security

BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT

GU

September 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	GU Exchange
Website	gu.exchange
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/GUFactory/live-contracts/tree/ad347d571d8a318fa794631cbebfcc7cf8319cbb/src
Resolution 1	https://github.com/GUFactory/live-contracts/tree/4a3a21459a614e7fbe366fe64f52207ace231c3c/src
Resolution 2	https://github.com/GUFactory/live-contracts/tree/43f7828fda8c0eabcb95f1e7027f145469f8654f/src
Resolution 3	https://github.com/GUFactory/live-contracts/tree/7b6dd40298b76e6199b4733b9ef368ae03ac7678/src
Resolution 4	https://github.com/GUFactory/live-contracts/tree/bdc3d1a71d663fb79525146a8d70e0c03342cc8e/src

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	5	5		
Medium	11	9		
Low	6	3		3
Informational	5	1		3
Governance				
Total	27	18		6

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

2. Detection

Given the high number of resolution rounds, we strongly recommend that the team proceed with an additional peer review before deployment to further ensure the safety of the codebase.

GuFactory

The GuFactory is a simple factory contract which allows the permissionless GuCoin token deployment against a nominal fee in ETH by invoking the deploy function.

The contract owner has the privilege to set several important variables, namely:

- bondingCurve: Determines the curve which is used for buy and sells during the initial period
- virtualReserves: Determines the ETH balance which backs the initialSupply (only virtually)
- initialSupply: Determines the initial GU balance
- maxSupply: Determines the supply at the end of the bonding curve period
- protocolFee: Fee which is transferred to GuFactory during LP seeding
- creatorFee: Fee which is transferred to token creator during LP seeding
- creationFee: Fee which must be provided as msg.value for token deployment

Furthermore, the owner can withdraw any fees which have been accrued in the contract during token deployments.

Privileged Functions

- withdraw
- setBondingCurve
- updateInitialParams
- updateFees).

Issue_01	Lack of validation against duplicate token creations
Severity	**Medium**
Description	The deploy function allows for the permissionless deployment of new GuCoin tokens with the following parameters: - name - symbol - description - image There is currently no validation that a token with these parameters has not already been created. This can be abused by malicious actors to deploy a token with the same parameters as a famous previously/to be deployed token and trick users into buying the wrong token.
Recommendations	Consider if it is desired to prevent the creation of tokens with the same parameters. It has to be noted that this could result in a DoS (although the likelihood is low because a fee has to be paid for token deployments).
Comments / Resolution	Resolved, such a check has been implemented. It has to be noted that this can be used to prevent the creation of legit tokens via frontrunning, depending on the fee, this is however negligible.

Issue_02	marketcap calculation is incorrect
Severity	**Low**

Description	The marketcap determination is done as follows:
	
	uint256 price = bondingCurve.getAmountOutSell(address(gu), 1 ether);
	*uint256 marketCap = price * maxSupply;*
	
	This is simply incorrect as it multiplies the current price (at beginning of the bonding curve) with maxSupply instead of the final price.

Recommendations	Consider using the final price instead. if we however assume to calculate the marketcap at the time of initialization, one should use the current price multiplied with the initial supply.
Comments / Resolution	Resolved, the marketCap for the initial state is now correctly represented.

Issue_03	Lack of zero check for _initialSupply
Severity	**Informational**
Description	The _updateInitialParams function does not check if _initialSupply is zero. In such a scenario where it is accidentally set to zero, the initialPrice calculation will revert.
Recommendations	Consider incorporating the mentioned check.
Comments / Resolution	Resolved.

GuCoin

The GuCoin contract is an extension of LayerZero's OFT token contract with built-in logic to support buys and sales during the initial bonding curve period and a LP seed mechanism. It is solely meant to be deployed by the GuFactory contract and the default contract owner will be the factory owner.

Whenever users buy or sell this token via the GuBondingCurve contract, the mint and burnCurve functions are invoked which will either mint tokens to the recipient or burn tokens from the recipient.

During the initial bonding time, which is until the desired MAX_SUPPLY has been reached, tokens are non-transferrable, which has the background to prevent any price manipulations on the pool contract. (The same applies for cross-chain transfers)

Once the MAX_SUPPLY has been reached, the LP tokens will be created which then allows for selling and buying into the pool. It is furthermore important to mention that three different fees are taken during the _seedLP function from the ETH balance:

a) Fee to the token creator
b) Fee to the factory
c) Fee to the _seedLP function caller

While these fees do not have an impact on the pool's price (unlike for UNIV2 implementations), they will result in less liquidity for the ETH side range.

Appendix: Liquidity Range Determination

Once the bonding curve period has passed (which is whenever the MAX_SUPPLY has been reached). The LP will be automatically created by providing token0 below the pair price and token1 above the pair price.

There are two scenarios based on the token sortings.

Scenario 1: GU/WETH

This scenario is triggered whenever the GU token address is smaller than the WETH token address and it denominates the GU price in WETH.

Example: Price = 10e18
-> 1e18 GU is worth 10e18 WETH
-> an increasing price reflects an increasing GU value

This scenario also aligns with our bonding curve price mechanism as with continuous progress the GU price increases. This means that seedTick > initialTick because the initialPrice is for example 1e18 and the seedPrice is for example 10e18.

Token0Range is determined as follows:

lowerTick: seedTick / TICK_SPACING * TICK_SPACING + TICK_SPACING

upperTick: MAX_TICK / TICK_SPACING * TICK_SPACING

Using this methodology, it is ensured that token0 is only provided above the seedTick.

Invariant: lowerTick >= seedTick

Token1Range is determined as follows:

lowerTick: initialTick / TICK_SPACING * TICK_SPACING

upperTick: seedTick / TICK_SPACING * TICK_SPACING - TICK_SPACING

Using this methodology, it is ensured that token1 is only provided below the seedTick.

Invariant: seedTick <= upperTick

Scenario 2: WETH/GU

This scenario is triggered whenever the WETH token address is smaller than the GU token address and it denominates the WETH price in GU.

Example: Price = 10e18
-> 1e18 ETH is worth 10e18 GU
-> an increasing price reflects a decreasing GU value

This scenario does not align with our bonding curve as it basically represents the inverse price. If the bonding curve progresses, this will increase the GU value and decrease the WETH value. This means that initialTick > seedTick because the initialPrice is for example 10e18 and the seedPrice is for example 1e18 (because the inverse is fetched, more about this specific scenario can be found in both price determination Appendix)

Token0Range is determined as follows:

lowerTick: seedTick / TICK_SPACING * TICK_SPACING + TICK_SPACING

upperTick: initialTick / TICK_SPACING * TICK_SPACING

Using this methodology, it is ensured that token0 is only provided above the seedTick.

Invariant: lowerTick >= seedTick

Token1Range is determined as follows:

lowerTick: MIN_TICK / TICK_SPACING * TICK_SPACING

upperTick: seedTick / TICK_SPACING * TICK_SPACING - TICK_SPACING

Using this methodology, it is ensured that token1 is only provided below the seedTick.

Invariant: upperTick <= seedTick

Appendix: Initial/Listing Price Determination

To determine the correct liquidity ranges, it is essential to add liquidity for token0 above the pool's price and liquidity for token1 below the pool's price. Whenever the token is deployed, it will automatically invoke createAndInitializePoolIfNecessary on Algebra's Position Manager. This will then ensure that the price is immutable until liquidity is added.

Theoretically, one could simply provide liquidity for token0 from seedTick to MAX_TICK and liquidity for token0 from MIN_TICK to seedTick (GU/WETH). However, in an effort to benefit from Algebra's concentrated liquidity model, also the initialPrice (which is supposed to be the beginning price of the curve) is incorporated into the range determination. We have already explained in the above appendix how the range is determined in detail. This Appendix explains the initial and listing price determination.

