

ABRACADABRA MONEY · SECURITY REVIEW

Abracadabra Money PurrSwap Balancer Aggregator.

A security review of PurrSwap Balancer Aggregator.

CLIENT

Abracadabra
Money

PROJECT

PurrSwap Balancer
Aggregator

AUDITORS

Nicholas Chew,
0xCiphky .,
Zdravko Hristov

AUDIT WINDOW

March 5 – March
29, 2025

FINAL REPORT

March 29, 2025

§ 00

Contents.

90 pages: overview, confidence ranking, scope, 72 findings, disclaimer, and about Guardian.

§ 01	Overview	P. 03
§ 02	Confidence Ranking	P. 04
§ 03	Scope & Classifications	P. 05
	Files in Scope	P. 06
§ 04	Findings	P. 07
	Main Review · 60 findings	P. 10
	Remediation Review · 12 findings	P. 76
§ 05	Disclaimer	P. 89
§ 06	About Guardian	P. 90

§ 01

Overview.

Guardian reviewed PurrSwap Balancer Aggregator across 2 review rounds.
The final report includes 7 Critical, 7 High, 18 Medium, 40 Low findings.

AUDIT FIRM	REPOSITORY
Guardian	https://github.com/Abracadabra-money/abracadabra-money-contracts , https://github.com/GuardianOrg/purrswap-contracts-team1
CLIENT	METHODOLOGY
Abracadabra Money	Smart contract · Manual · Main Review · Remediation Review
PROJECT	MAIN REVIEW COMMIT
PurrSwap Balancer Aggregator	484cb7dfddb0
PREPARED BY	
Nicholas Chew, 0xCiphky ., Zdravko Hristov	
AUDIT WINDOW	
March 5 – March 29, 2025	
FINAL REPORT	
March 29, 2025	

1.1 Confidence Ranking

1 / 5

Very Low Confidence

Guardian assigns a Confidence Ranking of 1 out of 5, combining finding severity, remediation outcome, and the reviewer’s assessment of latent risk. The full scale is on the next page.



1.2 Findings at a Glance

SEVERITY	TOTAL	PENDING	ACK	PARTIAL	RESOLVED	FIX REVIEW
■ Critical	7	0	2	0	5	0
■ High	7	0	2	0	4	1
■ Medium	18	0	9	0	5	4
■ Low	40	0	21	0	12	7
■ Informational	0	0	0	0	0	0
All severities	72	0	34	0	26	12

§ 02

Confidence Ranking.

Guardian assigns a project-level confidence score using the scale below. The ranking combines finding severity, remediation outcome, and the reviewer's assessment of latent risk.

CONFIDENCE RANKING	DEFINITION AND RECOMMENDATION	RISK PROFILE
5 Very High Confidence	Codebase is mature, clean, and secure. No High or Critical vulnerabilities were found. Follows modern best practices with high test coverage and thoughtful design. Recommendation: code is highly secure at time of audit.	0 High/Critical findings and few Low/Medium severity findings.
4 High Confidence	Code is clean, well-structured, and adheres to best practices. Design patterns are sound and test coverage is strong. Recommendation: suitable for deployment after remediations; consider periodic review with changes.	0-1 High/Critical findings per engagement week and little to no Medium severity findings.
3 Moderate Confidence	Code is functional but shows gaps in defensive programming or test coverage. Recommendation: remediate the findings and re-review the changes before deployment.	Some High/Critical findings, with various Medium severity findings.
2 Low Confidence	Recurring issues suggest systemic gaps rather than isolated mistakes. Recommendation: remediate, then commission a follow-up review at a frozen commit.	Over 6 High and Critical findings across the engagement.
1 Very Low Confidence ASSIGNED	The volume and severity of findings indicate the codebase is not ready for production. Recommendation: substantial rework, then a full re-review.	Over 12 High and Critical findings across the engagement.

§ 03

Scope & Classifications.

Guardian reviewed PurrSwap Balancer Aggregator in

<https://github.com/Abracadabra-money/abracadabra-money-contracts>,

<https://github.com/Guardian0rg/purrswap-contracts-team1>. The scope

includes 138 source lines across 2 files.

3.1 Severity Matrix

	IMP · HIGH	IMP · MED	IMP · LOW
LIK · HIGH	Critical	High	Medium
LIK · MED	High	Medium	Low
LIK · LOW	Medium	Low	Low

IMPACT · HIGH

Significant loss of assets, harm to a group of users, or core protocol functionality is disrupted.

IMPACT · MEDIUM

Small amount of funds can be lost, or ancillary functionality is affected. Users may experience reduced or delayed receipt.

IMPACT · LOW

Unexpected behavior in some functionality that is notable but does not meet the criteria for higher severity.

LIKELIHOOD · HIGH

Possible with reasonable assumptions that mimic on-chain conditions; cost is low compared to amount gained.

LIKELIHOOD · MEDIUM

Possible only in uncommon cases or requires a large amount of capital relative to disruption caused.

LIKELIHOOD · LOW

Unlikely to ever occur in production.

Files in Scope.

FILE	NSLOC	TOTAL
MAIN REVIEW		
src/libraries/BalancerV2VaultReentrancyLib.sol	11	34
src/oracles/aggregators/BalancerV2WeightedPoolAggregator.sol	58	76
REMEDIATION REVIEW		
src/libraries/BalancerV2VaultReentrancyLib.sol	11	34
src/oracles/aggregators/BalancerV2WeightedPoolAggregator.sol	58	76

§ 04

Findings.

Grouped by round. Visible IDs reset by severity within each round, so a finding is named the same way in the report as it was in the channel it was written in. Status is where each finding ended up; its own page says how.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
MAIN REVIEW					
C-01	Fees Can Be Constantly Claimed	Logical Error	CRITICAL	RESOLVED	p. 10
C-02	Stale _I_ Usage in _adjustTarget And buyShares	Logical Error	CRITICAL	RESOLVED	p. 11
C-03	Untracked Shares Allow Fee Drain Exploit	Logical Error	CRITICAL	RESOLVED	p. 12
C-04	Static _I_ Should Not Be Read Directly	Logical Error	CRITICAL	RESOLVED	p. 13
C-05	Fee Claim Manipulation via LP Token Transfers	Logical Error	CRITICAL	RESOLVED	p. 14
C-06	No Validation For Oracles Used In Pools	Logical Error	CRITICAL	ACKNOWLEDGED	p. 16
C-07	Wrong initial target updates	Logical Error	CRITICAL	ACKNOWLEDGED	p. 17
H-01	Fee Misallocation When Selling Shares Through Router	Logical Error	HIGH	RESOLVED	p. 19
H-02	PurrLP Cannot Be Used Without An Oracle	Logical Error	HIGH	ACKNOWLEDGED	p. 20
H-03	Fees Wrongly Assigned To Router	Logical Error	HIGH	RESOLVED	p. 21
H-04	Stale Price Used For Initial Buy & Targets	Logical Error	HIGH	RESOLVED	p. 22
H-05	_RState_ Is Set To ONE Despite The Initial Reserves	Logical Error	HIGH	ACKNOWLEDGED	p. 23
H-06	Pool Aggregator OOG When Fetching	DoS	HIGH	RESOLVED	p. 26
M-01	Incompatibility With Balancer Pools	Validation	MEDIUM	RESOLVED	p. 27
M-02	Fees Are Not Automatically Claimed	Validation	MEDIUM	RESOLVED	p. 28
M-03	Unrestricted Pool Creation	Validation	MEDIUM	ACKNOWLEDGED	p. 29
M-04	Unfair Fee Distribution With FoT Tokens	Logical Error	MEDIUM	ACKNOWLEDGED	p. 30
M-05	Risk Of Overflow On Sqrt Calculation	Logical Error	MEDIUM	ACKNOWLEDGED	p. 31
M-06	Absence of Oracle Will Lead To Stale _I_	Validation	MEDIUM	ACKNOWLEDGED	p. 32
M-07	Wrong Usage Of Weighted Pool's totalSupply	Logical Error	MEDIUM	RESOLVED	p. 33
M-08	Oracle Might Return Stale Or Incorrect Data	Validation	MEDIUM	ACKNOWLEDGED	p. 34
M-09	Blacklisted Users Cannot Claim Fees	DoS	MEDIUM	ACKNOWLEDGED	p. 35
M-10	Risk Of Overflow On Sqrt Calculation	Logical Error	MEDIUM	ACKNOWLEDGED	p. 36
M-11	getPrice Lacks Oracle Validation	Oracle	MEDIUM	ACKNOWLEDGED	p. 37

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
M-12	Preminted fees not included in totalSupply	Logical Error	MEDIUM	RESOLVED	p. 38
M-13	Pool Ratio Can Be Skewed Due To Rounding	Logical Error	MEDIUM	ACKNOWLEDGED	p. 39
M-14	Flashloan Sync Issues	Logical Error	MEDIUM	RESOLVED	p. 41
L-01	AggregatorPriceProvider Cannot Add or Update	Code Best Practices	LOW	ACKNOWLEDGED	p. 43
L-02	Missing Checks In AggregatorPriceProvider	Code Best Practices	LOW	ACKNOWLEDGED	p. 44
L-03	No Fallback Oracle	Code Best Practices	LOW	ACKNOWLEDGED	p. 45
L-04	Unused Function _resetTargetAndReserve	Code Best Practices	LOW	RESOLVED	p. 46
L-05	Wrong FeesCollected Event Is Emitted	Code Best Practices	LOW	RESOLVED	p. 47
L-06	_L_ Should Account For Token Decimals	Validation	LOW	ACKNOWLEDGED	p. 48
L-07	onlyKnownPool Restriction Can Be Bypassed	Validation	LOW	RESOLVED	p. 49
L-08	Removed Pools Are Still Functioning	Unexpected Behavior	LOW	ACKNOWLEDGED	p. 50
L-09	Unused predictDeterministicAddress Function	Code Best Practices	LOW	RESOLVED	p. 51
L-10	Unneeded _updateFor(address(0)) Call	Validation	LOW	RESOLVED	p. 52
L-11	Fee Model Should Not Change	Logical Error	LOW	ACKNOWLEDGED	p. 53
L-12	No Check For Active L2 Sequencer	Logical Error	LOW	ACKNOWLEDGED	p. 54
L-13	treasuryFeeBips Is Not Fixed	Logical Error	LOW	ACKNOWLEDGED	p. 55
L-14	Sandwich Large Trades To Steal Fees	Frontrunning	LOW	ACKNOWLEDGED	p. 56
L-15	Unused Imports, Events And Errors	Code Best Practices	LOW	RESOLVED	p. 57
L-16	Missing Checks In latestAnswer	Validation	LOW	ACKNOWLEDGED	p. 58
L-17	Missing nonReentrant On correctRState	Code Best Practices	LOW	RESOLVED	p. 59
L-18	Oracle Incompatibility	Validation	LOW	ACKNOWLEDGED	p. 60
L-19	Router Decimal Validation Can Be Bypassed	Validation	LOW	ACKNOWLEDGED	p. 61
L-20	Oracle Cannot Be Updated	Code Best Practices	LOW	RESOLVED	p. 62
L-21	Target Fluctuations Because of Rounding	Informational	LOW	ACKNOWLEDGED	p. 63
L-22	WETH Should Be WHYPE	Informational	LOW	ACKNOWLEDGED	p. 64
L-23	Not All Tokens Implement decimals()	DoS	LOW	ACKNOWLEDGED	p. 65

