BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT

Ichi

Vaults

December 2024

Disclaimer:

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

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

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

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

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

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

1. Project Details

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

Project	Ichi - Vaults
Website	ichi.org
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/ichifarm/algebra-integral-ichi-vaults/tree/2ef9f7fb08ec4e7e0e394ae0f7791a5f053047c9/contracts
Resolution 1	https://github.com/ichifarm/algebra-integral-ichi-vaults/tree/ae224c51b036a3ad9b10416aa8dc3fe798457cd8/contracts

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	4	2	1	1
Medium	4	1		3
Low	9			9
Informational	11	1		10
Governance	1			1
Total	29	4	1	24

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	Architecture is not compatible with transfer-tax tokens
Severity	**Informational**
Description	This architecture 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. For example, if direct interactions with the ICHIVault contract are made, shares are calculated based on the provided input amounts and not on the effective received output amounts. Furthermore, interactions via the ICHIVaultDepositGuard contract would not work due to the intermediate transfer-step.
Recommendations	Consider not deploying vaults with transfer-tax tokens or ask projects to specifically whitelist all necessary ICHI contracts.
Comments / Resolution	Acknowledged.

Common

Enum

The Enum contract is a very simple contract which is only meant to be used by the VaultSlippageCheckV2_1 and VaultSlippageV2 contract to determine the call type (call, delegatecall) for the gnosis transaction.

Privileged Functions

- none

External

ContractImports

The ContractImports contract solely imports the BasePluginV1Factory contract from Algebra.

Privileged Functions

- none

No issues found.

InterfaceImport

The InterfaceImports contract imports various interfaces which are used by Algebra.

Privileged Functions

- none

No issues found.

Lib

ICHIVaultDeployer

The ICHIVaultDeployer library is used within the ICHIVaultFactory to deploy a new iteration of the ICHIVault contract.

Privileged Functions

- none

No issues found.

OracleLibrary

The OracleLibrary is used within the UV3Math library and is mainly responsible for interactions with Algebra's volatility oracle and to fetch the SPOT and TWAP price from a pair via the getQuoteAtTick function during a deposit within a vault.

Appendix: Volatility Oracle

The OracleLibrary consults Algebra's VolatilityOracle to get the timeWeightedAverageTick. The oracle stores data by recording timepoints in a circular buffer each time a new data point is available when a swap is executed, potentially every block.

Each timepoint captures essential information such as the current tick, cumulative tick, and cumulative volatility at a specific timestamp.
This approach allows the oracle to maintain a history of price and volatility data efficiently, overwriting the oldest data when the buffer is full.

This data can then be used to fetch the current timepoint and a timepoint in the past to calculate the TWAP tick within this period. If an exact timepoint is not available, the oracle interpolates by using the nearest timepoint.

The way how the TWAP tick is calculated is as follows:

a) Fetch current timepoint and historic timepoint
b) Deduct old cumulativeTick from new cumulativeTick to get tickCumulativesDelta
c) Divide tickCumulativesDelta by the period
d) Receive the average tick within the TWAP period

Furthermore, this library exposes a function that returns a quote at the current tick. If for example a WETH/USDC pair is consulted, this function will return how much USDC corresponds to the provided amount of WETH. This function is considered within the SPOT price calculation during a deposit and attempts to use maximum precision for calculations

Privileged Functions

- none

Issue_02	Circular buffer limitation
Severity	**Low**
Description	Algebra employs a volatility oracle that stores timepoints in a circular buffer with a fixed size of uint16. This buffer efficiently maintains a recent history of price and volatility data by overwriting the oldest timepoints when new ones are added beyond its capacity. Timepoints are potentially written at each block, which could be as frequent as every second, depending on the block time. The getTimepoint function reverts if secondsAgo (the time difference specified for historical data retrieval) is too high because it requests data older than the oldest timepoint currently stored in the buffer. Since the buffer only retains a limited number of recent timepoints, any attempt to access data beyond this range cannot be fulfilled. The function includes checks to prevent access to unavailable historical data, and if the requested periodsAgo exceeds the buffer's capacity, it

	triggers an error like targetIsTooOld(). This mechanism ensures that only valid and accurate data is used in calculations, maintaining the integrity of the oracle's outputs. However, in the following case: a) Block time is very short b) twapPeriod is very long (which is desired to prevent manipulation) c) Swaps and corresponding writes happen as frequent as every block The deposit function will revert because the call to the oracle reverts.
Recommendations	Consider manually ensuring that the twapPeriod is a reasonable but safe value.
Comments / Resolution	Acknowledged. The client added the following comment: Operational procedures ensure that the twapPeriod is always set to a reasonable and safe value.

Issue_03	Unexpected zero return value
Severity	**Informational**
Description	The getQuoteAtTick function returns the corresponding quote amount for a base amount. The function uses most accurate precision to calculate the outcome. However, in certain unexpected scenarios, it may happen that the return value is erroneous and returns zero, which would then result in unexpected side-effects.
Recommendations	Consider reverting if the return value is zero.
Comments / Resolution	Acknowledged. The client added the following comment: In the unlikely case that the getQuoteAtTick function returns 0, the deposit function, which calls getQuoteAtTick internally, will still revert.

Issue_04	Unused functions
Severity	**Informational**
Description	The OracleLibrary contains various functions which remain essentially unused within the current architecture. This will increase contract size for no limit and can confuse third party reviewers.
Recommendations	Consider if it is desired to remove these functions. Alternatively, this can be safely acknowledged.
Comments / Resolution	Acknowledged.

UV3Math

The UV3Math library is a helper library which consults UniswapV3/Algebra native libraries as well as the OracleLibrary. Furthermore, it exposes functionality to calculate the Unique Vault Symbol.

It is used within the ICHIVault and VaultSlippageCheckV2 contracts.

Appendix: IV Symbol

Example:

- ammName() returns "Algebra",
- value = 42,
- allowToken0 = true,
- and the pool tokens are WETH (token0) and DAI (token1),

then the computed symbol would be: "IV-Algebra-42-WETH-DAI".

Changing allowToken0 to false would flip the token order: "IV-Algebra-42-DAI-WETH".

It is important to mention that the allowToken1 value will have no impact on the symbol.

Privileged Functions

- none

No issues found.

Core

IchiVaultDepositGuard

The ICHIVaultDepositGuard contract is a clearance contract which is meant to be applied in front of the ICHIVault contract.

It allows users to deposit while providing an external to address and specify the desired minimum output amount of shares which is declared as the minimumProceeds parameter.

Withdrawals can be made similarly, to an external to address while ensuring that the minimum output amounts are met, which is defined within minAmount1 and minAmount1.

On top of that, both the deposit and withdraw function have a corresponding native functionality which allows a deposit via native ETH and withdrawing native ETH.

Appendix: Core Invariants:

INV 1: Deposits can only made in one of both tokens

INV 2: Native deposits must use forwardNativeDepositToICHIVault

INV 3: Native withdrawals must use forwardNativeWithdrawFromICHIVault

INV 4: Received share amount must be >= minimumProceeds

INV 5: Withdrawal must result in amounts which are >= minAmount0/1

INV 6: Native ETH can only be accepted via payable function by users or via receive by the WETH contract

Privileged Functions

- none

Issue_05	Native transfer does not work with contracts that expose logic during fallback/receive
Severity	**Low**
Description	The contract uses the standard transfer pattern for native ETH transfers. This will never work if the recipient is a smart contract with logic within the fallback/receive function due to the hardcoded gas limit of the transfer call.
Recommendations	Consider using call instead of transfer.
Comments / Resolution	Acknowledged. The client added the following comment: In general, the deposit guard is intended to be used by end users. However, if a smart contract is developed to interact with the deposit guard, that contract can handle the wrapping and unwrapping of ETH when using the guard. This means that contracts which are implemented on top of ICHI should consider handling the ETH -> WETH conversion on their own.

ICHIVault

The ICHIVault contract provides a single-sided liquidity interface to an Algebra pool. It allows depositing one token to receive fungible liquidity tokens that represent a share of the underlying positions. The vault manages NFT-based liquidity positions, periodically rebalances them, collects and distributes fees, and ensures fair pricing through TWAP and spot checks.

Appendix: Share Calculation

The IchiVault contract uses a proportional share calculation mechanism which is similar to ERC4626 vaults when calculating a user's share amount based on the provided value. This is done as follows:

a) Convert a user's total deposit amount to tokenY using the lower price from SPOT or TWAP:

uint256 deposit0PricedInToken1 = deposit0.mul((price < twap) ? price : twap).div(PRECISION);

shares = deposit1.add(deposit0PricedInToken1);

b) Convert the vaults total amounts to tokenY using the higher price from SPOT or TWAP:

uint256 pool0PricedInToken1 = pool0.mul((price > twap) ? price : twap).div(PRECISION);

c) Calculate a users received shares proportionally on the provided amount, existing amount and existing share supply:

shares = shares.mul(totalSupply()).div(pool0PricedInToken1.add(pool1));

Appendix: Algebra Interaction

The contract interacts with the Algebra NFPM to add and remove liquidity. More specifically, during a rebalance, liquidity is completely removed and re-added and during a withdrawal, liquidity is withdrawn in the proportional manner based on the user's provided share amount.