Initial Price Determination:

As explained already, the initial price is supposed to be the beginning price of the bonding curve. It is simply calculated within the _computeSqrtPriceAndTick function via the initial supply and the VIRTUAL_BALANCE (the ETH amount which is supposed to virtually back the initial supply).

The math is as follows:

Scenario 1: GU/WETH
sqrtX96Price = sqrt((ethAmount * 2 ** 36) / GUSupply) * 2 ** 78

Scenario 2: WETH/GU
sqrtX96Price = sqrt(GUSupply / ethAmount) * 2 ** 96

The goal of this math is to properly reflect sqrtX96Price based on the totalSupply() at the time of token deployment and VIRTUAL_BALANCE.

As elaborated, in scenario 1 (GU/WETH) initialPrice is smaller than seedPrice because with bonding curve progress the GU token becomes more valuable (the ratio of GU/WETH shifts towards WETH). We have this formally verified as follows with example values:

Pair 1: GU/WETH

Case 1 (initialPrice):

GuSupply = 100e18
WETHSupply = 10e18

Price is 0.1x96
> Given 1e18 GU, receive 0.1e18 WETH (ignore slippage)

Case 2 (seedPrice):

GuSupply = 5000e18
WETHSupply = 25000e18

Price is 5x96
> Given 1e18 GU, receive 5 WETH (ignore slippage)

In Pair 2, (WETH/GU) initialPrice is higher than seedPrice because with bonding curve progress, the ratio from WETH/GU shifts towards WETH, making GU more valuable. However, because this pair represents the price inverse, this will decrease seedPrice over time, making it necessary to allocate token0 above the seedPrice and token1 between initialPrice and seedPrice. This is correctly handled in the range determination process.

Pair 2: WETH/GU

Case 1(initialPrice):

WETHSupply = 10e18
GuSupply = 100e18

Price is 10x96
> Given 1e18 GU, receive 0.1e18 WETH (ignore slippage)

Case 2 (seedPrice):

WETHSupply = 25000e18
GuSupply = 5000e18

Price is 0.2x96
> Given 1e18 GU, receive 5e18 WETH (ignore slippage)

Appendix: OFT Integration

The GuCoin contract leverages LayerZero's OFT standard which allows for simple cross-chain transfers via the deployed LayerZero Endpoint. Users can simply invoke the send function to initiate a cross-chain transfer to the desired EID (considering it has a corresponding peer set).

The flow for cross-chain transfers is as follows:

```
> OFT.send(_sendParams, _fee, _refundAddress)
        -> _debit(_amountLd, _minAmountLd, _dstEid)
                -> _debitView(_amountLd, _minAmountLd, _dstEid)
                -> _burn(msg.sender, amountSendLd)
        -> _buildMsgAndOptions(_sendParam, amountReceivedLd)
        -> _lzSend(_eid, message, options, MessagingFee)
                -> _payNative(fee.nativeFee)
                -> Endpoint.send(MessagingParams, _refundAddress)

> Endpoint.send(MessagingParams, _refundAddress)
        -> _send(msg.sender, _params)
                -> _outbound(_sender, dstEid, receiver)
                -> getSendLibrary(_sender, dstEid)
                -> SendLibrary.send
        -> _payNative(nativeFee, suppliedNative, _sendLibrary, _refundAddress)
```

> OFFCHAIN

> Endpoint.verify(Origin, _receiver, _payloadHash)

-> _inbound(_origin, _receiver, _payloadHash)

> Endpoint.lzReceive(_origin, _receiver, _guid, _message, _extraData)

-> _clearPayload(_receiver, _srcEid, _sender, _nonce, _payload)

-> OFT.lzReceive(_origin, _guid, _message, msg.sender, _extraData)

> OFT.lzReceive(_origin, _guid, _message, _executor, _extraData)

```
-> _lzReceive(_origin, _guid, _message, _executor, _extraData)
        -> _credit(toAddress, amountLD, srcEid)
                -> _mint(_to, _amountLD)
```

Privileged Functions

- transferOwnership
- renounceOwnership
- setPeer
- setDelegate
- setPreCrime
- setEnforcedOptions
- setMsgInspector

Issue_04	Determination of seedPrice potentially allows for flash-thefting funds from bonding curve
Severity	**High**
Description	Within the constructor, the _computeSqrtPriceAndTick function is invoked to calculate the following parameters: sqrtX96Price tick sqrtX96Price is calculated for two scenarios: GU / WETH pair: _marketCap / _supply WETH / GU pair: _supply / _marketCap This approach is simply incorrect because there is no guarantee this aligns with the price of the bonding curve at the current state. For example, the initial tick is calculated using accETH and totalSupply() after the initial minting with the goal to represent the initial price for the first purchase of 1 nominal token via the bonding curve. Similar to this, the seed tick is calculated using (TARGET_ETH - virtualReserves) and (MAX_SUPPLY - initialSupply), which simply corresponds to the ETH amount which needs to be provided to fulfill the bonding curve period and the corresponding distributed supply. While these calculations are technically correct as they reflect the price

	based on supply/reserves*, they are not forcefully aligned on the bonding curve price. This will expose a large issue if the final bonding curve price is lower than the seedPrice, as the last buyer can now effectively flash-theft tokens from the pair by immediately dumping his purchased tokens for a profit. Of course this also goes the other way around where users can buy cheaper from the pair compared to the last bonding curve purchase. Ideally, it's a neatless transition *Additionally, the calculation doesn't incorporate the real circulating supply plus all reserves (virtually). A correct solution would be to incorporate the initialSupply as well as the VIRTUAL_BALANCE to reflect the “fair” price
Recommendations	Consider fetching prices from the bonding curve for the corresponding state, ensuring a smooth transition from the bonding curve period to the lp period. This change must be carefully validated in accordance with the bonding curve math audit, we suggest to include the additional validation of this into the math audit procedure, as without the bonding curve audit it will not be possible to validate the correctness.
Comments / Resolution	This will be subject to validation within the bonding curve math audit. The status will be updated. Resolved, the seedPrice reflects now the end price on the bonding curve.

Issue_05	Malicious user can initialize a pool with wrong price while token is not yet deployed
Severity	**High**

Description	During the GU token deployment, createAndInitializePoolIfNecessary is invoked with the desired sqrtX96Price:
	
	lp = IPositionManager(POSITION_MANAGER).createAndInitializePoolIfNecessary(token0, token1, sqrtX96Price);
	
	A malicious user can frontrun this call/the whole token deployment (due to the deterministic nature of the token address and corresponding pool), setting an incorrect sqrtX96Price. This in turn will result in the createAndInitializePoolIfNecessary call to not change the pool state because the pool has already been deployed :
	
	else {
	
	uint160 sqrtPriceX96Existing = IAlgebraPool(pool)._getSqrtPrice();
	
	if (sqrtPriceX96Existing == 0) {
	_initializePool(pool, sqrtPriceX96);
	}
	
	Since the sqrtX96Price can be set to any arbitrary value, the determined tokenRange0/tokenRange1 will be completely incorrect.
	
	This will have several undesired side-effects such as one side not being added to the LP pair which results in stuck funds within the GuCoin contract. It will essentially break the whole system irreversibly.

Recommendations	A fix for this issue is non-trivial. The code must be changed as follows: a) Check if such a pool is already existing

	b) If such a pool is already existing, check the current price of the pool c) If the current price of the pool does not match the desired sqrtX96Price, a special deployment path must be crafted: -> Deploy the token with some initial supply to deployer -> Allow the transfer of the token -> Manually arbitrage the price to the desired sqrtX96Price -> Once this is done, continue with the standard business flow **Optionally, a different method can be used:** a) Deploy tokens with a custom salt, this way, users would explicitly need to frontrun a creation instead of being able to deterministically fetch the token address based on the nonce b) Check if the token already has a pool, if yes, simply revert This will still be vulnerable to frontrunning, however, we doubt anyone would exploit this for a longer time No matter what solution will be used, it is of utmost importance to revert if a pool has already been created and the price is different than the desired sqrtX96Price to prevent a loss of funds.
Comments / Resolution	Resolved, a check has been introduced which reverts if the pool is already initialized. On top of that, a salt is incorporated in the token deployment which prevents from determining the token address beforehand. That being said, it is still theoretically possible to determine a token address when the creation is in the mempool, initialize the corresponding token pool and prevent the creation process. However, this issue is more a theoretical than a practical issue because the attacker would need to constantly monitor the mempool and frontrun all creation calls, thus we do not consider it as a problem.