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
L-24	Lack of Oracle Price Validation Checks In PurrLP	Code Best Practices	LOW	RESOLVED	p. 66
L-25	Aggregator Integration Issues	Code Best Practices	LOW	ACKNOWLEDGED	p. 67
L-26	Possible Division By 0 Weight	Code Best Practices	LOW	ACKNOWLEDGED	p. 68
L-27	Zero Address Token Aggregator Allowed	Code Best Practices	LOW	RESOLVED	p. 69
L-28	DOS for aggregators with > 18 decimals	DoS	LOW	RESOLVED	p. 70
L-29	Lack of Sippage Protection	Informational	LOW	ACKNOWLEDGED	p. 71
L-30	Division Before Multiplication	Code Best Practices	LOW	RESOLVED	p. 72
L-31	Auto-Compounding Fees Not Customizable	Informational	LOW	ACKNOWLEDGED	p. 73
L-32	Direct Donations Can Create Imbalances	Unexpected Behavior	LOW	ACKNOWLEDGED	p. 74
L-33	Managed Pools Lack Aggregator Flexibility	Informational	LOW	ACKNOWLEDGED	p. 75

REMEDIATION REVIEW

H-01	Pre-deployment Token Transfers Can Bypass First Deposit Ratio Enforcement	Validation	HIGH	REPORTED	p. 76
M-01	Incorrect Refund In createPoolETH Function	Logical Error	MEDIUM	REPORTED	p. 77
M-02	Salt collision for router pools	Unexpected Behavior	MEDIUM	REPORTED	p. 78
M-03	Incorrect router ETH refund	Unexpected Behavior	MEDIUM	REPORTED	p. 80
M-04	Pool Ratio Skewed Due To Rounding	Rounding	MEDIUM	REPORTED	p. 81
L-01	Flashloan Interface, Event And Error Not Removed	Informational	LOW	REPORTED	p. 82
L-02	Obsolete Flashloan Artifacts Remain After Feature Removal	Code Best Practices	LOW	REPORTED	p. 83
L-03	createPools Calls Through The Router Can Be Front-run	Warning	LOW	REPORTED	p. 84
L-04	sync Can Use getEffectiveI() Directly	Code Best Practices	LOW	REPORTED	p. 85
L-05	Unused Event and Error	Best Practices	LOW	REPORTED	p. 86
L-06	correctRState Can Be Reentered	Warning	LOW	REPORTED	p. 87
L-07	Pool Creation Griefed By Front-running	Frontrunning	LOW	REPORTED	p. 88

CRITICAL C-01

RESOLVED

Fees Can Be Constantly Claimed

CATEGORY	Logical Error
LOCATION	PurrLP.sol
REVIEW	Main Review

DESCRIPTION

When a user first calls `claimFees`, they fully synchronize their personal index to the latest global index and collect accrued fees. If that same user then transfers their LP shares to a different address the receiving address can re-claim the same backlog of fees. This is because the new address's index was never updated to the current global index at the time of receiving the shares. This effectively creates a second claim for the same fees. This step can be repeated multiple times until draining the contract.

The underlying cause is that no function automatically updates the sender and receiver's personal indexes during an LP token transfer.

RECOMMENDATION

Override `_beforeTokenTransfer(from, to, amount)` in the `PurrLP` contract, ensuring both `from` and `to` are updated at the moment of the transfer:

```
``solidity
function _beforeTokenTransfer(address from, address to, uint256 amount) internal override {
    if (from != address(0) && to != address(0) && from != to) {
        _updateFor(from);
        _updateFor(to);
    }
}
``
```

RESOLUTION

Status: Resolved.

CRITICAL C-02

RESOLVED

Stale `_I_` Usage in `_adjustTarget` And `buyShares`

CATEGORY	Logical Error
LOCATION	PurrrLP.sol
REVIEW	Main Review

DESCRIPTION

In the `PurrrLP` contract, an oracle can be assigned to override the static `_I_` value during trades. The `getPMMState` function checks if an oracle was set during initialization and if so, each trade will use the returned price from the oracle as `_I_`. However, certain other functions continue to rely on `_I_` directly, ignoring any oracle updates. Specifically, the `_adjustTarget` and `buyShares` functions compute or expand the pool targets using the static `_I_` set at initialization rather than fetching a fresh oracle price. In scenarios where the actual market price has shifted significantly, these code paths will anchor to an outdated `_I_`, causing the pool to drift from the correct ratio. Liquidity providers may deposit or expand the pool at an obsolete price and any target adjustments fail to reflect the oracle's data. This mismatch can produce arbitrage opportunities or incorrectly valued liquidity shares.

RECOMMENDATION

Refactor the contract so that whenever `_I_` is referenced in `_adjustTarget` and `buyShares`, it first checks whether an oracle is set. If so, it should fetch the current oracle price rather than defaulting to the static `_I_`.

RESOLUTION

Status: Resolved.

CRITICAL C-03

RESOLVED

Untracked Shares Allow Fee Drain Exploit

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/purrswap-contracts-team2/blob/484cb7dfddb01f691c489ac0e46676fe702a7ffd/contracts/PurrLP.sol#L537
REVIEW	Main Review

DESCRIPTION

The claimFees function enables users to claim their share of the PurrLPFees, which is collected and stored in the PurrLPFees contract when the AUTO_COMPOUND_FEES flag is set to false. The issue arises because the _updateFor function—which calculates a user's earned fees—relies on balanceOf(recipient) to determine the user's supplied amount without verifying if the user previously had that amount of shares. As a result, an attacker can repeatedly claim fees by transferring their shares to a different account and claiming fees again, eventually draining all fees from the contract.

RECOMMENDATION

Update the fee logic so that instead of relying on balanceOf(recipient) at claim time, the contract records and uses each user's most recent balance at the moment of any share-modifying action (i.e., when buying or selling shares).

PROOF OF CONCEPT https://github.com/GuardianOrg/purrswap-contracts-team2/blob/f316d0199cbcaa1c6ff9b20c7680efecb3a9bc81/test/Guardian/drain_fees_poc.t.sol#L70

RESOLUTION

Status: Resolved.

CRITICAL C-04

RESOLVED

Static `_I_` Should Not Be Read Directly

CATEGORY Logical Error

LOCATION `PurrrLP.sol`

REVIEW Main Review

DESCRIPTION

The variable `_I_` is set during deployment and represents the external market price in the PMM algorithm. This variable is never updated and relies on an external oracle to consistently reflect real-time market conditions. To achieve this, the `getPMMState` function returns `state.i`, which is set to the oracle price. However, `_I_` is used directly in certain functions like `buyShares` (initial minting) and `_adjustTarget`. In contrast, trading functions like `sellBase` and `sellQuote` use `getPMMState` and thus utilize the most recent oracle price when available.

The biggest concern is with `_adjustTarget`, which is called after every trading function to recalculate target reserves. Since the market price is expected to diverge from the initial `_I_`, the pool will always be adjusting its targets based on an outdated price, leading to suboptimal liquidity provision.

RECOMMENDATION

Ensure that the functions `buyShares` and `_adjustTarget` are reading from `getPMMState` instead of `_I_`.

RESOLUTION

Status: Resolved.

CRITICAL C-05

RESOLVED

Fee Claim Manipulation via LP Token Transfers

CATEGORY	Logical Error
LOCATION	PurrLP.sol: 552-576
REVIEW	Main Review

DESCRIPTION

The `PurrLP::_updateFor` function is vulnerable to a `collusion attack` where multiple users can repeatedly claim additional pool fees by strategically transferring LP tokens. This allows them to extract a disproportionate share of fees, `disadvantaging` other legitimate liquidity providers. `_updateFor(recipient)` calculates `claimable fees` (`claimable0` and `claimable1`) based on the `current LP token balance` (`balanceOf(recipient)`) of the recipient. It multiplies this balance by the difference between the global fee index (`index0`, `index1`) and the user's last recorded index (`supplyIndex0`, `supplyIndex1`).

Let's consider the following scenario:

1. Let's assume User A and User B holds LP tokens.
2. User B transfers `all their LP tokens` to User A.
3. User A calls `claimFees()`. Their claimable fees are calculated based on `User A's new large LP token balance` (part of which belonged to User B) and transferred.
4. After the fee claiming `User A transfers all of his LP tokens to User B`.
5. User B calls `claimFees()`. Because `_updateFor(User B)` is called before the claim, the claimable fees are calculated based on User B's new (and large) LP token balance, even though those tokens just had their associated fees claimed by User A.
6. User B transfers the LP tokens to User C, and the process repeats.

This cycle allows the colluding users to repeatedly claim fees based on the same underlying liquidity thus greiving the other legitimate liquidity providers.

RECOMMENDATION

It is recommended to call `_updateFor` inside the `_transfer` internal function of the LP contract to update the `checkpoints` (`supplyIndex0` and `supplyIndex1`) during the `LPToken transfers` for both the `to` and `from` addresses of the transfer.

C-05 Fee Claim Manipulation via LP Token TransfersCONTINUED

— RESOLUTION

Status: Resolved.

CRITICAL C-06

ACKNOWLEDGED

No Validation For Oracles Used In Pools

CATEGORY	Logical Error
LOCATION	Router.sol, Factory.sol
REVIEW	Main Review

DESCRIPTION

External oracles were introduced to the PMM as a way to obtain the `i` value, which reflects the external price of a token pair. When creating a pool, the deployer provides an oracle address. However, there is no validation for the oracle address provided, representing a significant security risk, as a malicious oracle can manipulate the `i` value and control pricing.

This risk is particularly dangerous because it may not be immediately visible—it could only present itself mid-transaction when the oracle is queried.

Potential Attacks:

- **Price Manipulation:** The oracle returns an artificially low `i` value mid-swap, resulting in the user receiving significantly fewer tokens (or none at all).
- **Gas Griefing:** The oracle performs computationally intensive operations when queried, causing transactions to become excessively expensive or fail due to insufficient gas.
- **Read-only Reentrancy:** While most of `PurrrLP`'s functions are protected by the `nonReentrant` modifier, external integrators may not have this protection. A malicious oracle could re-enter mid-transaction, potentially leading to unintended side effects.

While `PurrrLP` is designed to be permissionless, this flexibility introduces significant security risks—especially for pools deployed through the protocol's `Router` and `Factory` contracts. Users may assume these pools are inherently safe, but without oracle validation, they remain just as vulnerable to manipulation as manually deployed pools.

RECOMMENDATION

- The `Factory` contract should enforce that only protocol-approved oracles are used. The protocol can maintain a whitelist of valid token-pair oracles.
- At the `Pool` level, arbitrary oracles could still be allowed, but this risk should be explicitly documented for users.

RESOLUTION

Status: Acknowledged.

CRITICAL C-07

ACKNOWLEDGED

Wrong initial target updates

CATEGORY Logical Error
LOCATION PurrrLP.sol
REVIEW Main Review

DESCRIPTION

As described in the [following issue](https://www.notion.so/guardianaudits/C-03-dec7075995b249a2b5924ce6d99a5b87?pvs=4) from the previous audit, if the ratio used in `buyShares` is not 1:1, then `R_STATE` will be set to `ONE`. During the initial mint, `_setReserves()` will also set the base and quote reserve to the raw contract balance of these tokens, resulting in a `R = 1` state even though the targets don't match the reserves.

