GA GUARDIAN

Baseline

Mercury AMM

Security Assessment

May 18th, 2026

Summary

Audit Firm Guardian

Prepared By Owen Thurm, Minato Namikazi, Zdravko Hristov, CuriousApple, Vladimir Zotov

Client Firm Baseline

Final Report Date May 18, 2026

Audit Summary

Baseline engaged Guardian to review Mercury AMM in scope. From January 21 through May 15, 2026, a team of 5 auditors reviewed the source code across six review rounds. All findings and follow-up results are recorded in this report.

Confidence Ranking

Given that most High and Critical issues were identified across iterative review rounds, Guardian assigns a Confidence Ranking of 4 to the protocol.

The progression of findings and remediations over multiple rounds supports a strong level of confidence in the reviewed codebase. Guardian advises the protocol to consider periodic review with future changes.

For detailed understanding of the Guardian Confidence Ranking, please see the rubric on the following page.

✅ Verify the authenticity of this report on Guardian's GitHub: https://github.com/guardianaudits

Guardian Confidence Ranking

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. Low risk of latent critical issues.	0 High/Critical findings and few Low/Medium severity findings.
4: High Confidence	Code is clean, well-structured, and adheres to best practices. Only 1 Significant issue was uncovered per week. 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 issues. Varied Low severity findings.
3: Moderate Confidence	Medium-severity and occasional High-severity issues found. Code is functional, but there are concerning areas (e.g., weak modularity, risky patterns). No critical design flaws, though some patterns could lead to issues in edge cases. **Recommendation:** Address issues thoroughly and consider a targeted follow-up audit depending on code changes.	1-2 High/Critical findings per engagement week.
2: Low Confidence	Code shows frequent emergence of Critical/High vulnerabilities. Audit revealed recurring anti-patterns, weak test coverage, or unclear logic. These characteristics suggest a high likelihood of latent issues. **Recommendation:** Post-audit development and a second audit cycle are strongly advised.	2-4 High/Critical findings per engagement week. Or additional High/Critical findings uncovered in remediation review which have not been resolved and confirmed by Guardian.
1: Very Low Confidence	Code has systemic issues. Multiple High/Critical findings (≥5/week), poor security posture, and design flaws that introduce compounding risks. Safety cannot be assured. **Recommendation:** Halt deployment and seek a comprehensive re-audit after substantial refactoring.	≥5 High/Critical findings and overall systemic flaws.

Table of Contents

Project Overview

Project Summary

Project Name	Baseline
Language	Solidity
Codebase	https://github.com/0xBaseline/mercury
Commit(s)	**Main Review:** 513ed6e354c668669011f2a0c71d5ea649052e00 **Final Round:** e16a15903babb28b0230b58b712ebef6c1ab9cb2

Audit Summary

Delivery Date	May 18, 2026
Audit Methodology	Static Analysis, Manual Review, Fuzzing

Vulnerability Summary

Vulnerability Level	Total	Pending	Declined	Acknowledged	Partially Resolved	Resolved
Critical	4	0	0	0	0	4
High	9	0	0	0	1	8
Medium	28	0	0	2	3	23
Low	33	0	0	1	0	32
Info	35	0	0	7	0	28

Audit Scope & Methodology

BToken.sol
Component.sol
Relay.sol
ConfigScript.sol
BController.sol
BCredit.sol
BFactory.sol
BHook.sol
BLens.sol
BStaking.sol
BSwap.sol
BlockPricingLib.sol
CollateralLib.sol
CurveLib.sol
FeeLib.sol
GuardLib.sol
MakerLib.sol
NativeLib.sol
NormalizeLib.sol
StateLib.sol
SwapContextLib.sol
SweepLib.sol

Audit Scope & Methodology

Vulnerability Classifications

Severity	Impact: *High*	Impact: *Medium*	Impact: *Low*
Likelihood: *High*	🔴 Critical	🟠 High	🟡 Medium
Likelihood: *Medium*	🟠 High	🟡 Medium	🟢 Low
Likelihood: *Low*	🟡 Medium	🟢 Low	🟢 Low

Impact

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

Medium A small amount of funds can be lost or ancillary functionality of the protocol is affected. The user or protocol may experience reduced or delayed receipt of intended funds.

Low Can lead to any unexpected behavior with some of the protocol's functionalities that is notable but does not meet the criteria for a higher severity.

Likelihood

High The attack is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount gained or the disruption to the protocol.

Medium An attack vector that is only possible in uncommon cases or requires a large amount of capital to exercise relative to the amount gained or the disruption to the protocol.

Low Unlikely to ever occur in production.

Audit Scope & Methodology

Methodology

Guardian is the ultimate standard for Smart Contract security. An engagement with Guardian entails the following:

- Two competing teams of Guardian security researchers performing an independent review.
- A dedicated fuzzing engineer to construct a comprehensive stateful fuzzing suite for the project.
- An engagement lead security researcher coordinating the 2 teams, performing their own analysis, relaying findings to the client, and orchestrating the testing/verification efforts.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross-referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts. Comprehensive written tests as a part of a code coverage testing suite.
- Contract fuzzing for increased attack resilience.

Round One Findings & Resolutions

ID	Title	Category	Severity	Status
C-01	Reserve Tokens Can Be Drained	Logical Error	Critical	Resolved
C-02	Protocol Reserves Can Be Drained	Validation	Critical	Resolved
H-01	Pool Fees Are Lost When Claimed	Logical Error	High	Resolved
H-02	Vault Withdrawals DOS Due To Solvency Check	DoS	High	Resolved
H-03	Stale K Enables Reserve Extraction	Logical Error	High	Resolved
M-01	Configured feeRecipient Ignored In Payouts	Logical Error	Medium	Resolved
M-02	Staking Rewards Can Be Globally Griefed & Frozen	DoS	Medium	Resolved
M-03	Reverting Swaps Due To Underflow	DoS	Medium	Resolved
M-04	User Favorable Rounding In Sell Path	Rounding	Medium	Resolved
M-05	BLV Translation Uses Outdated State	Logical Error	Medium	Resolved
M-06	Vault Exit Loss Not Propagated To Curve Math	Math	Medium	Resolved
M-07	Zero Share Deposits Can Brick Protocol Sweeps	DoS	Medium	Resolved
M-08	takeReserves() Can Result In 0 Shares Minted	Unexpected Behavior	Medium	Resolved

Round One Findings & Resolutions

ID	Title	Category	Severity	Status
M-09	BCredit.defaultSelf Off Curve	Logical Error	Medium	Resolved
M-10	Swap Fee Can Be Bypassed When Selling All Tokens	Gaming	Medium	Acknowledged
M-11	Entering Vault For A bToken Bricks Its Pool	DoS	Medium	Resolved
M-12	Convexity Parameter Is Being Overrelaxed	Math	Medium	Resolved
M-13	Underreported Supply Drains Shared Reserves	Gaming	Medium	Resolved
L-01	Swaps Bricks When convexityExp > 2e18	DoS	Low	Resolved
L-02	Unaccounted Dust Desync Reserves Accounting	Unexpected Behavior	Low	Resolved
L-03	Pool Creation May Fail Due To Underflow	DoS	Low	Resolved
L-04	Unused Code In Quote Functions	Superfluous Code	Low	Resolved
L-05	Smaller Swaps Reverting Due To Fee Rounding To 0	DoS	Low	Resolved
L-06	Remove Final Correction From Buy/sell Paths	Best Practices	Low	Resolved
L-07	Asymmetrical Validation For Buy Path	Validation	Low	Resolved
L-08	Incorrect Event Emission For Hook Swaps	Events	Low	Resolved

Round One Findings & Resolutions

ID	Title	Category	Severity	Status
L-09	solveBuy() Can Revert Due To Overflow	Math	Low	Resolved
L-10	Buy Fee Path Underflow Can Revert Swaps	DoS	Low	Resolved
L-11	Buffer Can Grow During Sells	Math	Low	Resolved
L-12	Hook Sweep Can Revert Batched Swaps	DoS	Low	Resolved
L-13	Recorded Reserves Drift Due To Vault Operations	Math	Low	Resolved
L-14	Hook Sweep Can Under-credit Vault	Math	Low	Resolved
L-15	Harvest Buffer Not Retained On Vault Exit	Math	Low	Resolved
L-16	Harvest Buffer Can Be Dynamically Adjusted	Suggestion	Low	Resolved
L-17	Pool Reserves Overstated On createPool()	Rounding	Low	Resolved
L-18	Duplicate Claim Leaves Enable Griefing	Trust Assumptions	Low	Resolved
L-19	Leverage Fee Charged On Max Borrow	Math	Low	Resolved
L-20	Ratio Rounding Favors Users On Buys And Sells	Rounding	Low	Resolved
I-01	Missing Guard Causes Sell Side Reverts	Informational	Info	Resolved