The NFPM source code can be found here:

https://github.com/cryptoalgebra/Algebra/blob/e1f358c1c2c9f6cda455c6f0934b333af624e964/src/periphery/contracts/NonfungiblePositionManager.sol

These are the only interaction points with the Algebra protocol besides the swap.

Appendix: Rebalance

The rebalance function allows the contract owner to remove the basePosition and limitPosition, eventually swap tokens and then re-adding both positions again to a new range.

Appendix: Fee Distribution

Whenever a deposit, withdrawal, rebalance or fee collection call is executed, the pool state is updated via a zero burn, fees are collected towards the ICHIVault address and then distributed among different recipients with the leftover remaining in the vault for value appreciation. The fee distribution is as follows:

a) AMM fee is distributed to the ammFeeRecipient
b) BASE fee is distributed to the affiliate and toRecipient address (depending on baseFeeSplit)
c) Leftover amount will remain within the ICHIVault contract

It is of utmost importance that this is happening before any deposit/withdrawal, as otherwise the share calculation will result in an erroneous outcome.

Appendix: Swap Callback

The algebraSwapCallback function is called by the corresponding pool whenever a swap has been invoked and is responsible for transferring the input token amount from the contract to the pool.

Appendix: Swap Logic

Within the rebalance function, the swapQuantity parameter determines if a swap is executed, into which direction and whether it is a exactInput or exactOutput swap.

Since the swapQuantity can be positive and negative, the two following scenarios derive from this:

a) swapQuantity > 0:

A swap from X -> Y with exactInput is being executed

b) swapQuantity < 0:

A swap from Y -> X with exactOutput is being executed

Appendix: BalanceOf logic

The contract serves as an ERC20 token which implements the balanceOf function, allowing users to use their shares in other protocols. The balance of a user is always the corresponding share amount which is received during the deposit function which always reflects the proportional share of the underlying vault value from a user. This means the balanceOf will return a nominal value which corresponds to a fluctual USD value.

Appendix: Core Invariants

INV 1: Shares must be calculated by incorporating all tokens in the system

INV 2: Shares must be calculated by converting token0 to token1

INV 3: Withdrawal must proportionally

INV 4: Fee must be collected before getTotalAmounts is consulted

INV 5: BalanceOf must reflect a user’s share amount

INV 6: Share amount must always accurately reflect a users proportional share on the underlying vault value.

INV 7: Deposits/Withdrawals must only be executed by ICHIVaultDepositGuard

INV 8: The initial share amount must be scaled by 1000

INV 9: Users must always receive a share amount which is proportional to their deposit amount on the overall pool amount

INV 10: The pool state must be updated and fees must be collected before any deposit/withdraw/rebalance

INV 11: Withdraw must calculate pro-rata share on base and limit position and on unused tokens

INV 12: deposit0PricedInToken0 must always use price which is against the favor of the user (lower price)

INV 13: pool0PricedInToken1 must always use price which favors the vault (higher price)

INV 14: fee0 and fee1 must be completely consumed within _distributeFees

INV 15: rebalance must remove liquidity completely from both tokenIds

INV 16: Any deviation from TWAP to SPOT price during deposits must always result in a disadvantage for the user

INV 17: The supply must either be >= 1000 or 0

INV 18: Any deviation from SPOT and TWAP price within deposit is reflected as fee for the depositor

INV 19: Only single sided deposits must be allowed

INV 20: Specific deviations between SPOT and TWAP price are only allowed if no swap has happened within the current block

INV 21: System value must be solely within basePosition/limitPosition and the vault balance

INV 22: Contract balance must always be >= collected fees

Privileged Functions

- transferOwnership
- renounceOwnership
- resetAllowances
- setTwapPeriod
- setHysteresis
- setAmmFeeRecipient
- setAffiliate
- setDepositMax

Issue_06	Owner can steal funds via various pathways
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. The contract owner has several privileges which can result in a loss of funds: a) updating the twapPeriod b) bringing tick to extreme case plus rebalancing the vault c) swapping forth and back and and accruing fees d) swapping without slippage e) hysteresis manipulation
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged. The client added the following comment: Operational procedures ensure that the owner of the vault is a Gnosis multi-signature contract with trusted signers.

Issue_07	Edge-case allows exploiter to drain value from the vault by abusing deposits and withdrawals via share calculation manipulation
Severity	**High**
Description	The current share calculation is handled in such a way that a users tokenX deposit is multiplied with the TWAP or SPOT price which then returns deposit0PricedInToken: *uint256 deposit0PricedInToken1 = deposit0.mul((price < twap) ? price : twap).div(PRECISION);* Afterwards, the tokenY amount is added on top of that value which represents tokenX in tokenY + tokenY: *shares = deposit1.add(deposit0PricedInToken1);* Then the vault value from tokenX in tokenY is calculated, similar to the above calculation: *uint256 pool0PricedInToken1 = pool0.mul((price > twap) ? price : twap).div(PRECISION);* And the tokenY amount is added on top of that: *pool0PricedInToken1.add(pool1)* Once both values are determined, it is calculated how the percentual share of a user's deposit is, compared to the pool amount: *shares = shares.mul(totalSupply()).div(pool0PricedInToken1.add(pool1));* Based on that, the user will receive a proportional amount of the pool as shares. Within the user deposit calculation, the lower price from either TWAP

or SPOT is chosen, which decreases the user's nominal value in tokenY:

(price < twap) ? price : twap)

Within the pool value calculation, the higher price from either TWAP or SPOT is chosen, which increases the pool's nominal value in tokenY:

(price > twap) ? price : twap)

The idea behind this logic is to always favor the vault and result in less minted shares for the user.

Under most circumstances, this works flawless and will result in a deposit which favors the vault instead of the user.

However, based on the explanation above, Bailsec crafted a specific state where it is possible to exploit the vault's safety mechanism against it.

Two different facts must be given:

a) A user must deposit tokenX where the multiplicator price is higher than the real price

b) The vault must be in a state where where the total vault value mainly consists out of tokenY

State a) can be achieved if the current SPOT price is below the TWAP price, which means the SPOT price should be chosen for a user to determine the provided value. A swap manipulation in the same block, before the deposit, allows for moving the SPOT price towards the TWAP price which results in a higher deposit value for the user.

	This also results in the higher price being used for the vault value calculation, however, if the vault is moved towards a state where there is more tokenY centered than tokenX, this becomes vulnerable. Given these theoretical ideas, Bailsec crafted a scenario where it is possible to exploit the vault by depositing tokenX and withdrawing tokenX and tokenY, gaining a nominal USD value gain based on the real price. The root-cause of this exploit allows for various different PoCs and all of them are based on the fact that a user deposits tokenX at a higher price than the current SPOT price while the vault does not experience that price increase because it mainly consists of tokenY instead of tokenX. **Written PoC:** **STATUS QUO:** SPOT = 10 USD TWAP = 15 USD DOT/USDC pool range = [16095, 29958] (5$ to 20$; equally centered around the current price) currentTick = 23027 (10 USD) liquidity = 1100e18 tokenX = 101,87e18 tokenY = 1018,85e18 At this point, the exact same USD value is on the right side and on the left side. If a user would deposit into the vault at the current state, he would not gain a benefit. This can now be exploited by swapping tokenY to tokenX, which has the following effects:

a) **Increase the price to 15$**
b) **Remove tokenX from the vault, add tokenY to the vault**

Both results are exactly the necessary states which have been described above.

Iteration 1:

1. Swap 750Y in
> get 60X out
-> [16095; 29958]
-> currentTick = 19844
-> liquidity = 1100e18
> brings price price from 250541448375047931186413801569 to 304560650089319070454739222252
> brings tick from 19844 to 26932
> amountX = 40e18
> amountY = 1768e18

2. deposit 10x (which is equivalent to 10 USD but valued at 15 USD due to the exploit step)

> calculate deposit0InToken1:
10 * 15 = 150

> calculate poolInToken1
pool0PricedInToken1 = 40 * 15 = 600
poolAll = 600 + 1768 = 2368

> user gets 6.33% of the pool

3. swap the 60e18 tokenX back; this will give initial provided Y amount (fees are ignored for simplicity)

> amountX (vault liq)= 101,87e18

> amountY (vault liq) = 1018,85e18
> balanceX (in vault) = 10
> price = 10

4. withdraw 6.33% (1100e18 * 0.0633 = 69.63)

> get out from liquidity the following (6.33% each)

> amountX = 6,42e18 tokenX
> amountY = 64,21e18 tokenY
> balanceX (in vault) = 10 = 0.633 tokenX

total calculated in tokenX using current price of 10:

6.42 tokenX + 0.633 tokenX + (64.21 tokenY*1/10) = 13.474 tokenX

User has deposited 10 tokenX and got 13.474 tokenX out, thus stole from the vault, these steps can be likely looped to drain the whole vault.