This leads to wrong target updates used for swap calculations and users receiving more or less from the token they are swapping depending on which one.

In the attached POC you can find the following case

MIM-DAI Pair, `I = 3000 DAI`

1. LP adds 2 MIM and 12000 DAI
2. `R = 1`, `BASE_TARGET = 2`, `QUOTE_TARGET = 6000`
3. A `sellQuote` of only 1 DAI is executed which makes `R > 1`
4. `_adjustTarget()` is called which adjusts the `_BASE_TARGET`, but uses the delta between `_QUOTE_RESERVE` and `_QUOTE_TARGET`. Because `QUOTE_RESERVE` is `12000e18 + 1`, but `_QUOTE_TARGET` is `6000`, the `_BASE_TARGET` will experience a huge jump in its value, making the targets `4:6` when the real ratio is `1:6`.

You will also find two examples of user selling `12000DAI` immediately after initialization and when there is a 1 wei `sellQuote()` in between. Because of the wrong target update, there is a difference in the value received (~1.33 MIM vs ~0.8 MIM).

RECOMMENDATION

During the initial mint, set the reserves to the initial targets and refund the excess amount of tokens to the LP. By doing so, you will ensure a true equilibrium for the `R = 1` state.

PROOF OF CONCEPT <https://github.com/GuardianOrg/purrrswap-contracts-team2/pull/3>

C-07 Wrong initial target updates

CONTINUED

— RESOLUTION

Status: Acknowledged.

HIGH **H-01**

RESOLVED

Fee Misallocation When Selling Shares Through Router

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/purrrswap-contracts-team2/blob/484cb7dfddb01f691c489ac0e46676fe702a7ffd/contracts/PurrLP.sol#L453
REVIEW	Main Review

— DESCRIPTION

The sellShares function internally calls the `_updateFor(msg.sender)` function to update a user's earned fees, which can later be claimed using the `claimFees` function. However, when a user sells shares through the router contract, `msg.sender` becomes the router's address instead of the individual user's address. This causes any earned fees from the sold shares to be attributed to the router rather than the user. As a result, the user loses the fees they have accumulated from those shares, and since the router contract lacks functionality to call `claimFees`, those fees become permanently stuck in the contract.

— RECOMMENDATION

Modify the sellShares function so that it uses the actual user's address rather than `msg.sender` when updating the fee balance when called from the router. This could be achieved by passing the originating user address as an additional parameter .

PROOF OF CONCEPT https://github.com/GuardianOrg/purrrswap-contracts-team2/blob/d6c57b9a936a15f966c2ef47c1a1b3ed128d3339/test/Guardian/router_fee_misallocation_poc.t.sol#L80

— RESOLUTION

Status: Resolved.

HIGH **H-02**

ACKNOWLEDGED

PurrLP Cannot Be Used Without An Oracle

CATEGORY	Logical Error
LOCATION	PurrLP.sol: 131
REVIEW	Main Review

— DESCRIPTION

The PurrLP contract can be initialized with or without an oracle. If no oracle is set, the `getPMMState` function will return the `_I_` variable stored in the contract.

The issue is that, in the old contract (MagicLP), the `_I_` variable could be updated periodically through the admin function `setParameters` (<https://github.com/Abracadabra-money/abracadabra-money-contracts/blob/main/src/mimswap/MagicLP.sol#L502>). However, in PurrLP, that function has been removed.

This means that if no oracle is set, `_I_` would be a static value that can never be changed. Since `_I_` represents the price between the base and quote token (e.g. WETH/USDC), it is expected to change over time. Without an oracle or an admin update mechanism, price will be lower than market price, and tokens can be extracted at a cheaper cost.

Additionally, it may be required for `_K_` value to be updated to match the changing volatility of a token pair. This also cannot be done without the `setParameters` function.

— RECOMMENDATION

Either enforce that an oracle address must be set (i.e., ensure it is not a zero address) or re-implement the `setParameters` and `pause` functions to allow updates to `_I_` and `_K_` values.

— RESOLUTION

Status: Acknowledged.

HIGH **H-03**

RESOLVED

Fees Wrongly Assigned To Router

CATEGORY Logical Error

LOCATION PurrrLP.sol

REVIEW Main Review

— DESCRIPTION

When the `sellShares` function is called through the `Router`, the contract uses `_updateFor(msg.sender)` to account for external fees. Because `msg.sender` is the router address in this context, the fees end up credited to the router rather than the end user who owned and burned the LP shares. This misallocation causes the true LP holder to lose out on fees that should have accrued to their account.

— RECOMMENDATION

Consider calling `_updateFor` with the `to` address instead of `msg.sender` in the `sellShares` function.

— RESOLUTION

Status: Resolved.

HIGH **H-04**

RESOLVED

Stale Price Used For Initial Buy & Targets

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/purrswap-contracts-team2/blob/d6c57b9a936a15f966c2ef47c1a1b3ed128d3339/contracts/PurrLP.sol#L641
REVIEW	Main Review

— DESCRIPTION

The PurrLP contract uses the `I` variable to represent the base token price during the initial buy (`buyShares`) and in the `_adjustTarget` function. This variable is set once during initialization and is not updated afterward. In contrast, other parts of the contract fetch the latest price via the oracle (using `IPurrLPOracle(*ORACLE*).price(address(*BASE_TOKEN*))`). This means that after initialization, the `buyShares` and `_adjustTarget` functions rely on a potentially stale price, which can lead to mispricing in the initial buy and inaccurate target adjustments. Furthermore, since the rest of the code uses the latest price from the oracle, this creates a discrepancy between how trades are priced and how the pool's targets are adjusted—potentially resulting in imbalanced pool reserves and opportunities for arbitrage.

— RECOMMENDATION

Update the `buyShares` and `_adjustTarget` functions to fetch the latest price using the oracle instead of relying on the outdated `I` variable.

PROOF OF CONCEPT https://github.com/GuardianOrg/purrswap-contracts-team2/blob/c370d1b130b21215261b25e6ced14187864c31ff/test/Guardian/stale_price_poc.t.sol#L70

— RESOLUTION

Status: Resolved.

HIGH **H-05**

ACKNOWLEDGED

`_RState_` Is Set To ONE Despite The Initial Reserves

CATEGORY Logical Error

LOCATION `PurrrLP.sol`

REVIEW Main Review

— DESCRIPTION

The `PurrrLP` contract, designed to implement the Dodo's PMM model, suffers an issue in the initialization and management of its `_RState_` variable. This variable, an enum with possible values `RState.ONE` (represented numerically as 0), `RState.ABOVE_ONE` and `RState.BELOW_ONE`, is intended to reflect the balance of the pool's reserves, specifically, the ratio of base tokens (`B`) to quote tokens (`Q`) relative to their target values (`B0` and `Q0`). In the PMM model, `_RState_` dictates which pricing formula the contract uses for trades: a balanced-state formula when `_RState_` is `ONE`, an excess-quote-token formula when `_RState_` is `ABOVE_ONE` or an excess-base-token formula when `_RState_` is `BELOW_ONE`. However, a flaw in how `_RState_` is handled upon deployment and after the initial liquidity deposit creates a window of mispricing that can significantly distort the first trades.

Upon deployment of the `PurrrLP` contract, the `_RState_` variable defaults to `RState.ONE` because its enum value of 0 is the natural initial state of an unassigned variable in Solidity. This default setting implies that the pool's reserves are perfectly balanced, meaning the actual amount of base tokens (`B`) matches the target amount (`B0`) and the actual amount of quote tokens (`Q`) matches the target amount (`Q0`). In an ideal scenario, this assumption might hold if the first liquidity provider deposits tokens in exact accordance with the pool's target ratio, defined by `_I_` (the target price). However, the contract does not enforce this constraint. Instead, it allows the first liquidity provider, through the `buyShares` function, to deposit arbitrary amounts of base and quote tokens. For example, a depositor might supply a skewed ratio, say, far more base tokens than quote tokens relative to the target, resulting in reserves where `B` significantly exceeds `B0` while `Q` falls short of `Q0`, or vice versa. Despite this potential imbalance, the contract does not update `_RState_` after this initial deposit. It remains set to `RState.ONE`, incorrectly signaling a balanced pool. This mischaracterization becomes problematic when the first trade occurs. The `PurrrLP` contract provides functions like `sellBaseToken` and `sellQuoteToken`, which rely on queries such as `querySellBase` or `querySellQuote` to determine the price of a trade. These

H-05 `_RState_` Is Set To ONE Despite The Initial Reserves

CONTINUED

queries, in turn, consult the `PMMPricing` library, which uses `_RState_` to select the appropriate pricing logic. When `_RState_` is `ONE`, the library invokes `_ROneSellBaseToken` or `_ROneSellQuoteToken`, functions tailored for a balanced pool where the reserves align with their targets. These functions assume a specific relationship between `B`, `Q`, `B0` and `Q0`, yielding a price that reflects an equilibrium state. However, if the initial deposit created a skewed ratio, say, `B` much greater than `B0` and `Q` less than `Q0`, which should correspond to `RState.BELOW_ONE`, the actual reserve state does not match this assumption. The first trader, whether selling base tokens or quote tokens, receives a price quote based on the incorrect balanced-state formula rather than the appropriate formula for an imbalanced pool (e.g., `_RBelowOneSellBaseToken` or `_RAboveOneSellQuoteToken`). This discrepancy leads to a mispriced trade. The consequences of this mispricing are not trivial. Depending on the direction of the skew and the trade, the first trader might receive an overly favorable or unfavorable price. The root of this bug lies in the `buyShares` function, which handles liquidity deposits. When the first liquidity provider adds tokens `totalSupply() == 0`, this initial liquidity deposit is forced to match the price ratio declared by `_I_`:

- It checks whether the deposited quote is less than $(\text{base} * _I_)$.
- Depending on which side is "shorter," it limits the effective number of "shares" so that `_BASE_TARGET_` and `_QUOTE_TARGET_` end up being shares and `shares * _I_`, respectively.

This implementation ensures the final `base:quote` ratio in the pool will be exactly `_I_` after that first deposit. In simpler terms, no matter how many tokens are deposited initially, the pool itself only "counts" as many tokens as needed to match `_I_`. If the first depositor tries to deposit more of one side than `_I_` implies, the "excess" effectively gets ignored in the share calculation, making the pool start out at a ratio that is locked to `_I_`. However, at the end of the `buyShares` logic, the function still call `_setReserve(baseBalance, quoteBalance);` when, in this situation, it should have ignored the excess of base or quote tokens. The contract here is assuming that aligning the targets with the actual reserves preserves the `RState.ONE` condition, but this assumption only holds if the initial deposit matches the intended target ratio `_I_`. Only after the first trade modifies the reserves does the contract potentially reassess `_RState_`, flipping it to `ABOVE_ONE` or `BELOW_ONE` as needed.

— RECOMMENDATION

Update the `buyShares` function to enforce the target ratio `_I_` for the initial deposit, refunding excess base or quote tokens to the first depositor:

- Calculate the expected Q based on deposited B and `_I_` (or vice versa).
- Accept only the matching amounts, refunding the rest.
- Leave `_RState_ = ONE`, as the pool will be balanced.

H-05 _RState_ Is Set To ONE Despite The Initial ReservesCONTINUED

— RESOLUTION

Status: Acknowledged.