Round One Findings & Resolutions

ID	Title	Category	Severity	Status
I-02	Excess Repayment Funds Not Refunded	Informational	Info	Resolved
I-03	Repay Preview Revert When Debt Is 0	Suggestion	Info	Resolved
I-04	harvestYield Has A Return But Doesn't Return It	Informational	Info	Resolved
I-05	Missing Upper Bound Blocks Pool Creation	Validation	Info	Resolved
I-06	isExecutor() Can Use Dirty Address	Best Practices	Info	Resolved
I-07	Relay Routes Are Unnecessary	Superfluous Code	Info	Acknowledged
I-08	Incorrect Ceil Comments	Best Practices	Info	Resolved
I-09	Redundant mulWad() In Active Price Computation	Superfluous Code	Info	Resolved
I-10	Unreachable Edge Cases In computeSwap()	Suggestion	Info	Resolved
I-11	Binary Search Loop Can Break Earlier	Gas Optimization	Info	Resolved
I-12	Dead Pool.shareBalance State Field	Superfluous Code	Info	Resolved
I-13	K Decreases Due To Computation Inconsistencies	Rounding	Info	Acknowledged
I-14	Vaults Are A Security Risk	Warning	Info	Acknowledged

Round One Findings & Resolutions

ID	Title	Category	Severity	Status
I-15	Unused previewTakeReserves() Helper	Superfluous Code	Info	Resolved
I-16	Outdated Refund Ordering Comment	Documentation	Info	Resolved
I-17	Invariant Error Lacks Drop Context	Best Practices	Info	Resolved
I-18	Check Computed Blv Against Book Price	Best Practices	Info	Resolved
I-19	Lack Of BToken Parameters Validation	Validation	Info	Acknowledged

C-01 | Reserve Tokens Can Be Drained

Category	Severity	Location	Status
Logical Error	● Critical	BFactory.sol	Resolved

Description

In `BFactory.createPool()`

```
    // pool asset accounting
    pool.totalReserves = (params.initialPoolReserves + params.initialDebt).toUint128();
    // only transfer initial reserves from caller.
    pool.reserve.takeReserves(msg.sender, params.initialPoolReserves);
```

We only collect initialPoolReserves from the creator, then credit the pool with initialDebt as if it were reserves by adding it into pool.totalReserves.

A permissionless pool creator can set initialDebt arbitrarily large and make the AMM believe the pool has huge reserves.

The sell path uses curve params derived from totalReserves, so the curve now believes the pool is backed by a huge amount of reserves.

```
    params_.reserves = NormalizeLib.normalizeWad(pool.totalReserves, pool.reserveDecimals);
```

And every place that pays out reserves uses the ERC20 balance held by the Relay, not per-pool balance

```
    reserve.giveReserves(msg.sender, _deltaUserReserves.toUint256(), false)
```

`VaultLib.giveReserves()` pulls from

```
    State.Allocation storage allocation = State.meta().allocations[_reserve];
```

An attacker can drain reserve tokens of other pools by selling, as demonstrated in the PoC
https://github.com/GuardianOrg/mercury-team1-1768594959614/blob/master/test/unit/InitialDebtReserveDrain.poc.t.sol

Recommendation

If initialDebt needs to be reflected in accounting, consider tracking it separately

Resolution

Baseline:
https://github.com/0xBaseline/mercury/blob/e81f3a58403bde43d7b37b9a77bdc3d851ae8f47/src/components/BFactory.sol#L228.

C-02 | Protocol Reserves Can Be Drained

Category	Severity	Location	Status
Validation	● Critical	.	Resolved

Description

Related with the added solvency check for C-01

```
    (uint256 maxBorrow,uint256 fee) = BCredit(address(this)).getBorrowForCollateral();
    require(maxBorrow + fee >= params.initialDebt, InsolventInitialCreditPosition());
```

getBorrowForCollateral() depends on blvPrice, and blvPrice depends on pool.totalReserves, and we already added `initialDebt` into pool.totalReserves

By inflating `initialDebt` , we inflate pool.totalReserves then inflate bookPrice / BLV, which inflate maxBorrow + fee So attacker could still drain pool reserves of any token

https://gist.github.com/GuardianAudits/8c790f2784f855ad91ff1ef85688885b

Recommendation

consider removing initialDebt for permissionless pool creation

Resolution

Baseline:
https://github.com/0xBaseline/mercury/blob/7d128420821b922870e5a844083b309cd96ec0c6/src/libraries/GuardLib.sol#L89.

H-01 | Pool Fees Are Lost When Claimed

Category	Severity	Location	Status
Logical Error	High	BFactory.sol	Resolved

Description

In `createPool` the `params.creator` and `params.feeRecipient` values are validated but never persisted to pool state
We never do `pool.creator = params.creator;` and `pool.feeRecipient = params.feeRecipient;` so after pool creation, `State.pool(params.bToken).creator` and `State.pool(params.bToken).feeRecipient` stays at its default value `address(0)` And because `pool.creator` is never set, the intended creator can't transfer creator later with `transferCreator()`
In `_claimPoolFees`

```
    pool.reserve.giveReserves(pool.creator, creatorFees, false);
    pool.reserve.giveReserves(meta.protocolFeeRecipient, protocolFees, false);
```

Because both values are 0, any fees collected will be irreversibly lost when claiming it

Recommendation

Inside `createPool`, store the creator and the fee recipient

```
    pool.creator = params.creator;
    pool.feeRecipient = params.feeRecipient;
```

.

Resolution

Baseline:
https://github.com/0xBaseline/mercury/blob/e81f3a58403bde43d7b37b9a77bdc3d851ae8f47/src/components/BFactory.sol#L201.

H-02 | Vault Withdrawals DOS Due To Solvency Check

Category	Severity	Location	Status
DoS	High	VaultLib.sol: 312-314	Resolved

Description

`VaultLib.depositToVault` records `depositedReserves` using `previewRedeem(shares))`, so the tracked assets are rounded down and are equal to the amount of assets owned by the contract.
`VaultLib.withdrawFromVault()` then burns shares via `vault.withdraw(_reserveAmount, ...)`, which internally uses previewWithdraw (rounds up). After this burn, `_ensureVaultSolvency()` checks `previewRedeem(remainingShares) >= depositedReserves`. Because `previewWithdraw()` can burn extra shares for the same asset amount, the remaining share balance can preview to less than the tracked reserves even when the vault is solvent. This will result in unexpected DoS cases while normally using the protocol.
Example:

- TotalAssets=2, totalShares=3.
- Deposit 2 assets → receive 3 shares, depositedReserves=2.
- Withdraw 1 asset → previewWithdraw(1)=2 shares burned.
- DepositedReserves is decreased to 1
- Remaining shares=1, previewRedeem(1)=0 while depositedReserves=1, so the check reverts despite no real loss.

Recommendation

Reconsider if this check is really needed. You can also switch to tracking shares and comparing them against the real share balance of the protocol.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/7352fe69811a35a143821ffedf7023961ea80c2a/src/libraries/VaultLib.sol#L136.

H-03 | Stale K Enables Reserve Extraction

Category	Severity	Location	Status
Logical Error	High	.	Resolved

Description

updateAccounting() computes and adds liquidityFee_ into pool.totalReserves
recordSwap() decides whether to recompute maker.lastInvariant, the curve K using a different threshold test
Here liquidity fee gating uses a ratio, wad division

```
    // _updateAccounting
    liquidityFee_ = _feesReceived - LIQUIDITY_GROWTH_FEE_SHARE.mulWad(_feesReceived);
    if (uint256(pool.totalBTokens).divWad(pool.totalSupply) >= 0.95e18) {
    liquidityFee_ = 0;
    }
```

If the pool has >= 95% of supply inside totalBTokens/totalSupply >= 95% , then no liquidityFee If < 95% , then liquidityFee is kept inside the pool and added into pool.totalReserves but here, Invariant recompute gating uses a floored product

```
    // _recordSwap
    if (pool.totalBTokens < pool.totalSupply.mulWad(0.95e18)) {
    newInvariant = CurveLib.computeInvariant(getCurveParams(_bToken));
    }
    maker.lastInvariant = newInvariant;
```

So invariant recompute happens if

```
    totalBTokens  <  floor( totalSupply * 0.95 )
```

The vulnerability here is that we are using two different definitions of inside / outside the 95% safety band
Liquidity fee uses

```
    floor(totalBTokens * 1e18 / totalSupply )  >=  0.95e18
```