Vault at the beginning:

> 101,87 * 10 = 1018
> 1018,85
> 2036 USD
> 1100e18 LIQ

Vault after:

> tokenX = 95,45 * 10 = 954,5
> tokenY = 954
> tokenX = 9.367 * 10 = 93.6
= 2002 USD
> 1030.67e18 LIQ

Iteration 2:

NEW STATUS QUO:

-> [16095; 29958]

-> currentTick = 23027

-> liquidity = 1030.67e18

-> amountX = 94.56e18

-> amountY = 953e18

-> balanceX = 9.367

-> SPOT = 10 USD

-> TWAP = 15 USD

1. Swap 700 Y in to bring price to 15

> new price = 304350828448189709171104114894

> new currentTick = 26918

> new liquidity distribution

    > tokenX = 37,8e18

    > tokenY = 1654.0e18

2. deposit 10x (which is equivalent to 10 USD but valued at 15 USD due to the exploit step)

> calculate deposit0InToken1:

10 * 15 = 150

> calculate poolInToken1

pool0PricedInToken1 = 37.8* 15 = 567

poolAll = 567 + 1654 + (9.367*15) = 2361

> user gets 6.35% of the pool

3. Swap back:

-> [16095; 29958]
-> currentTick = 23027
-> liquidity = 1030e18
-> amountX = 94.56e18
-> amountY = 953e18
-> balanceX = 19.367
-> SPOT = 10 USD
-> TWAP = 15 USD

4. Withdraw 6.35%

> 1030.67 * 0.0635 = 65,44
> tokenX = 6,06e18
> tokenY = 60.6e18
> tokenX = 1.22e18
> user got: 133.4 USD while depositing 100 USD

Vault at the beginning:

> tokenX = 95,45 * 10 = 954,5
> tokenY = 954
> tokenX = 9.367 * 10 = 93.6
= 2002 USD
> 1030.67e18 LIQ

Vault after:

> tokenX = 89,37 * 10 = 893,70
> tokenX = 893,81
> tokenX = 18,14 * 10 = 181,14
= 1967 USD
> 965e18 liquidity

Iteration 3:

NEW STATUS QUO:

-> [16095; 29958]
-> currentTick = 23027
-> liquidity = 965e18
-> amountX = 89,37e18
-> amountY = 893,81e18
-> balanceX = 18,14
-> SPOT = 10 USD
-> TWAP = 15 USD

1. Swap 670 Y in to bring price to 15
> new price = 305549602659563067132190430299
> new currentTick = 26996
> new liquidity distribution
> tokenX = 34,43
> tokenY = 1563,81

2. deposit 10x (which is equivalent to 10 USD but valued at 15 USD due to the exploit step)

> calculate deposit0InToken1:
10 * 15 = 150

> calculate poolInToken1
pool0PricedInToken1 = 34,43* 15 = 516
poolAll = 516 + 1563,81 + (18.14*15) = 2351.19

> user gets 6.38% of the pool

3. Swap back:

-> [16095; 29958]
-> currentTick = 23027
-> liquidity = 965e18
-> amountX = 89,37e18
-> amountY = 893,81e18
-> balanceX = 28,14
-> SPOT = 10 USD
-> TWAP = 15 USD

4. Withdraw 6.38%

> 965 * 0.0635 = 61.27
> tokenX = 5,67
> tokenY = 56,75
> tokenX = 1.8
> user got: 131 USD while depositing 100 USD

Vault at the beginning:

> tokenX = 89,37 * 10 = 893,70
> tokenX = 893,81
> tokenX = 18,14 * 10 = 181,14
= 1967 USD
> 965e18 liquidity

Vault after:

> tokenX = 83,69 * 10 = 836.90
> tokenY = 837,06
> tokenX = 26.34 * 10 = 263.34
= 1937 USD
> 904e18 liquidity

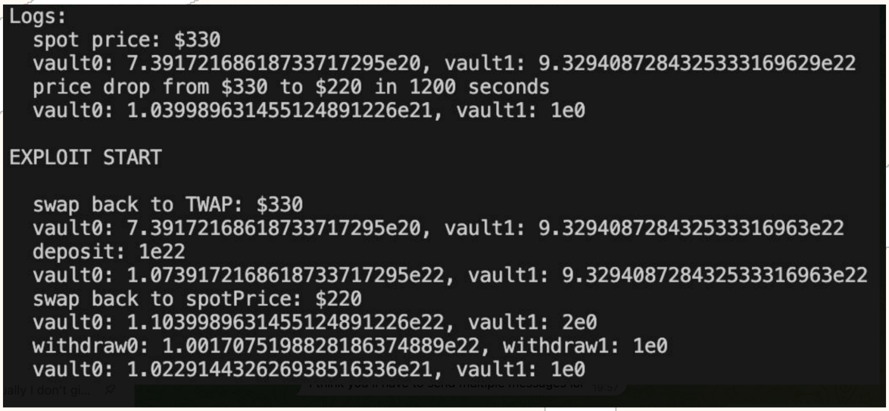
	Swap tokenX to tokenY to achieve current liquidity ratio and re-add tokenX/Y back to the liquidity to determine full post-exploit vault liquidity:
	
	-> swap X to Y:
	
	tokenX = 13.37
	tokenY = 133.7
	
	receive: 142e18 LIQ
	
	end liquidity after 3 iterations:
	
	904e18 + 142e18 = 1046e18
	
	Start liquidity = 1100e18
	
	The vault effectively lost 54e18 liquidity tokens, making it not only less worth in nominal tokenX / tokenY / USD value but also in nominal liquidity tokens (which is the most important measurement of a lp position's value).
	
	With different numbers, this will be more severe. For example if the first deposit consists of 100 tokenX, the vault will already lose 158e18 liquidity tokens.
	
	Coded PoC (different values):
	
	function setUp() public {
	    vm.createSelectFork("https://rpc.ankr.com/bsc", 44727636);
	
	    deal(token0, address(this), 1e36);
	    deal(token1, address(this), 1e36);
	}

```
function test_Deposit_DropInPrice() public {
    vm.label(address(token0), "Token0");
    vm.label(address(token1), "Token1");
    vm.label(address(pool), "Pool");
    vm.label(address(vault), "Vault");

    IV3Pool.GlobalState memory globalState =
IV3Pool(pool).globalState();

    sqrtPriceTWAP = globalState.price;
    sqrtPriceSpot = globalState.price * 8165 / 10000;

    twapPrice = _convertPrice(sqrtPriceTWAP);
    spotPrice = _convertPrice(sqrtPriceSpot);
    console.log("spot price: $%i", twapPrice / 1e18);

    MockERC20(token0).approve(address(vault), 1e36);

    vault.collectFees();
    (uint256 totalAmount0, uint256 totalAmount1) =
vault.getTotalAmounts();
    console.log("vault0: %e, vault1: %e", totalAmount0, totalAmount1);

    (, uint32 time,,,,) =
IV3Pool(pool).timepoints(globalState.timepointIndex);

    vm.roll(block.number + 1);
    vm.warp(time + 1200);

    IV3Pool(pool).swap(address(this), sqrtPriceTWAP > sqrtPriceSpot,
1e36, sqrtPriceSpot, "");
    console.log("price drop from $%i to $%i in 1200 seconds", twapPrice
/ 1e18, spotPrice / 1e18);

    vault.collectFees();
```

	```
	    (totalAmount0, totalAmount1) = vault.getTotalAmounts();
	    console.log("vault0: %e, vault1: %e", totalAmount0, totalAmount1);

	    (int256[] memory amounts0, int256[] memory amounts1) = (new int256[](2), new int256[](2));

	    console.log("\nEXPLOIT START\n");

	    (amounts0[0], amounts1[0]) =
	        IV3Pool(pool).swap(address(this), sqrtPriceTWAP < sqrtPriceSpot, 1e36, sqrtPriceTWAP, "");
	    console.log("swap back to TWAP: $%i", twapPrice / 1e18);

	    vault.collectFees();
	    (totalAmount0, totalAmount1) = vault.getTotalAmounts();
	    console.log("vault0: %e, vault1: %e", totalAmount0, totalAmount1);

	    vault.deposit(deposit0, 0, address(this));
	    console.log("deposit: %e", deposit0);

	    vault.collectFees();
	    (totalAmount0, totalAmount1) = vault.getTotalAmounts();
	    console.log("vault0: %e, vault1: %e", totalAmount0, totalAmount1);

	    (amounts0[1], amounts1[1]) =
	        IV3Pool(pool).swap(address(this), sqrtPriceTWAP > sqrtPriceSpot, 1e36, sqrtPriceSpot, "");
	    console.log("swap back to spotPrice: $%i", spotPrice / 1e18);

	    vault.collectFees();
	    (totalAmount0, totalAmount1) = vault.getTotalAmounts();
	    console.log("vault0: %e, vault1: %e", totalAmount0, totalAmount1);

	    (uint256 amount0, uint256 amount1) = vault.withdraw(vault.balanceOf(address(this)), address(this));
	```

```
    console.log("withdraw0: %e, withdraw1: %e", amount0, amount1);

    vault.collectFees();
    (totalAmount0, totalAmount1) = vault.getTotalAmounts();
    console.log("vault0: %e, vault1: %e", totalAmount0, totalAmount1);
}
```

Output result:



```
Logs:
  spot price: $330
  vault0: 7.391721686187337172 95e20, vault1: 9.329408728432533316 9629e22
  price drop from $330 to $220 in 1200 seconds
  vault0: 1.0399896314551248 91226e21, vault1: 1e0

EXPLOIT START

  swap back to TWAP: $330
  vault0: 7.391721686187337172 95e20, vault1: 9.3294087284325333169 63e22
  deposit: 1e22
  vault0: 1.073917216861873371 7295e22, vault1: 9.3294087284325333169 63e22
  swap back to spotPrice: $220
  vault0: 1.1039989631455124891226e22, vault1: 2e0
  withdraw0: 1.0017075198828186374889e22, withdraw1: 1e0
  vault0: 1.0229144326269385163 36e21, vault1: 1e0
```

The output demonstrates how tokenX was siphoned out from the vault.