HIGH **H-06**

RESOLVED

Pool Aggregator OOG When Fetching

CATEGORY	DoS
LOCATION	BalancerV2VaultReentrancyLib.sol
REVIEW	Main Review

DESCRIPTION

The `BalancerV2WeightedPoolAggregatorPool.latestAnswer()` function uses `BalancerV2VaultReentrancyLib.ensureNotInVaultContext()` which executes a `staticcall` to the vault.

The `staticcall` will always revert either because:

- a reentrancy happens and the `nonReentrant` modifier reverts with data;
- there is no reentrancy, but the `nonReentrant` tries to modify the state in a `staticcall`, reverting with empty data.

If the revert data is empty, the contract allows fetching the price. When the storage is modified in a `staticcall`, all of the forwarded gas is used up. Because there is no hard limit set when calling the vault, all 63/64th of the gas in the aggregator will be wasted on each legitimate fetch call.

Because of this behavior, fetching prices requires extremely high amounts of gas, which grow more the more `poolTokens` there are. 1/64 of the gas sent to the main call should cover all of the logic inside the aggregator. As seen in the POC, a simple price fetching for two tokens requires more than 3M gas. Once integrated into the Cauldrons, the aggregator will most likely cause DOS of the system because of its high gas expenditure.

RECOMMENDATION

Hardcode the `gas` to a value around `10_000`, like than in the [original implementation] (<https://github.com/balancer/balancer-v2-monorepo/blob/36d282374b457dddea828be7884ee0d185db06ba/pkg/pool-utils/contracts/lib/VaultReentrancyLib.sol#L78>).

PROOF OF CONCEPT <https://gist.github.com/ZdravkoHr/cf848ee4669cdb7c2d5ad11da8544631>

RESOLUTION

Status: Resolved.

MEDIUM **M-01**

RESOLVED

Incompatibility With Balancer Pools

CATEGORY	Validation
LOCATION	BalancerV2WeightedPoolAggregator.sol
REVIEW	Main Review

DESCRIPTION

As mentioned in the [BalancerV2 docs](https://docs-v2.balancer.fi/concepts/advanced/valuing-bpt/valuing-bpt.html#getting-bpt-supply), there are three potential functions to query when determining the BPT supply depending on pool type. The right function should be chosen accordingly to the pool type:

- `getActualSupply` : This is the most common/current function to call. It is used by the most recent versions of Weighted and Stable Pools. It accounts for pre-minted BPT as well as due protocol fees.
- `getVirtualSupply` : This is used by Linear Pools and "legacy" Stable Phantom Pools. It accounts for pre-minted BPT but does not take due protocol fees into account. The omission of protocol fee calculations in Linear Pools is intentional since they do not pay protocol fees.
- `totalSupply` : Which should only be used for older "legacy" pools. The original Weighted and Stable Pools do not have pre-minted BPT, so they follow the typical convention of using `totalSupply` to account for issued pool shares.

`totalSupply` will always return `type(uint211).max` in Balancer Pools with pre-minted BPT.

RECOMMENDATION

Be aware of this limitation and avoid using the `BalancerV2WeightedPoolAggregator` with Balancer Pools with pre-minted BPT.

RESOLUTION

Status: Resolved.

MEDIUM **M-02**

RESOLVED

Fees Are Not Automatically Claimed

CATEGORY Validation
LOCATION PurrrLP.sol
REVIEW Main Review

— DESCRIPTION

When a user transfers LP tokens without first calling `claimFees`, the contract does not automatically update or settle their unclaimed fees. Consequently, fees accumulated for the transferred shares will be lost. This stems from the absence of a mandatory fee settlement step prior to reducing the sender's LP balance. Users often remain unaware that they must manually call `claimFees` before transferring shares, leading to lost rewards.

— RECOMMENDATION

Modify the transfer flow so that each holder's fees are automatically claimed before shares leave their wallet.

— RESOLUTION

Status: Resolved.

MEDIUM **M-03**

ACKNOWLEDGED

Unrestricted Pool Creation

CATEGORY Validation
LOCATION Factory.sol
REVIEW Main Review

— DESCRIPTION

The `Factory` contract allows any user to create new pools without prior validation or permission checks. While this enables permissionless innovation, it also poses a significant risk that an attacker or inexperienced deployer could set critical parameters in a harmful way. For example, a malicious deployer could specify a completely incorrect target price `_I_` or choose `K = 0` for a token pair where that setting makes no economic sense (non-stable pool). Worse yet, a deployer could assign a malicious oracle that returns manipulated prices, forcing unsuspecting liquidity providers into providing capital at unfair ratios. Because the `Factory` does not vet these parameters or oracles, liquidity providers joining such pools are exposed to extreme economic vulnerabilities or outright malicious configurations.

— RECOMMENDATION

Consider adding robust warnings and documentation that emphasize the high risk of joining pools with suspicious tokens, unrecognized oracles or extreme parameters such as $K = 0$ for non-stable tokens.

— RESOLUTION

Status: Acknowledged.

MEDIUM **M-04**

ACKNOWLEDGED

Unfair Fee Distribution With FoT Tokens

CATEGORY Logical Error
LOCATION PurrLP.sol: 508, 525
REVIEW Main Review

DESCRIPTION

The PurrLP contract presents a bug when used with `fee-on-transfer` (FOT) tokens. The contract calculates `claimable fees` based on the `intended fee amount`, but the `PurrLPFees` contract receives less than that due to the `FOT token's built-in fee`. (The Sponsor said that any token can be used in the protocol hence `FOT` tokens are a valid concern here).

The core issue is that the `fee calculation` in the `PurrLP::_updateBaseFees` and `PurrLP::_updateQuoteFees` (`index0, index1`) uses the `pre-fee-deduction amount`, while the `PurrLPFees` contract holds the `post-fee-deduction amount`.

This discrepancy leads to following issues:

1. Denial of Service (DoS): The `PurrLPFees` contract can `run out of tokens` before all users have claimed their fees. `Later users` will be `unable to claim`, causing a revert.
2. Unfairness: Early claimers get more shares than they truly deserve, while later claimers unable to claim since not enough fee left in the `PurrLPFees` contract.

RECOMMENDATION

It is recommended to `track the actual amount of tokens` received by `PurrLPFees` after the `FOT deduction` and use that for `fee calculations`.

RESOLUTION

Status: Acknowledged.

MEDIUM **M-05**

ACKNOWLEDGED

Risk Of Overflow On Sqrt Calculation

CATEGORY Logical Error
LOCATION Math.sol: 98
REVIEW Main Review

— DESCRIPTION

The original finding M-02 \ Risk of Overflow on Sqrt Calculation ([<https://www.notion.so/guardianaudits/M-02-1848bda5828c8107802fccdddb2b0021?pvs=4>])(<https://www.notion.so/guardianaudits/M-02-1848bda5828c8107802fccdddb2b0021?pvs=4>) remains unaddressed.

The risk of overflow has increased due to the expectation that `PurrLP` will be permissionless and could be used with exotic tokens. Such tokens may have large decimals, which can result in significantly larger `delta` or `i` values. This increases the likelihood of an overflow occurring in the function `_SolveQuadraticFunctionForTarget`.

During fuzz testing, overflow risk was also observed in `_SolveQuadraticFunctionForTrade` when calculating `squareRoot`.

— RECOMMENDATION

Consider using libraries that support intermediate overflows. Furthermore, clearly document the risks with using large K values, and tokens with large decimals.

— RESOLUTION

Status: Acknowledged.

MEDIUM **M-06**

ACKNOWLEDGED

Absence of Oracle Will Lead To Stale `_I_`

CATEGORY Validation
LOCATION `Factory.sol`, `PurrrLP.sol`
REVIEW Main Review

— DESCRIPTION

If a new `PurrrLP` pool is deployed without specifying an oracle, the `_I_` value remains the static ratio defined at initialization. Because asset prices typically evolve in real time, that constant `_I_` quickly becomes inaccurate relative to the market. This inaccuracy leads to large arbitrage opportunities or perpetual mispricing in the pool.

— RECOMMENDATION

Enforce in the `Factory` contract that a valid oracle is provided during pool deployment so the pool's `_I_` can update as the market shifts. If tokens are expected to maintain a stable price, use an oracle that reliably confirms this peg; otherwise, choose an external feed that reflects the asset's volatility.

— RESOLUTION

Status: Acknowledged.

MEDIUM **M-07**

RESOLVED

Wrong Usage Of Weighted Pool's totalSupply

CATEGORY Logical Error
LOCATION BalancerV2WeightedPoolAggregator.sol: 37
REVIEW Main Review

DESCRIPTION

The `latestAnswer` function will obtain a price for Balancer Pool Token (BPT) by fetching the pool's TVL and dividing it by the pool's total supply of BPT tokens.

The problem with the current approach is that the pool's `totalSupply` function is used, instead of `getActualSupply`.

As stated in the [weighted pool contract](https://github.com/balancer/balancer-v2-monorepo/blob/ac63d64018c6331248c7d77b9f317a06cced0243/pkg/pool-

weighted/contracts/WeightedPool.sol#L328-L330) `getActualSupply` should always be used instead of `totalSupply`. This is because "the Pool owes debt to the Protocol in the form of unminted BPT, which will be minted immediately before the next join or exit." Such BPT tokens need to be accounted for because "even if they don't yet exist, they will effectively be included in any Pool operation that involves BPT."

As a result from this, some differences might arise between the values returned from `totalSupply` and the ones returned by `getActualSupply`.

Looking at the [USDC/STG pool]

(https://etherscan.io/address/0x3ff3a210e57cfe679d9ad1e9ba6453a716c56a2e#readContract) at block 22017511, the `totalSupply` was 13581803776383036010328725 but the `getActualSupply` was 13589503378492964934776332 -- a difference of 7699602109928924447607 (7699e18) which will greatly impact the BPT token's price calculation.

RECOMMENDATION

Use `getActualSupply` instead of `totalSupply` to obtain the pool's BPT supply.

RESOLUTION

Status: Resolved.

MEDIUM **M-08**

ACKNOWLEDGED

Oracle Might Return Stale Or Incorrect Data

CATEGORY Validation

LOCATION AggregatorPriceProvider: 30

REVIEW Main Review

DESCRIPTION

In `AggregatorPriceProvider`, the function `getPrice` uses Chainlink's `latestRoundData` but there is no check if the return value indicates stale data. This could lead to potentially incorrect values being returned from the oracle.

See [Chainlink Docs](https://docs.chain.link/data-feeds/price-feeds/historical-data).

RECOMMENDATION

Add the following checks to verify the correctness of the data returned:

```
``plain text
(uint80 roundID, int256 answer, uint256 timestamp , uint256 updatedAt, uint80 answeredInRound)
= aggregator.latestRoundData();
if (updatedAt + tokenHeartbeat[token] < block.timestamp) revert StalePrice();
require(timestamp != 0, "Round not complete");
require(answeredInRound >= roundID, "Stale Price");
require(answer > 0, "Invalid Price");
``
```

RESOLUTION

Status: Acknowledged.

MEDIUM **M-09**

ACKNOWLEDGED

Blacklisted Users Cannot Claim Fees

CATEGORY	DoS
LOCATION	PurrLPFees.sol: 19-28
REVIEW	Main Review

DESCRIPTION

Liquidity providers (LPs) can claim their earned fees via `PurrLP::claimFees`, which internally calls `PurrLPFees::claimFeesFor`. This function transfers both base token and quote token fees in a single transaction.