Invariant refresh uses

```
    totalBTokens  <  floor(totalSupply * 0.95)
```

Those differ at the boundary whenever 0.95 * totalSupply is not an integer, creating a fee kept but K not updated inconsistent state
https://gist.github.com/GuardianAudits/83303e7af8afb67a41b0b84355200a69 That leaves the system in an inconsistent state
pool.totalReserves has been increased by a fee that is intended to grow liquidity but maker.lastInvariant K is still the pre fee value
Since CurveLib.computeSwap() treats lastInvariant as the source of truth
Pricing is now computed using stale K against a reserves value that already includes extra buffer after every swap ends exactly at totalBTokens = floor(0.95 * totalSupply), the pool reserves already include the liquidity fee, but K lastInvariant is still the old value, so the next swap that moves totalBTokens one step below that floor refreshes lastInvariant, causing a sudden curve jump
An attacker can arbitrage this in one transaction
1 - Buy a tiny amount while K is stale (cheaper)
2 - That buy crosses the floor, so _recordSwap updates lastInvariant
3 - The update now locks in the previously added liquidity fee into the curve
4 - Immediately sell back the same amount against the updated curve (higher price), pocketing reserves
In the poc, the attacker buys 1900 bTokens for 90 reserves using stale K,
Crossing the boundary refreshes K after the buy, attacker sells back for 1906 reserves, netting 1816 profit
https://gist.github.com/GuardianAudits/1bbd127b943b837305385d214ddb6de6

Recommendation

make the liquidity fee decision and the invariant refresh decision use the same predicate

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/77094087b38cb7ccd0a76dd572714a6fc31bade1/src/libraries/MakerLib.sol#L235.

M-01 | Configured feeRecipient Ignored In Payouts

Category	Severity	Location	Status
Logical Error	Medium	BController.sol	Resolved

Description

The configurable feeRecipient set via setFeeRecipient is never used when distributing creator fees. ClaimPoolFees/claimPoolFees send creator fees to pool.creator, making FeeRecipientSet ineffective. This misroutes funds compared to configuration intent and prevents creators from delegating revenue to a separate recipient without transferring creator ownership

Recommendation

.
We need to replace the payout target for creator fees with pool.feeRecipient if set, falling back to pool.creator only if feeRecipient is unset.

Resolution

Baseline:
https://github.com/0xBaseline/mercury/blob/e81f3a58403bde43d7b37b9a77bdc3d851ae8f47/src/components/BController.sol#L229.

M-02 | Staking Rewards Can Be Globally Griefed & Frozen

Category	Severity	Location	Status
DoS	Medium	BStaking.sol	Resolved

Description

An attacker can keep the global accumulator from ever increasing, so _getEarned() returns the old prevEarned for a very long time while pool.pendingYield continues to grow. The attacker can then stake right before distribution resumes and capture rewards accrued while they were not staked, or simply keep repeating the DoS.

getAccumulator() can determine that there is not enough yield to distribute.

```
    uint256 yield = min(tokensPerSecond_ * timeElapsed, pendingYield);
    uint256 accumulatorIncrease = yield.fullMulDiv(RAY, totalStaked);
    // @audit, not enough yield to distribute evenly, return early
    if (accumulatorIncrease == 0) return (accumulator_, 0, tokensPerSecond_);
```

So if yield * RAY < totalStaked, no accumulator increase happens, and `newYield_ = 0`

_sync() still updates `lastUpdated` even when nothing was distributed

```
    (accumulator, newYield, tokensPerSecond) = getAccumulator(_bToken);
    // update the last updated timestamp every time
    if (staking.lastUpdated != block.timestamp)
    staking.lastUpdated = block.timestamp.toUint32();
    // @audit, return early if user's accumulator up to date
    if (accumulator == staking.accounts[_user].userAccumulator) return;
```

An attacker can therefore keep timeElapsed tiny forever because it is always block.timestamp - lastUpdated. That keeps yield tiny, keeps accumulatorIncrease at 0, and prevents rewards from moving from pool.pendingYield into the stakers accumulator, so users' earned balances do not grow.

An attacker only needs a public function that calls `_sync()`

BStaking.deposit() is public and does not require `_amount > 0`

Calling deposit(_bToken, attacker, 0) is enough to run _sync(). The attacker can therefore send a cheap transaction repeatedly to keep resetting lastUpdated, and stakers cannot accrue earned rewards.

This can persist indefinitely as long as the attacker keeps calling.

Recommendation

We need to prevent `lastUpdated` from being advanced when the accumulator didn't advance

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/3ab36e01c5dadf20ce472440c447fb867a279e48/src/components/BStaking.sol#L211.

M-03 | Reverting Swaps Due To Underflow

Category	Severity	Location	Status
DoS	Medium	CurveLib.sol: 171,165	Resolved

Description

`CurveLib.computeFee()` computes the `IL` fee by subtracting `ΔB` from `|Δc| * marginalPremium`.

```
    // IL Fee = difference between marginal price and integral price
    fee_ = absDelta.mulWadUp(marginalPremium) - (newBufferDown - buffer);
```

In the ideal case, the LHS of the equation should always be greater than the RHS. However, for some swaps LHS is actually lower, which results in swaps reverting due to underflow.

The same is true for the else branch as well:

```
    fee_ = (buffer - _newBuffer) - (absDelta.mulWad(marginalPremium));
```

Recommendation

Consider using `zeroFloorSub()` when calculating the IL fee.

Buy:

```
    fee_ = (absDelta.mulWadUp(marginalPremium)).zeroFloorSub(newBufferDown - buffer);
```

Sell:

```
    fee_ = (buffer - _newBuffer).zeroFloorSub(absDelta.mulWad(marginalPremium));
```

In addition, you can add a maximum tolerable negative delta validation.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/4735c459372aea7027f20c2f53019558e3f90ab7/src/libraries/CurveLib.sol#L152.

M-04 | User Favorable Rounding In Sell Path

Category	Severity	Location	Status
Rounding	Medium	CurveLib.sol: 121	Resolved

Description

In `CurveLib.computeSwap` the sell branch is explicitly documented as “protocol-favorable” rounding. It uses `fullMulDivUp` for `newBuffer`, which rounds up to make reserves larger and the user payout smaller. However, the BLV component is added with `mulWad`, which rounds down:

```
    newReserves = newBuffer + BLV * c1
```

Using `mulWad` here slightly reduces `newReserves`, which makes the user receive more than the protocol-favorable rounding intent. This is inconsistent with the stated rounding policy for sells and can leak small amounts of value on each trade.

Over time, this can lead to swaps where the resulting `K` is smaller than the previous one even though `BLV` and `n` didn’t change. Such swaps will revert with the `InvariantDecreased()` error.

Recommendation

Use `mulWadUp` for the BLV term in the sell branch so that `newReserves` is rounded up consistently:

```
    - newReserves = newBuffer + BLV.mulWad(c1)
    + newReserves = newBuffer + BLV.mulWadUp(c1)
```

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/3ae71f63f48baa4fa50d0abcbabf55a7aaff801f/src/libraries/CurveLib.sol#L121.

M-05 | BLV Translation Uses Outdated State

Category	Severity	Location	Status
Logical Error	Medium	MakerLib	Resolved

Description

MakerLib.recordSwap stores a memory copy of State.Pool before mutating state and then uses that snapshot to decide whether to translate the BLV floor price
(checking if pool.totalBTokens < pool.totalSupply * threshold), Because this check happens after updateAccounting() but reads from the stale memory copy,
BLV translation decisions can be made using pre trade values, causing incorrect or delayed BLV updates and inconsistent curve evolution
Swaps that cross the 95% boundary can wrongly skip or wrongly trigger BLV translation (one trade late/early)

Recommendation

.
make the BLV translation threshold read post trade storage, not a pre trade memory snapshot

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/e21011f5a7214ea3182a1849523d9651b1f008db/src/libraries/MakerLib.sol#L175.