Recommendations	A proper solution is the implementation of an oracle. However, alternatively this issue may be partially resolved by the implementation of multiple TWAP prices with different periods and choose between these to use the lowest TWAP price for the user's deposit calculation and the highest TWAP price for the vault's value calculation.
Comments / Resolution	Partially resolved. To understand the solution and the potential impact of the solution, we need to re-elaborate the share calculation mechanism and the invariants. Afterwards we need to re-elaborate the edge-case which allows the exploit:

	When users deposit tokenX, the SPOT and TWAP price are fetched which represents tokenXInTokenY. The conservative approach uses the smaller of both prices to calculate a user's token0InToken1 amount. This will result in: **INV 12: deposit0PricedInToken0 must always use price which is against the favor of the user (lower price)** Furthermore, the vault's AUM value in tokenY is calculated by using the larger of both prices to convert tokenX to tokenY This will result in: **INV 13: pool0PricedInToken1 must always use price which favors the vault (higher price)** During the normal business logic, any deviation will result in a fee for the user. The exploit scenario was now to abuse an edge-case where the SPOT price is below the TWAP price (or vice-versa in case tokenY would be deposited, however, ICHI only works with tokenX). In that scenario, it was possible to execute a swap which does the following: a) Move the SPOT price towards the TWAP price b) Convert any eventual tokenX to tokenY via the curve price between SPOT and TWAP, which means tokenX will result in less tokenY, compared to if it would be swapped at the TWAP price. (In our example this was the price between 10 and 15, which means tokenX will be swapped to tokenX between 10 and 15) The proposed solution was to implement a reliable oracle. However, since ICHI vaults are meant to be compatible with various tokens, including those which do not have an oracle, ICHI decided to

	incorporate another TWAP price with a shorter period. In fact, the period can be set by the contract owner. The idea of this second TWAP period is to: a) Amplify the above identified invariants by choosing the lower price for user value calculations and the higher price for vault value calculations. b) (Partially) prevent the above mentioned exploit where a second TWAP (which is closer to the initial SPOT price than the main TWAP) is used for the user calculation. For the above mentioned exploit this means even if the exploiter swaps the SPOT price to 15 USD, the incorporation of a second, faster oracle will be near to the initial SPOT price, potentially around 11 USD, which will make the exploit less profitable. On top of that, the hysteresis check was made more strict, which now also incorporates the deviation from the SPOT price to the second TWAP. Applied on our example, if the second TWAP returns 12 and the SPOT price was manipulated to 15, the transaction will revert if the swap has happened within the same block. However, that also means that under specific circumstances, the second TWAP price may result in a higher deviation from the main TWAP price compared to the SPOT price, which can then result in a higher fee for users. Therefore, it should be carefully considered for each vault, IF the second TWAP should be activated and with which period. The client added furthermore the following comment: **Summary**: After carefully reviewing the reported scenario, we have identified that while a theoretical exploit can be described under specific conditions, the vault’s underlying incentive structure and trading strategy are designed to compensate arbitrageurs for returning deposited tokens.

	This mechanism effectively self-corrects any temporary imbalance and mitigates the described exploitation pathway. Simply put, it is a desired feature in our design. Nonetheless, for situations where users don't want this feature, we implemented additional control using multiple TWAP calculations which can be turned on to eliminate the incentive for all practical purposes. **Reasoning**: 1. **Non-Issue Justification**: The core premise of the described scenario involves temporarily manipulating price inputs to gain an advantage in share issuance. Our system's economic design inherently addresses such discrepancies. As tokens become imbalanced, the vault pays increasingly higher premiums, incentivizing arbitrageurs to supply the missing assets. This external market correction mechanism naturally closes the gap: o **Built-In Self-Correction**: Any profit realized by a participant through the described manipulation is counterbalanced by a surplus of value delivered to the vault's liquidity providers by pushing the vault closer to target inventory levels. Over time, as the vault returns to balance, these incentives diminish to zero, eliminating any ongoing exploitable condition. o **Dynamic Incentive Alignment**: The system's equilibrating forces ensure that attempts to repeatedly exploit short-term price divergences become uneconomical. The cost of continuously manipulating the pool outweighs the possible one-off gains, maintaining long-term stability. 2. **Enhancement — Two TWAP Integration**: Although the vault design naturally mitigates the purported exploit through its economic incentives, we recognize the value of implementing a more robust pricing framework to address even the theoretical risk. o **Two TWAP Periods**: We integrated two TWAP calculations to ensure that transient price fluctuations are properly accounted for.

	○ **Selection Logic**: Deposits will be valued conservatively using the lowest price between the spot price and enabled TWAPs, while the vault's aggregated valuation will rely on the highest price between the spot price and enabled TWAPs. This ensures that share issuance remains robust against rapid, short-term price manipulations. ○ **Transparency & On-Chain Disclosures**: The methodology for computing and selecting TWAP values will be fully documented and disclosed on-chain, enhancing transparency and ensuring that all stakeholders are aware of the valuation logic. 3. **Conclusion**: Given the vault's original design, the reported issue does not present a direct, long-term economic vulnerability. The incentive mechanism to compensate arbitrageurs already provides a self-healing equilibrium. However, to address the theoretical scenario thoroughly, the implementation of two TWAP calculations offers an additional safety net.

Issue_08	Lack of LST fee incorporation within getTotalAmounts will result in loss for depositor
Severity	**High**

Description	Within deposit the getTotalAmount function plays a fundamental part in the share calculation as this function essentially returns the tokenX and tokenY amount within all positions and within the pool:
	
	function getTotalAmounts() public view override returns (uint256 total0, uint256 total1) {
	(, uint256 base0, uint256 base1) = getBasePosition();
	(, uint256 limit0, uint256 limit1) = getLimitPosition();
	total0 = IERC20(token0).balanceOf(address(this)).add(base0).add(limit0);
	total1 = IERC20(token1).balanceOf(address(this)).add(base1).add(limit1);
	}
	
	The integrity of this function is of utmost importance as any deviation from the system value will either result in a profit or in a loss for the depositor (due to the proportional calculation):
	
	shares = shares.mul(totalSupply()).div(pool0PricedInToken1.add(pool1));
	
	uint128 liquidityToDecrease = uint128(uint256(positionLiquidity).mul(shares).div(totalSupply));
	
	However, this function does not incorporate the pending LST fee into the accounting:
	
	struct PositionWithdrawalFee {
	uint32 lastUpdateTimestamp; // last increase/decrease liquidity timestamp
	***uint128 withdrawalFeeLiquidity**; // liqudity of accumulated withdrawal fee*

```
        }
```

This means if a user deposits, the total system value will be calculated without the pending fee and if then rebalance is triggered, the total system value will be immediately decreased because this liquidity is burned from the position:

```
        if (positionWithdrawalFeeLiquidity > 0) {
        (amount0, amount1) = pool._burnPositionInPool(tickLower,
tickUpper, positionWithdrawalFeeLiquidity);
        FeesVault[] memory vaults =
withdrawalFeePoolParams[address(pool)].feeVaults;
        if (vaults.length == 0) {
                pool.collect(defaultWithdrawalFeesVault, tickLower,
tickUpper, uint128(amount0), uint128(amount1));
        } else {
                for (uint i = 0; i < vaults.length; i++) {
                uint16 feePart = vaults[i].fee;
                pool.collect(
                vaults[i].feeVault,
                tickLower,
                tickUpper,
                uint128((amount0 * feePart) / FEE_DENOMINATOR),
                uint128((amount1 * feePart) / FEE_DENOMINATOR)
                );
                }
        }
        }
```

Recommendations	Consider incorporating _positionsWithdrawalFee[params.tokenId].withdrawalFeeLiquidity into the _getPositionAmounts function and decreasing the positionLiquidity by the withdrawalFeeLiquidity to then return the correct corresponding tokenX and tokenY amounts.