However, if an LP gets blacklisted by either the base or quote token, they can still withdraw liquidity using `Router::removeLiquidityOneSide` by choosing the token they are not blacklisted from.

But this is not possible for fee claims because both tokens are transferred together and there is no option to claim only one token or convert fees to the non-blacklisted token.

As a result, fees remain stuck in `PurrLPFees.sol` indefinitely unless the LP is whitelisted again.

RECOMMENDATION

Allow partial fee claims, enabling LPs to withdraw only the token they are not blacklisted from. Alternatively, implement a swap mechanism to convert fees to the non-blacklisted token before distribution.

RESOLUTION

Status: Acknowledged.

MEDIUM **M-10**

ACKNOWLEDGED

Risk Of Overflow On Sqrt Calculation

CATEGORY Logical Error

LOCATION Math.sol

REVIEW Main Review

— DESCRIPTION

Within the usage of more volatile assets, there is now a greater risk of overflow when solving the quadratic functions.

For example, the function `_SolveQuadraticFunctionForTarget` risks overflow within the else-if case: `else if ((ki * delta) / ki == delta)`

Depending on the `_K_` and `_I_` values (which can be up to `10^36` and `10^18`) respectively, the multiplication with a large delta will overflow, which effects the `adjustedTarget` function and hence most protocol actions.

Another example is `uint256 squareRoot = DecimalMath.mulFloor((DecimalMath.ONE - k) * 4, DecimalMath.mulFloor(k, V0) * V0);` within `_SolveQuadraticFunctionForTrade` which can overflow when multiplying `K * V0 * V0` ultimately leading to failed trades.

— RECOMMENDATION

Consider using libraries that support intermediate overflows. Furthermore, clearly document the risks with using large K values, and tokens with large decimals.

— RESOLUTION

Status: Acknowledged.

MEDIUM **M-11**

ACKNOWLEDGED

getPrice Lacks Oracle Validation

CATEGORY	Oracle
LOCATION	https://github.com/Abracadabra-money/abracadabra-money-contracts/blob/e5f9b907185e0f4f6b9728572f880b87a8754518/src/periphery/AggregatorPriceProvider.sol#L25
REVIEW	Main Review

— DESCRIPTION

The `getPrice` function retrieves the price using Chainlink's `latestRoundData()` method. However, it does not validate the returned data for staleness or correctness, which could lead to potential issues if the data is outdated or invalid.

— RECOMMENDATION

Implement validation checks on the data retrieved from `latestRoundData()`. Specifically, ensure that the price is greater than zero and that the data is not stale by verifying that `updatedAt` is within the expected `tokenHeartbeat` threshold.

— RESOLUTION

Status: Acknowledged.

MEDIUM **M-12**

RESOLVED

Preminted fees not included in totalSupply

CATEGORY Logical Error
LOCATION BalancerV2WeightedPoolAggregator.sol
REVIEW Main Review

DESCRIPTION

`BalancerV2WeightedPoolAggregator.latestAnswer()` uses the `totalSupply()` function of the balancer pool to calculate the fair LP price. This is incorrect for the new balancer pools - for them `getActualSupply()` should be used instead because it includes the fees not yet minted. In result, the aggregator will return wrong result for the LP price.

RECOMMENDATION

Add a `try/catch` where you use the `getActualSupply()` if available and the `totalSupply` otherwise.

RESOLUTION

Status: Resolved.

MEDIUM **M-13**

ACKNOWLEDGED

Pool Ratio Can Be Skewed Due To Rounding

CATEGORY Logical Error
LOCATION `PurrrLP.sol`
REVIEW Main Review

DESCRIPTION

When shares are calculated in `PurrrLP.buyShares()`, the base and quote balances are computed based on the initial price `_I_`. The calculations ensure the tokens are proportionally added. Because `mulFloor(baseBalance, _I_)` rounds down, an attacker can abuse this for some tokens and skew the pool ratio.

Suppose `quoteBalance = 1`, `baseBalance = 19999`, `I = 1e14`. Then we have:

```
1 < 19999 * 1e14 / 1e18 => 1 < 1 => False
```

Therefore `shares = 19999`.

It sets the targets:

```
_BASE_TARGET_ = 19999
```

```
_QUOTE_TARGET_ = 19999 * 1e14 / 1e18 = 1
```

The result is the ratio `1:19999`, when the intended ratio from `I` is `1:1000`.

RECOMMENDATION

Consider increasing the minimum shares to be minted requirement.

PROOF OF CONCEPT <https://github.com/GuardianOrg/purrrswap-contracts-team2/pull/5>

RESOLUTION

Status: Acknowledged.

Guardian: The main audit finding 'M-09 Pool Ratio Can Be Skewed Due To Rounding' has not been fixed. While the Router contract now performs adjustment of BASE/QUOTE input amounts via `_adjustAddLiquidity` this does not resolve the issue as `_adjustAddLiquidity` has the same logic as `PurrrLP::buyShares`.

The pool ratio can still be skewed due to rounding which leads to a discrepancy between `_BASE_TARGET_ / _QUOTE_TARGET_` and the `i` value.

M-13 Pool Ratio Can Be Skewed Due To Rounding

CONTINUED

****Recommendation****Use `mulCeil` instead of `mulFloor` when calculating `shares`:```plain text`

```
shares = quoteBalance < DecimalMath.mulFloor(baseBalance, _i) ?  
DecimalMath.divFloor(quoteBalance, _i) : baseBalance;  
``
```

MEDIUM **M-14**

RESOLVED

Flashloan Sync Issues

CATEGORY Logical Error
LOCATION `PurrLP.sol`
REVIEW Main Review

DESCRIPTION

There are several concerns about the `PurrLP.flashloan()` function:

- it can be called with 0 amounts to execute `sync` and bypass the `onlyImplementationOwner` modifier.
- if the if branches are not entered, the `Rstate` won't be modified and can lead to a situation where an incongruity happens between the `Rstate` and the actual targets.

`flashloan()` checks if the contract balance of the two tokens is below the recorded balance in the respective `RESERVE` variable and if it is, the `Rstate` is recalculated.

However, it's possible to skip both these checks if the balances are equal to or greater than the reserves. Then `_sync()` will be called, the reserves will be updated with the new balances and the targets will be adjusted in `_adjustTarget()` without updating the `rState`.

This ability to execute `_sync()` and change the targets allows an attacker to do the following:

For example, `I = 4e18`:

- initialize the pool with `2000 BASE` and `8000 QUOTE` (`I = 4`; `R = ONE`)
 - sell just 1 wei of `QUOTE`, this sets `R = ABOVE_ONE`
 - execute the flashloan and transfer `3000 BASE` there.
 - `_adjustTarget()` is called and since `R == ABOVE_ONE`, the `BASE_TARGET` will be adjusted.
- Since `QUOTE_RESERVE - QUOTE_TARGET = 1 wei`, the `BASE_TARGET` is set to `BASE_RESERVE = ~5000` (2000 from initialization and 3000 from flashloan)
- In result the new targets are `BT = 5000`, `QT = 8000`, the price ratio has been completely skewed

RECOMMENDATION

Consider reworking the flashloan function to work with exact amounts, i.e no surpluses. This will allow you not to call `_sync()` and avoid all the issues.

PROOF OF CONCEPT <https://github.com/GuardianOrg/purrrswap-contracts-team2/pull/4>

M-14 Flashloan Sync Issues

CONTINUED

— RESOLUTION

Status: Resolved.

LOW **L-01**

ACKNOWLEDGED

AggregatorPriceProvider Cannot Add or Update

CATEGORY Code Best Practices
LOCATION AggregatorPriceProvider.sol
REVIEW Main Review

— DESCRIPTION

In the current implementation, all aggregator references in the `AggregatorPriceProvider` contract are set during construction and stored in the aggregators mapping. Once deployed, no administrative function exists to add new tokens or change existing aggregator addresses. This restricts the contract to the initial mapping configuration and prevents reconfiguration if aggregator addresses change or new tokens need coverage. If any aggregator feed becomes invalid or compromised, integrators cannot simply update the address; they must redeploy the entire contract.

— RECOMMENDATION

Either incorporate an `onlyOwner` function to add or update `aggregators[token]` after deployment or design the solution so that aggregator addresses can be retrieved from a more flexible registry. Otherwise, any aggregator change or new token listing will require redeploying the contract, which is costly and disruptive.

— RESOLUTION

Status: Acknowledged.

LOW **L-02**

ACKNOWLEDGED

Missing Checks In AggregatorPriceProvider

CATEGORY	Code Best Practices
LOCATION	AggregatorPriceProvider.sol: 31
REVIEW	Main Review

— DESCRIPTION

Within the `AggregatorPriceProvider.getPrice` function, there is no check to ensure the retrieved price from the underlying aggregator is positive, nor any mechanism to detect if the data is stale. If the aggregator feed returns zero, a negative value or a price timestamp beyond a freshness threshold, `getPrice` still returns it as-is. This can result in inaccurate on-chain pricing data being used by dependent systems, leading to potentially unsafe operations or unexpected reverts.

— RECOMMENDATION

Consider updating the `getPrice` function by adding checks for price positivity and staleness. One approach is to store the block timestamp or round timestamp from the aggregator, compare it to a maximum staleness window and revert or return an error if the data is outdated. Similarly, require the raw aggregator price to exceed zero to protect against zero or negative readings, which likely indicate invalid data.

— RESOLUTION

Status: Acknowledged.

LOW **L-03**

ACKNOWLEDGED

No Fallback Oracle

CATEGORY Code Best Practices
LOCATION `PurrLP.sol`: 223-225
REVIEW Main Review

— DESCRIPTION

The `PurrLP::getPMMState` function will set the `state.i` to the `retrieved oracle price` if the oracle has been set.

The `getPMMState` function is being used in the `querySellBase` and `querySellQuote` functions which are in turn used by the `sellBase` and `sellQuote` functions to determine the amount of `quote` and `base` tokens to send to the `recipient` and to later `adjust the target` values for both the `quote` and `base` tokens.

Hence `receiving` the `correct prices` from the oracle is critical for the `correct functioning` of the `PurrLP` contract.

But there is no mechanism in place to handle the fallback scenario in case the oracle fails or starts providing stale price data. This will lead to `DoS` of the `PurrLP` trades or `incorrect execution` of the `PurrLP` trades.

— RECOMMENDATION

Hence it is recommended to add a fallback oracle in the `PurrLP` contract incase the original oracle fails or start providing stale price data.

— RESOLUTION

Status: Acknowledged.

LOW **L-04**

RESOLVED

Unused Function `_resetTargetAndReserve`

CATEGORY Code Best Practices

LOCATION `PurrrLP.sol`: 586

REVIEW Main Review

DESCRIPTION

In `PurrrLP.sol`, the function `_resetTargetAndReserve` is unused and should be removed.

RECOMMENDATION

Remove the function `_resetTargetAndReserve`.

RESOLUTION

Status: Resolved.

LOW **L-05**

RESOLVED

Wrong FeesCollected Event Is Emitted

CATEGORY	Code Best Practices
LOCATION	PurrLP.sol: 516 PurrLP.sol: 533
REVIEW	Main Review

DESCRIPTION

The `PurrLP::_updateQuoteFees` and `PurrLP::_updateBaseFees` functions emit the `FeesCollected` event as shown below (when the `auto compounding is on`):

```
``solidity
emit FeesCollected(_treasury, 0, amount);
`
```

But the `amount` emitted here is wrong since the `_treasury` is only transferred `(amount * _treasuryFeeBips) / BIPS` amount when the auto compounding is on.