M-06 | Vault Exit Loss Not Propagated To Curve Math

Category	Severity	Location	Status
Math	Medium	VaultLib	Resolved

Description

`ExitVaultUnsafe` realizes a loss but doesn't propagate it into pool / curve accounting,
`BController.exitVaultUnsafe(ERC20 _reserve)` is exposed as a Relay route and calls
`VaultLib.exitVaultUnsafe(ERC20 _reserve)`

```
    function exitVaultUnsafe(ERC20 _reserve) external permissioned nonReentrant returns (
    uint256 amount_,
    uint256 loss_
    ...
    emit VaultExited(address(_reserve), amount_, 0, loss_);
    }
```

And in `VaultLib.exitVaultUnsafe`

```
    redeemed_ = vault.redeem(vault.balanceOf(address(this)), address(this), address(this));
    uint256 depositedReserves = allocation.depositedReserves;
    // requires a loss
    ...
    allocation.idleReserves = 0;
    allocation.vault = ERC4626(address(0));
```

ExitVaultUnsafe successfully exits an ERC4626 vault when there is a loss, `redeemed_ < depositedReserves`, and returns `loss_`, but it does not apply that loss anywhere else
In particular, it does not update
State.pool(bToken).totalReserves, the reserves number used by the AMM curve / pricing
State.maker(bToken).lastInvariant, the K source of truth used by `CurveLib.computeSwap` So after an unsafe exit
Actual reserves available to pay users are smaller because the vault lost funds But Internal accounting still thinks the old amount exists After `exitVaultUnsafe`, users can trade / withdraw based on stale, overstated `pool.totalReserves`. Early sellers can extract the remaining reserves at the pre loss price until the contract balance is exhausted, leaving late users unable to redeem

Recommendation

If markets will not kept open after exitVaultUnsafe, then it would be safe
Otherwise apply the realized vault loss to the internal pool accounting and curve invariant for every pool that uses `_reserve`

Resolution

Baseline:
https://github.com/0xBaseline/mercury/blob/717218cc7a92dd79b275ece952864a42f3193f87/src/components/BController.sol#L113.

M-07 | Zero Share Deposits Can Brick Protocol Sweeps

Category	Severity	Location	Status
DoS	Medium	SweepLib	Resolved

Description

Inside `VaultLib.depositIdleReserves()`

```
    function depositIdleReserves(ERC20 _reserve) internal {
    State.Allocation storage allocation = State.meta().allocations[_reserve];
    ERC4626 vault = allocation.vault;
    ...
    }
    }
```

And `depositIdleReserves()` which is reachable from user flows via `SweepLib.executeSweep()`

```
    if (VaultLib.isAllocated(allocation.vault)) {
    if (VaultLib.convertDepositToAssets(reserve, reserveBalance) > 0) {
    VaultLib.depositToVault(allocation, reserve, reserveBalance);
    ...
    }
    }
```

A normal user triggered action that causes a sweep can end up calling `depositIdleReserves()`, which is gated by `getHarvestableYield()`
`depositIdleReserves()` can attempt to deposit `idleReserves` even when `convertDepositToAssets(_reserve, idleReserves) == 0` That case means
vault.previewDeposit(idleReserves) is effectively 0 shares or redeem value 0
Yet if `getHarvestableYield(allocation) > idleReserves` because there is enough yield, then
PrecisionError = idleReserves - 0 = idleReserves
Condition `harvestableYield > precisionError` becomes `harvestableYield > idleReserves`
Deposit is attempted anyway, It could cause a global DoS, common with Solmate ERC4626
Many ERC4626 vaults revert on `deposit()` if `shares == 0` So if `idleReserves` is still too small to mint shares
`depositToVault()` calls `vault.deposit(idleReserves, address(this))`, Vault reverts, the entire sweep reverts and because sweeps are invoked in many important paths (swaps, staking syncs, fee claims) this can brick core interactions
This DoS is triggerable because users can create tiny `idleReserves` via very small swaps or any path that causes `reserveBalance` in the pool manager to be too small to deposit, and later force sweeps again.

Recommendation

We must not attempt to deposit idle reserves unless the vault will mint non zero shares

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/master/src/libraries/SweepLib.sol#L107.

M-08 | takeReserves() Can Result In 0 Shares Minted

Category	Severity	Location	Status
Unexpected Behavior	Medium	VaultLib.sol: 76-78	Resolved

Description

`SweepLib.executeSweep()` checks the assets that were just swept will result in non-zero assets if deposited to the vault. If not, the idle reserves are increased instead.

```
    VaultLib.convertDepositToAssets(reserve, reserveBalance) > 0
```

However, this logic is absent in `takeReserves()`. There `depositToVault()` is called with the exact amount unconditionally. If this amount is small enough to cause 0 shares minting, the result is either:

• Loss of assets, since the relay receives 0 shares

• DoS for the whole transaction if the vault implementation reverts on 0 shares minted, for example Solmate.

Two very common paths enabling this issue are `BSwap.buyTokensExactIn()` and `BSwap.sellTokensExactOut()` because they use an estimation amount and the difference between it and the actual user amount is stored in `dustFee`, which is later passed as an argument to `takeReserves()`.

Recommendation

Add the same check to `takeReserves()` - if the current amount will mint 0 value, add it to idle reserves and call `depositIdleReserves()`.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/94a9b5d2c3c6e18229f0b4d65d19ec9159743d96/src/libraries/VaultLib.sol#L116.

M-09 | BCredit.defaultSelf Off Curve

Category	Severity	Location	Status
Logical Error	Medium	BCredit	Resolved

Description

In `BCredit.defaultSelf()`

```
    pool.totalBTokens += collateral.toUint128();
    pool.totalReserves -= debt.toUint128();
```

This function directly mutates the AMM core state variables `totalBTokens`, `totalReserves` that drive pricing inside `MakerLib`, `CurveLib` But it does not update the curve source of truth invariant

State.maker(_bToken).lastInvariant That invariant is what `CurveLib.computeSwap()` uses to calculate swap deltas.

The AMM uses `maker.lastInvariant` as K

Swaps don't recompute K from the current reserves / supply, they use

maker.lastInvariant, stored K

Plus current `pool.totalReserves` / `pool.totalBTokens` / `circ` So `lastInvariant` must remain consistent with pool.totalReserves, pool.totalBTokens, circ, BLV, convexityExp`

defaultSelf() changes pool.totalBTokens, pool.totalReserves But leaves `maker.lastInvariant` unchanged. That means after a default:

pool.totalReserves / pool.totalBTokens reflect a new curve point But `maker.lastInvariant` still represents the old curve surface So the pool is now off curve, and the next swap will compute prices from a stale K

Recommendation

. After mutating `pool.totalBTokens` / `pool.totalReserves` in `defaultSelf()`, recompute and store the invariant

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/efff95228c9d02d54292a8a0c4b5a34bb022b79a/src/components/BCredit.sol#L263.

M-10 | Swap Fee Can Be Bypassed When Selling All Tokens

Category	Severity	Location	Status
Gaming	Medium	CurveLib.sol: 187	Acknowledged

Description

When all tokens are being sold to the pool, a flat `swapFee` is applied on top of the `BLV` price and the user receives reserves below `BLV` per bToken.

```
    if (c1 == 0) {
    uint256 blvValue = _params.BLV.mulWad(_params.circ);
    uint256 receipt = blvValue.mulWad(WAD - _params.swapFee);
    return (int256(receipt), _params.reserves - receipt);
    }
```

However, during normal sells `_computeFee()` applies the fee only to the `marginalPremium`, which depends on the buffer.

```
    fee_ += absDelta.mulWadUp(marginalPremium.mulWadUp(_p.swapFee));
```

In a situation where a user will sell all the bTokens to the pool, they can split the swap in two parts: large and small one. For the big swap, they will end up paying `swapFee` only on the new `marginalPremium` which is small because the new buffer is small as well. Then they will pay `swapFee` for `BLV` only for the small swap.

Recommendation

Consider applying the `swapFee` to `BLV + marginalPremium` in `computeFee()`

```
    - fee_ += absDelta.mulWadUp(marginalPremium.mulWadUp(_p.swapFee));
    + fee_ += absDelta.mulWadUp((_p.BLV + marginalPremium).mulWadUp(_p.swapFee));
```

Resolution

Baseline: Acknowledged.

M-11 | Entering Vault For A bToken Bricks Its Pool

Category	Severity	Location	Status
DoS	Medium	BController.sol: 100	Resolved

Description

When a bToken is also used as a reserve token, calling `setVault()` on that reserve deposits the entire bToken balance held by the protocol into the vault (enterVault uses `_reserve.balanceOf(address(this))`). This drains the liquid bToken inventory that backs the bToken's own pool, leaving the pool unable to transfer bTokens to users (e.g., swaps,
Staking withdrawals), effectively bricking the pool until the vault is exited.

Recommendation

When setting a vault for a reserve that is also a bToken with an active pool, only deposit the excess reserves above `totalBTokens + totalStaked + credit.accounts[address(this)].collateral` (or the pool's required liquid inventory), not the full balance.

Resolution

Baseline:
https://github.com/0xBaseline/mercury/blob/16a40bcf8d713d6a1c1ccc06dd8788c372a96933/src/components/BController.sol#L95.