Issue_06	Tick range determination can result in lowerTick >= upperTick for certain edge-cases and weird bonding curve implementations
Severity	**High**

Description	A core invariant when adding liquidity to the Algebra DEX is that lowerTick and upperTick should never be the same tick.
	In certain edge-cases, specifically in these where the initialPrice is not too much different from the seedPrice, it may happen that lowTick accidentally becomes the same as highTick (due to TICK_SPACING deduction).
	
	Consider the following example:
	
	initialPrice = 0.994018
	seedPrice = 1
	
	initialTick = -60
	seedTick = 0
	
	token1Range determination:
	
	lowTick = initialTick / TICK_SPACING * TICK_SPACING
	> -60
	
	highTick = seedTick / TICK_SPACING * TICK_SPACING - TICK_SPACING
	
	> -60
	
	In that scenario, lowTick == highTick and the liquidity addition for token1 reverts:
	
	if (topTick <= bottomTick) revert IAlgebraPoolErrors.topTickLowerOrEqBottomTick();

	Potential other scenarios due to truncation may exist as well. (Consider example if initialTick = -59 and seedTick = 0, which results in highTick > lowTick).
Recommendations	Consider: a) Ensuring that the initial and target parameters are always correctly determined b) Implementing a check which reverts upon token deployment whenever lowerTick >= upperTick is detected during the initial range settings It should furthermore also be checked that initialTick can never be seedTick. However, we doubt that this scenario would ever happen based on initial configurations. One would intentionally need to set the initial config wrong.

Comments / Resolution	Resolved, the following checks have been implemented:
	
	if (initialTick == seedTick) revert InvalidLiquidityRanges();
	
	if (token0LowTick >= token0HighTick || token1LowTick >= token1HighTick) {
	revert InvalidLiquidityRanges();
	}
	
	Failed resolution (3rd commit):
	
	A change has been implemented:
	
	These checks are now not executed upon deployment but only upon LP seeding, which makes the whole practice redundant as the liquidity addition reverts anyways. The initial goal of preventing locked funds by deployment revert is not reached.

	Resolution 3: **Resolved**, check is now executed upon deployment.

Issue_07	Liquidity range calculations are insufficiently thought out
Severity	**Medium**
Description	The liquidity ranges for token0 and token1 are calculated as follows: **SCENARIO 1: GU/WETH** token0 (GU) seedTick <-> MAX_TICK token1 (WETH) initialTick <-> seedTick **SCENARIO 2: WETH/GU** token0 (WETH) seedTick <-> initialTick token0 (GU) initialTick <-> MAX_TICK This exposes the following bugs: a) There is no guarantee that the provided WETH amount will be consumed if the whole outstanding supply is sold. In some scenarios

	(as already discussed with the developer), some WETH remains permanently untouched. b) There is no guarantee that the provided WETH amount will be sufficient to cover the outstanding supply. In some scenarios not all tokens can be sold because the WETH liquidity is already consumed. a&b lies within the nature of how Algebra calculates liquidity and amounts. In an ideal scenario, one would check how much liquidity corresponds to the outstanding GU supply on the desired WETH range and only add the exact WETH amount which converts to the equal liquidity, to ensure that the circulating GU supply can consume all WETH liquidity. However, this process is non-trivial and requires some refactoring effort using Algebra's LiquidityAmounts library: -> Given a predetermined liquidity range; calculate how much liquidity is received by adding the whole leftover GU amount in this range -> getLiquidityForAmount0 -> For the same range, calculate the needed amount of WETH to match this liquidity -> getAmount1ForLiquidity -> Refund the WETH which was not added to the range However, even this approach is just an example approach as one would need to incorporate scenario b) as well which means the desired range must be decreased if the needed amount of WETH to reach the desired liquidity is not existent in the contract. !!! IMPORTANT !!! These calculations must be done with the ETH post-fee amount, as fees are taken during the _seedLP function which then results in less liquidity for the same range.

	——------------------------——--------------------------——------------------------ c) The range for GU/WETH of token0: seedTick <-> MAX_TICK , respectively for WETH/GU of token1: MIN_TICK <-> seedTick is unreasonable as nobody is willing to pay the corresponding margin prices, which not only results in less efficiency but also in unreasonable high swap slippage.
Recommendations	Consider revisiting this logic and eventually apply necessary changes. Any potential changes must be carefully re-audited in context with Algebra’s integration and potential caveats. It is expected that additional time (outside of the standard resolution round) needs to be allocated to carefully verify the correctness in terms of edge-cases.
Comments / Resolution	The logic for this has been refactored. As per our legal contract, refactorings (even these recommended) are not covered by the resolution round. Additional resources are necessary to validate the fix. The status of this is subject to updates. Resolved, the efficiency has been greatly improved and now relies on a different strategy. However, it still contains minor flaws. Furthermore, it has to be noted that the GU range is still always until the marginal tick (MAX_TICK or MIN_TICK). **Resolved,** minor flaws have been fixed but the marginal ticks are still existent.

Issue_08	_computeSqrtPriceAndTick does not properly work for WETH/GU pairs
Severity	**Medium**
Description	The _computeSqrtPriceAndTick function calculates the price based on a specific _marketCap and _supply. This will not work for the WETH/GU case where _marketCap > _supply because it will result in zero (or precision loss): *sqrtX96Price = uint160((Math.sqrt(_supply / _marketcap) * 2 ** 96));* Also it will result in truncation in certain scenarios. If for example _marketCap = 1e18 and _supply = 10e18, it will return a price of 9 instead of 10. Moreover, it will not work in the GU/WETH case in the scenario where _supply is unreasonably large compared to _marketCap.
Recommendations	Consider scaling these calculations sufficiently to avoid such errors. A dynamic approach should be used to ensure that each scenario is covered: _supply > _marketcap _marketcap > _supply for both GU/WETH and WETH/GU. We recommend additional fuzz tests. One can furthermore introduce assert checks which ensure that **invariants** are always correct. This would prevent incorrect deployment in all edge-cases.
Comments / Resolution	**Failed resolution**, the _computeSqrtPriceAndTick function was adjusted to handle specific edge-cases. However, truncation is still a

	problem during divisions. **Illustrated Example**: GU/WETH _marketCap = 1.9e18 _supply = 1e18 sqrtX96Price = 79228162514264337593543950336 which corresponds to a price of 1. However, the correct result would be a price of 1.9. Moreover, the in the issue described scenario: *Also it will result in truncation in certain scenarios. If for example _marketCap = 1e18 and _supply = 10e18, it will return a price of 9 instead of 10.* Is still present **Resolution 3: Resolved.**

Issue_09	initialTick determination does not necessarily convert to bonding curve start price
Severity	**Medium**
Description	The initialTick is calculated using the initial accETH amount and the initialSupply. This tick is then used to determine the lower (or higher if token pair is inverse) boundary for the range with the goal to properly reflect the starting price of the bonding curve. However, since the bonding curve can be changed, there is no such guarantee that these settings always align. This is similar to "Incorrect determination of seed price potentially allows for flash-thefting funds from bonding curve"
Recommendations	Consider refactoring that approach and fetching the initial price from the bonding curve.

Comments / Resolution	This will be subject to validation within the bonding curve math audit. The status will be updated.
	
	Failed resolution, prices do not align.
	