RECOMMENDATION

Recommended to store the `(amount * _treasuryFeeBips) / BIPS` value in a local variable and emit that variable in the `FeesCollected` event when the `auto compounding is on`.

RESOLUTION

Status: Resolved.

LOW **L-06**

ACKNOWLEDGED

I Should Account For Token Decimals

CATEGORY Validation
LOCATION `PurrrLP.sol`
REVIEW Main Review

— DESCRIPTION

The `Factory` contract lets users set the price ratio `_I_` of the deployed `PurrrLP` without enforcing a specific decimal format. Although the code often uses arithmetic with a "`1e18`" style from DODO libraries, it does not actually require `_I_` to be passed in 18-decimal form. This flexibility means the deployer is free to use an 8-decimal ratio, for example, if combining an 18-decimal token (like `WETH`) with a 6-decimal token (like `USDC`). If `_I_` is not correctly scaled to reflect each token's actual decimals, the pool will end up at a mismatched or nonsensical price, leaving it highly susceptible to arbitrages. The contract does not self-adjust or validate the ratio against the tokens' decimals, so any misalignment can produce a major mispricing from day one.

— RECOMMENDATION

When deploying a new pool, ensure that the `_I_` value accurately encodes the intended ratio in a consistent numeric scale. The code and front end must handle cases where base or quote tokens differ from the usual 18 decimals, factoring that difference into the ratio calculation. If, for instance, the base token is 18 decimals and the quote token is 6 decimals, confirm that `_I_` incorporates this 12-decimal offset if you intend a `1e18`-style ratio. Alternatively, provide documentation and checks in the factory contract that measure `_I_` against the tokens' decimals at deployment to catch potential misconfigurations early.

— RESOLUTION

Status: Acknowledged.

LOW **L-07**

RESOLVED

onlyKnownPool Restriction Can Be Bypassed

CATEGORY Validation
LOCATION Factory.sol, Router.sol
REVIEW Main Review

— DESCRIPTION

The `Router` won't allow executing actions for an unknown pool. The pool is checked to be known by querying the `Factory.poolExists()` function. However, `Factory.create()` is permissionless. This means any non removed pool will satisfy the condition in `onlyKnownPool`, not only the ones created from the router itself.

— RECOMMENDATION

Modify the `Factory.poolExists` mapping to `address creator => address pool => bool exists`. Then modify the check in `onlyKnownPool` to allow only pools deployed by the router.
IMPORTANT: you should use `msg.sender` for the creator, not `tx.origin`.

— RESOLUTION

Status: Resolved.

LOW **L-08**

ACKNOWLEDGED

Removed Pools Are Still Functioning

CATEGORY Unexpected Behavior

LOCATION `PurrLP.sol`

REVIEW Main Review

— DESCRIPTION

Whenever a pool is removed by calling `Factory.removePool()`, the router stops supporting that pool, however the pool itself is still functioning as before.

— RECOMMENDATION

Consider allowing only `PurrLP.sellShares()` if the pool is removed from the factory so LPs can withdraw their funds.

— RESOLUTION

Status: Acknowledged.

LOW **L-09**

RESOLVED

Unused `predictDeterministicAddress` Function

CATEGORY Code Best Practices

LOCATION `Factory.sol`

REVIEW Main Review

DESCRIPTION

The `predictDeterministicAddress` function in the `Factory` contract is never actually called by any other portion of the code or external contract. It represents a leftover utility that does not serve a current purpose and can cause confusion for developers or auditors.

RECOMMENDATION

Remove the `predictDeterministicAddress` function from the `Factory` contract to streamline the code and avoid misleading future integrators.

RESOLUTION

Status: Resolved.

LOW **L-10**

RESOLVED

Unneeded `_updateFor(address(0))` Call

CATEGORY	Validation
LOCATION	PurrrLP.sol
REVIEW	Main Review

DESCRIPTION

Within the `buyShares` function, the `PurrrLP` contract mints `1001` shares to `address(0)` and calls `_updateFor(address(0))`. Since the zero address is not an actual participant and cannot claim fees, this action creates meaningless accounting data.

RECOMMENDATION

Avoid calling `_updateFor` in the internal `_mint` function if the `to` address is the zero address.

RESOLUTION

Status: Resolved.

LOW **L-11**

ACKNOWLEDGED

Fee Model Should Not Change

CATEGORY Logical Error

LOCATION PurrrLP.sol

REVIEW Main Review

— DESCRIPTION

The function `setAutoCompoundFees` allows the implementation owner to switch between auto-compounding fees (where fees remain in the pool) and non auto-compounding fees (where fees are sent to an external contract for manual claiming).

The issue arises because all existing pools reference the fee model from the implementation contract. This means that a pool deployed under the assumption of auto-compounding fees could later be forced into a non auto-compounding model due to a contract-wide change.

This is particularly problematic for pool deployers that are smart contracts, as they may not have the necessary logic to manually claim fees. If the fee model is switched after deployment, these contracts could permanently lose unclaimed fees.

— RECOMMENDATION

Instead of dynamically reading the fee model from the implementation contract, enshrine the fee model at the time of pool deployment, making it immutable per pool. This ensures stability for deployed pools while still allowing the implementation owner to modify the fee model for future pools.

— RESOLUTION

Status: Acknowledged.

LOW **L-12**

ACKNOWLEDGED

No Check For Active L2 Sequencer

CATEGORY	Logical Error
LOCATION	AggregatorPriceProvider.sol
REVIEW	Main Review

— DESCRIPTION

Using Chainlink in L2 chains such as Arbitrum requires to check if the sequencer is down to avoid prices from looking like they are fresh although they are not according to their [recommendation]

(<https://docs.chain.link/data-feeds/l2-sequencer-feeds#arbitrum>)

In the `getPrice` function, the protocol calls `aggregator.latestRoundData` to retrieve the price from Chainlink. However, it does not check to see if the sequencer is down.

— RECOMMENDATION

Implement the recommended Chainlink [example code](<https://docs.chain.link/data-feeds/l2-sequencer-feeds#example-code>).

— RESOLUTION

Status: Acknowledged.

LOW **L-13**

ACKNOWLEDGED

treasuryFeeBips Is Not Fixed

CATEGORY Logical Error
LOCATION PurrrLP.sol: 170-174
REVIEW Main Review

— DESCRIPTION

In the `PurrrLP::_updateBaseFees` and `PurrrLP::_updateQuoteFees` functions are used to calculate the respective fee amounts. When the `auto compound is off` the `_LP_FEE_RATE_` is used to calculate the fee amount eligible for the `Liquidity providers`. Since the `_LP_FEE_RATE` can not be changed after initialization the `Liquidity providers` know exactly the percentage of fees they are going to receive via trades.

But the issue here is if the `auto compound is switched on` later then an additional `treasuryFeeBips` is counted on top of the `LPFee` and this `treasuryFeeBips` amount can be updated by the `implementation contract owner`. Hence the Liquidity Providers are no longer eligible for a fixed fee amount (since the `treasuryFeeBips` can be changed later) though they provided liquidity expecting that. Hence this could be opportunity cost to the Liquidity Providers and it is beyond their control.

— RECOMMENDATION

Hence it is recommended to keep the `treasuryFeeBips` fixed after the initialization similar to `_LP_FEE_RATE_`. This will ensure the Liquidity Provider fee distribution is fair, irrespective of whether the `auto compounding is on or off`

— RESOLUTION

Status: Acknowledged.

LOW **L-14**

ACKNOWLEDGED

Sandwich Large Trades To Steal Fees

CATEGORY Frontrunning
LOCATION PurrrLP.sol: 453-497
REVIEW Main Review

DESCRIPTION

The `PurrrLP.sol` contract is vulnerable to a form of `unfair fee claiming` due to the absence of a `lockup period` for `LP tokens` after they are minted via `buyShares`. This allows an attacker to `front-run large trades` (which updates the global fee indices (`index0` and `index1`), `deposit liquidity` just before `fees` are generated, and then `immediately withdraw liquidity` after the `fees` are accrued, disproportionately `claiming a share` of the fees that should have gone to `existing genuine liquidity providers`.

RECOMMENDATION

Consider adding a `lockup period` for `LP tokens` after they are minted (via `buyShares`). Hence once the `LP tokens` are minted the liquidity providers will not be able to redeem them during the lockup period. This will prevent users stealing fees using the same liquidity by `front-running` as described above.

RESOLUTION

Status: Acknowledged.

LOW **L-15**

RESOLVED

Unused Imports, Events And Errors

CATEGORY Code Best Practices

LOCATION PurrLP.sol

REVIEW Main Review

DESCRIPTION

The following imports, events and errors are unused in PurrLP:

- `import {IWETH} from "../interfaces/IWETH.sol";`
- `event TargetChanged`
- `event OracleChanged`
- `error ErrInvalidSignature()`

RECOMMENDATION

Remove the unused imports, events and errors.

RESOLUTION

Status: Resolved.

LOW **L-16**

ACKNOWLEDGED

Missing Checks In latestAnswer

CATEGORY	Validation
LOCATION	BalancerV2WeightedPoolAggregator.sol: 35-47
REVIEW	Main Review

— DESCRIPTION

In `BalancerV2WeightedPoolAggregator::latestAnswer`, the function retrieves the `weights` and `tokens` arrays. However, two issues could lead to reverts:

1. Each `weight` should be checked to ensure it is not `0`. A zero weight would cause a division by zero, reverting the entire transaction.
2. The function iterates using `tokens.length` but assumes both arrays (tokens and weights) have the same length. If they are mismatched, an out-of-bounds error will occur when accessing `weights[i]`, causing the transaction to revert mid-loop.

— RECOMMENDATION

Before entering the loop, validate that `tokens.length == weights.length` and ensure that no weight is `0` to prevent unexpected failures.

— RESOLUTION

Status: Acknowledged.

LOW **L-17**

RESOLVED

Missing nonReentrant On correctRState

CATEGORY Code Best Practices

LOCATION PurrrLP.sol

REVIEW Main Review

DESCRIPTION

Finding L-03 \ correctRState LP Price Manipulation (<https://www.notion.so/guardianaudits/L-03-1858bda5828c80488a6ec72be2b8d056>) remains unaddressed.

RECOMMENDATION

Add a `nonReentrant` modifier to function `correctRState`.

RESOLUTION

Status: Resolved.

LOW **L-18**

ACKNOWLEDGED

Oracle Incompatibility

CATEGORY	Validation
LOCATION	PurrrLP.sol
REVIEW	Main Review

— DESCRIPTION

When launching a `PurrrLP` pool with a 18 decimals base token and a 6 decimals as the quote token, similar to the [MIM/USDC currently deployed in Arbitrum] (<https://arbiscan.io/address/0x8279699D397ED22b1014fE4D08fFD7Da7B3374C0>), the target price `_I_` must account for the six-decimal nature of the quote token. A 1:1 ratio from a user's perspective equates to an `_I_` of `1e6` on chain. However, most oracles natively operate in 8 or 18 decimals, leaving no straightforward price feed that produces a price with 6-decimal scaling. In this situation a custom oracle implementation with 6 decimals will be necessary.

— RECOMMENDATION

Consider implementing an oracle "adapter" that normalizes its output price to the required decimals.