M-12 | Convexity Parameter Is Being Overrelaxed

Category	Severity	Location	Status
Math	Medium	CurveLib.sol: 306	Resolved

Description

The protocol adapts the curve's convexity (`n`) using `CurveLib.computeMinimumConvexityExp` so a single curve can be consistent with both the current state and the historical max state `(maxCirc, maxReserves)`. The invariant is:

```
K = B * (x/c)^n
B = y - BLV*c
x = totalSupply - c
```

Taking logs:

```
ln K = ln B + n(ln x - ln c)
```

Solving for `n` using the current state and the max state (and eliminating `K`) gives:

```
n = (ln B_max - ln B_now) / [(ln x_now - ln c_now) - (ln x_max - ln c_max)]
```

Here, `x_now` is the remaining supply (`supply = totalSupply - circ`). The code uses `ln(totalSupply)`, which inflates the denominator and yields a smaller `n` than required. This flattens the curve, so the resulting curve no longer passes through the max point as intended. Practically, it underprices buys in later states and weakens the intended safety margin above BLV.

```
int256 lnSupply = (_params.totalSupply).toInt256().lnWad();
```

Recommendation

Use `supply` instead

```
function computeMinimumConvexityExp(
CurveParams memory _params,
uint256 _maxReserves,
...
);
}
```

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/abc71eba553151253c7eb914fe38c79cc0789d6d/src/libraries/CurveLib.sol#L299.

M-13 | Underreported Supply Drains Shared Reserves

Category	Severity	Location	Status
Gaming	Medium	BController.sol: 145	Resolved

Description

`setBTokenDeployment()` allows setting `State.pool(_bToken).totalSupply` to a value lower than the actual `BToken.totalSupply()`. A pool creator can then hold more `BToken` units than the pool accounting Recognizes, stake them via `BStaking.deposit()`, and inflate `getMaxBorrow()` / `borrow()` because credit capacity is computed from collateral and `blvPrice` without capping to accounted supply or Available reserves. The `borrow()` payout is sourced from shared reserve custody via `VaultLib.giveReserves()`, so the attacker can drain reserves funded by other pools.
The risk is further amplified if there is an upgradeability logic. For example, a token with a fixed `totalSupply` may later introduce a logic for minting and successfully drain the pool reserves.

Recommendation

Cap collateralization to accounted circulating supply, e.g., enforce `totalStaked + _amount <= State.pool(_bToken).totalSupply - State.pool(_bToken).totalBTokens` in `BStaking.deposit()`. Also prefer allowing tokens with fixed total supply.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/77094087b38cb7ccd0a76dd572714a6fc31bade1/src/components/BStaking.sol#L81.

L-01 | Swaps Bricks When convexityExp > 2e18

Category	Severity	Location	Status
DoS	Low	MakerLib()	Resolved

Description

_initialActivePrice can push the curve into the non quadratic convexityExp > 2 regime at initialization And that regime immediately hits a code bug path due to a units mismatch introduced during initialize() If _initialActivePrice is high enough, MakerLib.initialize() sets `maker.convexityExp > 2e18`

For `convexityExp > 2e18`, every swap goes through `_updateCurveConvexity()`

_updateCurveConvexity() compares native decimal pool totals against WAD normalized max values that were stored during initialization

```
    require(pool.totalReserves > maker.maxReserves);
```

With common reserves like USDC 6 decimals, this comparison will always fail, so the first buy and any trade that tries to push circulating > maker.maxCirc reverts, DoSing buys / growth for that pool

Recommendation

Consider normalizing current `circulating` and `totalReserves` to WAD before comparing and assigning

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/506fea7bff1c90f6bd860f068629f98006109c9e/src/libraries/MakerLib.sol#L258.

L-02 | Unaccounted Dust Desync Reserves Accounting

Category	Severity	Location	Status
Unexpected Behavior	Low	MakerLib,186	Resolved

Description

Both exactIn and exactOut paths perform an extra reserve transfer called "dust" after executing the swap. This transfer pulls additional reserves from the user (buyExactIn pulls the difference up to amountIn; sellExactOut pulls back any overage above target) and increases the vault's actual reserves. However, these extra reserves are not reflected in pool.totalReserves nor distributed via MakerLib's accounting. This creates a persistent mismatch between actual reserves and the curve's tracked reserves, underpricing the pool and leaking fees away from the configured fee distribution logic.

Recommendation

We could incorporated the dust into MakerLib accounting

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/2fcc5507549a0b0b98b4e7ff3e4964cc1f6e557f/src/components/BSwap.sol#L117.

L-03 | Pool Creation May Fail Due To Underflow

Category	Severity	Location	Status
DoS	Low	CurveLib.sol: 240-266,242	Resolved

Description

`CurveLib.computeInitialCurveParams()` decides whether it can use the `n = 2` branch based on `quadraticPrice >= _initialActivePrice`. However, quadraticPrice is computed with divWadUp (rounding up). This makes quadraticPrice a ceiling, not the exact value. The inequality required for the `n = 2` BLV formula to be safe is:
`QuadraticPrice supply circ <= 2 reserves totalSupply` That inequality holds in exact math, but a 1-wei upward rounding in quadraticPrice can flip it when multiplied by very large supply * circ. As a result, the code can enter the n=2 branch even though the exact inequality fails, and the subtraction
`2 reserves totalSupply - initialActivePrice circ supply`
Underflows and reverts.
This results in unexpected DoS for some pools.

Recommendation

You can:
1. Make the `quadraticPrice` calculation equivalent to the one in `computeActivePrice()`

```
    •         uint256 quadraticPrice = _initialBLV + FixedPointMathLib.divWadUp(
    •             buffer.mulWad(2e18).mulWad(_params.totalSupply),
    •             _params.supply.mulWad(_params.circ)
    ...
    +             _params.supply.mulWad(_params.circ)
    +         );
```

1. Round the `convexityExp` up so `n < 2` is avoided

```
    uint256 priceDelta = _initialActivePrice - _initialBLV;
    •           uint256 term = FixedPointMathLib.fullMulDiv(priceDelta, _params.supply,
_params.totalSupply);
    •           convexityExp_ = FixedPointMathLib.fullMulDiv(term, _params.circ, buffer);
    +           uint256 term = FixedPointMathLib.fullMulDivUp(priceDelta, _params.supply,
_params.totalSupply);
    +           convexityExp_ = FixedPointMathLib.fullMulDivUp(term, _params.circ, buffer);
```

As an additional safety measure, you can assert the computed `convexityBuffer` is not below 2.

```
    if (convexityExp < 2e18) revert()
```

If this behavior is not problematic, you can also revert with a clear reason instead of letting it underflow.

Resolution

Baseline:
https://github.com/0xBaseline/mercury/blob/7e2716658107ac392d3d241ddcc03f349314c3bb/src/libraries/CurveLib.sol#L231.

L-04 | Unused Code In Quote Functions

Category	Severity	Location	Status
Superfluous Code	Low	BSwap.sol: 212-222,312-318,282	Resolved

Description

`BSwap.quoteBuyExactIn` and `BSwap.quoteSellExactOut` both compute an initial `delta` estimate (based on `reservesIn / priceWithFee`), cap it, and then never use it. Each function immediately calls its respective solver (`_solveBuy` / `_solveSell`), which performs an independent bracketed binary search. As a result, the entire `delta` block in both functions is dead code: it does not affect outputs, gas usage in the solver, or correctness.

This unused computation makes the code harder to reason about and may mislead reviewers into thinking the solver is seeded with this value. It also adds unnecessary computation and leads to increased gas costs.

Recommendation

Remove the unused `delta` computation blocks in both `quoteBuyExactIn` and `quoteSellExactOut`.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/9d25aed78e413b5c675d1b408cba61a7811c17fa/src/components/BSwap.sol#L195.

L-05 | Smaller Swaps Reverting Due To Fee Rounding To 0

Category	Severity	Location	Status
DoS	Low		Resolved

Description

`MakerLib.swapTokens` computes swap fees in WAD via `CurveLib.computeSwap` and then denormalizes them to native reserve decimals with `NormalizeLib.denormalizeWad`. For low-decimal reserve tokens (e.g., 6 decimals), small swaps can produce a non-zero `feesWad` that still rounds down to `0` in native units. The current guard `require(feesReceived_ != 0 && deltaUserReserves_ != 0, InvalidOutput());` treats that rounding artifact as an error and reverts. This creates a denial-of-service on small trades: the curve returns a valid quote and the user pays rounded-against amounts, but the swap fails solely because the denormalized fee is zero. It also makes it harder to sell tokens as the pool supply increases. If the revert were simply relaxed without any additional logic, the rounding loss would silently divert fees away from the intended recipients. The fee is already embedded in the swap curve output, so users would still pay it, but the fee would remain inside pool reserves instead of being recorded as creator/protocol/staker fees. That is an accounting drift and changes fee distribution semantics.