	PoC:
	
	initialReserves = 1000e18
	initialGu = 1000e18
	MAX_SUPPLY = 5000e18
	targetETH = 5000e18
	
	Calculation of initialPrice via GuLiquidityManager:
	
	initialPrice = initalReserves / (MAX_SUPPLY-initialGu)
	
	> 1000e18 / (5000e18-4000e18)

	> 0.25 **Calculation using CurveMath.saleTargetAmount:** saleTargetAmount(5000000000000000000000, 1000000000000000000000, 1, 1000000000000000000) > 0.2 (199960007998400319) **Resolution 3: Resolved.** **Resolution 4: This was changed again, not resolved.**

Issue_10	Slightly discontinuous liquidity range in edge-case
Severity	**Informational**
Description	The _configureLiquidityRanges function is responsible for crafting token0Range and token1Range. In an optimal scenario, the liquidity range is continuous, for token0Range starting directly at the next possible tick from seedTick and for token1Range. This is not granted in several different edge-cases. We will just highlight one example edge-case to showcase the issue. Determination of token0Range: *token0Ranges.push(* *IPositionManager.LiquidityRange({* *lowTick: seedTick / TICK_SPACING * TICK_SPACING + TICK_SPACING,* *highTick: (initialTick > seedTick ? initialTick : TickMath.MAX_TICK) / TICK_SPACING * TICK_SPACING* *})* *);* Example 1: seedTick is 953 (corresponds to 1.1) lowTick = seedTick / TICK_SPACING * TICK_SPACING + TICK_SPACING > 953 / 60 * 60 + 60 > 900 + 60 This is perfectly fine, it is necessary to add TICK_SPACING on top of the truncated result to ensure lowTick > seedTick Example 2: seedTick is -23028 (corresponds to 0.1)

	lowTick = seedTick / TICK_SPACING * TICK_SPACING + TICK_SPACING > -23028 / 60 * 60 + 60 > -22980 + 60 > -22920 As one can see in our example, the addition of +60 is not necessary in this calculation because the truncation result already brings lowTick > seedTick. The addition of 60 simply results in slightly discontinuous liquidity which worsens swapping experience.
Recommendations	At this point we do not recommend a change because in the worst case, changes could result in some reverts during the liquidity addition event, if not properly secured. Additionally this small empty tick range is not a big deal. However, given this edge-case in mind, we encourage the developer to execute further fuzzing for these cases.
Comments / Resolution	The logic for this has been refactored. As per our legal contract, refactorings (even these recommended) are not covered by the resolution round. Additional resources are necessary to validate the fix. The status of this is subject to updates. Acknowledged.

GuBondingCurve

The GuBondingCurve contract is the entry contract for buying and selling tokens during the initial bonding period.

This contract is deployed as a standalone contract and interacts with tokens based on the input parameter during the buy and sell function calls. Notably, the GuCoin token contract must have the correct bondingCurve address set in an effort to allow buying and selling.

For each buy and sell transaction, a percentual fee based on the provided / received ETH amount is calculated and taken from the user. The fee can be determined via the setTxFee function by the factory owner and can be set up to 5%.

Appendix: supplyLeft excess

The buy function allows users to buy tokens by providing a specific amount of ETH. The corresponding amount of tokens is calculated by the post-fee ETH amount. In the scenario where the received token amount would result in exceeding the MAX_SUPPLY of the token, the following steps are taken:

1. Calculate how much tokens can be purchased until the MAX_SUPPLY is reached
2. Calculate the cost to purchase these tokens
3. Calculate the fee based on the cost
4. Refund the leftover ETH amount after the cost and the fee to the buyer

Appendix: Fee Application

Whenever users buy and sell tokens, a percentual fee is applied as follows:

a) Buying tokens: The fee is pro-rata deducted from the provided amount of ETH. If for example a user provides 100 ETH and the fee is 5%, the user will receive tokens only worth 95 ETH and 5 ETH will stay in the contract to be later claimed by governance:

   uint256 fee = _calculateFee(msg.value);

   *(amount * txFee) / 10_000*

b) Selling tokens: The fee is pro-rata deducted from the received amount of ETH from a token sale. If for example a user receives 100 ETH from a sale and the fee is 5%, the user will receive 95 ETH and 5 ETH will stay in the contract to be later claimed by governance.

   uint256 fee = _calculateFee(amountOut);

   (bool ok,) = msg.sender.call{value: amountOut - fee}("");

Appendix: Bonding Curve

A bonding curve is a mathematical formula that defines the relationship between the price of a token and its total supply. It creates a dynamic pricing mechanism where the cost to buy or sell a token depends on how many tokens are currently in circulation. As more tokens are purchased and the supply increases, the price per token rises according to the curve. Conversely, as tokens are sold and the supply decreases, the price per token falls.

Within this scope, bonding curves are used to automatically calculate the buy and sell prices of tokens during the bonding period of a token launch.

When someone wants to **buy** tokens:

- The bonding curve formula calculates the amount of tokens which are received for a specific provided amount of ETH
- The price per token increases slightly with each purchase, reflecting the new total supply.

When someone wants to **sell** tokens:

- The bonding curve determines how ETH someone will receive when selling a specific amount of tokens.
- The price per token decreases slightly with each sale, again reflecting the adjusted total supply.

Inherent Risks of Bonding Curves

While bonding curves offer an elegant solution for dynamic pricing, they carry inherent risks, especially if there are flaws in the mathematical implementation.

Potential issues could be:

Mathematical Errors: Inaccurate calculations can lead to incorrect pricing, allowing users to exploit these discrepancies.

Rounding Errors: Truncation in solidity's math operations can result in inaccurate outputs which can potentially be exploited.

Example Exploit:

Imagine a scenario where there's a flaw in the bonding curve's math that doesn't correctly account for small incremental sales. Here's how an attacker might exploit this:

Buying Tokens in Bulk:

- The attacker purchases **10 tokens** in a single transaction.
- Due to the bonding curve, the price per token increases slightly as the total supply increases.
- Let's say the total cost for 10 tokens comes to **$100**, so **$10 per token** on average.

Selling Tokens Individually:

- Instead of selling all 10 tokens at once, the attacker sells **1 token at a time** across **10 separate transactions.**
- If there's a flaw in the bonding curve math, each individual sale might not decrease the price per token accurately.
- Suppose the bonding curve overestimates the price for small sales due to rounding errors or incorrect calculations.

Profiting from the Flaw:

- The attacker might receive **$10.1%** for each token sold individually because the price doesn't decrease properly.
- Selling 10 tokens individually yields **$101** in total.
- The attacker originally spent $100 to buy the tokens and now receives $101 from selling them, netting a **$1 profit**.

How to properly secure this mechanism:

While the formal verification of the math can be considered, this may still result in a deviation due to the actual solidity implementation having its own caveats (truncation).

Therefore, we always recommend intensive fuzzing of the buy and sell calculation mechanisms, including the following invariants:

buy(m) + buy(n) >= buy(m+n)

sell(m) + sell(n) >= sell(m+n)

buy(m) + buy(n) >= sell(m + n)

sell(m) + sell(n) <= buy(m + n)

Privileged Functions

- withdraw
- setTxFee

Issue_11	Insufficient input validation within sell function allows for stealing accumulated ETH fees
Severity	**High**
Description	The sell function allows an arbitrary token parameter, this parameter is used on the following occasions: a) getAmountSell: To determine how much ETH one receives when selling a specific amount of token. A user can simply create a malicious token with valid return values for totalSupply() and reserveBalance() to return the desired ETH amount to drain b) token.burnCurve: The malicious token can just implement an empty execution upon the call of this function. c) token.MAX_SUPPLY(): This is just used during the event Using this trivial exploit, one can deploy a malicious token with the needed functions and return values and drain all ETH which is sitting in the contract by invoking the sell function with the malicious token parameter.
Recommendations	Consider validating the token parameter sufficiently. A registry should be implemented to keep track of all deployed tokens.
Comments / Resolution	Resolved, a check has been implemented which ensures that the token is a valid, factory-deployed token.

