqrtPrice to tick conversion happens.

Issue_13	Added Hypervisor contracts can never be removed
Severity	**Medium**
Description	The addPosition function allows a new HypervisorNFPM contract to be added to the system with a corresponding version. If a contract is added, it will always allow users to deposit to that contract. However, there is currently no way to remove it.
Recommendations	Consider allowing the removal of pools.
Comments / Resolution	Acknowledged.

Issue_14	Lack of slippage parameter for final received output amount during withdraw
Severity	**Medium**
Description	The withdraw function withdraws the proportional amount from the pair and from the idle balance based on the provided share amount and the totalSupply. This function exposes a minAmounts parameter which validates the received output amount from the pair withdrawals. However, for the idle output amount, there is no slippage check at all. This will become problematic in the following scenario: a) There is currency no liquidity added and all funds are sitting in the vault b) At least one of both tokens (tokenX/tokenY) is a low decimal token c) The overall share value is very large d) The user withdraws a proportional small share amount Consider the following numerical value: sharesToBurn * amountX / totalShares 100e39 * 100e6 / 1e50 = 0.1 The user will burn 100e39 shares while not receiving any output token.
Recommendations	Consider implementing a slippage check during the withdraw function which validates the final output amount for both tokens.

Comments / Resolution	Acknowledged.

Issue_15	Standard first deposit is never permitted
Severity	**Low**
Description	The clearDeposit function exposes the following check: *require(total0 > 0 && total1 > 0, "cannot determine ratio");* We have already elaborated how it is trivial to bypass this check - however, this could result in unexpected side-effects where a legitimate user attempts to create the first deposit as it will inherently revert b/c there are neither assets in the pool, nor in the vault. This can also become a problem if there is only one user left in the vault and withdraws all shares which then renders new deposits impossible.
Recommendations	Consider documenting this behavior.
Comments / Resolution	Acknowledged.

Issue_16	Division by zero DoS if fee is set to zero
Severity	**Low**
Description	The setFee function allows the owner to set an arbitrary value for the fee. If the fee is ever set to 0 either intentionally or accidentally, the zeroBurn function will always revert due to a division by zero error.
Recommendations	Consider validating the setFee function accordingly.
Comments / Resolution	Resolved.

Issue_17	Protocol fee disregarded within getDepositAmount
Severity	**Low**
Description	Within the HypervisorNFPM contract, we already raised an issue which allowed us to drain the contract due to the lack of protocol fee incorporation when considering the fee for the ratio mandate check. This issue will be fixed by simply calling zeroBurn prior to the ratio mandate enforcement. However, it has to be noted that the getDepositAmount function will still use the manual fee calculation methodology, which should then incorporate the protocol fee to allow for correct view-only purposes.
Recommendations	This function can be simply removed since it isn’t used anymore due to enforced zeroBurn within the clearing process.
Comments / Resolution	Resolved.

Issue_18	Redundant approval
Severity	**Informational**
Description	During the addPosition function, allowance is granted to the "pos" address. This is redundant as the contract has never actually any tokens to spend.
Recommendations	Consider removing this approval.
Comments / Resolution	Acknowledged.

Issue_19	Unused variables
Severity	**Informational**
Description	Variables which are unused will unnecessarily increase the contract size for no reason and will confuse third-party reviewers: *uint256 public constant PRECISION = 1e36;* *uint256 internal constant Q128 = 1 << 128;*
Recommendations	Consider removing these variables
Comments / Resolution	Acknowledged.

HypervisorNFPM

Within the deposit function, the directDeposit case has been refactored. This was not recommended by Bailsec and was not audited. However, since the team ensured that directDeposit always remains false, this change will never find its way being called during the production (unless due to a misconfiguration)

The HypervisorNFPM contract is an Algebra position manager contract that allows users to mint shares by providing tokenX/tokenY which is then used to mint an Algebra liquidity position.

All deposited funds are aggregated and follow a share calculation principle, similar to that from ERC4626 vaults but with incorporation of the position balances and denominated in tokenY. Whenever a new deposit is executed, the amount of shares which is received based on the provided tokenX/tokenY amount and the overall amount of tokenX/tokenY in the system, which is calculated as follows:

> ((depositAmountX * price) + depositAmountY) * totalSupply / ((AUMAmountX * price) + AUMAmountY)

A depositor will thus receive the proportional amount of shares based on the provided value, the totalSupply and the existing value in the vault.

Once a withdrawal is executed, the user will simply receive the proportional amount of tokens based on the provided share amount and the totalSupply. This amount will be proportionally withdrawn from the existing liquidity and the idle balance.

Shares are denominated as ERC20 token and can then be deposited into the MultiFeeDistribution contract in an effort to farm different reward tokens. This is tied to the fact that the minted tokenIds will be used to enter Algebra's farming module in order to farm rewards.

The management of the vault is solely handled by governance via the following functions:

- rebalance
- compound
- decreaseLiquidity
- (mintLiquidity)

Furthermore, the contract exposes a directDeposit mode where the deposit immediately adds liquidity upon the deposit call. This should however remain disabled as it allows for MEV.
The contract applies a performance fee which is 20% by default but can be set up to 100% and is taken from the accrued swap fees.

Appendix: FeeGrowth calculation

In Algebra, pools accumulate fees during swaps. If a swap from X -> Y is executed, the pool accumulates Y, if the swap is vice-versa, the pool accumulates X. Since v3 usually has different positions over different ranges, it is not possible to simply accrue these fees over the full range. Instead, fees must be accrued by these positions which are currently in range.

To accommodate that, a totalFeeGrowthToken0/1 variable was implemented for tokenX/tokenY. This variable tracks the fee per liquidity, scaled by 1x128.

> totalFeeGrowthToken += fees * (1<<128) / liquidity

Example:

A swap has happened and accrued 1e18 fees in tokenX while the active liquidity is 1100e18:

totalFeeGrowthToken += 1e18 * (1<<128) / 1100e18
totalFeeGrowthToken = 3.039e35

Now another swap happens which accrues 10e18 fees in tokenX while the activeLiquidity is 1000e18

totalFeeGrowthToken = 3.039e35
3.039e35 += 10e18 (*1<<128) / 1000e18

3.039e35 += 3.402e36
totalFeeGrowthToken = 3.705e36

Now since we are aware that totalFeeGrowthToken aggregates the fee per unit of liquidity, the following calculation allows us to determine how much fees a specific position has accrued:

> (innerFeeGrowthToken - innerFeeGrowthTokenLast) * liquidity

Now as we can see, the variable which is used includes “inner”. We need to understand that, as already explained, each v3 position incorporates its very own storage mechanism.