Recommendation

Accumulate fee “dust” in WAD and only realize it once it reaches at least 1 native unit, while relaxing the revert to check that the WAD-level fee is non-zero. This prevents small-trade DoS caused by rounding.

Add a WAD remainder bucket to pool state:

```
--- a/src/libraries/StateLib.sol
+++ b/src/libraries/StateLib.sol
@@ struct Pool {
...
+        uint256 feeRemainderWad;
}
```

Accumulate fee dust and relax the revert to check `feesWad` instead of `feesReceived_`:

```
--- a/src/libraries/MakerLib.sol
+++ b/src/libraries/MakerLib.sol
@@ function swapTokens(...)
...
+
+        require(feesWad != 0 && deltaUserReserves_ != 0, InvalidOutput());
```

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/a1037957095eb560f887477cc1bde5339667e61d/src/libraries/MakerLib.sol#L155.

L-06 | Remove Final Correction From Buy/sell Paths

Category	Severity	Location	Status
Best Practices	Low	BSwap.sol: 486-490,394	Resolved

Description

At the end of the `_solveBuy()` and `_solveSell()` functions, there is a correction logic which modifies the delta in a way that makes the user lose more value.

`_solveBuy()`:

```
    // CONSERVATIVE: Reduce to ensure reserve-side swap is always worse than token-side.
    // Scale reduction based on decimal precision difference to handle 8-decimal reserves.
    uint256 reduction = FixedPointMathLib.max(1, delta_ / 10_000_000); // 0.00001% reduction
(10x tighter)
    if (delta_ > reduction) delta_ -= reduction;
```

`_solveSell()`:

```
    // CONSERVATIVE: Increase to ensure reserve-side swap is always worse than token-side.
    // Scale increase based on decimal precision difference to handle 8-decimal reserves.
    uint256 maxDelta = _pool.totalSupply - _pool.totalBTokens;
    uint256 increase = FixedPointMathLib.max(1, delta_ / 10_000_000); // 0.00001% increase
(10x tighter)
    if (delta_ + increase <= maxDelta) delta_ += increase;
```

Since both of the functions are wrappers around `CurveLib.computeSwap()`, no further adjustment to the delta is needed. The binary search approach is already not in favor of the user due to the dust they have to pay, adjusting the delta makes them incur unnecessary loses.

Recommendation

Remove the final correction logic from the two functions.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/d04e852b3db75c963883c9004b410704d5713765/src/components/BSwap.sol#L322.

L-07 | Asymmetrical Validation For Buy Path

Category	Severity	Location	Status
Validation	Low	BSwap.sol	Resolved

Description

`BSwap.quoteBuyExactIn()` limits certain buys, where the reserves received are more than 95% of the current reserves.

```
    if (reservesInWad > p.reserves * 95 / 100)
    revert AmountExceedsLiquidity();
```

However, this check is not present in `buyTokensExactOut()`, allowing users to bypass it by using the other function.

Recommendation

Consider whether the check is needed, since it limits the token in, not the token out. If it's - add it to the other function as well, otherwise remove it.

Resolution

Baseline: Resolved.

L-08 | Incorrect Event Emission For Hook Swaps

Category	Severity	Location	Status
Events	Low	MakerLib.sol: 206	Resolved

Description

`MakerLib._recordSwap()` emits the `Swap` event with `msg.sender` as the user.

```
    emit Swap(
    _bToken,
    msg.sender,
    ...
    liquidityFee
    );
```

For swaps performed through the `BHook` contract, `msg.sender` will always be the `Relay` itself. This leads to incorrect value emitted and makes it harder to track swaps offchain.

Recommendation

If emitting the correct user is important, you can introduce permissioned wrapper functions around `BSwap` that forward the sender.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/8aee8c26cc685862d19bc8ecebfc038be889bde9/src/libraries/MakerLib.sol#L207.

L-09 | solveBuy() Can Revert Due To Overflow

Category	Severity	Location	Status
Math	Low	BSwap.sol: 354,413	Resolved

Description

In `BSwap._solveBuy()`, the code multiplies `totalBTokens * 99`. If `totalBTokens * 99 > type(uint128).max`, the operation will revert (checked arithmetic in Solidity >=0.8), causing swaps/liquidity operations that reach this branch to fail even though the values may be otherwise valid for the surrounding logic. This creates a hard cap on `totalBTokens` and can brick functionality once the pool grows beyond `type(uint128).max / 99`.

Recommendation

Cast `totalBTokens` to uint256 before multiplying.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/d5c2211c2b73543b9ca1291a53d2c38cd0d2caab/src/components/BSwap.sol#L335.

L-10 | Buy Fee Path Underflow Can Revert Swaps

Category	Severity	Location	Status
DoS	Low	CurveLib.sol: 171	Resolved

Description

In `CurveLib._computeFee` (buy branch), `newBufferDown` is recomputed and then used in `fee_ = absDelta.mulWadUp(marginalPremium) - (newBufferDown - buffer)`.
The whole recomputation path is rounded down (including division and multiplication steps, as well as `powWad`), which can produce `newBufferDown < buffer` even while processing a buy. When that happens, `(newBufferDown - buffer)` underflows and the swap reverts, causing DoS for otherwise valid buy paths.

Recommendation

Prevent fee-path underflow by using a bounded buffer delta, e.g. `bufferIncrease = newBufferDown.zeroFloorSub(buffer)` before subtraction.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/f716d5c08fe72cade906cd017f938a3ce2c3650f/src/libraries/CurveLib.sol#L153.

L-11 | Buffer Can Grow During Sells

Category	Severity	Location	Status
Math	Low	CurveLib.sol: 184	Resolved

Description

The sell path can violate expected buffer monotonicity due to compounded upward rounding. In `CurveLib.computeInvariant()`, lastInvariant (K) is rounded up.. In `CurveLib.computeSwap()`, sell quotes derive newBuffer from `lastInvariant` using upward rounding as well. This is usually done to round in favor of the protocol and require more reserves from the user. However, the new buffer may become greater than the current one, especially for smaller sells. This will result in negative `invariantDelta` and the sell will actually behave like a buy for the reserves. In `CurveLib._computeFee()` (sell branch), the fee path performs an unsigned subtraction equivalent to (buffer - _newBuffer) before IL-premium adjustment. When `_newBuffer` exceeds `buffer`, this subtraction underflows and the swap reverts, leading to unexpected failures, and inconsistent sell behavior.

Recommendation

If this is acceptable, consider reverting with appropriate error if the new buffer increases during swaps. Otherwise, the buffer calculation should be reconsidered.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/f716d5c08fe72cade906cd017f938a3ce2c3650f/src/libraries/CurveLib.sol#L167.

L-12 | Hook Sweep Can Revert Batched Swaps

Category	Severity	Location	Status
DoS	Low	MakerLib.sol: 226	Resolved

Description

`MakerLib._updateAccounting()` calls `SweepLib.sweep()` at the start of every swap. When swaps are routed through the hook, the input token is recorded as an outstanding balance and settled later by the router. If an integrator batches multiple hook swaps in the same unlock and defers settlement until the end (a common v4 routing pattern), the second swap will try to sweep the first swap's unpaid outstanding balance and revert when poolManager.take cannot transfer funds that haven't been settled yet. This makes multi-swap batching through the hook incompatible unless each swap is settled before the next one, reducing composability and potentially breaking routers that rely on open deltas.

Recommendation

Either:

• Skip `SweepLib.sweep()` if the pool manager is unlocked and the hook still has an unpaid delta, and defer sweeping to a later action.

• Require integrators to settle after each hook swap when batching (and document the constraint).

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/d288a2edbb0a24a14e663617539d99b89c9f035c/src/libraries/SweepLib.sol#L63.

L-13 | Recorded Reserves Drift Due To Vault Operations

Category	Severity	Location	Status
Math	Low		Resolved

Description

When a swap occurs, `CurveLib` computes the amount of `Δreserves` and returns it as `deltaResWad`. If the direction is buying and a vault is set for that reserve, the received tokens will be immediately deposited to that vault which will lead to a precision loss. To mitigate that, `MakerLib` overcharges the user in such a way that after the deposit, the system will have its balance increased with at least `Δreserves`

```
    deltaUserReserves_ = -(
    VaultLib.convertAssetsToDeposit(
    State.pool(_bToken).reserve,
    NormalizeLib.denormalizeWadUp(uint256(-deltaResWad), rDec)
    ).toInt256());
```

Then `deltaUserReserves_` is passed to `recordSwap()`, where `totalReserves` will be increased with that value and the user will have to pay the exact same amount. Because of this, the rounding error that's covered by the user will be attributed to `totalReserves` even though this amount will be lost.