Issue_12	Bonding curve implementation requires inverse formula
Severity	**Medium**
Description	Currently, the contract allows for the following interactions: a) Buy X ETH worth of GU b) Sell x GU worth for ETH The flawless bonding curve implementation for this mechanism is inherently hard and can result in undesired side-effects. A simpler implementation would be: a) Buy X GU with ETH b) Sell x GU for ETH This is because determining how many tokens a user can receive for a specific amount of ETH requires solving the inverse of the bonding curve equation, which can be complex, especially with non-linear curves. This might necessitate iterative methods or approximations, increasing the risk of errors. This risk is furthermore amplified due to solidity truncating during divisions.
Recommendations	Consider focusing on the token amount specification: 1) Allow users to purchase a specific amount of tokens with the provided msg.value and refund the excess. 2) Allow users to sell a specific amount of tokens (Already applied)
Comments / Resolution	Resolved.

Issue_13	Game-theoretical approach can be used to achieve advantage for LP seeding
Severity	**Low**
Description	The GuCoin contract incentivizes users to seed the LP (achieve the MAX_SUPPLY) by redistributing a share of the fees towards the caller. This competition opens up a game-theoretical exploit by frontrunning a transaction which triggers the LP seeding mechanism by burning some tokens beforehand such that the initial caller won't trigger the LP seeding. Afterwards one can simply invoke the buy function with the leftover amount to receive the seeding rewards.
Recommendations	We do not recommend a change here. This issue can be safely acknowledged.
Comments / Resolution	Acknowledged.

Issue_14	Missing safeguard for GU => TVL mapping
Severity	**Low**
Description	Currently, there is no explicit safeguard which maps the corresponding ETH that backs a token. This can become problematic in the case where the bonding curve is flawed as this means one can potentially drain more ETH than the amount which is backing the token. In fact, such a safeguard is present in each corresponding GuCoin contract as accETH variable. However, this variable includes the VIRTUAL_BALANCE (is not backed), which then allows in case of issues to drain the whole reserves including VIRTUAL_BALANCE, which is then taken from other tokens' reserves.
Recommendations	Consider implementing such a safeguard.
Comments / Resolution	Resolved, such a safeguard has been implemented within the GuCoin contract.

Issue_15	Lack of pausing functionality
Severity	**Low**
Description	Currently, the contract does not expose a pausing functionality. Such a function can be specifically helpful in the context of bonding curve sales, as one can quickly intervene if malicious behavior is spotted. Additionally, one could even think of implementing a transaction limitation which prevents users from buying and immediately selling (with a grace period of a few seconds). This can also be done optionally and toggled off after some time. (Safe Launch Mechanism).
Recommendations	Consider thinking about additional safeguards.
Comments / Resolution	Resolved.

Issue_16	getAmountOutBuySupplyCapped does not consider the fee
Severity	**Informational**
Description	A new function getAmountOutBuySupplyCapped was introduced which serves as a view-function and handles the scenario where a purchase exceeds the reserve. Since the fee is not incorporated with this function, it will be falsified.
Recommendations	Consider incorporating the fee.
Comments / Resolution	Acknowledged.

Issue_17	Selling tokens does not work for contracts without fallback/receive function
Severity	**Informational**
Description	The sell function transfers native ETH to the recipient. This will never work for contracts without a receive/fallback function.
Recommendations	We do not recommend a change but simply recommend to keep this in mind.
Comments / Resolution	Acknowledged.

GuLiquidityManager

The GuLiquidityManager is a standalone contract which incorporates all logic for the correct tick determination. This logic has been externalized from the GuCoin contract towards this contract. Furthermore, the tick determination logic has been changed such that it now follows a different approach than in the initial commit.

Notable, the actual setting of the range, ie:

GU/WETH:

Token0Range is determined as follows:

lowerTick: seedTick / TICK_SPACING * TICK_SPACING + TICK_SPACING

upperTick: MAX_TICK / TICK_SPACING * TICK_SPACING

Using this methodology, it is ensured that token0 is only provided above the seedTick.

Invariant: lowerTick >= seedTick

Token1Range is determined as follows:

lowerTick: initialTick / TICK_SPACING * TICK_SPACING

upperTick: seedTick / TICK_SPACING * TICK_SPACING - TICK_SPACING

Using this methodology, it is ensured that token1 is only provided below the seedTick.

Invariant: seedTick <= upperTick

Is still following the exact same approach. It has just been changed how seedTick and initialTick are derived.

Appendix: Constant Product Formula

The constant product formula is the model which is used in UniswapV2 pools which determines how swaps are executed. The formula is:

x * y = k

Where:

- **x** = the quantity of Token A in the liquidity pool.
- **y** = the quantity of Token B in the liquidity pool.
- **k** = a constant value representing the product of the quantities of the two tokens, which must remain unchanged for every swap.

This relationship implies that the total value (product of the token reserves) in the liquidity pool remains constant after every trade/reserve change. The constant **k** ensures that liquidity always exists on both sides of the trading pair, but the prices will adjust based on the relative supply of each token in the pool.

This formula is used within the CurveMath contract and during the targetETH amount determination.

Appendix: targetETH derivation

To be able to ensure an efficient liquidity range, the targetETH amount must be derived based on the virtualReserves and the MAX_SUPPLY. This is done as follows:

1. Calculate k based on initial reserves:

k = (maxSupply - initialSupply) * virtualReserves

2. Calculate targetETH based on k and MAX_SUPPLY:

targetETH = k / initialSupply + virtualReserves

Following this approach (if bug-free), it is ensured that the targetETH in the corresponding range is aligned with the circulating supply of GU tokens.

Appendix: Price Determinations

Contrary to the initial approach *"Appendix: Initial/Listing Price Determination"* , a new price determination flow was developed which has its root in the bonding curve itself. Prices are simply calculated based on the initial state of the bonding curve and the final state of the bonding curve.

Below we will illustrate different prices based on the sorting of the tokens

Example values (fetched from the testing architecture):

INITIAL_RESERVES = 0.0345e18

INITIAL_SUPPLY = 150 000 000e18

MAX_SUPPLY = 1 000 000 000e18

targetETH = 0.23e18

Initial Price (GU/WETH)

> initialWETH / (MAX_SUPPLY - initialGu)

> 0.000000000040588

> 1 GU is worth 0.000000000040588 WETH

Listing Price (GU/WETH)

> targetETH / initialGu

> 0.000000001533

> 1 GU is worth 0.000000001533

As the bonding curve matures, the GU value increases, which is correlated with the GU/WETH pair price

—-------------------------

Initial Price (WETH/GU)

> (MAX_SUPPLY - initialGu) / initialWETH

> 24637681159

> 24637681159 GU are needed for 1 WETH

Listing Price (WETH/GU)

> initialGu / targetETH

> 652173913

> 652173913 GU are needed for 1 WETH

As the bonding curve matures, the GU value increases, which is inversely correlated to the WETH/GU pair price.

Issue_18	Uniswap’s k invariant is violated
Severity	**Medium**

Description	The targetETH amount is calculated using the k invariant from UniswapV2.
	
	Ideally, this is done with the following steps:
	
	a) Determine k via the known initial reserves
	
	b) Calculate targetETH with the known k and one side of the reserves.
	