Here are a few keypoints:

a) Whenever a position is newly added, it consists of upper and lower tick

   i) If the tick had liquidity before, outerFeeGrowth is not adjusted
   ii) If the tick had no liquidity before and the tick is <= currentTick, outerFeeGrowth is set to totalFeeGrowth
   iii) If the tick had no liquidity before and the tick is >= currentTick, outerFeeGrowth is not set

b) Whenever a position is newly added, feeGrowthInside will be set to the current feeGrowthInside for the range

c) Whenever a swap happens and a tick for a position is crossed, the outerFeeGrowth of the tick will be set to [feeGrowthTotal - outerFeeGrowth]

d) outerFeeGrowth always corresponds to the feeGrowth outside of the position to either the left or the right side (depending on lower/upper)

e) Depending on the currentTick and the range, the following calculations are used to calculate feeGrowthInside:

   i) **currentTick < lowerTick** (lowerTick was crossed; X > Y):
   > innerFeeGrowth = lower.outerFeeGrowth - upper.outerFeeGrowth

ii) **currentTick >= upperTick** (upperTick was crossed; Y > X):
> innerFeeGrowth = upper.outerFeeGrowth - lower.outerFeeGrowth

iii) **lower < current = upperTick** (upperTick was crossed; Y > X)
> innerFeeGrowth = upper.outerFeeGrowth - lower.outerFeeGrowth

iv) **lowerTick < currentTick < upperTick** (no tick was crossed):
> innerFeeGrowth = totalFeeGrowth - lower.outerFeeGrowth - upper.outerFeeGrowth

v) **lowerTick = currentTick** < upperTick (lowerTick was not crossed):
> innerFeeGrowth = totalFeeGrowth - lower.outerFeeGrowth - upper.outerFeeGrowth

f) The last scenario: lowerTick = currentTick < upperTick is a special scenario, as one would assume the calculation must be as follows:

> innerFeeGrowth = lower.outerFeeGrowth - upper.outerFeeGrowth.

The reason is because one would assume that is that the scenario:

lowerTick = currentTick

resulted in a cross of the tick and thus lower.outerFeeGrowth = [feeGrowthTotal - outerFeeGrowth]. This is the only thing that would make sense.
However, in fact, in that scenario, the tick has not crossed. This is because of the edge-case within SwapCalculation._calculateSwap for the zeroToOne scenario where currentTick = nextTick - 1. Consider the following example: lowerTick = 0; currentTick = 10. X -> Y swap is being executed nextTick = 1, which means the lowerTick is not crossed. However, currentTick will be set to nextTick - 1 which means now that the condition: currentTick = lowerTick is met without the tick being crossed and thus tick = 0 still remains the same outerFeeGrowth as before and the calculation must be executed as if the currentTick is inside the range.

g) The calculation explicitly assumes underflows/overflows which is derived from the "unchecked" setting. There are specific scenarios where this is necessary to keep the

integrity of the fee calculation mechanism. For example, in the following situation, this becomes relevant: https://github.com/Uniswap/v3-core/issues/573.
More specifically, the following conditions must be true:
a) currentTick is lower than lowerTick
b) lowerTick of the position was not yet initialized
c) upperTick was initialized. If now a position is added based on these properties, innerFeeGrowth will be calculated as follows:
> innerFeeGrowth = lower.outerFeeGrowth - upper.outerFeeGrowth
This means it becomes negative.

Example:

a) currentTick = 0
b) lowerTick = 1
c) upperTick = 10
d) lowerTick.outerFeeGrowth = 0
e) upperTick.outerFeeGrowth = 100
f) totalFeeGrowth = 100

This will result in: innerFeeGrowth = 0 - 100

Appendix: Farming Module

Whenever new tokenIds are minted upon rebalance or via mintLiquidity, these tokenIds will immediately enter Algebra's Farming module via the approveForFarming call on the NFPM followed by the enterFarming call on the FarmingCenter.

Once a tokenId enters the farming module, it will receive a base token and eventually a bonus token as reward which is related to the setting for the incentiveKey. These rewards are then forwarded to the MultiFeeDistribution contract for distribution among stakers of the HypervisorNFPM ERC20 token.

Entering the Farming module requires two transactions:

a) Approving the tokenId on the NFPM to enter the FarmingCenter
b) Enter farming for a specific IncentiveKey on the FarmingCenter

Each incentiveKey has the following properties and represents a unique farming campaign:

- rewardToken
- bonusRewardToken
- pool
- nonce

The corresponding incentiveId is just the keccak256 of the incentiveKey struct.

Core Invariants:

INV 1: getUnclaimedFees must represent balance of tokenX/Y which is currently claimable, deducted by protocol fees.

INV 2: _collectAndClaim is always tied to incentive where tokenId has entered

INV 3: During withdraw, user will receive the proportional tokenX/tokenY amount from pair and idle balances for burned share amount to total share amount

INV 4: Deposits must only be allowed via the UniProxyETH contract

INV 5: Withdraw must only be allowed for from = msg.sender

INV 6: _approveAndEnterFarming must always fetch current key from incentiveMaker

INV 7: zeroBurn must claim all due fees which are tied to a tokenId

INV 8: zeroBurn must be triggered before deposits and withdrawals

INV 9: _collectAndClaim must be triggered before deposits and withdrawals

INV 10: Rewards can still be collected and claimed even if the incentiveKey is deactivated

INV 11: A tokenId must always be tied to the corresponding incentiveId once it has entered farming

INV 12: getTotalAmounts must always return all tokenX/tokenY in the system in case it is consulted by _deposit

INV 13: After zeroBurn, the tokensOwed0/1 values for a tokenId within the NFPM must always be zero

INV 14: _approveAndEnterFarming must always consult the incentiveManager for a potentially new incentiveKey

INV 15: During a deposit, getTotalAmounts must be considered before the depositor's asset are transferred in.

Privileged Functions

- transferOwnership
- renounceOwnership
- setWhitelist
- removeWhitelisted
- setFee
- setTickSpacing
- toggleDirectDeposit
- setProtocolAddresses
- updateIncentiveMaker
- setNftIds
- decreaseLiquidity
- mintLiquidity
- rebalance
- compound
- transferReceiver

Issue_20	Governance Issue: Full control over managed funds
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For instance, there are several scenarios for a compromised owner to withdraw all funds from the contract/hold funds hostage: - via malicious rebalancing - via the mintLiquidity function - via changing tokenIds - manipulating whitelistedAddress - ... There are multiple other issues which can result in a loss of funds if wrongly configured.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_21	Contract can be drained via sophisticated mandate bypass mechanism
Severity	**High**
Description	Currently, the main safeguard for the contract against price manipulation, followed by single sided deposits is the enforcement of the ratio mandate. This is done via the getTotalAmountsPlusFees function which includes the "to be claimed" fees into the calculation. In itself, this is correct. However, a protocol fee is taken upon zeroBurn which then results in a deviation from the actual ratio of the vault compared to the allowed ratio for deposits. This deviation can be abused to drain the pair: **Status Quo** > price = 1 > [0.909; 1.1] > poolX = 51.19e18; poolY = 51.19e18 > liquidity = 1100e18 > fee = 50% > totalSupply = 102.38 **Attacker swaps towards right side:** > price = 1000000000000 > [0.909; 1.1] > poolX = 0; poolY = 104.8 > unclaimedX = 1; unclaimedY = 1 **Attacker deposits, bypassing the mandate:** > poolX = 0; poolY = 104.8 > unclaimedX = 1; unclaimedY = 1

	> depositX = 1; depositY = 105.8 > ratio is accepted **Calculate how much shares attacker receives:** > fees are claimed and 50% protocol fee taken > poolX = 0; poolY = 104.8 > idleX = 0.5; idleY = 0.5 (these are the fees now) > totalX = 0.5; totally = 105.3 > poolValue = (0.5 * 1000000000000) + 105.3 > poolValue = 500000000105 > depositValue = (1 * 1000000000000) + 105.8 > depositValue = 1000000000105 calculate shares for attacker > 1000000000105 * 100 / 500000000105 > 200 **Attacker withdraws 200 shares:** > poolX = 0; poolY = 104.8 > idleX = 1.5; idleY = 105.3 > totalX = 1.5; totalY = 210.1 > receivedX = 200 * 1.5 / 300 > receivedX = 1 > receivedY = 200 * 210.1 / 300 > receivedY = 140 Attacker deposits 1x; 104.8y Attacker received 1x; 140y An attacker can afterwards execute a swap in the range of the baseTokenId or limitTokenId such that fees in tokenX will be accumulated to repeat this attack. This can all be done in one

	transaction until the protocol is emptied. This attack is so sophisticated that upon the first few hours after such an exploit, the root-cause still wouldn't be known.
Recommendations	Consider simply collecting all fees before the ratio calculation.
Comments / Resolution	Resolved.