Comments / Resolution	Resolved, the _getPositionAmounts function was adjusted and is now fetching the up-to-date withdrawal fee via _getLatestWithdrawalFeeLiquidity(positionId). This will invoke calculateLatestWithdrawalFeesLiquidity(positionId) on the NFPM which then fetches the up-to-date fee: *latestWithdrawalFeeLiquidity = positionWithdrawalFee.withdrawalFeeLiquidity + _calculateWithdrawalFees(* *pool,* *positionWithdrawalFee.lastUpdateTimestamp,* *position.tickLower,* *position.tickUpper,* *position.liquidity* *);* This fee is then deducted from the position liquidity and used to calculate the vault's owned assets: *uint128 withdrawalFeesLiquidity = _getLatestWithdrawalFeeLiquidity(positionId);* *liquidity = positionLiquidity > withdrawalFeesLiquidity* *?* *uint128(uint256(positionLiquidity).sub(uint256(withdrawalFeesLiquidity)))* *: 0;* This amount correctly reflects the vault's “real” owned assets and thus can be used for the calculation of shares.

Issue_09	Lack of LST fee incorporation into withdraw allows users to bypass the LST fee
Severity	**High**
Description	The LST fee plays a fundamental part in the protocol and ensures position can only be decreased while the corresponding fee is being paid: *function decreaseLiquidity(* *DecreaseLiquidityParams calldata params* *)* *external* *payable* *override* *isAuthorizedForToken(params.tokenId)* *checkDeadline(params.deadline)* ……. *if (positionWithdrawalFeeLiquidity > 0) {* *(amount0, amount1) = pool._burnPositionInPool(tickLower, tickUpper, positionWithdrawalFeeLiquidity);* This invariant is not enforced within the withdraw function, users can simply withdraw their share and bypass the LST fee, which will then result in all other users in the vault bearing this loss. This is due to the fact that the withdraw function triggers the decreaseLiquidity function only after liquidityToDecrease is being calculated and liquidityToDecrease is being calculated based on positionLiquidity which does not incorporate the LST fee: a) fetch position liquidity without LST fee: *// Get position info*

```
        (
        ,
        ,
        ,
        ,
        ,
        ,
        uint128 positionLiquidity,
        ,
        ,
        uint128 tokensOwed0,
        uint128 tokensOwed1
        ) = _nftManager().positions(positionId);
```

b) Calculate liquidityToDecreased based on positionLiquidity (which is the wrong value because it doesnt include pending LST fee):

```
uint128 liquidityToDecrease =
uint128(uint256(positionLiquidity).mul(shares).div(totalSupply));
```

c) Decrease the liquidity by the wrong value, allowing user to bypass the fee:

```
        if (liquidityToDecrease > 0) {
        // Decrease liquidity
        (amount0, amount1) = _nftManager().decreaseLiquidity(
                INonfungiblePositionManager.DecreaseLiquidityParams({
                tokenId: positionId,
                liquidity: liquidityToDecrease,
                amount0Min: 0,
                amount1Min: 0,
                deadline: block.timestamp
                })
        );
```

	The crux hereby is that the full liquidityToDecrease will be accounted to the user, while the LST fee will simply be deducted from the remainder of the position: a) Burn the fee: *amount0, amount1) = pool._burnPositionInPool(tickLower, tickUpper, positionWithdrawalFeeLiquidity);* c) Burn liquidityToDecrease (params.liquidity): *uint128 liquidityDeltaWithoutFee = params.liquidity > positionLiquidity - positionWithdrawalFeeLiquidity* *? positionLiquidity - positionWithdrawalFeeLiquidity* *: params.liquidity;* *(amount0, amount1) = pool._burnPositionInPool(tickLower, tickUpper, liquidityDeltaWithoutFee);* **PoC:** > liquidity = 100e18 > liquidityFee = 10e18 > shares = 100e18 > Alice = 90e18 > Bob = 10e18 Alices withdraws 90e18 shares: > liquidityToDecrease = 90e18 > Alice receives 90e18 liquidity (as tokenX/tokenY) > 10e18 liquidityFee is burned > the positionId has 0 liquidity left > Bob is left with nothing

Recommendations	Consider properly incorporating the LST fee into the positionLiquidity when calculating a user's proportional liquidity based on shares. Furthermore, proper fuzzing tests are mandatory to be provided with this fix by the client. This fix will furthermore require additional resources by Bailsec to formally verify correctness.
Comments / Resolution	Resolved, the _withdrawFromPosition function now fetches the withdrawal fee via the _getLatestWithdrawalFeeLiquidity function, similar as within _getPositionAmounts. This fee is then deducted from the positionLiquidity and used to calculate how much liquidity is owed to the user by the incorporation of the proportional amount of owned shares to the total vault supply. Counter PoC: > adjustedLiquidity = 90e18 (with deducted fee) > liquidityToDecrease = 90e18 * 90e18 / 100e18 > liquidityToDecrease = 81e18 > 81e18 + 10e18 liquidity is burned > Bob is left with 9e18 liquidity

Issue_10	Lack of slippage check can result in sandwich attack during rebalance swap
Severity	**High**
Description	First of all, we must elaborate that this issue is not corresponding to frontrunning a rebalance via a swap, as this attack-vector is effectively mitigated due to the VaultSlippageCheckV2_1 contract. The rebalance function executes a swap in either direction based on the swapQuantity parameter. If positive, a swap from X -> Y is executed, if negative from Y -> X. Since the VaultSlippageCheckV2_1 contract incorporates a slippage check which ensures that the pool price cannot be manipulated, a swap is usually considered as safe. However, swaps are liquidity dependent and in times of thin liquidity, the swap which starts at the expected tick is being executed until the input amount is consumed. This can be exploited by a malicious user as follows: 1. rebalance function is invoked with swapQuantity = -1000e18 which aims to swap tokenY to tokenX 2. Malicious user is the majority provider of the pool in the current range. This user detects such a swap in the mempool and removes the liquidity from the range, resulting in no liquidity or thinner liquidity, based on other liquidity providers. Furthermore, this user provides tokenX on a tick which reflects a very high price for tokenX. 3. The swap is being executed, starting from the current tick (which is still within the slippage range). Due to the thin liquidity, the swap absorbs all the liquidity until it reaches the expensive tick where tokenX

	is being added, consuming a partial (or full) amount of tokenX. 4. The exploiter gained profit by forcing the vault to execute the swap up to the provided tokenX on the expensive tick while the vault now experienced a large loss. This can essentially mean the vault swaps 10000000 tokenY for 1 tokenX (depending on the tick where the exploiter has placed the tokenX liquidity). If under any circumstances, tokens are swapped within this implementation, this is directly exploitable.
Recommendations	Consider providing the limitSqrtPrice as parameter for the rebalance function which is then used for the swap call.
Comments / Resolution	Acknowledged, the client will not use the swap during rebalance.

Issue_11	Malicious user can create vault for a new pair
Severity	**Medium**
Description	Currently, it is expected that the frontend shows only vaults which are created by ICHI and not by any other addresses. This ensures that vaults are not drainable by any malicious owner interactions (as long as the owner is sufficiently guarded). There is however an issue which is related to the fact that there is currently no TWAP check whenever a new ICHIVault iteration is deployed. Since Algebra pools can be deployed permissionless, a scenario can occur where a malicious user creates a pair and seeds this pair with very low liquidity, just with the intention to keep the pair at a wrong price which then results in very cheap manipulation of the TWAP price (if every few blocks a small swap is happening). In the scenario where a pair has low liquidity, there is also no interest for MEV bots to rebalance the pair to the correct market price. If ICHI now deploys a vault to such a corresponding, beforehand manipulated pair, this pair will have a wrong TWAP price. Once the vault is then deployed, a malicious user can just deposit tokenX while the SPOT + TWAP price corresponds to the wrong (increased price), allowing the attacker to gain a large amount of shares. The attacker can then simply swap the pair to the normal spot price which means that all subsequent users will only receive a very small share amount compared to the attacker, allowing the attacker to drain a majority of the pool at any desired time. This issue is only rated as medium because the execution is non-trivial (but still possible practically).

Recommendations	Consider adding a TWAP check upon contract deployment which only allows deployment if the TWAP price matches a desired range. This will prevent any situations where the ICHI deployer accidentally deploys a vault for a pair with a manipulated price.
Comments / Resolution	Acknowledged, the client added the following comment: Deploying a vault does not automatically place liquidity into the pool. While the pool may initially have an incorrect price, deposits into the vault remain safe, as the vault only holds the deposit token until the first rebalance. Operational procedures ensure that the first rebalance does not occur when the pool has an incorrect price or insufficient liquidity for arbitrage.

Issue_12	Vault can be bricked temporarily by exploiting check within NFPM
Severity	**Medium**

Description	The rebalance function calls _dismantlePosition which then calls NFPM.decreaseLiquidity if there are valid positions as (basePositionId != 0; limitPositionId != 0):
	
	_dismantlePosition(basePositionId);
	_dismantlePosition(limitPositionId);
	
	`if (positionId != 0) {`
	`_nftManager().decreaseLiquidity(`
	`    INonfungiblePositionManager.DecreaseLiquidityParams({`
	`    tokenId: positionId,`
	`    liquidity: _getPositionLiquidity(positionId),`
	`    amount0Min: 0,`
	`    amount1Min: 0,`
	`    deadline: block.timestamp`
	`    })`
	`);`
	
	As one can see here, the NFPM.decreaseLiqudity call is enforced as long as the positionId is non-zero.
	