	The calculation within the GuLiquidityManager is incorrect as it is done as follows (see PoC below):
	
	*uint256 k = (maxSupply - initialSupply) * virtualReserves / (10 ** 18);*
	*uint256 targetETH = k * (10 ** 18) / initialSupply + virtualReserves;*
	
	—-----------------—-----------------—---------------------------
	
	Example using incorrect formula:
	
	INITIAL_RESERVES = 10e18
	INITIAL_SUPPLY = 150000000e18
	MAX_SUPPLY = 1000000000e18
	TARGET_ETH = to be calculated
	
	*uint256 k = (maxSupply - initialSupply) * virtualReserves / (10 ** 18);*
	*uint256 targetETH = k * (10 ** 18) / initialSupply + virtualReserves;*

k = (1000000000e18-150000000e18) * 10e18 / 1e18
k = 8.5e27

targetETH = 8.5e27 * 1e18 / 150000000e18 + 10e18
targetETH = 66.66666e18

Now we need to ensure the UniswapV2 invariant for the initial and seed state (we use the determination logic from the GuLiquidityManager):

initialPrice = VIRTUAL_BALANCE / (MAX_SUPPLY - INITIAL_SUPPLY)

seedPrice = TARGET_ETH / initialSupply

k (initial state) = 10e18 * (1000000000e18-150000000e18)
k (initial state) = 8.5e45

k (seed state) = 66.666666e18 * 150000000e18
k (seed state) = 1e46

The invariant is therefore **violated.**

——

To ensure that the k invariant is not violated, one can simply adjust the formula slightly:

```
uint256 k = (maxSupply - initialSupply) * virtualReserves
uint256 targetETH = k/ initialSupply
```

k = (1000000000e18-150000000e18) * 10e18 k = 8.5e45

targetETH = 8.5e45 / 150000000e18
targetETH = 56.66666e18

Now we need to ensure the UniswapV2 invariant for the initial and seed state (we use the determination logic from the GuLiquidityManager):

initialPrice = VIRTUAL_BALANCE / (MAX_SUPPLY - INITIAL_SUPPLY)

seedPrice = TARGET_ETH / initialSupply

k (initial state) = 10e18 * (1000000000e18-150000000e18)
k (initial state) = 8.5e45

k (seed state) = 56.666666e18 * 150000000e18
k (seed state) = 8.5e45

Furthermore, we can now prove the correctness of the range using Algebra’s LiquidityAmounts library:

> getLiquidityForAmount1(85935000000000000000000000, 486965041500000000000000000, 56666600000000000000)
> 111951478176998655415652

> getAmount0ForLiquidity(85935000000000000000000000, 486965041500000000000000000, 111951478176998655415652)
> 849999082492218987366642812

The targetETH amount in the range is perfectly aligned to the circulating GU token supply of 850 000 000 tokens, making the liquidity provision perfectly efficient.

Furthermore, the 1e18 truncation is useless.

Recommendations	Consider following the correct displayed formula. If additionally on top the initialPrice is changed as recommended within: "initialPrice does not align with initial price on bonding curve", it is mandatory to re-audit the fixed flow. Fuzzing is explicitly expected after this has been changed.
Comments / Resolution	Resolved, the initialPrice is now determined as follows: VIRTUAL_BALANCE / MAX_SUPPLY using the previous existing k and targetETH calculation, this will result in a constant k. **However, the 1e18 truncation is still existent, which can become an issue if the values used are not rounded numbers. For example:** INITIAL_WETH = 0.123456789456789987e18 INITIAL_GU = 1.123123123123123123e18 MAX_SUPPLY = 123.456789456789456789e18 uint256 k = ((maxSupply - initialSupply) * virtualReserves) / (10**18) -> 15102921688032642006 uint256 targetETH = (k * (10**18)) / initialSupply + virtualReserves; -> 13570710591902965892 initial state: MAX_SUPPLY * VIRTUAL_RESERVE -> 15241578862978105835252783549384371743 final state: targetETH * initialSupply

	-> 152415788629781058349897923895855207 16 Consider simply fixing the k calculation, it shouldn't divide by 1e18 to multiply it again just after.

Issue_19	initialPrice does not align with initial price on bonding curve
Severity	**Medium**

Description	The bonding curve is using UniswapV2's k invariant to determine prices based on the WETH reserves and the GU supply "to be purchased + INITIAL_SUPPLY".
	
	If we now apply this logic, that means that the first purchase would reflect the following initial reserves:
	
	GU = MAX_SUPPLY
	WETH = VIRTUAL_BALANCE
	
	this is due to the fact how purchaseTargetAmount is invoked:
	
	return CurveMath(formula).purchaseTargetAmount(gucoin.MAX_SUPPLY() - gucoin.totalSupply() + gucoin.INITIAL_SUPPLY(), gucoin.reserveBalance(), reserveWeight, amount);
	
	If we compare this with the initialPrice within the GuLiquidityManager:
	
	_computeSqrtPriceAndTick(token, token0, gucoin.VIRTUAL_BALANCE(), gucoin.MAX_SUPPLY() - gucoin.INITIAL_SUPPLY());

	*sqrtX96Price = Math.sqrt((_marketcap * precision) / _supply) * (2 ** 96) / (10 ** 9);*
	
	It becomes clear that this does not align, as here reserves will be:
	
	GU = MAX_SUPPLY - INITIAL_SUPPLY
	WETH = VIRTUAL_BALANCE
	
	Illustrated:
	
	INITIAL_RESERVES = 10e18
	INITIAL_SUPPLY = 150000000e18
	MAX_SUPPLY = 1000000000e18
	
	Price on the bonding curve for 1 wei (getAmountOutBuy): 99 999 999
	
	Price reflected in the GuLiquidityManager by initialPrice = 85 000 000

Recommendations	The fix for this issue would be to simply follow the same initial price calculation as with the bonding curve: INITIAL_RESERVES / MAX_SUPPLY If that issue is fixed together with *“Uniswap’s k invariant is violated”*, the whole flow needs to be re-audited. Therefore, it may be worth a consideration to acknowledge this issue (if the impact is accepted) and simply adjust the targetETH calculation formula. This will be assessed based on the overall changes once the resolution code is provided.
Comments / Resolution	Resolved.

Issue_20	Inefficient WETH liquidity range
Severity	**Medium**

Description	The current way how the targetETH, seedTick and initialTick is calculated is incorrect and will always result in suboptimal efficiency for the WETH range:
	
	Illustrated GU/WETH pair:
	
	initialReserves = 10e18
	initialSupply = 150000000e18
	
	MAX_SUPPLY = 1000000000e18
	TARGET_ETH = 66.666666e18
	
	initialPrice = 0.000000011764 (8593240000000000000000000)
	seedPrice = 0.00000044 (52554000000000000000000000)
	
	calculating liquidity for targetETH in this range using Algebra's LiquidityAmounts library:
	
	getLiquidityForAmount1 = 1189483240494806341194 71
	
	calculate amount0 which is needed to cover the calculated liquidity in the same range:
	
	getAmount0ForLiquidity = 917 361 236e18
	
	This means the provided range can absorb up to 917 361 236e18 GU instead of the circulating amount of MAX_SUPPLY - initialSupply (850 000 000e18).
	
	The result of this is that there will always be some leftover WETH if all 850 000 000 GU tokens are sold into the range.

Recommendations	The root-cause of this deviation is the same root-cause as within *"Uniswap's k invariant is violated"*. Fixing this issue will also fix the efficiency problem. The fix must be carefully validated for GU/WETH and WETH/GU.
Comments / Resolution	Resolved.

Issue_21	Precision loss within _computeSqrtPriceAndTick can result in incorrect sqrtX96Price
Severity	**Medium**
Description	This issue is described within "*_computeSqrtPriceAndTick does not properly work for WETH/GU pairs*" in the GuCoin section.
Recommendations	Consider using sufficient precision for all calculations.
Comments / Resolution	Resolved.