— RESOLUTION

Status: Acknowledged.

LOW **L-19**

ACKNOWLEDGED

Router Decimal Validation Can Be Bypassed

CATEGORY Validation

LOCATION Router.sol: 761, Factory.sol: 76

REVIEW Main Review

— DESCRIPTION

In Router.sol, the _validateDecimals check prevents pool creation if:

- The tokens have zero decimals
- A token has more than 18 decimals.
- The decimal difference between tokens is greater than 12.

This ensures that extreme decimal values do not cause overflow or precision issues.

However, this validation can be bypassed by calling create directly in the factory contract. Pools created this way are still added to the Factory's pools mapping, which may give users a false sense of security, assuming they were properly validated.

— RECOMMENDATION

In the factory contract's create function, enforce that only the Router contract can call it. This ensures all pools go through proper validation

— RESOLUTION

Status: Acknowledged.

LOW **L-20**

RESOLVED

Oracle Cannot Be Updated

CATEGORY Code Best Practices

LOCATION `PurrLP.sol`: 131

REVIEW Main Review

DESCRIPTION

Once the `_ORACLE_` is initialized in the `PurrLP::init` there is no ability to update it should the need arises in the future (due to an oracle failure etc.). Without a functioning oracle, all trade activity will fail. The presence of an `OracleChanged` event also suggests that an update function should have been implemented.

RECOMMENDATION

Consider implementing an admin-only oracle update function.

RESOLUTION

Status: Resolved.

LOW **L-21**

ACKNOWLEDGED

Target Fluctuations Because of Rounding

CATEGORY Informational
LOCATION PurrrLP.sol, PMMPricing.sol
REVIEW Main Review

— DESCRIPTION

When targets are computed in `PurrrLP._adjustTarget()` and `PMMPricing.adjustedTarget()`, `reciprocalFloor()` is used to get the QUOTE/BASE price. This function performs division and rounds down the result. This difference in the price will cause the targets to slightly fluctuate over time.

— RECOMMENDATION

Be aware of this risk.

— RESOLUTION

Status: Acknowledged.

LOW **L-22**

ACKNOWLEDGED

WETH Should Be WHYPE

CATEGORY Informational

LOCATION Router.sol

REVIEW Main Review

— DESCRIPTION

The Router offers functionality for working with ETH by wrapping it to WETH. However, the protocol will be deployed to Hyperliquid where the native token is not ETH, but HYPE and there is no WETH, but WHYPE.

— RECOMMENDATION

Rename the ETH/WETH to HYPE/WHYPE.

— RESOLUTION

Status: Acknowledged.

LOW **L-23**

ACKNOWLEDGED

Not All Tokens Implement decimals()

CATEGORY DoS
LOCATION PurrLP.sol: 197
REVIEW Main Review

— DESCRIPTION

The PurrLP is designed to work permissionlessly with all types of tokens. However, in the `decimals` view function, the base token is expected to implement: `IERC20Metadata(* BASE_TOKEN *).decimals()`. According to EIP-20, it is optional for tokens to implement the `decimals()` function. So PurrLP's `decimals` function should be able to handle tokens that do not implement `decimals()`.

— RECOMMENDATION

In PurrLP's `decimals` function, add a `try-catch` and return the standard `18` decimals if the token does not implement `decimals()`.

— RESOLUTION

Status: Acknowledged.

LOW **L-24**

RESOLVED

Lack of Oracle Price Validation Checks In PurrLP

CATEGORY Code Best Practices

LOCATION PurrLP.sol: 224

REVIEW Main Review

— DESCRIPTION

In the `getPMMState` function, the contract retrieves the latest price using `IPurrLPOracle(*ORACLE*).price(address(*BASE_TOKEN*))` when an oracle is defined. However, there are no checks to validate this fetched price, which can lead to potential issues if the oracle returns unexpected data.

— RECOMMENDATION

Implement validation checks on the price retrieved from the oracle. At a minimum, replicate the checks used in the `init` function to ensure the price is neither zero nor exceeds `MAX_I`.

— RESOLUTION

Status: Resolved.

LOW **L-25**

ACKNOWLEDGED

Aggregator Integration Issues

CATEGORY	Code Best Practices
LOCATION	BalancerV2WeightedPoolAggregator.sol
REVIEW	Main Review

— DESCRIPTION

The `BalancerV2WeightedPoolAggregator` contract uses the same interface as the chainlink oracles.

However, its `latestRoundData()` hardcodes zeroes for anything besides the price.

It's common for integrators to check whether the returned data is stale and etc... Because the returned values are 0, these calls will always revert.

— RECOMMENDATION

Document that integrators should expect 0s for these return values.

— RESOLUTION

Status: Acknowledged.

LOW **L-26**

ACKNOWLEDGED

Possible Division By 0 Weight

CATEGORY	Code Best Practices
LOCATION	BalancerV2WeightedPoolAggregator.sol
REVIEW	Main Review

— DESCRIPTION

The `BalancerV2WeightedPoolAggregator.sol` divides by `weight` without checking if it's 0 or not. In practice, the weight should never be 0, but if it happens, this will DOS the entire function. On the other hand, a division by zero will also occur if `totalSupply` is 0.

— RECOMMENDATION

Consider returning a 0 price if `totalSupply` is equal to 0.

— RESOLUTION

Status: Acknowledged.

LOW L-27

RESOLVED

Zero Address Token Aggregator Allowed

CATEGORY	Code Best Practices
LOCATION	AggregatorPriceProvider.sol
REVIEW	Main Review

DESCRIPTION

Once set in the constructor, the aggregators in `AggregatorPriceProvider.sol` can't be changed. The constructor doesn't check for `address(0)` in the `_tokenAggregators` array. If the deployer makes a mistake and passes `address(0)`, the aggregator can never be set again.

RECOMMENDATION

Consider reverting the transaction if any of the aggregators is `address(0)`.

RESOLUTION

Status: Resolved.

LOW L-28

RESOLVED

DOS for aggregators with > 18 decimals

CATEGORY	DoS
LOCATION	AggregatorPriceProvider.sol
REVIEW	Main Review

— DESCRIPTION

The code in `AggregatorPriceProvider.sol` adjust the returned price by the aggregator so the end result will be in 18 decimals. However, it does so by subtracting the aggregator decimals from `WAD`

```
price * (int256(10) ** (WAD - decimals))
```

This means for any aggregators which have `decimals > 18`, the transaction will be reverting.

— RECOMMENDATION

Change the calculation to

```
price * (int256(10) ** WAD) / (10 ** decimals)
```

— RESOLUTION

Status: Resolved.

LOW **L-29**

ACKNOWLEDGED

Lack of Slippage Protection

CATEGORY Informational

LOCATION `PurrLP.sol`

REVIEW Main Review

— DESCRIPTION

There is no slippage protection in the functions of the `PurrLP` contract. This exposes users who directly interact with the contract without the router to a risk of losing their assets.

— RECOMMENDATION

Document the risk.

— RESOLUTION

Status: Acknowledged.

LOW **L-30**

RESOLVED

Division Before Multiplication

CATEGORY Code Best Practices

LOCATION Math.sol

REVIEW Main Review

DESCRIPTION

The `Math._GeneralIntegrate()` function computes $(v0^2 / v1) / v2 * k$ as `penalty`. Notice how two divisions are executed before the actual multiplication. This will make the `GeneralIntegrate()` return inaccurate value.

RECOMMENDATION

Rewrite the formula to $v0^2 * k / (v1 * v2)$.

RESOLUTION

Status: Resolved.

LOW **L-31**

ACKNOWLEDGED

Auto-Compounding Fees Not Customizable

CATEGORY Informational

LOCATION PurrLP.sol: 159

REVIEW Main Review

— DESCRIPTION

The autoCompoundFees variable is retrieved from the original implementation contract for every pool, meaning it cannot be customized on a per-pool basis. This limits the flexibility of the protocol, as all pools must follow the same auto-compounding setting, even if some pools would benefit from different fee handling strategies.

— RECOMMENDATION

Consider storing autoCompoundFees as a state variable at the pool level to allow per-pool customization.

— RESOLUTION

Status: Acknowledged.

LOW **L-32**

ACKNOWLEDGED

Direct Donations Can Create Imbalances

CATEGORY Unexpected Behavior

LOCATION PurrLP.sol: 399

REVIEW Main Review

— DESCRIPTION

The PurrLP contract uses the RState, which is derived from token reserves and targets, to optimize trade calculations. However, in certain scenarios, RState may not accurately reflect the true contract state. If donations are made to the pool, for instance, if liquidity is added in an imbalanced manner, the excess amount may lead to a mismatch between the stored RState and actual reserves state. This discrepancy affects future traders, as they may not receive the optimized trade value expected under the protocol's model.

— RECOMMENDATION

Be aware of potential discrepancies in RState when liquidity is added in an imbalanced manner or through direct donations. These situations could lead to inconsistencies that impact future traders.

— RESOLUTION

Status: Acknowledged.

LOW **L-33**

ACKNOWLEDGED

Managed Pools Lack Aggregator Flexibility

CATEGORY Informational

LOCATION AggregatorPriceProvider.sol: 21

REVIEW Main Review

— DESCRIPTION

The AggregatorPriceProvider contract sets the aggregators during its construction, meaning that no new ones can be added later. This setup works fine for Balancer's standard Weighted Pools. However, for Managed Pools—which are Weighted Pools with mutable tokens and weights—this limitation creates an issue. If new tokens are added to a Managed Pool, their aggregator cannot be registered in the current implementation.

— RECOMMENDATION

Be aware of this limitation when working with Managed Weighted Pools.

— RESOLUTION

Status: Acknowledged.

HIGH **H-01**

Pre-deployment Token Transfers Can Bypass First Deposit Ratio Enforcement

CATEGORY	Validation
LOCATION	PurrrLP.sol
REVIEW	Remediation Review

— DESCRIPTION

The `PurrrLP` contract attempts to ensure that the initial deposit of base and quote tokens aligns with the target price specified by `_I_` (or by an oracle) when `totalSupply() == 0`. This mechanism, implemented via an `_adjustAddLiquidity` internal function in the `Router`, forces the very first liquidity provider to deposit base and quote tokens in the correct proportion, preserving the PMM model's assumptions about the pool's initial balance and respecting the `RState.ONE`. However, because these pool addresses can be precomputed before deployment, it is possible for an attacker to transfer tokens to the soon-to-exist pool address before the pool contract is actually deployed and initialized. Once the contract is deployed and calls `init`, it sets `_BASE_RESERVE_` and `_QUOTE_RESERVE_` to whatever tokens are already present, ignoring whether these tokens match the intended ratio. No LP shares are minted for those pre-deposited tokens, but the pool's reserves are nonetheless "skewed" from the moment of deployment. This leaves the pool in a potentially imbalanced state (e.g., `_RState * _` remains set to `ONE` by default, even though the actual base-to-quote ratio could diverge significantly). Subsequent calls to `buyShares` or the normal "initial deposit" logic see those pre-initialization tokens already in the reserves and do not enforce the ratio on them. As a result, the first trades against the pool can be mispriced, and the protocol's assumption that the pool was initially "balanced at ratio `_I_`" is violated.

— RECOMMENDATION

Enforce the correct ratio from the very start by handling any pre-deployed token transfers in the `init` function. This can be achieved by transferring to the pool deployer any base or quote token present in the `PurrrLP` contract upon completion of the `init` function.