Issue_22	Edge-case during directDeposit state allows for thefting from a vault
Severity	**High**
Description	Throughout the report, we already mentioned multiple issues with the setting of directDeposit = true, as it essentially allows users to trigger the liquidity addition. It is an important invariant to not allow users to add liquidity, especially when there is no price deviation check. This is a known issue for V3 vaults because it can result in users triggering the liquidity addition, which results in the liquidity addition happening at the wrong price, even with the correct tick settings. If for example, the base position consists of [0,909; 1.1] while at the same time the price is 1, it will naturally add liquidity 50% in tokenY and 50% in tokenX, geometrically around the price. This is a desired state. If however a user swaps the price to the right side, let's say to 1.1, the newly added liquidity is only in tokenY, while at the same time, the real price still is 1, which means tokenY is being added to a range where tokenX is worth more than the real price, at [1; 1.1], which thus allows the user to swap tokenX to tokenY for an inflated value and thus theft a share from the vault.

	There are now multiple different scenarios where this root-cause can be abused, just to name a few: a) Inspecting a pool with an idle balance and directDeposit = true b) Inspecting a pool with unclaimed fees which then flow in the idle balance and directDeposit = true c) Multiple other scenarios with edge-cases and directDeposit = true (incoming user deposits/idle limit position/...) Since the Gamma team agreed to never set directDeposit to true, we do not spend further time searching for edge-cases which allow users to trigger adding liquidity at an undesired price.
Recommendations	Consider never allowing for directDeposit = true.
Comments / Resolution	Acknowledged, the client will never set directDeposit to true.

Issue_23	Ratio enforcement can be abused to frontrun depositors resulting in a loss of tokens for depositors
Severity	**High**
Description	Currently, there is no deviation check for the pair price to the real price. This can be abused by a malicious user to frontrun a deposit, by executing a swap on the pair which increases or decreases the price. The root-cause is that the execution of a swap will result in the user receiving the output token at a higher price than the current price (as the curve increases during purchases) which then has the effect that the user must provide more of the input token. This will result in the pair "accruing value" which then together with the ratio mandate enforcement results in a nominal loss for the depositor as the share value calculation results in less shares compared to if there would be no swap before the deposit. This issue is more amplified in case directDeposit is turned on but still exists in case directDeposit is turned off. In case directDeposit is turned on, the swapper will even profit due to the increased liquidity density (similar as in the above described issue). In case directDeposit is turned off, all vault participants will share the profited amount. However, the directDeposit scenario would require minIn parameters to be loose. **INCREASE PRICE; directDeposit = false** > pool: 51.19 tokenY; 51.19 tokenX; price = 1; [0.90909; 1.1]; liquidity = 1100e18 > idle = 100 tokenX > totalX = 151.19e18 > totalY = 51.19e18 > totalSupply = 204.8e18 **Attacker swaps Y in and move price to 1.1**

> pool = 104.8e18 tokenY
> idle = 100e18 tokenX

Victim deposits

> user deposits 100e18 tokenX ; 104.8e18 tokenY
> userValue = (amountX * price) + amountY
> (100e18 * 10) + 104.8e18
> 1104.8e18
> poolValue = (amountX * price) + amountY
> (100e18 * 10) + 104.8e18
> 1104.8e18
> 1104.8e18 * 204.8e18 / 1104.8e18
> user gets 204.8e18
> totalSupply = 409.6e18

Attacker swaps x back and moves price to 1

> pool: 51.19 tokenY; 51.19 tokenX; price = 1; [0.90909; 1.1]; liquidity = 1100e18
> idle = 200e18 tokenX; 104.8e18 tokenY
> totalX = 251.19e18
> totalY = 154.78e18
> user withdraws 204.8e18 shares
> from pool:
    > amountX = 204.8e18 * 51.19e18 / 409.6e18
    > amountX = 25.595e18
    > amountY = 25.595e18
> from idle:
    > amountX = 204.8e18 * 200e18 / 409.6e18
    > amountX = 100e18
    > amountY = 204.8e18 * 104.8e18 / 409.6e18
    > amountY = 52.4e18
> amountX = 125.595e18

	> amountY = 77.995e18 > total = 203.59 > user effectively lost tokens; 1,21 tokens lost > these tokens go towards all previous depositors; 1.18 tokens won (precision deviation 0.03) ## **INCREASE PRICE; directDeposit = true** > pool: 51.9e18 tokenY; 51.9e18 tokenX; price = 1; [0.909; 1.1]; liquidity = 1100e18 > totalSupply = 104.8e18 **Attacker swaps Y in** > attacker provides 53.68 tokenY; receives 51.19 tokenX > pool = 104.8e18 tokenY **Victim deposits (liquidity is added)** > 104.8e18 tokenY > receive 50% > receive 104.8e18 shares > totalSupply = 209.6e18 > pool: tokenY = 209.6e18 > liquidity = 2200e18 **Attacker swaps X in (reverse swap)** > user provides: 102.38 tokenX > user receives: 107.37 tokenY > pool: tokenX = 102.38; tokenY = 102.38

	Victim withdraws 100% of his shares (50% of all)
	
	> from pool:
	  > 51.19 tokenX
	  > 51.19 tokenY
	
	> total = 102.38e18
	> victim lost tokens because the pre-deposit swap expected the victim to deposit more tokens than the pair in reality has (since the swap accrued some funds), the leftover tokens will remain there for the swapper back
	> back runner profited

Recommendations	Consider implementing a price deviation check.
Comments / Resolution	Partially resolved, a slippage mechanism within the UniProxyETH's deposit function has been implemented. A price deviation check is still missing.