Issue_22	Liquidity ranges are not set upon deployment
Severity	**Medium**
Description	*The root-cause of this issue is incorrect externalization, against our recommendation to not doing this before the audit is finalized:* Contrary to the previous iteration, liquidity ranges are now only determined upon LP seeding within the _configureLiquidityRanges function during the seedLp function. Previously, these have already been determined upon deployment of the token (see: https://github.com/GUFactory/live-contracts/blob/4a3a21459a614e7fbe366fe64f52207ace231c3c/src/GuCoin.sol#L260) As we have already identified, there are multiple scenarios in the case of incorrect initial parameter setting within the factory, where lower and upper ticks can be equal. To combat this issue, we have recommended to add explicit invariant checks and revert upon violation of these. Those invariant checks have been correctly added but these are now only invoked once the LP is seeded: *if (token0LowTick >= token0HighTick || token1LowTick >= token1HighTick) {* *revert InvalidLiquidityRanges();* *}* This is a huge blunder as it **makes these checks literally void** as it would revert anyways upon LP seeding. The original idea was to implement these invariant checks during the contract deployment, which would then **revert immediately and prevent potentially lost funds in the future**.
Recommendations	Consider refactoring the LP seeding mechanism to match the initial

	flow. This includes creating a storage slot for each token's liquidity ranges upon the createAndInitializePool call and then fetching this storage slot upon seedLp. A careful additional check is necessary to ensure it is now flawless.
Comments / Resolution	Resolved.

Issue_23	Fee deduction results in less range efficiency
Severity	**Low**

Description	Within *"Appendix: Price Determinations"*, we have elaborated how the initial and seeding price is determined. Ideally, this is done in such a way that the circulating supply: (MAX_SUPPLY - initialSupply) results in the exact same liquidity as the targetETH in the range. Due to the fact that not the full targetETH is being added to the range but several fees are taken:
	
	*IERC20(WETH).transfer(creator, ethBalance * CREATOR_FEE / 10_000);*
	*IERC20(WETH).transfer(owner(), ethBalance * PROTOCOL_FEE / 10_000);*
	// Set refund fee to half the creator fee
	*IERC20(WETH).transfer(caller, ethBalance * CREATOR_FEE / 20_000);*
	
	This will result in the calculated efficiency never being met.

Recommendations	A fix for this issue is non-trivial because the targetETH is already the perfect corresponding amount to the circulating supply (if the calculation is fixed, see other issues).

	Moreover, since the impact is rather low, we are of the opinion that this issue can be acknowledged.
Comments / Resolution	Acknowledged.

Issue_24	targetETH includes VIRTUAL_BALANCE
Severity	**Low**
Description	The targetETH calculation relies on the inclusion of the VIRTUAL_BALANCE. Due to the fact that the contract however only receives the following ETH amount during the bonding period: targetETH - VIRTUAL_BALANCE This will result in less WETH being added to the range than expected, falsifying the efficiency in the exact same way as *"Fee deduction results in less range efficiency"* (additionally)
Recommendations	We recommend sticking to the current implementation and acknowledging this issue. Fixing this bug would result in another refactoring of the formula.
Comments / Resolution	Acknowledged.

GuLiquidityManager bdc3d1a commit

The latest iteration of the GuLiquidityManager is within the bdc3d1a commit. The following changes have been implemented:

a) The redundant 1e18 truncation within the k and targetETH calculation has been removed

b) The liquidity pool existence check has been outsourced into the _checkIfLiquidityPoolExists function

c) virtualReserves are decreased by 25% whenever the initialPrice is calculated. The reason for this change is to decrease the impact of the virtualReserves, as virtualReserves are not existing in the LP and thus users will inherently experience disadvantages.

The biggest challenge is the offset of the virtualReserves by 25% whenever the initialPrice is calculated, a few notes need to be made on this:

a) The initialPrice does not align with the starting price of the bonding curve

b) The k invariant is violated in terms of pricing

c) Other side-effects may occur for different settings (incorrect pricings, violation of states, etc)

To prevent undesired side-effects, the GU team decided to set the following default settings:

INITIAL_RESERVES: 7.5 ether
INITIAL_SUPPLY: 65_000_000
MAX_SUPPLY: 100_000_000
Protocol fee 1.5% (150)
Creator fee 1.5% (150)
Bonder fee 0.75% (75)

We will conduct a case study to demonstrate the impact of the virtualReserve offset using the default settings.

Scenario 1: No offset

a) Calculation of k:

uint256 k = (maxSupply - initialSupply) * virtualReserves
-> (100000000e18-65000000e18) * 7.5e18
-> 2.625e44

b) Calculation of targetETH:

uint256 targetETH = k / initialSupply + virtualReserves
-> 2.625e44 / 65_000_000e18 + 7.5e18
-> 11,5384615384615e18

c) Calculation of initial pair state:

INITIAL_RESERVES / MAX_SUPPLY

> 7.5e18 / 100_000_000e18

price = 0.000000075

sqrtPriceX96 = 21697500000000000000000000

d) Calculation of final pair state:

Final Pair State:

targetETH / INITIAL_SUPPLY

> 11,5384615384615e18 / 65_000_000e18

price = 00000017751

sqrtPriceX96 = 33380809075959786591714528

e) Calculate liquidity corresponding to (targetETH - virtualReserves) between initialPrice and seedPrice:

LiquidityAmounts.getLiquidityForAmount1(21725727873737242772113352, 33363941666774741885899097, 4038461538461500000)
> 27492181598197486239539

f) Calculate amount corresponding to received liquidity between initialPrice and seedPrice:

LiquidityAmounts.getAmount0ForLiquidity(21725727873737242772113352, 33363941666774741885899097, 27492181598197486239539)
> 34972238373430802381327012 (34_972_223)

As one can see, if 34_972_223 GU tokens are sold, this would result in receiving 4.038 WETH. (For this example, we ignore any swap fee)

The problem now is that not all WETH is being added to the liquidity due to the fee application (see: ***Fee deduction results in less range efficiency***), which means that (depending on the fee size), there will always be some leftover GU tokens after all raised WETH has been taken out from the range. To counter this, the VIRTUAL_RESERVES have been decreased by 25% during the initialPrice determination which increases the range of where WETH is being added and decreases the initialPrice.

Scenario 1: Implemented offset GU/WETH

a) Calculation of k:

uint256 k = (maxSupply - initialSupply) * virtualReserves
-> (100000000e18-65000000e18) * 7.5e18
-> 2.625e44

b) Calculation of targetETH:

uint256 targetETH = k / initialSupply + virtualReserves
-> 2.625e44 / 65_000_000e18 + 7.5e18
-> 11,5384615384615e18

c) Calculation of initial pair state:

INITIAL_RESERVES * 0.75 / MAX_SUPPLY

> 5.625e18 / 100_000_000e18

price = 0.00000005625

sqrtPriceX96 = 18790543303508072947060818

d) Calculation of final pair state:

Final Pair State:

targetETH / INITIAL_SUPPLY

> 11,5384615384615e18 / 65_000_000e18

price = 0.000000177514

sqrtPriceX96 = 33380805571322422036599904

e) Calculate liquidity corresponding to (targetETH - virtualReserves - fees) between initialPrice and seedPrice (modulated):

LiquidityAmounts.getLiquidityForAmount1(18812177021695284541616474, 33363941666774741885899097, 3887019230769193750)
> 21163164662341257107281

f) Calculate amount corresponding to received liquidity between initialPrice and seedPrice (modulated):

LiquidityAmounts.getAmount0ForLiquidity(18812177021695284541616474, 33363941666774741885899097, 21163164662341257107281)
> 38874019374093296297114106 (38_874_019 GU)

As one can see, if 38_874_019 GU tokens are sold, this would result in receiving 3.887 WETH, however, the circulating supply is only 35_000_000 GU.