	If we now inspect NFPM.decreaseLiquidity, we can spot the following validation:
	
	require(params.liquidity > 0);
	
	This means, if the liquidity parameter is zero, the call will revert and thus the whole rebalance call will revert.
	
	A malicious user can exploit this fact and brick a newly deployed vault as follows:

	a) Alice deposits funds into the contract and is the only depositor b) The contract owner invokes rebalance which then adds liquidity by using the funds which Alice has just deposited into the contract c) Alice calls withdraw with all her shares. This call will burn all liquidity which is entitled to the tokenId. However, tokenId is not set to zero, nor burned. d) The vault is effectively bricked because even if someone else deposits now, rebalance can never be invoked because tokenId != 0 but liquidity = 0 and thus the enforced decreaseLiquidity call will always revert, resulting in a DoS of the rebalance call.
Recommendations	There are multiple possibilities to fix that issue. While one possibility is to implement a code-based solution which resets the tokenId in a scenario where all shares are withdrawn or a check which simply returns early if there is no liquidity left. However, these changes are intrusive and may expose unexpected side-effects in other scenarios. Therefore, we remain cautious and recommend the owner to execute the first deposit which is never being withdrawn and thus ensures that basePositionId and limitPositionId can never result in zero liquidity via a simple withdraw call. If something like that still happens, in the worst case, anyone can simply call increaseLiquidity permissionless on the NFPM for the corresponding tokenId.
Comments / Resolution	Resolved, the _dismantlePosition function now implements an if-clause which only decreases liquidity if in fact the positionLiquidity is larger than zero.

Issue_13	Rebalance slippage check within VaultSlippageChecV2 is void if swap is executed during rebalance
Severity	**Medium**
Description	The rebalance function is only meant to be called by the owner which flows through the VaultSlippageCheckV2 contract and exposes the deviation check: *// Check if the current tick is within the allowed range* *bool isWithinRange = (currentTick >= (expectedTick - range)) && (currentTick <= (expectedTick + range));* This check works as expected. A small side-effect was not incorporated into the thought process: The fact that the optional swap will inherently change the price/tick of the pool. This will then effectively result in a tick which is deviated from the desired tick, making the isWithinRange check void, which then will further result in undesirable liquidity composition.
Recommendations	Consider providing a minSqrtPrice parameter which not only ensures that the desired slippage is correct but also ensures that the initial deviation is still preserved.
Comments / Resolution	Acknowledged, the client will not use the swap during rebalance. Furthermore, the client developed a VaultSlippageCheckV3 contract which can be used for that purpose. This contract is not audited by Bailsec.

Issue_14	Fee changes will be applied in hindsight
Severity	**Medium**
Description	The _distributeFees function distributes fees to different participants based on the determined percentage values. These values can be changed within the ICHIVaultFactory contract and changes will be applied in hindsight, without beforehand claiming fees. This can result in a disadvantage for users if fees have not been updated/claimed for some time and then suddenly percentages towards external recipients increase.
Recommendations	Consider ensuring that fees are always updated before variables are changed.
Comments / Resolution	Acknowledged. Client indicated that it is sufficient to regularly call collectFees

Issue_15	Farming module is not incorporated
Severity	**Low**

Description	Algebra's NFPM includes a built-in farming module which allows for reward distribution on tokenIds:
	
	`function enterFarming(IncentiveKey memory key, uint256 tokenId) external override isApprovedOrOwner(tokenId) {`
	`bytes32 incentiveId = IncentiveId.compute(key);`
	`if (address(incentiveKeys[incentiveId].pool) == address(0)) incentiveKeys[incentiveId] = key;`
	
	`require(deposits[tokenId] == bytes32(0), 'Token already farmed');`
	`deposits[tokenId] = incentiveId;`
	`nonfungiblePositionManager.switchFarmingStatus(tokenId, true);`
	
	`IAlgebraEternalFarming(eternalFarming).enterFarming(key, tokenId);`
	`}`
	
	`function approveForFarming(`
	`uint256 tokenId,`
	`bool approve,`
	`address farmingAddress`
	`) external payable override isAuthorizedForToken(tokenId) {`
	`address newValue;`
	`if (approve) {`
	`require(farmingAddress == farmingCenter);`
	`newValue = farmingAddress;`
	`}`
	`farmingApprovals[tokenId] = newValue;`
	`}`
	
	This module is completely ignored by the ICHI implementation, effectively rendering it impossible for ICHI positions to participate in

	farming campaigns.
Recommendations	Consider if it is desired to participate in farming. If yes, this requires additional implementation.
Comments / Resolution	Acknowledged.

Issue_16	Uncovered edge-case within _mintPosition
Severity	**Low**

Description	The _mintPosition function exposes different conditions with corresponding validations. More specifically, it ensures that provided desired amounts match with the tick setup.
	
	This means for example, in the scenario where liquidity is only added below/equal currentTIck, only tokenY must be considered.
	
	Furthermore, the following scenario enforces that both tokens are desired:
	
	// If current tick is within or on the boundaries of our range, we need both tokens
	if (currentTick >= tickLower && currentTick < tickUpper) {
	if (amount0Desired == 0 || amount1Desired == 0) {
	return 0;
	}
	}
	
	The current function does not cover the following edge-case:
	
	currentTick = lowerTick & currentTick < upperTick & **currentPrice = price(currentTick)**

In that scenario, it is allowed to only provide tokenX, however, the function returns 0 if amount1Desired = 0, while in fact it is possible to only provide amount0Desired.

To further understand this edge-case, we have provided the following explanation:

> token0/token1 distribution is not based on currentTick but on currentPrice

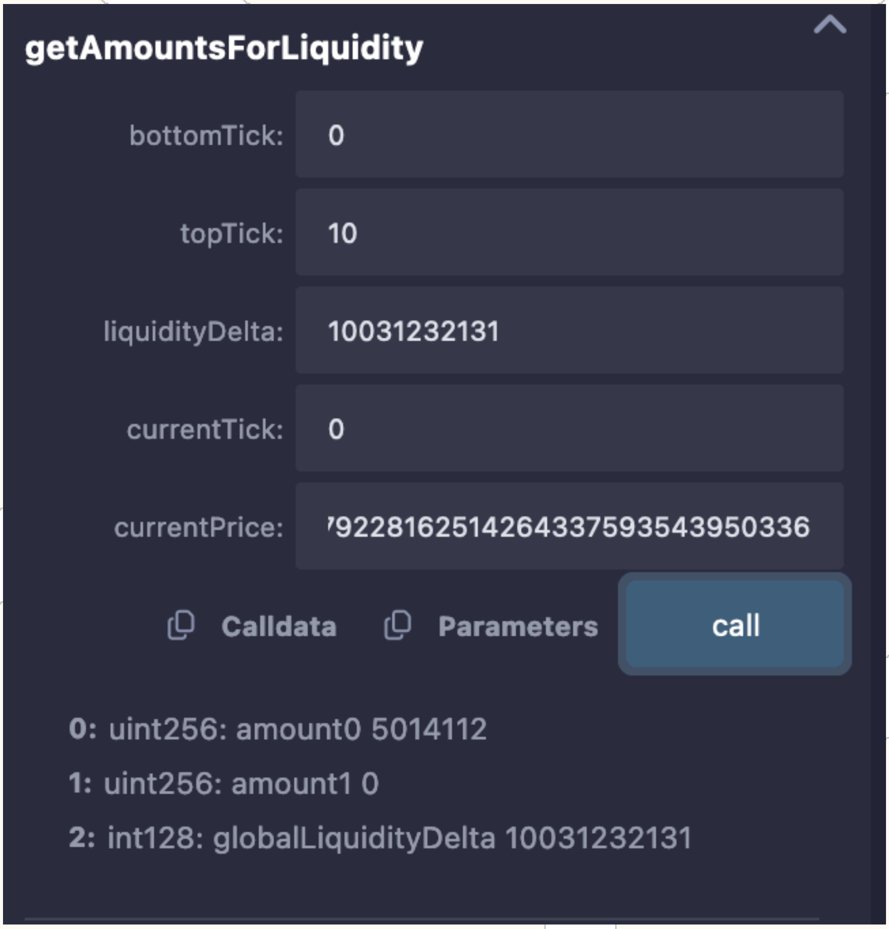
> currentPrice to currentTick conversion always rounds down