Issue_24	minIn checks can be bypassed via different routes
Severity	**High**
Description	minIn checks within the deposit function are usually sufficient against price manipulations as the user can carefully determine to which ratio his deposit should be added (which is inherently based on the price). However, there are several routes to bypass this: a) If there is an idle balance in the vault, minIn checks will not only relate to the user's deposit but also incorporate the idle balance during the liquidity addition. This makes this safeguard essentially void. b) If the deposit is stuck in the mempool and in the meantime directDeposit is set to false, another deposit happens and directDeposit is set to true again c) Any accrued fees will manipulate the balance which is added to the liquidity and thus result in a similar issue as a) d) If there is only one shareholder, he can trivially frontrun a incoming deposit by transferring funds directly to the vault (in the expected ratio such that the user's deposit does not revert) these funds will then make the minIn check but at the same time go towards the shareholder such that no loss was experienced The issue behind the manipulation of minIn has idea that a deposit is frontrun with a swap such that the depositor adds liquidity to the range for a different than expected price and experiences a loss during the swap back (see second example in "Ratio enforcement can be abused to frontrun depositors resulting in a loss of tokens for depositors"
Recommendations	Consider incorporating a price deviation check.

Comments / Resolution	Acknowledged, no price deviation check has been implemented. However, the client ensured that directDeposit will never be set to true.

Issue_25	Share inflation attack possibility for first depositor
Severity	**High**
Description	The vault calculates the received share amount by simply following the rule of three: > ((depositAmountX * price) + depositAmountY) * totalSupply / ((AUMAmountX * price) + AUMAmountY) The first depositor can deposit a small amount of tokenY, followed by a large donation. This will then result in a large divisor where the share amount for all subsequent depositors rounds either down or completely results in zero.
Recommendations	Consider including a price deviation check within the ClearingV3 contract as well as a minSharesOut parameter within the UniProxyETH contract.
Comments / Resolution	Acknowledged, the client will always be the first depositor.

Issue_26	First depositor can break the vault by inflating the denomination of shares
Severity	**High**
Description	The first depositor will always receive exactly the share amount which is corresponding to depositInY: *shares = deposit1 + FullMath.mulDiv(deposit0, price, PRECISION);* This can be abused by swapping the price to the right side followed by a small deposit of tokenX such that the share amount becomes near uint256.max. Subsequently, all further deposit attempts by other users revert due to an overflow and the vault is essentially rendered unusable.
Recommendations	Consider including a price deviation check within the ClearingV3 contract.
Comments / Resolution	Acknowledged, the client will always be the first depositor.

Issue_27	First depositor can break the reward calculation within the MultiFeeDistribution contract
Severity	**High**
Description	As already explained within the issue above: The first depositor will always receive exactly the share amount which is corresponding to depositInY: *shares = deposit1 + FullMath.mulDiv(deposit0, price, PRECISION);* This can be abused by swapping the price to the right side followed by a small deposit of tokenX such that the share amount becomes large. In this scenario, the target of the attack is **not the revert of all subsequent deposits** but just the fact that the totalStakes amount within the MultiFeeDistribution contract becomes inflated, which then always results in zero rewards due to the large divisor: *if (totalStakes > 0) {* *uint256 additionalRewards = currentBalance + unclaimedAmount - r.amount;* *newRewardPerToken += additionalRewards * 1e50 / **totalStakes**;* *}* Furthermore, it has to be noted that generally a large totalSupply offers more space for errors as users can mint proportionally smaller liquidity which results in zero output during withdrawals but still burns a user's shares if the liquidity is zero.: *PositionValue.liquidityForShares(self, baseNftId, shares, totalSupply),*
Recommendations	Consider including a price deviation check within the ClearingV3 contract.

Comments / Resolution	Acknowledged, the client will always be the first depositor.

Issue_28	Raw MEV possibility via compound/rebalance
Severity	**High**
Description	The contract is inherently vulnerable to MEV attacks during various different functions such as adding liquidity, rebalancing, compounding etc. We have already described how it is possible to frontrun a liquidity addition in an effort to profit from the increased liquidity density during the swap-back and thus essentially steal a part of the added funds. To a degree, these attacks can be mitigated via the minIn/minOut parameters. However, this only works if in fact the overall funds in the system remain consistent b/w transaction initiation and transaction execution. A sophisticated attacker can simply frontrun a compound/rebalance and execute a normal deposit which increases the idle balance of the vault. This will have the effect that compound/rebalance will have higher nominal tokenX/tokenY amount to add which then in turn falsifies the intention of minIn, as the input amounts will now be inherently higher than anticipated. (we have already explained the insufficient safety of minIn during another issue, however, since this issue inherently exposes a risk which can be exploited to steal a part of the system value, we considered it as mandatory to create a separate issue for that scenario). This now allows for swapping the pair towards one side while still ensuring that minIn slippage parameters pass, allowing a malicious user to drain a share of the system's funds. It is essentially possible whenever rebalance / compound is called.

	This is specifically an issue for compound, as during rebalance it may be possible that amountOut prevents the decreaseLiquidity call. However, it is important to understand that any swap will be in favor of the pair which means that amountOut often may be loosely set (b/c actually there is no possible loss to occur during a withdrawal, only profit)
Recommendations	Consider implementing a targetSqrtPrice parameter for all governance functions, especially for rebalance and compound (as these rely on balanceOf instead of a custom amount input parameter).
Comments / Resolution	Acknowledged, the client will use strict slippage parameters.

Issue_29	Overflow revert possibility for pairs with very large price during price calculation
Severity	**Medium**
Description	The price is currently calculated as follows: *uint256 price = FullMath.mulDiv(uint256(sqrtPrice) * uint256(sqrtPrice), PRECISION, 2**(96 * 2));* This will not work if the price is very high and sqrtPrice*sqrtPrice results in an overflow. More specifically, if tokenX is a token with 6 decimals and tokenY is a token with 21 decimals, a value of 1 will already result in a price of 1e15 which translates into 2505414483750479286512002635546469086 If now the price is higher than 1, the possibility for overflow is given which results in a revert of the deposit call.
Recommendations	Consider using the following calculation for the price:

	FullMath.mulDiv(uint256(sqrtPrice) * 1e18, uint256(sqrtPrice) * 1e18, 2**(96 * 2));
Comments / Resolution	Resolved.

Issue_30	Protocol fee change will be applied in hindsight
Severity	**Medium**
Description	The fee variable determines how much of the accrued swap fees go towards the protocol. A change of this parameter will not only become effective at the time of the change but will also be applied in hindsight if swap fees are not claimed beforehand.
Recommendations	Consider claiming swap fees before the fee is updated.
Comments / Resolution	Acknowledged

Issue_31	Change of tokenId will not change range
Severity	**Medium**
Description	The contract incorporates a function which allows for the change of tokenId's. This could be specifically useful in scenarios where the tokenId is burned via a normal withdrawal but not reset and also if a tokenId has been minted via mintLiquidity. Even though these scenarios are rare, it can theoretically still happen. A small problem with such a change is the fact that the baseLower/upper; limitLower/upper variables are not adjusted which will then result in potentially incorrect liquidity additions during compound.
Recommendations	Consider adjusting the range as well.
Comments / Resolution	Acknowledged