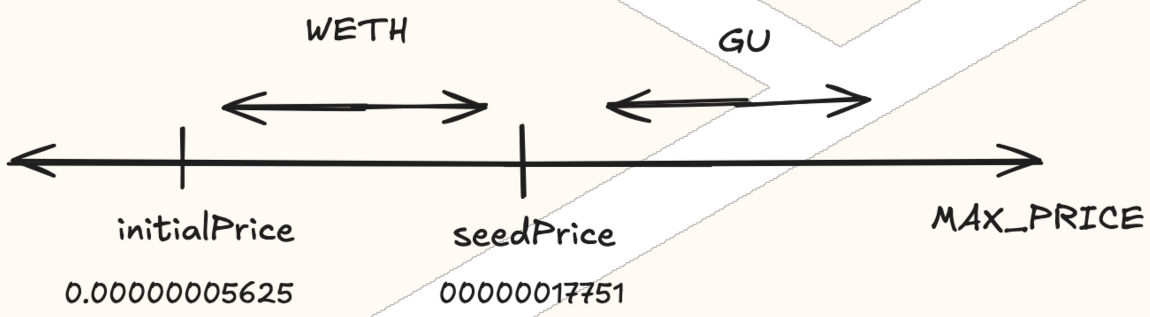
g) Calculate how much WETH is left after 35_000_000 GU has been sold:

getLiquidityForAmount0(19667812734493782430767490, 33363941666774741885899097, 35000000000000000000000000)
> 21165281190460504721332

getAmount1ForLiquidity(19667812734493782430767490, 33363941666774741885899097, 21165281190460504721332)
> 3658830532897154002

(raisedETH - fee) - receivedETH
> 3.887 - 3.658 = 0.229 ETH
> 0.229 WETH is left after the while circulating GU supply was sold.

Illustrated trading range:



Scenario 1: Implemented offset WETH/GU

a) Calculation of k:

uint256 k = (maxSupply - initialSupply) * virtualReserves
-> (100000000e18-65000000e18) * 7.5e18
-> 2.625e44

b) Calculation of targetETH:

uint256 targetETH = k / initialSupply + virtualReserves
-> 2.625e44 / 65_000_000e18 + 7.5e18
-> 11,5384615384615e18

c) Calculation of initial pair state:

MAX_SUPPLY / INITIAL_RESERVES * 0.75

> 100_000_000e18 / 5.625e18

price = 17777778

sqrtPriceX96 = 334025933160138447294381294616576

d) Calculation of final pair state:

Final Pair State:

INITIAL_SUPPLY / targetETH

> 65_000_000e18 / 11.5384615384615e18

price = 5633333

sqrtPriceX96 = 188045224461183414131519308503503

e) Calculate liquidity corresponding to (targetETH - virtualReserves - fees) between initialPrice and seedPrice:

LiquidityAmounts.getLiquidityForAmount0(188140292237643207939482149717954, 333672266008743486154215807002823, 3887019230769230771)
> 2116316466234145867O862

f) Calculate amount corresponding to received liquidity between initialPrice and seedPrice:

LiquidityAmounts.getAmount1ForLiquidity(188140292237643207939482149717954, 333672266008743486154215807002823, 21163164662341458670862)
> 38874019374093666543562498 (38_874_019e18)

As one can see, if 38_874_019 GU tokens are sold, this would result in receiving 3.887 WETH, however, the circulating supply is only 35_000_000 GU.

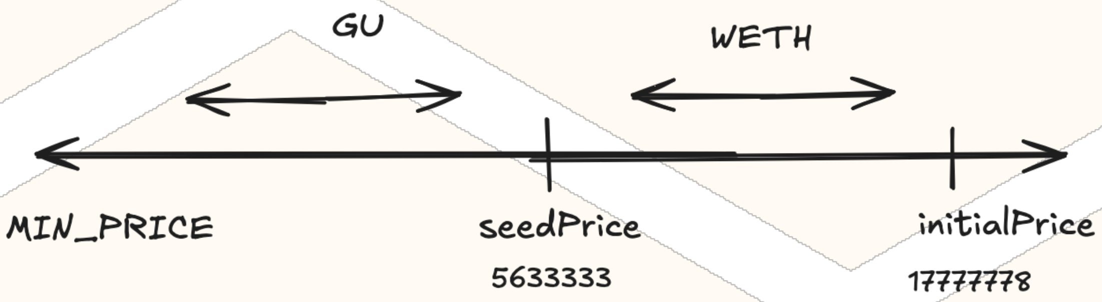
g) Calculate how much WETH is left after 35_000_000 GU has been sold:

getLiquidityForAmount1(188140292237643207939482149717954, 319156065807855739260721268542938, 35000000000000000000000000)
> 21165281190460504721334

getAmount0ForLiquidity(188140292237643207939482149717954, 319156065807855739260721268542938, 21165281190460504721334)
> 3658830532897154002

(raisedETH - fee) - receivedETH
> 3.887 - 3.658 = 0.229 ETH
> 0.229 WETH is left after the while circulating GU supply was sold.

Illustrated trading range:



The following invariants must always be enforced to ensure a smooth trading experience:

INV 1: Selling the circulating GU supply should not result in all WETH being taken from the range

INV 2: The leftover WETH amount after all GU has been sold should not be too high

Finally, we would like to highlight the fact that for the GU/WETH pair, token 1 is still distributed until MAX_TICK and for the WETH/GU pair, token0 is still distributed until MIN_TICK. This can result in undesired slippage and should be carefully tested to be sure it aligns with the intended swapping dynamics.

Issue_25	initialPrice does not align with start of the bonding curve
Severity	**Medium**
Description	Due to the fact that the VIRTUAL_RESERVES are decreased by 25% upon the initialPrice calculation, the resultPrice will be lower than the real initialPrice on the bonding curve. If the whole GU supply of 35_000_000 GU tokens is being sold, this will yield a resultPrice of 0.0000000616244 (see https://github.com/cryptoalgebra/Algebra/blob/master/src/core/contracts/libraries/PriceMovementMath.sol#L129) However, the initial price of the bonding curve is 0.000000075, users may therefore experience a loss which is beyond the initial price of the bonding curve. For WETH/GU: initialPrice on bonding curve: 13333333 resultPrice after swap: 16227328
Recommendations	This issue is inherently present with the offset practice of the VIRTUAL_RESERVES. It cannot be fixed without refactoring the code.
Comments / Resolution	

Issue_26	Increased slippage due to price decrease
Severity	**Informational**
Description	Due to the decreased initialPrice, GU sales will inherently experience a higher slippage because the range is increased, compared to the previous iteration where the range was smaller and thus liquidity more concentrated, this iteration will result in less concentrated liquidity.
Recommendations	This impact is inherently existent due to the range increase, there is no fix for it.
Comments / Resolution	

CurveMath

The CurveMath contract provides utility functions to facilitate calculations for buying and selling specific amounts of tokens. The contract achieves this by using the K-constant invariant from UniswapV2, ensuring that the reserve balances and token supply maintain a consistent ratio.

The main calculations are:

Buy

amountEth = (supply * reserve) / (supply - amountToken) - reserve

amountToken = supply - (supply * reserve) / (reserve + amountEth)

Sell

amountEth = reserve - (supply * reserve) / (supply + amountToken)

Issue_27	Unnecessary roundings can break the curve
Severity	**High**

Description	Consider the following function:
	
	function getK(uint256 _supply, uint256 _reserveBalance) internal pure returns (uint256) {
	    *return _supply * _reserveBalance / (10 ** 18);*
	}
	
	The K constant is divided by 10**18 for no real reason as it is then multiplied again by 10**18 when calculating the amounts:
	
	*uint256 _newReserve = k * (10 ** 18) / _newSupply + 1;*
	
	*uint256 _newSupply = k * (10 ** 18) / _newReserve + 1;*
	
	*uint256 _newReserve = (k * (10 ** 18) / _newSupply) + 1;*
	
	Due to this unnecessary rounding, the calculation will be either:
	- slightly wrong
	- even revert with some specific supplies (for example, if k is rounded down to 0, then the subtraction will most of the time revert)

Recommendations	Consider updating the getK function to return the real K _supply * _reserveBalance and to update the different mentioned lines from the k * 10**18 / XYZ to k / XYZ
Comments / Resolution	Resolved.