> invariant: currentTick <= currentPrice

> most of the time, the currentPrice is not exactly sitting at currentTick

> **but sometimes, currentPrice exactly corresponds to currentTick**

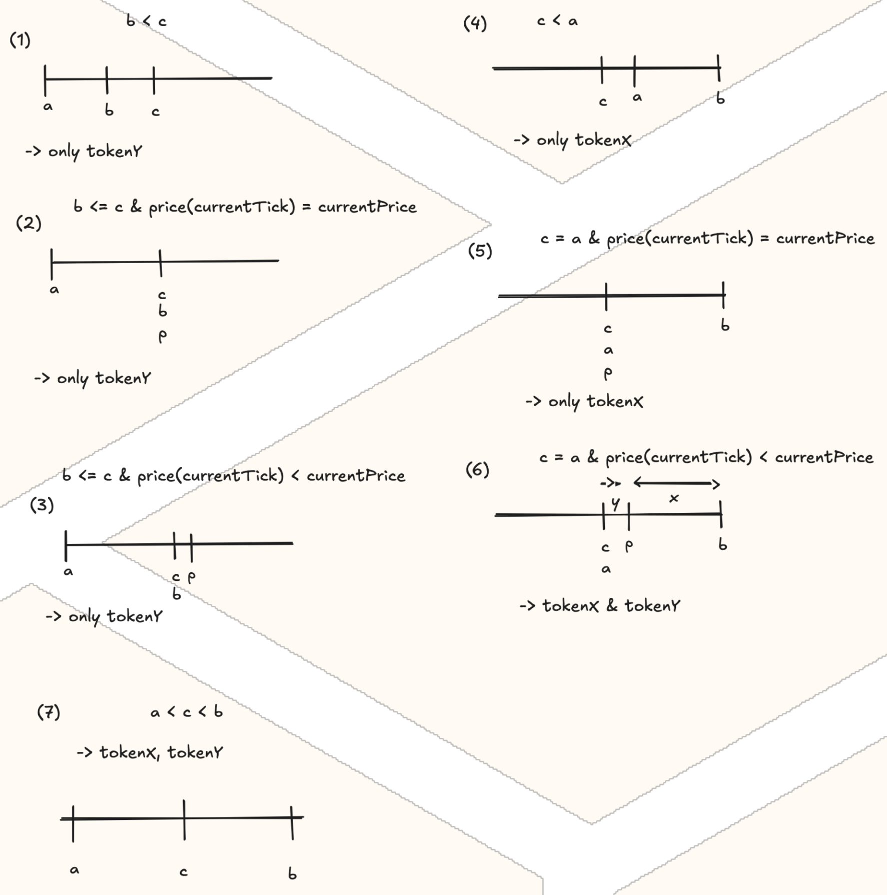
See remix example:


getAmountsForLiquidity

```
bottomTick: 0
topTick: 10
liquidityDelta: 10031232131
currentTick: 0
currentPrice: '922816251426433759354395O336
```

```
0: uint256: amount0 5014112
1: uint256: amount1 0
2: int128: globalLiquidityDelta 10031232131
```

We have furthermore illustrated all different possibilities:

	 (1) b < c a b c -> only tokenY (2) b <= c & price(currentTick) = currentPrice a c b p -> only tokenY (3) b <= c & price(currentTick) < currentPrice a c p b -> only tokenY (4) c < a c a b -> only tokenX (5) c = a & price(currentTick) = currentPrice c a p b -> only tokenX (6) c = a & price(currentTick) < currentPrice y x c a p b -> tokenX & tokenY (7) a < c < b -> tokenX, tokenY a c b
Recommendations	Consider if it is ever desired to add liquidity to that specific edge-case, if yes, consider simply incorporating this edge-case as a condition for the _mintPosition function. Alternatively, this issue can be acknowledged.
Comments / Resolution	Acknowledged.

Issue_17	Deposit requirement can result in temporarily DoS
Severity	**Low**
Description	The following check is incorporated within the deposit function: *require(pool0 > 0 || pool1 > 0 || _totalSupply == 0, "IV.deposit: empty");* While it is generally a valid invariant that totalSupply != 0 means the pool has funds, due to the incorporation of the withdrawal fee, this invariant does not hold in the specific edge-case where the vault has outstanding shares but the withdrawal fee is larger than the vault's balance, resulting in the DoS of deposits.
Recommendations	Since this is such a rare edge-case, it is very unlikely to happen, thus we do not recommend a change.
Comments / Resolution	Acknowledged.

Issue_18	Usage of tick for SPOT price calculation is inaccurate
Severity	**Low**
Description	Within the deposit function, the SPOT price is fetched by using the currentTick of a pool. Depending on tickSpacing, which is for most DEXes between 60 and 100, the price can be up to 0.059999 to 0.0999% different from the current tick. In the scenario where the price is lower/higher than the tick, the returned SPOT price which is based on the tick will be higher than the actual SPOT price of the pair, which then will result in a falsified calculation.
Recommendations	Consider if it makes sense to refactor this function and determine the actual currentPrice which then is used to quote the token amount.
Comments / Resolution	Acknowledged. The client added the following comment: The difference between the spot price and the price derived from the current tick is minimal and does not significantly impact the accuracy of the hysteresis check or the valuation of deposits. This negligible discrepancy ensures that both processes remain reliable.

Issue_19	Withdrawals can result in DoS due to the MIN_SHARES implementation
Severity	**Low**
Description	Currently, on withdrawal the vault checks that the totalSupply is always greater than MIN_SHARES, or equal to 0. This is added to prevent any inflation attack. However, if we take an edge case where: - Alice receives 1000 shares - Bob receives 1000 shares - Alice withdraws 500 shares - Bob withdraws 500 shares Now both users are stuck unless one of them transfer its shares to the other. This issue was raised as low because it is very unlikely, and most of the time 1000 shares will be worth almost nothing. However, it is still possible to inflate shares so that 1000 shares is not a dust amount anymore (eg, someone deposits 1 token0, withdraw all its shares minus 1000, and then transfer 1000 token0 directly to the vault, now 1 share is equal to 1 token0).
Recommendations	If this poses an issue, consider forcing the owner to execute the first deposit which is never withdrawn and thus ensuring that no one can lock any amount of shares.
Comments / Resolution	Acknowledged. Client indicated that a small deposit on the beginning will be made.

Issue_20	Lack of deviation check during withdraw
Severity	**Low**
Description	The withdraw function calculates the proportional amount of the liquidity position for a user based on the provided shares and calculates what the user will receive as tokenX and tokenY. Currently, there is no price deviation check at the beginning of the withdraw function. While that is generally not considered as mandatory because every swap will actually favor the LP in terms of USD value, it is possible for a user to swap before the withdraw function such that the pool only consists of tokenX, then withdraws and swaps back. While that means the nominal USD value of the pool will increase and the user will lose a small amount of tokens, it also will result in the pool to end up with more tokenY than tokenX, which is undesired by the client.
Recommendations	Consider if such a validation is desired or if this scenario can be accepted.
Comments / Resolution	Acknowledged. The client added the following comment: We acknowledge the identified scenario and find it acceptable under our current design philosophy. While it's true that a user could perform a sequence of actions—specifically withdrawing, then swapping out of the pool—to momentarily alter the composition of the underlying assets, the economic outcome does not harm the vault or other LPs. In fact, it results in the manipulator spending more capital than they would have through a straightforward withdrawal and one-directional swap to a single asset. The mechanism ensures that the manipulator cannot leave the vault with fewer tokens than the market deems appropriate post-swap, as they are constrained by the on-curve liquidity and market response. Over time, these attempts translate to a net benefit for remaining LPs,

	as the manipulator's added complexity and costs effectively funnel more value into the pool (and therefore vault) than a standard, more transparent withdrawal strategy would. Given that each swap ultimately aims to maintain or enhance the pool's overall USD value, and any attempt to force undesirable balances involves added expense to the manipulator, we are comfortable with this behavior. It is effectively self-regulating and beneficial for the other LPs.

Issue_21	Fee recipient blacklisting can result in DoS of all functions
Severity	**Low**
Description	Upon the _distributeFees function a push over pull patter is implemented to transfer out fees to the different fee recipients. If one of these recipients is blacklisted for token0 or token1, this call will revert and thus results in a DoS of all major functions.
Recommendations	Consider if that exposes an issue, if yes consider implementing a pull over push pattern. However, in such a scenario the owner can also intervene and change the recipient addresses, hence it can be safely acknowledged.
Comments / Resolution	Acknowledged. In the unlikely event that a fee recipient is blacklisted, the vault owner retains the ability to replace the recipient, ensuring that the system can continue operating.

Issue_22	Payer parameter within algebraSwapCallback is redundant
Severity	**Informational**

Description

The algebraSwapCallback function is invoked by the pair to retrieve the corresponding input amount. The data parameter includes the address of the payer, which is however always address(this) in that scenario:

```
IAlgebraPool(pool).swap(
        address(this),
        swapQuantity > 0,
        swapQuantity > 0 ? swapQuantity : -swapQuantity,
        swapQuantity > 0 ? UV3Math.MIN_SQRT_RATIO + 1 : UV3Math.MAX_SQRT_RATIO - 1,
        abi.encode(address(this))
);
```

Hence, the else callpath is redundant as it is never being triggered:

```
if (amount0Delta > 0) {
if (payer == address(this)) {
        IERC20(token0).safeTransfer(msg.sender, uint256(amount0Delta));
} else {
        IERC20(token0).safeTransferFrom(payer, msg.sender, uint256(amount0Delta));
}
} else if (amount1Delta > 0) {
if (payer == address(this)) {
        IERC20(token1).safeTransfer(msg.sender, uint256(amount1Delta));
} else {
        IERC20(token1).safeTransferFrom(payer, msg.sender, uint256(amount1Delta));
}
```

	}
Recommendations	Consider simplifying the function.
Comments / Resolution	Resolved.