Issue_32	Sophisticated scenario allows for stealing part of reward tokens
Severity	**Medium**
Description	The contract exposes a mintLiquidity function which in itself is less likely to be used but still possible. In a scenario where such a new tokenId is minted, it will automatically enter the farming module and accrues rewards. However, these rewards are never actually collected and claimed for the simple reason that collect only happens for base and limit tokenId. This opens the window for a front-running attack where a user deposits a large amount into the MultiFeeDistribution contract just before the tokenId is assigned to base/limit in order to then collect and claim rewards. Once that has happened, the user can immediately withdraw again and as a result of that action stole a majority of the rewards which have been accrued by this tokenId. An even more sophisticated method is that a user directly interacts via NFPM.increaseLiquidity (permissionless function), this will then collect rewards for that tokenId and assign them towards the HypervisorNFPM which can then be swiftly claimed via a simple deposit call. Following this methodology, a user does not even need to rely on the setting of the tokenId to base/limit but can immediately steal rewards.
Recommendations	Consider keeping that scenario in mind. Ideally, the mintLiquidity function is never used.
Comments / Resolution	Acknowledged

Issue_33	Contract is not pausable
Severity	**Medium**
Description	The contract inherits the Pausable contract and applies the whenNotPaused modifier on various functionalities: a) deposit b) depositAndStake c) withdraw However, it fails to actually expose a pause/unpause function which means it is never possible to trigger the paused state.
Recommendations	Consider implementing functionality which allows for pausing the contract in emergency situations. We recommend only adding this in explicit emergency situations. Additional time must be allocated to ensure emergency withdrawing does not mess up with reward integrity.
Comments / Resolution	Resolved.

Issue_34	minAmounts during withdraw is void if directDeposit happened before the execution
Severity	**Medium**
Description	The minAmounts parameter within the withdraw function determines how much of tokenX/tokenY a user wants to withdraw from the baseTokenId and limitTokenId based on the provided share amount and the tokenIds liquidity. In the scenario where a directDeposit is executed exactly before the execution of the withdraw function, it will increase the liquidity of the pair, resulting in a manipulated output parameter (because the idle balance now was transferred to the pair) which then makes the minAmounts parameter void.
Recommendations	Consider implementing a price deviation check.
Comments / Resolution	Acknowledged.

Issue_35	Contract does not automatically enter farming if currentIncentiveKey.pool = address(0) due to erroneous early return
Severity	**Medium**
Description	Whenever the contract is deployed, currentIncentiveKey is potentially set as follows: *currentIncentiveKey = incentiveMaker.poolToKey(_pool);* In a scenario where the contract is deployed while there was not yet an incentiveKey for the corresponding pool but at a later point in time the incentiveKey for this pool is set within the incentiveMaker, there is no way to enter farming without updating the whole incentiveMaker address via updateIncentiverMaker. The reason for that is that the _approveAndEnterFarming function returns early if there is no currentIncentiveKey: *// Early return if pool address is zero* *if (address(currentIncentiveKey.pool) == address(0)) {* *return;* *}* Indeed, currentIncentiveKey will be adjusted if it is different from the old one: *// Get the key from incentive maker* *IncentiveKey memory key = incentiveMaker.poolToKey(address(pool));* *// Check if there's an existing deposit and if its incentive is deactivated* *bytes32 incentiveId = keccak256(abi.encode(key));* *if (incentiveId != bytes32(0) &&*

	farmingCenter.eternalFarming().isIncentiveDeactivated(incentiveId)) { *return;* *}* *if (* *address(key.rewardToken) != address(currentIncentiveKey.rewardToken) ||* *address(key.bonusRewardToken) != address(currentIncentiveKey.bonusRewardToken) ||* *address(key.pool) != address(currentIncentiveKey.pool) ||* *key.nonce != currentIncentiveKey.nonce) {* *currentIncentiveKey = key;* *}* However, due to the early return, this check is never actually executed.
Recommendations	An ideal solution would be to refactor the _approveAndEnterFarming function to check this state before the early return. However, given the significant amount of existing issues and the likelihood that a significant amount of changes will be applied to the codebase, we do not recommend applying this change as well. Instead, we recommend simply calling the updateIncentiveMaker function with the same incentiveMaker address as before, if that scenario occurs..
Comments / Resolution	Acknowledged.

Issue_36	IncentiveId will never become bytes(0)
Severity	**Medium**
Description	The following check is exposed (after the early return) within the _approveAndEnterFarming function: *if (incentiveId != bytes32(0) && farmingCenter.eternalFarming().isIncentiveDeactivated(incentiveId)) {* *return;* *}* This has the core assumption that, if incentiveId == bytes(0), the function returns and this is not related to currentIncentiveKey but rather to the return value from the external call: *IncentiveKey memory key = incentiveMaker.poolToKey(address(pool));* However, this will never be the case because even with the default values from the incentiveKey, it will not result in bytes(0). This will then always revert in such a scenario, resulting in a DoS, until the currentIncentiveKey.pool is manually set to address(0).
Recommendations	Consider manually setting currentIncentiveKey.pool to address(0) in such a case.
Comments / Resolution	Acknowledged.

Issue_37	Temporary DoS if liquidity is fully withdrawn
Severity	**Low**
Description	The withdrawal of liquidity will never reset tokenIds in case the liquidity afterwards is zero. A problem in that scenario is that all subsequent deposits as well as rebalances will revert due to the fact the zeroBurn always attempts to collect from the unreset tokenId's which then reverts within the NFPM: *function collect(* *CollectParams calldata params* *) external payable override* ***isAuthorizedForToken****(params.tokenId) returns (uint256 amount0, uint256 amount1) {*
Recommendations	Consider manually setting the emptied tokenId to zero in such a scenario.
Comments / Resolution	Acknowledged.

Issue_38	Blacklisting of MultiFeeDistribution for rewardToken or bonusRewardToken will result in a revert of _collectAndClaim
Severity	**Low**
Description	The _collectAndClaim function claims rewards towards the receiver address which is generally considered as the MultiFeeDistribution contract. If that contract is blacklisted for any of both reward tokens, this call will always revert.
Recommendations	Consider monitoring for such a scenario and manually change the receiver.
Comments / Resolution	Acknowledged.

Issue_39	Incentive switch can result in temporarily unaccrued rewards
Severity	**Low**
Description	The incentiveId can be switched out via the updateIncentiveMaker function. This allows new minted tokenIds to enter the new incentive. If however, no rebalance is executed immediately afterwards, both tokenIds will remain assigned to the old incentiveId which eventually does no longer accrue rewards.
Recommendations	Consider rebalancing after the incentiveId has been changed.
Comments / Resolution	Acknowledged.

Issue_40	Change of farmingCenter within EternalFarming can result in DoS of protocol
Severity	**Low**
Description	During every interaction with the protocol, the _collectAndClaim function is invoked which collects and claims rewards from Algebra's farming module. Special attention must be given to the collectRewards function as this can only be called by the FarmingCenter. If now the farmingCenter variable within the EternalFarming contract is changed, this call will essentially revert and DoS all protocol functionalities. The following notes have to be added: a) The farmingCenter variable can be changed within the vault contract. However, that does not guarantee that the new farmingCenter is compatible b) In such a scenario, it can also result in issues with the NFPM, which however lies not within our control.
Recommendations	Consider simply decreasing liquidity via the decreaseLiquidity function (in case it still works via NFPM) in such a scenario and setting the tokenIds to zero.
Comments / Resolution	Acknowledged.