— RESOLUTION

Status: Reported during the fix review period.

MEDIUM **M-01**

Incorrect Refund In createPoolETH Function

CATEGORY	Logical Error
LOCATION	Factory.sol
REVIEW	Remediation Review

DESCRIPTION

Within the `createPoolETH` function, the contract attempts to refund any leftover ETH to the sender by transferring it as WETH tokens. The code deposits only a portion of `msg.value` into WETH (the amount actually needed for the liquidity addition), but then calls `address(weth).safeTransfer` for the remainder. Since the Router only holds exactly `ethAmountIn` worth of WETH (the portion minted), it does not have the extra leftover portion in WETH form. If `leftover > 0`, this transfer fails because the contract's WETH balance is insufficient, causing the transaction to revert. As a result, unused ETH cannot be reclaimed by the user and legitimate liquidity operations involving surplus ETH can break.

RECOMMENDATION

Replace the final leftover transfer call with either a raw ETH transfer or a properly handled WETH transfer. The simplest fix is to perform a direct ETH transfer for any remainder, for example:

`refundTo.safeTransferETH(msg.value - ethAmountIn)`. If the intended behavior is to refund WETH, the code must mint WETH for the entire `msg.value` up front, then transfer the remainder out as WETH. Either approach ensures that users can safely reclaim their unspent funds without causing the transaction to revert.

RESOLUTION

Status: Reported during the fix review period.

MEDIUM **M-02**

Salt collision for router pools

CATEGORY	Unexpected Behavior
LOCATION	Factory.sol:L69
REVIEW	Remediation Review

— DESCRIPTION

The `poolExists` mapping was reworked to include the creator as suggested in [L-12] (<https://www.notion.so/guardianaudits/L-12-1b58bda5828c80688733d9d2f412b99b?pvs=4>) to differentiate between pools created by the router and any other pools. Because of this `Factory.create()` assigns `creator = msg.sender` and then uses the creator to compute the salt. For calls coming from the router the `creator` will always be the router which makes the salt behave as if the field of the creator is not used at all in these cases. There are 2 important outcomes from this change:

- Once a pool is created through the router, its exact settings can never be used again to create a new pool because the salt will be the same. This means if user A creates a pool with settings A through the router, user B can never use this settings to create its own pool.
- Building upon the previous bullet point, `Router.createPool()` can be frontrun by a malicious user and make it revert. This is unexpected because the malicious user can as well deposit funds and force the initial caller to not be the first depositor anymore. In addition, if `Router.createPool()` is used by a smart contract that doesn't handle this revert case, it can be DOSed permanently, depending on the implementation.

— RECOMMENDATION

There are several different solutions you can implement to fix the problem:

Solution 1

Instead of `creator`, pass `tx.origin` to `_computeSalt()`.

Pros:

- minimal changes
- `salt` stays unique per each user for different pool settings

Cons:

- The pools in the `userPools` array are accounted to the router instead of the original creator

Solution 2

Change back `creator` to `tx.origin` and add a new ``mapping(address pool => bool`

M-02 Salt collision for router pools

CONTINUED

isRouted) and mark each pool created through the router as true. Then modify the onlyKnownPool modifier to work with the mapping.

Pros:

- salt stays unique per each user for different pool settings
- userPools and poolExists` are still tied to the original creator

Cons:

- Requires more code changes

— RESOLUTION

Status: Reported during the fix review period.

MEDIUM **M-03**

Incorrect router ETH refund

CATEGORY	Unexpected Behavior
LOCATION	Router.sol#L144
REVIEW	Remediation Review

DESCRIPTION

The `Router.createPoolETH()` function was changed to adjust the amount of tokens added to the current price ratio. It computes the `ethAmountIn` and `tokenInAmount` amounts to be added, deposits `ethAmountIn` into `WETH`, sends the received `WETH` and its paired token to the `PurrLP` and finally buys shares. In cases where `msg.value > ethAmountIn` the excess `ETH` should be refunded to the user. Currently, the router transfers the excess to the user in the form of `WETH`. However, the router deposited only `ethAmountIn`. Because of this, all refunds will be reverting due to insufficient funds and the pool creations will be failing (unless the router somehow holds `WETH`).

RECOMMENDATION

Refund the excess amount in native token, not `WETH`.

RESOLUTION

Status: Reported during the fix review period.

MEDIUM **M-04**

Pool Ratio Skewed Due To Rounding

CATEGORY	Rounding
LOCATION	Router.sol, PurrLP.sol
REVIEW	Remediation Review

— DESCRIPTION

The main audit finding 'M-09 Pool Ratio Can Be Skewed Due To Rounding' has not been fixed.

While the Router contract now performs adjustment of BASE/QUOTE input amounts via

`_adjustAddLiquidity` this does not resolve the issue as `_adjustAddLiquidity` has the same logic as `PurrLP::buyShares`.

The pool ratio can still be skewed due to rounding which leads to a discrepancy between `_BASE_TARGET_ / _QUOTE_TARGET_` and the `i` value.

— RECOMMENDATION

Use `mulCeil` instead of `mulFloor` when calculating `shares`:

```
``plain text
shares = quoteBalance < DecimalMath.mulFloor(baseBalance, _i) ?
DecimalMath.divFloor(quoteBalance, _i) : baseBalance;
``
```

— RESOLUTION

Status: Reported during the fix review period.

LOW **L-01**

Flashloan Interface, Event And Error Not Removed

CATEGORY	Informational
LOCATION	PurrrLP.sol, IPurrrLP.sol
REVIEW	Remediation Review

— DESCRIPTION

In response to a main audit finding, the `flashloan` functionality was removed from the codebase. However, the associated function interfaces, event and error were not removed.

— RECOMMENDATION

Remove the 1) `Flashloan` event, 2) `ErrFlashLoanFailed` error, 3) `FlashLoanCall` from `ICallee.sol`, and 4) `flashLoan` from `IPurrrLP.sol`.

— RESOLUTION

Status: Reported during the fix review period.

LOW **L-02**

Obsolete Flashloan Artifacts Remain After Feature Removal

CATEGORY	Code Best Practices
LOCATION	PurrLP.sol
REVIEW	Remediation Review

— DESCRIPTION

The flashloan feature was removed from the codebase. However, remnants of this functionality remain defined in the contract: the flashloan interface, the associated event and the `ErrFlashLoanFailed` error. Because the underlying logic has been taken out, this code is now superfluous and their presence implies functionality that no longer exists.

— RECOMMENDATION

Cleanly remove all references to the obsolete flashloan feature. This includes any interfaces, events and custom errors (`ErrFlashLoanFailed`) associated with flashloans.

— RESOLUTION

Status: Reported during the fix review period.

LOW L-03

createPools Calls Through The Router Can Be Front-run

CATEGORY	Warning
LOCATION	Router.sol
REVIEW	Remediation Review

— DESCRIPTION

To differentiate router-created pools from others, the `poolExists` mapping was modified to include the creator address in computing the salt. However, for calls originating from the router, `creator = msg.sender` is always the router itself, rendering that field effectively unused. As a result, the same salt will be reused whenever identical pool settings are passed through the router, leading to two critical outcomes:

- If user A creates a pool with a particular set of parameters (baseToken, quoteToken, feeRate, etc.) via the router, user B cannot create another identical pool because the salt computation yields the same result, causing the second creation to revert.
- A malicious user can preemptively call the `Factory` with the same parameters and deposit funds before the genuine router call arrives. This forces the router's creation attempt to revert. If a downstream contract depends on `Router.createPool` and is not coded to handle reverts gracefully, it could be permanently DoSed, preventing that call from ever completing successfully.

— RECOMMENDATION

Consider adding `tx.origin` also to the `salt` computation in the `Factory` contract.

— RESOLUTION

Status: Reported during the fix review period.

LOW **L-04**

sync Can Use getEffectiveI() Directly

CATEGORY	Code Best Practices
LOCATION	PurrLP.sol 177, PurrLP.sol 430
REVIEW	Remediation Review

— DESCRIPTION

The `sync()` function builds the state by calling `getPMMState()` and then reads only the `i` field, which is equivalent to `getEffectiveI()`. Instead, it can directly call `getEffectiveI()`. The same is true for the `_sync()` call in the `sellShares()` function.

— RECOMMENDATION

```
``diff
function sync() external nonReentrant onlyClones onlyImplementationOwner {
- PMMPricing.PMMState memory state = getPMMState();
- _sync(state.i);
+ _sync(getEffectiveI());
}
\
`diff
- PMMPricing.PMMState memory state = getPMMState();
- _sync(state.i);
+ _sync(getEffectiveI())
``
```

— RESOLUTION

Status: Reported during the fix review period.

LOW **L-05**

Unused Event and Error

CATEGORY	Best Practices
LOCATION	https://github.com/GuardianOrg/purrswap-contracts-team2/blob/484cb7dfddb01f691c489ac0e46676fe702a7ffd/contracts/PurrLP.sol#L31
REVIEW	Remediation Review

— DESCRIPTION

The flashLoan function was removed in the recent update, but the associated FlashLoan event and ErrFlashLoanFailed error remain in the contract.

— RECOMMENDATION

Remove the unused FlashLoan event and ErrFlashLoanFailed error.

— RESOLUTION

Status: Reported during the fix review period.

LOW **L-06****correctRState Can Be Reentered**

CATEGORY	Warning
LOCATION	PurrLP.sol 138
REVIEW	Remediation Review

— DESCRIPTION

The `correctRState()` function lacks the `nonReentrant` modifier which means that if the `PurrLP` contract uses tokens with hooks they can call `correctRState()` in the middle of the execution. For example, `sellShares()` calls `_transferBaseOut()` and `transferQuoteOut()` before further interactions. This can lead to unexpected behaviors.

— RECOMMENDATION

Consider adding the `nonReentrant` modifier to `correctRState()`

— RESOLUTION

Status: Reported during the fix review period.

LOW **L-07**

Pool Creation Griefed By Front-running

CATEGORY	Frontrunning
LOCATION	Factory.sol
REVIEW	Remediation Review

— DESCRIPTION

PurrLP pools deployed from the Router contract have a predictable `salt` because the Factory contract uses `creator = msg.sender`, which is always the Router.

This allows grieving of pool deployments by front-running with another transaction to create the same pool. While there is no significant risk as the owner of the clone does not have any special privileges, this behavior may be unexpected for pool deployers.

— RECOMMENDATION

Consider updating the `Factory::create` function to use both `tx.origin` and `msg.sender`:

- Use `tx.origin` to compute the salt to ensure that the pool address is user-specific and unpredictable.
- Use `msg.sender` for `_addPool` to correctly associate the deployed pool with the Router.

— RESOLUTION

Status: Reported during the fix review period.

§ 05

Disclaimer.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Guardian to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Guardian's position is that each company and individual are responsible for their own due diligence and continuous security. Guardian's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by Guardian is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis.

Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract's safety and security. Therefore, Guardian does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

§ 06

About Guardian.

Guardian is a smart contract security firm. Every engagement is worked by two independent teams competing on the same scope, each with its own tooling and methods, so no single reviewer's blind spot survives.

\$45B+DIGITAL ASSETS UNDER
REVIEW**200+**

ENGAGEMENTS COMPLETED

500+CRITICAL & HIGH FINDINGS
REPORTED

Reach us at audits@guardianaudits.com.