For example, let's say the invariant computes `Δy` assets needed. In order to get these assets we have to deposit `Δy + ε` to the vault. The code will record an increase in `totalReserves` of `Δy + ε` even though the system received only `Δy`.

Recommendation

You can add a new parameter to `_recordSwap()` called `rawAmount` and use it to pass the value before the vault deposit if the direction is a buy.

```
    _recordSwap(_bToken, _deltaCirc, deltaUserReserves_, ... lizeWadUp(uint256(-deltaResWad), rDec)) :
int256(0));
```

`0` is being used as a default value when the swap is sell, so `_recordSwap` will handle that case.

```
    if (rawAmount == 0) {
    rawAmount = _deltaUserReserves;
    }
```

Finally, `_updateAccounting()` will use the raw amount instead.

```
    uint256 liquidityFee = _updateAccounting(_bToken, _deltaCirc, rawAmount, _feesReceived);
```

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/d8185686cab6b4d1acc3c199078f21c89ef95ead/src/libraries/MakerLib.sol#L145.

L-14 | Hook Sweep Can Under-credit Vault

Category	Severity	Location	Status
Math	Low	VaultLib.sol: 301	Resolved

Description

When a reserve has a vault set, the funds from swaps are deposited to it during settlement. Swaps use `convertAssetsToDeposit()` to charge the user an amount that would result in `targetAssets` after the deposit. With the same vault ratio at quote and deposit (r1 = r2), the rounding buffer (`VAULT_ROUNDING_BUFFER = 2`) ensures `previewRedeem(previewDeposit(X)) >= targetAssets`, so the pool does not end up under-credited.

For swaps performed through `BHook` the user amount is computed at time t1 using `convertAssetsToDeposit()`, but the actual vault deposit happens later when `sweep` executes. This introduces a ratio gap (`r1 = tS1/tA1` at quote, `r2 = tS2/tA2` at deposit), so the original rounding guarantee no longer necessarily holds.

Let `a` be the target assets. The rounding pipeline is:

$\mathrm{previewWithdraw}(a)=\lceil a\cdot r_1\rceil = a\cdot r_1 + \varepsilon,\quad \varepsilon\in[0,1)$

$\mathrm{assetsNeeded}=\mathrm{previewRedeem}(\mathrm{previewWithdraw}(a)+2)=\lfloor a+\frac{\varepsilon+2}{r_1}\rfloor=a+\lfloor\frac{\varepsilon+2}{r_1}\rfloor$

$\mathrm{sharesMinted}=\lfloor \mathrm{assetsNeeded}\cdot r_2\rfloor=\mathrm{assetsNeeded}\cdot r_2-\delta,\quad \delta\in[0,1)$

$\mathrm{creditedAssets}=\lfloor \frac{\mathrm{sharesMinted}}{r_2}\rfloor=a+\lfloor\frac{\varepsilon+2}{r_1}\rfloor-\lceil\frac{\delta}{r_2}\rceil$

$\mathrm{creditedAssets}<a\iff \lfloor\frac{\varepsilon+2}{r_1}\rfloor<\lceil\frac{\delta}{r_2}\rceil$

Case `r2 >= 1` (share price <= 1 asset): `ceil(δ/r2)` is 0 or 1. Loss is only possible when `floor((ε+2)/r1) = 0` and `δ > 0`, which happens when `(ε+2)/r1 < 1` (i.e., `r1 > 2+ε`). Using a buffer of one full asset's worth of shares (`previewWithdraw(1) ~= ceil(r1)`) guarantees `floor((ε+buffer)/r1) >= 1`, so the loss condition cannot hold when r2 >= 1.

Case `r2 < 1` (share price > 1 asset): `ceil(δ/r2)` can exceed 1, so even with a larger buffer a loss is still technically possible if r2 drops enough between t1 and t2.

Recommendation

For `Case r2 >= 1`, you can increase the share buffer in `convertAssetsToDeposit` to one full asset's worth of shares, and keep it at least `VAULT_ROUNDING_BUFFER`: use `max(previewWithdraw(1), VAULT_ROUNDING_BUFFER)` instead of a fixed +2. This removes the loss condition and improves accuracy under rate drift.

`Case r1 < 1` cannot be fully solved because `r2` is not known at `t1`, but it can be improved by frequent sweeps.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/4dd60579772b6eda212e23b8dd2d8ff55ff71fa0/src/libraries/VaultLib.sol#L305.

L-15 | Harvest Buffer Not Retained On Vault Exit

Category	Severity	Location	Status
Math	Low	VaultLib.sol: 217	Resolved

Description

`HARVEST_BUFFER` is used in `getHarvestableYield()` and in `switchVaults()`'s pre-check to ensure there is enough yield to cover precision errors on vault entry/exit. However, `exitVault()` computes `yield_ = redeemed_ - depositedReserves` and immediately pays out the full yield to the fee recipient. This means the buffer is not retained during a vault exit. As a result, any rounding losses are no longer offset by the intended buffer and are effectively borne by the pool's reserves instead of being absorbed by retained yield.

Recommendation

When exiting a vault, retain the buffer by subtracting `HARVEST_BUFFER` from the payout: compute `yield_ = (redeemed_ - depositedReserves).zeroFloorSub(HARVEST_BUFFER)` before distributing fees. Then update the switchVaults check to require `yield_ != 0` so the original safety threshold remains intact.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/80487f76ba426b9a304602be1f90d64a943a4edd/src/libraries/VaultLib.sol#L214.

L-16 | Harvest Buffer Can Be Dynamically Adjusted

Category	Severity	Location	Status
Suggestion	Low	VaultLib.sol: 20	Resolved

Description

`HARVEST_BUFFER` in `VaultLib.sol` is meant to absorb potential rounding losses from vault interactions. Currently it's hardcoded to 100 wei, but may not be enough if vault shares are expensive.
A mixed approach of using both a hardcoded buffer and dynamically adjusted one can be made. You can measure losses when a vault is entered, withdrawals are performed or idle funds are deposited and accumulate the losses to a dynamic buffer. Then, when using buffer logic you can utilize `max(HARVEST_BUFFER, dynamicBuffer)`.
Currently, `depositIdleReserves()` tries to use a part of the available yield in order to cover the loss from the deposit. However, the only thing it achieves is to ensure there is more extractable yield than the realized loss. Once this yield is extracted, the proxy contract will still bear a loss. With the dynamic buffer approach, you can add the `precisionError` to it, which will ensure that precision loss is actually not later withdrawn as yield and enough funds remain in the contract in order to sufficiently back `totalReserves`.

Recommendation

Consider implementing the dynamic buffer approach, especially in `depositIdleFunds()`.

Resolution

Baseline:
https://github.com/0xBaseline/mercury/blob/3d0ac7f711a6ca7b5a1d3da0136fb8a8ebc962b2/src/libraries/VaultLib.sol#L153.

L-17 | Pool Reserves Overstated On createPool()

Category	Severity	Location	Status
Rounding	Low		Resolved

Description

`createPool()` sets `pool.totalReserves` to `params.initialPoolReserves + params.initialDebt` before collecting reserves. It then calls `takeReserves()` to pull `params.initialPoolReserves` from the caller. If a vault is already allocated for the reserve (likely when the same reserve is shared across pools), `takeReserves()` deposits into the vault via `depositToVault()`, which can round down and credit fewer assets than the amount transferred. This creates an initial shortfall where `pool.totalReserves` is overstated relative to the vault-redeemable value.

Recommendation

When collecting initial reserves in `createPool()`, charge the user with `convertAssetsToDeposit()` so the vault credits the intended `params.initialPoolReserves`. This keeps `pool.totalReserves` consistent with the actual vault-credited assets.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/88cf1c0019a1aaa0309f921fa726a5fba86b62d3/src/components/BFactory.sol#L201.

L-18 | Duplicate Claim Leaves Enable Griefing

Category	Severity	Location	Status
Trust Assumptions	Low	BCredit.sol: 211	Resolved

Description

`claimCredit()` is permissionless and `_processClaim()` only tracks `credit.claimed[user]` per user. If a user appears twice in the Merkle root with different amounts, any third party can call `claimCredit()` with the smaller leaf first, setting `credit.claimed[user] = true` and permanently blocking the larger claim via `BCredit_AlreadyClaimed()`. This turns a root-construction mistake into a griefing vector where the user is forced to accept the lower allocation.

Recommendation

Either document that users can claim only once and this must be enforced when generating the Merkle root, or change the tracking to be per-leaf instead of per-user so multiple leaves for the same user can be claimed without griefing.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/6b7c7ccd878732e8308469d04ad0cff87de10b42/src/components/BCredit.sol#L172.