Issue_41	User can always empty active liquidity if activeDeposit = false at no cost
Severity	**Low**
Description	Currently, in the scenario where activeDeposit = false, a user's deposit will not be added as liquidity towards the pair but rather stays as idle balance until governance compounds or rebalances. However, the withdrawal proportionally withdraws from the pool and from the idle balance. This can be abused to "grief down the liquidity". The root-cause of this issue is that a withdrawal is always proportional on the pair and the idle balance based on the provided amount of shares and the total supply.
Recommendations	Consider keeping this issue in mind, turning activeDeposits on will introduce several issues.
Comments / Resolution	Acknowledged.

Issue_42	Overall restrictive check is pointless
Severity	**Low**
Description	Currently, the zeroBurn function first collects all fees and then potentially transfers out any protocol fee: *if (owed0/fee > 0 && token0.balanceOf(address(this)) > 0) token0.safeTransfer(feeRecipient, owed0/fee);* There are two notes about the balanceOf check: a) Inherently (if there are no edge-cases which we missed), the balanceOf check is redundant as post-claim the contract will always have the "owed0" amount. b) Additionally, it is only checked that balanceOf > 0 instead that balanceOf >= owed/fee If there is ever any edge-case which results in balanceOf < owed/fee, the balanceOf check becomes completely redundant and the transfer would result in a DoS.
Recommendations	Consider further inspecting if there are any potential side effects. Moreover we recommend executing a balanceOf >= owed/fee check.
Comments / Resolution	Acknowledged.

Issue_43	directDeposit not executed if totalSupply = 0
Severity	**Informational**
Description	Currently, a directDeposit is only executed in case totalSupply != 0: *uint256 total = totalSupply();* *if (total != 0) {* *uint256 pool0PricedInToken1 = FullMath.mulDiv(pool0, price, PRECISION);* *shares = FullMath.mulDiv(shares, total, (pool0PricedInToken1 + pool1));* *if (directDeposit) {* *// Check baseNft and potential liquidity* *if (baseNftId != 0) {* *nonfungiblePositionManager.computeAndIncreaseLiquidity(* *pool,* *baseNftId,* *baseLower,* *baseUpper,* *token0.balanceOf(address(this)),* *token1.balanceOf(address(this)),* *inMin[0],* *inMin[1]* *);* *}* Most of the time, this limitation makes total sense. However, if the contract owner beforehand manually transferred funds inside and called mintLiquidity, there would be a valid reason to execute direct deposits even if totalSupply = 0.
Recommendations	Since we anyways, always recommend to never set directDeposit = true, we recommend acknowledging this issue.

Comments / Resolution	Acknowledged.

Issue_44	Unused import(s)
Severity	**Informational**
Description	The contract contains one or more imports which are completely unused. This will increase contract size for no reason and can confuse third-party reviewers: *import "@cryptoalgebra/integral-farming/contracts/interfaces/IAlgebraEternalFarming.sol";* *import "@cryptoalgebra/integral-farming/contracts/interfaces/IAlgebraEternalVirtualPool.sol";*
Recommendations	Consider removing any unused imports.
Comments / Resolution	Resolved.

Issue_45	Unused variable(s)
Severity	**Informational**
Description	Variables which are unused will unnecessarily increase the contract size for no reason and will confuse third-party reviewers: *uint256 internal constant Q128 = 1 << 128;*
Recommendations	Consider removing any unused variables.
Comments / Resolution	Resolved.

Farming Module

RewardCalculations

The RewardCalculations contract is a helper library for the MultiFeeDistribution contract and is leveraged within the view-only totalUnclaimedRewards and getLastSecondRewards functions. It essentially mimics the reward calculation for an incentiveId within the EternalFarming contract.

Core Invariants:

INV 1: The getVirtualFeeGrowthInside function must mimic the calculation from Algebra

INV 2: totalRewardGrowth must be updated before innerFeeGrowth is calculated

Privileged Functions

- none

Issue_46	Lack of underflow possibility within getRewardsForPosition and getVirtualFeeGrowthInside
Severity	**High**
Description	Currently, there is no unchecked block within both aforementioned functions which allows for the correct calculation of the fee. Instead, it will not allow for underflow and thus the fee calculation can become flawed / revert in that specific edge-case as described within the appendix.
Recommendations	Consider following 1:1 Algebra's implementation
Comments / Resolution	Failed resolution, while an unchecked block was added within the getVirtualFeeGrowthInside function, the getRewardsForPosition and getLastSecondRewards functions still follow the wrong pattern. The correct way to do it can be found within Positions._recalculatePosition.

Issue_47	getFeeGrowthInside is not 1:1 taken from Algebra
Severity	**Informational**
Description	Algebra's feeGrowthInside calculation is as follows: *if (currentTick < topTick) {* *if (currentTick >= bottomTick) {* *innerFeeGrowth0Token = totalFeeGrowth0Token - lower.outerFeeGrowth0Token;* *innerFeeGrowth1Token = totalFeeGrowth1Token - lower.outerFeeGrowth1Token;* *} else {* *innerFeeGrowth0Token = lower.outerFeeGrowth0Token;* *innerFeeGrowth1Token = lower.outerFeeGrowth1Token;* *}* *innerFeeGrowth0Token -= upper.outerFeeGrowth0Token;* *innerFeeGrowth1Token -= upper.outerFeeGrowth1Token;* *} else {* *innerFeeGrowth0Token = upper.outerFeeGrowth0Token - lower.outerFeeGrowth0Token;* *innerFeeGrowth1Token = upper.outerFeeGrowth1Token - lower.outerFeeGrowth1Token;* *}* Which is in fact, from observation the same as the refactored code. However, we still always recommend following the original code instead of introducing redundant risk.
Recommendations	Consider following the exact same style as Algebra.
Comments / Resolution	Acknowledged.

MultiFeeDistribution

The MultiFeeDistribution contract is an ERC20 staking contract which distributes different reward tokens and is meant for HypervisorERC20 tokens as stakingToken. Unlike the standard Masterchef, rewards are not constantly accrued per second but rather dripped into the contract and increased on each update.
Any manager address can add new reward tokens while anyone can seed the contract with rewards.
However, the main business logic relies on reward distribution from Algebra's farming module. Furthermore, the contract is pausable via OpenZeppelin's Pausable implementation.

Appendix: Reward Drip Logic

When a yield-generating system wants to distribute new tokens to stakers, it:

1. **Transfers** new reward tokens into the contract's custody (into address(this)).
2. The contract sees that "extra" tokens have arrived (the difference between its new balance and a previously recorded r.amount).
3. It then increments a global ratio, typically like rewardPerToken += (newRewards * 1e50 / totalStakes).
4. This ratio (rewardPerToken) effectively tracks how many tokens each staked unit "owns" in newly arrived rewards.
5. Each user has a rewardPerToken[user] checkpoint, so their claimable portion is userDeposit * (globalRewardPerToken - userRewardPerTokenCheckpoint) / scalingFactor.

This approach ensures **no immediate user** gets an unfair advantage just by being present at that moment: the system lumps the tokens into "unclaimed, proportional to stake." So all stakers share them according to stake / totalStakes.