Issue_23	Rebalance frontrun can result in vault loss
Severity	**Informational**
Description	The rebalance function allows the owner to completely remove the liquidity from the basePosition and limitPosition and re-add it to two new positions based on the desired range. This function is usually meant to be called by the VaultSlippageCheckV2_1 function which ensures that the tick is not too deviated from the expected tick. This can result in issues if the pair is a stable pair and the range is not **very** strict. PoC: > Position is within tick 1 and 10 > currentTick = 5 > tokenX = 10; tokenY = 10 a) Rebalance is triggered with a range +-10 b) A malicious user swaps Y -> X; provides 20 tokenY and gets 10 tokenX out. The position is now as follows: > [1;10]

	> 30 tokenY > currentTick = 15 c) The rebalance is executed and 10 idle tokenY are added to the position, which results in the following position: > [1;10] > 40 tokenY > currentTick = 15 d) Malicious user swaps 10x back to the pair, receiving > 20 tokenY due to the increased liquidity density due to the liquidity addition. The vault effectively experienced a loss and the exploiter gained profit. *This issue is only marked as informational severity since the rebalance function is usually intended to be called by the VaultSlippageCheckV2_1 contract.
Recommendations	Consider always choosing a strict range.
Comments / Resolution	Acknowledged. The client added the following comment: Rebalances are executed either by the strategies or through the VaultSlippageCheck contracts. In both cases, the rebalance process is safeguarded against front-running.

Issue_24	shares reference is misleading
Severity	**Informational**
Description	Within the deposit function, the to received amount of shares is calculated as follows: a) Calculate deposit0PricedInToken1 b) Add deposit1 to deposit0PricedinToken1 c) Calculate pool0PricedInToken1 d) Calculate shares as follows: ***shares*** *= shares * totalSupply / (pool0PricedInToken1 + pool)* The problem hereby is that the actual shares declaration is only correct where it is highlighted. The previous share declaration should just be referenced as fullDepositInToken1, as it is (deposit0PricedInToken1 + deposit1).
Recommendations	Consider renaming the variable names.
Comments / Resolution	Acknowledged.

Issue_25	Deposit manipulation outcome in case of direct interaction
Severity	**Informational**
Description	The deposit function calculates a user's deposit always in tokenY based on the SPOT or TWAP price (whatever is lower). It further compares a user's deposit in tokenY to the overall vault value in tokenY and mints the proportional amount of shares to the user. A hysteresis safeguard is implemented which reverts if the SPOT price deviates too much from the TWAP price: *// Check price manipulation* *uint256 delta = (price > twap)* *? price.sub(twap).mul(PRECISION).div(price)* *: twap.sub(price).mul(PRECISION).div(twap);* *if (delta > hysteresis) require(checkHysteresis(), "IV.deposit: try later");* All these safeguards which are in place are sufficient to ensure users cannot steal funds from the vault. However, under specific circumstances, such as: a) Reasonable hysteresis value b) Reasonable position range c) Direct interaction with the vault It is possible to slightly manipulate the pool state while still resulting in a successful deposit such that the depositor will experience a disadvantage. PoC: 1. User detects a deposit in the mempool and frontruns the pool to receive such a state that the majority of the vaults position is in tokenY and the price is larger than before (while still being within the

	hysteresis range) 2. Depositor now deposits tokenY only which means the deposit doesn't benefit from a (slightly) increased price 3. Shares are calculated by using amountY from the vault which is effectively larger than amountX * price + amountY within the initial state (due to the swap) 4. The depositor thus experienced a disadvantage from the swap, receiving less shares.
Recommendations	This issue can be safely acknowledged due to the fact that it is intended to deposit via the VaultSlippageCheckV2_1 contract paired with a strict hysteresis check.
Comments / Resolution	Acknowledged. The client added the following comment: User deposits are protected through the ICHIVaultDepositGuard contract.

Issue_26	Incorrect divisor in delta calculation
Severity	**Informational**
Description	Currently, the delta is determined as follows: *// Check price manipulation* *uint256 delta = (price > twap)* *? price.sub(twap).mul(PRECISION).div(price)* *: twap.sub(price).mul(PRECISION).div(twap);* which corresponds in: a) (price - twap) * 1e18 / price b) (twap-price) * 1e18 / twap The main issue is that the denominator is the higher of both values which will represent a falsified deviation. PoC: price = 100e18 twap = 200e18 (200e18-100e18) * 1e18 / 200e18 = 0.5e18 Now the question is whether it is desired to get the deviation from price to twap, which would be 100% or the deviation from twap to price, which would be 50%. Usually with deviations, it is considered to be restrictive rather than loose which means the first example is desired and hence the lower divisor should be used.
Recommendations	Consider if that is desired, if not consider using the lower denominator.

Comments / Resolution	Acknowledged. The client added the following comment: The use for a larger denominator is acceptable in this case

Issue_27	Erroneous return value within baseLower and baseUpper in case of no existing positions
Severity	**Informational**
Description	The baseLower and baseUpper functions (as well as limitLower and limitUpper) return the corresponding tick for a positionId. If the positionId is however zero, that means there is no existing liquidity position. However, it still returns zero for the tick which is in fact a valid value.
Recommendations	Consider simply reverting if there is no active position.
Comments / Resolution	Acknowledged, the client added the following comment: Returning zeroes from the view function instead of reverting aligns with the design pattern we've adopted, providing a more flexible and streamlined approach. This method facilitates easier implementation of rebalancing strategies, as it avoids unnecessary transaction failures and simplifies the flow of logic.

VaultSlippageCheckV2_1

The VaultSlippageCheckV2_1 contract is an auxiliary contract which allows the owner to invoke the rebalance function with a slippage check for the currentTick.

This contract was developed to counter an exploit where a malicious user could frontrun the rebalance call to bring the pool to a suboptimal price which then allows stealing tokens from the vault reserves by swapping back after the rebalance.

Privileged Functions

- transferOwnership
- renounceOwnership
- rebalance

Issue_28	Unnecessary complex range logic due to int24 type of range
Severity	**Informational**
Description	The rebalance function exposes the int24 range parameter which allows the owner to provide a range for the deviation from the expected tick. This parameter can be positive and negative. While the validation still covers both scenarios, this is redundant, as a range of 10 still perfectly covers +-10.
Recommendations	Consider changing the range type to a proper uint value.
Comments / Resolution	Acknowledged.

VaultSlippageCheckV2

The VaultSlippageCheckV2 contract is similar to the VaultSlippageCheckV2_1 contract with the only difference that an additional check is being executed which prevents rebalances if a swap has happened already in the current block

Privileged Functions

- transferOwnership
- renounceOwnership
- rebalance

Issue_29	Unnecessary complex range logic due to int24 type of range
Severity	**Informational**
Description	The rebalance function exposes the int24 range parameter which allows the owner to provide a range for the deviation from the expected tick. This parameter can be positive and negative. While the validation still covers both scenarios, this is redundant, as a range of 10 still perfectly covers +-10.
Recommendations	Consider changing the range type to a proper uint value.
Comments / Resolution	Acknowledged.

ICHIVaultFactory

The ICHIVaultFactory allows the permissionless deployment of ICHIVault instances. Each address can create their own vault with the following settings:

a) tokenA: tokenX
b) tokenB: tokenX
c) allowTokenA: tokenX deposits allowed?
d) allowTokenB: tokenY deposits allowed?

Each vault is then stored in the getICHIVault mapping corresponding to the created key.

The factory owner can furthermore adjust multiple variables such as the fees and the fee recipients which will be reflected on all deployed ICHIVault iterations.

Appendix: Key Creation

Each vault has a unique identifier which is corresponding to the key. The key is derived as follows:

key = keccak256(abi.encodePacked(deployer, token0, token1, allowToken0, allowToken1));

The key is furthermore used to ensure a deployer cannot deploy the same vault twice.

Appendix: Default Fee Split

The default fee split is defined by three main parameters in the contract: baseFee, ammFee, and baseFeeSplit. By default, these are set as follows:

- ammFee = 0%
- baseFee = 20% (expressed as 2 * 10^17, which equals 0.2 or 20% of the fees)
- baseFeeSplit = 50% (expressed as 5 * 10^17, which equals 0.5 or 50%)

Here's how the split works in practice:

1. **Total Fees Accumulated:**
   The ICHIVaults collect certain swap fees from the underlying liquidity operations.
2. **Applying ammFee and baseFee:**
   Out of the total collected fees, a portion defined by ammFee is allocated to the AMM fee recipient. Another portion defined by baseFee is allocated for further distribution between the feeRecipient and the affiliate.
   Since by default ammFee = 0%, all the considered portion is effectively from the baseFee share (20%).
3. **Splitting the baseFee Portion:**
   Once the baseFee share (20%) is isolated, it is then split between the feeRecipient and the affiliate according to baseFeeSplit.
   With a baseFeeSplit of 50%, the 20% baseFee is divided equally:
   - 10% (half of 20%) goes to the feeRecipient.
   - The other 10% goes to the affiliate.

Privileged Functions

- transferOwnership
- renounceOwnership
- setFeeRecipient
- setAmmFee
- setBaseFee
- setBaseFeeSplit

No issues found.