Core Invariants:

INV 1: calculateClaimable must be called at the beginning of stake, unstake and getRewards

INV 2: calculateClaimable must update claimable, rewardPerToken and lastUpdate

INV 3: calculateClaimable must be called for each rewardToken

INV 4: rewardData.amount must always set to balanceOf during _updateRewards to properly reflect the current reward balance

INV 5: _updateReward must be called before _calculateClaimable

INV 6: userData[onBehalfOf].lastTimeUpdated is always > 0 after the first deposit and will never be set to zero again

INV 7: Any difference b/w balanceOf and rewardData[rewardToken].amount must be used to increase rewardData[rewardToken].rewardPerToken

INV 8: rewardPerToken must always be increased by accumulated token balance divided by totalStakes

INV 9: rewardData.amount must always be decreased by the amount of rewards which is transferred out

Privileged Functions

- transferOwnership
- renounceOwnership
- setManagers
- removeManagers
- setStakingToken
- addReward
- removeRewardToken
- recoverERC20

Issue_48	Rewards can be stolen if a rewardToken is removed
Severity	**High**
Description	**Scenario A: rewardToken is removed and added back** Currently, it is possible to remove a rewardToken and then re-add it back at a later point in time. (via addReward and removeRewardToken) Doing that will have a critical side-effect, as users can now actively manipulate their own due rewards and thus steal rewards from the contract. The root-cause of this issue stems from the fact that the removal of a rewardToken does not adjust the storage for this specific reward token. Consider the example a user would be entitled to 100e18 reward tokens based on his staked balance and the rewardPerToken value: *pendingReward = (newRewardPerToken - userRewardPerToken) * userData[account].tokenAmount / 1e50;* Now the rewardToken is removed and this does not adjust the user's entitlement, in fact, the user has theoretically still the same amount entitled, it just cannot be claimed since the loop does not iterate over the removed token. The user now increases tokenAmount by depositing a large amount. Due to the fact that the rewardToken has been removed, it will not loop over it during _updateReward, not during _calculateClaimable, which therefore means, the rewardToken's storage is not adjusted If the rewardToken is then re-added to the system via addReward, the _calculateClaimable function still follows this math:

	*pendingReward = (newRewardPerToken - userRewardPerToken) * userData[account].tokenAmount / 1e50;* But at the same time, tokenAmount has been increased which grants the user an erroneous large rewardToken amount, effectively allowing for stealing from the contract. **Scenario B: rewardToken is only removed and not added back** This scenario describes a more severe pathway to drain tokens from the contract. Essentially it leverages all facts described in the first scenario with a slightly different callpath: a) rewardToken is being removed b) User executes deposit to increase tokenAmount; rewardToken which was removed is not claimed nor updated due to not being in the array c) User calls getReward with the removed rewardToken as parameter, this will allow the user to completely steal all rewardTokens in the contract: *pendingReward = (newRewardPerToken - userRewardPerToken) * userData[account].tokenAmount / 1e50;* The additional root-cause here is that a user can essentially claim for a rewardToken which is not existent anymore. This has the idea that users can claim their past unclaimed rewards but introduces this critical side-effect.
Recommendations	Consider not allowing to re-add a rewardToken once it has been removed. Consider not allowing to claim rewardToken(s) which are not in the rewardTokens array.
Comments / Resolution	Failed resolution, if a rewardToken is removed, the exploiter can still execute a deposit which increases the exploiters amount, subsequently

	followed by a getReward call with the removed rewardToken as parameter to then receive an inflated amount of rewards. This will essentially allow the exploiter to steal all unclaimed rewards from other users.

Issue_49	Permanently stuck tokens if rewardToken is removed
Severity	**High**
Description	The removeRewardToken function allows for the removal of a rewardToken. This will not only become in effect immediately but will also result in all unclaimed tokens b/w [lastTimeUpdated; block.timestamp] to be stuck. Users can basically only claim what has already been entitled rewardPerToken but not what remains leftover since the lastUpdateTime and block.timestamp. (Besides in the above described edge-case). The reason for that is that the removal of the rewardToken will never trigger an update for rewardPerToken during _updateRewards since it is removed from the array. Furthermore, it will also become impossible to recover them due to the strict check within recoverERC20. It has to be noted that re-adding the rewardToken will not solve this problem because it introduces another issue.
Recommendations	Consider updating rewards before a rewardToken is removed. Furthermore, consider removing the strict check within recoverERC20 as the contract anyways exposes large governance flexibility. It has to be noted that an excess withdrawal will then result in users being unable to claim their leftover rewards. It must be carefully calculated how much rewards are considered as stuck.

Comments / Resolution	Resolved.

Issue_50	Overflow DoS possible in rewardPerToken computation
Severity	**High**
Description	Throughout the MultiFeeDistribution contract, we compute amounts related to the rewardPerToken using a numerator of 1e50 for a high degree of precision, e.g. in claimableRewards. The problem with this logic is that because the numerator is so large, there's a risk of overflow during the initial multiplication, which would cause execution to revert, causing a permanent DoS. Specifically, in this case, this can occur if `additionalRewards` is greater than ~1.1579208924E27. Furthermore, since anyone can transfer tokens into the contract, and we compute additionalRewards based on the token balance of the contract, an attacker could transfer enough tokens into the contract to intentionally trigger this.
Recommendations	Consider using FullMath.mulDiv, which will only revert if the final result would overflow, allowing for much larger values to be safely used. Furthermore, it's worth considering the totalSupply of tokens to be used to consider whether a smaller numerator may also be necessary to ensure that the final result of these computations also does not overflow. Note that this must be fixed everywhere in the contract that we're either multiplying or dividing by 1e50.
Comments / Resolution	Resolved.

Issue_51	Addition of stakingToken as rewardToken will break the protocol
Severity	**High**
Description	Currently, it is possible to add the stakingToken as a rewardToken. While that approach is slightly unconventional, it can still be a reasonable design choice to attempt distributing the HypervisorERC20 token as a reward token, additionally to the farming module rewards. This will however never work as any deposit transaction will increase the balance of the contract which will then inherently be considered as "reward drip" during the next _updateRewards call and will result in the distribution of deposited tokens as reward. Moreover, any manager can set this maliciously to drain stakingToken(s) from the contract.
Recommendations	Consider including an explicit check which prevents that.
Comments / Resolution	Resolved.

Issue_52	Lack of emergencyWithdraw functionality
Severity	**Medium**
Description	Currently, the contract does not expose an emergencyWithdraw functionality. In any scenario where there are issues with the reward calculation or rewardToken distribution, it may be possible that the stakeToken remains permanently locked.
Recommendations	Consider implementing an emergencyWithdraw function. Additional resources must be allocated to ensure the correctness of such a function without introduction of side-effects.
Comments / Resolution	Acknowledged.

Issue_53	Frontrunning possibility in case of non base/bonus rewards
Severity	**Medium**
Description	As already mentioned, the contract allows for the addition of multiple rewardTokens, outside of the base and bonus reward tokens from the farming module. These tokens would need to be dripped into the contract which then increases rewardPerToken on the next _updateReward call. This inherently exposes a frontrunning issue where a malicious user can frontrun a drip with a deposit just to collect a share of the newly introduced rewards.
Recommendations	Consider dripping reasonable amounts in frequent intervals.
Comments / Resolution	Acknowledged.

Issue_54	Manager addresses can DoS protocol via malicious rewardToken
Severity	**Medium**
Description	A malicious manager can add a rewardToken which reverts upon the balanceOf call and then results in a DoS of the _updateReward function, essentially DoS'ing the contract. This issue is rated as medium instead of governance due to the fact that managers are likely less trusted addresses.
Recommendations	Consider ensuring only trusted addresses will be added as manager.
Comments / Resolution	Acknowledged.

Issue_55	Potential storage collision during updates due to inheritance of ReentrancyGuard
Severity	**Low**
Description	The contract inherits the ReentrancyGuard instead of the ReentrancyGuardUpgradeable. This can result in storage collisions during upgrades due to the abundance of the gap.
Recommendations	Consider inheriting ReentrancyGuardUpgradeable.
Comments / Resolution	Acknowledged.

Issue_56	Violation of checks-effects-interactions pattern within _getRewards
Severity	**Low**
Description	Throughout the contract there are one or multiple spots which violate the checks-effects-interactions pattern, to ensure a protection against invalid states, all external calls should strictly be implemented after any checks and effects (state variable changes): *if (claimable[token][_user] > 0) {* *IERC20(token).safeTransfer(_user, claimable[token][_user]);* *r.amount -= claimable[token][_user];* *claimable[token][_user] = 0;* *emit RewardPaid(_user, token, claimable[token][_user]);* *}* Furthermore, the same issue is existing within _stake and _unstake
Recommendations	Consider adjusting the state before any external interaction.
Comments / Resolution	Resolved.

Issue_57	Potential OOG error if rewardTokens array becomes too large
Severity	**Low**
Description	Currently, there is no limitation in how many rewardTokens can be added to the system. In the scenario where this array becomes unreasonably high, it can result in a DoS of multiple functionalities.
Recommendations	Consider implementing a reasonable upper limit
Comments / Resolution	Acknowledged.

Issue_58	totalUnclaimedRewards returns erroneous value if reward token is a transfer-tax token
Severity	**Low**
Description	The totalUnclaimedRewards function calculates the due rewards based on the state within EternalFarming. This however does not account for any potential loss if the token is a transfer-tax token and will result in an erroneous return value.
Recommendations	There is no trivial fix for this issue as there is no simple way to virtually display the tax.
Comments / Resolution	Acknowledged.

Issue_59	claimableRewards only accounts for increase of base and bonus reward tokens
Severity	**Low**

Description	The claimableRewards function only increases rewardPerToken in the scenario that the token is the base or bonus token:
	
	*if (token == unclaimedAddresses[0] || token == unclaimedAddresses[1]) {
	uint256 unclaimedAmount = token == unclaimedAddresses[0] ? unclaimedAmounts[0] : unclaimedAmounts[1];
	uint256 currentBalance = IERC20(token).balanceOf(address(this));
	
	// Calculate new reward per token including unclaimed rewards
	if (totalStakes > 0) {
	uint256 additionalRewards = currentBalance + unclaimedAmount - r.amount;
	newRewardPerToken += additionalRewards * 1e50 / totalStakes;
	}
	}*
	
	In the scenario where the token is neither of both tokens, rewardPerTokens is not increased, even if balanceOf is larger than rewardData[rewardToken].amount

Recommendations	Consider either acknowledging this issue or calculating the difference and increasing rewardPerToken, similar as done within _updateReward.
Comments / Resolution	Acknowledged.

Issue_60	Event emission during _getReward is incorrect
Severity	**Informational**
Description	The _getReward function emits the following event: *emit RewardPaid(_user, token, claimable[token][_user]);* This is incorrect, as claimable is set to zero beforehand.
Recommendations	Consider emitting the event before claimable is reset.
Comments / Resolution	Resolved.

Issue_61	Unguarded updateReward function
Severity	**Informational**
Description	Currently, the updateReward function does not expose a reentrancy guard. While that usually does not expose any critical risk, we need to keep in mind that the _getReward function does not follow the CEI which can be abused to then reenter into updateReward. This exposes a significant user flexibility which is not essential. Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic.
Recommendations	Consider guarding the updateReward function.

Comments / Resolution	Resolved.

Issue_62	_earned is unused
Severity	**Informational**
Description	Functions which are unused will unnecessarily increase the contract size for no reason and will confuse third-party reviewers: *function _earned(* *address _user,* *address _rewardToken* *) internal view returns (uint256 earnings) {* *RewardData memory rewardInfo = rewardData[_rewardToken];* *UserData storage userInfo = userData[_user];* *return (rewardInfo.rewardPerToken - userInfo.rewardPerToken[_rewardToken]) * userInfo.tokenAmount;* *}*
Recommendations	Consider removing this function.
Comments / Resolution	Resolved.

Issue_63	First depositor will receive all idle rewards
Severity	**Informational**
Description	Currently, rewardPerToken is only updated if totalStakes > 0: *for (uint i; i < rewardTokens.length; i ++) {* *address rewardToken = rewardTokens[i];* *if (totalStakes > 0) {* *RewardData storage r = rewardData[rewardToken];* *uint256 currentBalance =* *IERC20(rewardToken).balanceOf(address(this));* *uint256 diff = currentBalance - r.amount;* *r.lastTimeUpdated = block.timestamp;* *r.rewardPerToken += diff * 1e50 / totalStakes;* *r.amount = currentBalance;* *}* *}* This means if there are any accrued rewards and no deposit has happened yet, the first depositor will receive all these accrued rewards after his deposit.
Recommendations	Consider if that is the expected design, if yes, this can be acknowledged.
Comments / Resolution	Acknowledged.

Issue_64	UserData.tokenClaimable is never used
Severity	**Informational**
Description	The UserData struct exposes the following variables: - tokenAmount - lastTimeUpdated - tokenClaimable - rewardPerToken The tokenClaimable variable is never really used, instead, rewards are accrued towards the following state variable: mapping(address => mapping(address => uint256)) public claimable;
Recommendations	Since the contract was already tested with the corresponding storage layout and this does not expose any harm, we do not recommend a change. However, in the future this should be removed.
Comments / Resolution	Acknowledged.

Issue_65	UserData.lastTimeUpdated is never used
Severity	**Informational**
Description	The UserData struct exposes the following variables: - tokenAmount - lastTimeUpdated - tokenClaimable - rewardPerToken The lastTimeUpdated variable is never really used besides in the following check: *if (userInfo.lastTimeUpdated > 0 && userInfo.tokenAmount > 0) {* *claimable[_rewardToken][_onBehalf] += (r.rewardPerToken - userInfo.rewardPerToken[_rewardToken]) * userInfo.tokenAmount / 1e50;* *}* Indeed, it would be possible to fully remove this variable without affecting the contract logic, since there is never a state where tokenAmount > 0 and lastTimeUpdated = 0.
Recommendations	Since the contract was already tested with the corresponding storage layout and this does not expose any harm, we do not recommend a change. However, in the future this should be removed.
Comments / Resolution	Acknowledged.

Issue_66	repetitive call of _updateReward within _getReward
Severity	**Informational**
Description	Within the _getReward function, _updateReward is called on every loop iteration. Generally speaking, one call would already be enough. This will unnecessarily waste gas.
Recommendations	We do not recommend a change due to the high amount of potential changes and the fact that this does not pose a security risk.
Comments / Resolution	Acknowledged.