L-19 | Leverage Fee Charged On Max Borrow

Category	Severity	Location	Status
Math	Low	BCredit.sol: 286	Resolved

Description

In `leverage()`, `getBorrowForCollateral()` computes `borrowAmount` and `fee` from `_totalCollateral` using `State.meta().originationFee`. After `buyTokensExactOut()` returns `reservesIn`, the code reduces `debt_` by the unused `borrowAmount`, but `fee` is not adjusted and is still distributed via `State.pool(_bToken).distributeFees()`. When `reservesIn` is lower than `borrowAmount`, the effective fee rate becomes `fee / reservesIn`, which is higher than the intended `originationFee` and can overcharge `leverage()` users while inflating `pendingYield` relative to the realized swap cost.

Recommendation

Recompute `fee` from the realized borrow (`reservesIn`) or from the final `debt_` after the refund, and use that recomputed `fee` in `distributeFees()`.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/0f1a32826dc29d0e84f6e772e36e7a64b8319772/src/components/BCredit.sol#L315.

L-20 | Ratio Rounding Favors Users On Buys And Sells

Category	Severity	Location	Status
Rounding	Low	CurveLib.sol: 95	Resolved

Description

In `CurveLib.computeSwap()`, both buy and sell branches compute `ratio = x1 / c1` using `divWadUp()`. Rounding the ratio up makes `powWad()` larger and reduces `newBuffer`. For buys, this
Lowers the required reserves, so users pay less and `effectivePrice` can fall below `activePrice()` as observed in `FoundryRound3::test_replay()`. For sells, the same rounding reduces `newReserves`,
Increasing user payout. This is opposite to the documented protocol-favorable rounding intent.

Recommendation

Use `divWad()` for the `ratio` in both branches to round against the user, and update the comments to match the rounding policy.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/7361f1f8ea9c1e2b4651f568088d34fb61184c70/src/libraries/CurveLib.sol#L91.

I-01 | Missing Guard Causes Sell Side Reverts

Category	Severity	Location	Status
Informational	Info	computeSwap()	Resolved

Description

The BUY branch protects powWad and expWad from overflowing by enforcing n times ln(ratio) less than or equal to approximately 135e18. The SELL branch computes the same powWad term for ratio equal to x1 divided by c1 and greater than 1, but omits the guard. For large x1 divided by c1 or high convexityExp, int256(ratio).powWad(n) overflows due to the expWad limit and reverts. This makes otherwise valid sell quotes or executions revert, causing a denial of service in high convexity or deep sell scenarios.

Recommendation

.
Mirror the BUY branch overflow guard in the SELL branch before calling powWad

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/f55c8ffaf890db97a526edd82e175f63410d4c6f/src/libraries/CurveLib.sol#L93.

I-02 | Excess Repayment Funds Not Refunded

Category	Severity	Location	Status
Informational	Info	BCredit.sol	Resolved

Description

Both repay() and repayWithNative() compute the actual debt to repay via previewRepay/repay (which caps to the user's remaining debt), but then unconditionally pull the full reservesIn/msg.value from the payer. Any excess over what is needed to repay debt is not refunded and is not credited to the user

Recommendation

We could only pull the exact amount required to cover debt, if the caller supplied a higher amount refund the difference by using `convertAssetsToDeposit`

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/954261a56243fae55f66605fe311ab839aba4202/src/components/BCredit.sol#L147.

I-03 | Repay Preview Revert When Debt Is 0

Category	Severity	Location	Status
Suggestion	Info	BCredit.sol	Resolved

Description

In previewRepay()

```
    debtToRepay_ = min(_debtAmount, _account.debt);
    collateralRedeemed_ = uint256(_account.collateral).mulDiv(debtToRepay_, _account.debt);
```

If _account.debt == 0, we divide by zero even though debtToRepay_ will be 0 So `previewRepay`, `previewRepayExactDebt`, and `repay` will revert

Recommendation

Consider handling zero debt early if (_account.debt == 0) return (0, 0);

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/df3b92f283e5e33359f1370ea58c7ab9149ab10c/src/components/BCredit.sol#L544.

I-04 | harvestYield Has A Return But Doesn’t Return It

Category	Severity	Location	Status
Informational	● Info	BController.sol	Resolved

Description

HarvestYield declares a return but doesn’t return it

```
    function harvestYield(ERC20 _reserve) external permissioned nonReentrant returns
(uint256) {
    uint256 yield_ = _reserve.harvestYield();
    emit Harvested(address(_reserve), yield_);
    }
```

Recommendation

Consider returning yield_

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/440850b0504dfaa1535635df1b26766ae3f5c788/src/components/BController.sol#L87.

I-05 | Missing Upper Bound Blocks Pool Creation

Category	Severity	Location	Status
Validation	● Info	BFactory.sol	Resolved

Description

In createBToken, totalSupply is only lower bounded, but the protocol later downcasts circulating supply to uint128
The user choose `_totalSupply` and the factory stores it as the pool supply
_validateBTokenParams() only enforces a minimum

```
    require(_totalSupply >= MIN_TOTAL_SUPPLY, TotalSupplyTooLow());
```

The rest of the system assumes that circulating supply can fit into uint128 But during `createPool()`, it does

```
    maker.maxCirc = NormalizeLib
    .normalizeWad(pool.totalSupply - pool.totalBTokens, bDec)
    .toUint128();
```

If `_totalSupply` is large enough such that
pool.totalSupply - pool.totalBTokens > type(uint128).max then `toUint128()` reverts, and the pool cannot ever be initialized for that bToken. So a user can successfully deploy a token via `createBToken()` but can not create a pool for it

Recommendation

Cap `_totalSupply` so it can not exceed `uint128` by adding an upper bound in _validateBTokenParams

Resolution

Baseline:
https://github.com/0xBaseline/mercury/blob/547384e00c259fa03c5a37fb290bbcf5fbdd7983/src/components/BFactory.sol#L303.

I-06 | isExecutor() Can Use Dirty Address

Category	Severity	Location	Status
Best Practices	Info	Component.sol: 64	Resolved

Description

`Component._isExecutor(address)` accepts an address and returns if this address is an executor. To do so, it hashes the address together with the mapping storage slot. However, it doesn't clean the upper 12 bytes of the address. This is fine for the current implementation, but if code changes and the address provided has dirty bytes, the returned value may be incorrect.

```
    function _isExecutor(address _user) internal view returns (bool isExecutor_) {
    assembly {
    mstore(0x00, _user)
    ...
    }
    }
```

Recommendation

Consider cleaning the upper 12 bytes of `_user`

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/46292a7c215c8bda575c43e2783fd4bf9703bcbf/src/Component.sol#L65.

I-07 | Relay Routes Are Unnecessary

Category	Severity	Location	Status
Superfluous Code	Info	Relay.sol: 66-84	Acknowledged

Description

A `_setRelayRoutes()` function was introduced to the `Relay` contract and is invoked in its constructor. It sets `address(this)` as a route for each function selector of the `Relay` contract.

```
    function _setRelayRoutes() internal {
    // state getters
    routes[this.admin.selector] = address(this);
    ...
    routes[this.executeActions.selector] = address(this);
    }
```

Doing this is unnecessary, because if the first 4 bytes of the calldata match any of the functions, they will be executed as a normal EVM call, and the `fallback` won't even be entered.

Recommendation

Remove the `_setRelayRoutes()` function.

Resolution

Baseline: Acknowledged.

I-08 | Incorrect Ceil Comments

Category	Severity	Location	Status
Best Practices	Info	BCredit.sol: 558-559	Resolved

Description

The comments in `BCredit.getBorrowForCollateral()` say the divisions there are using `ceil`, while it's actually `floor` that's happening.

```
    uint256 debtWad = blv.mulWad(collateralWad);  // ceil for obligation
    uint256 debt = NormalizeLib.denormalizeWad(debtWad, rDec);  // ceil
```

Recommendation

Remove the comments.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/c102fe997ee42f10134c27b3eddddfe33cff9ebf/src/components/BCredit.sol#L559.

I-09 | Redundant mulWad() In Active Price Computation

Category	Severity	Location	Status
Superfluous Code	Info	CurveLib.sol: 62	Resolved

Description

`CurveLib.computeActivePrice()` computes the premium to be added to `BLV`:

```
    uint256 premium = FixedPointMathLib.fullMulDiv(
    _params.reserves - _params.BLV.mulWad(_params.circ),
    _params.convexityExp.mulWad(_params.totalSupply),
    _params.supply.mulWad(_params.circ).mulWad(WAD)
    );
```

On the last line of the code snippet, we can see `mulWad(WAD)`. Because `mulWad` is equivalent to `x * y / WAD`, the end result will be `x * WAD / WAD = x`, which makes the operation unnecessary.

Recommendation

Delete the last `mulWad()`

```
    uint256 premium = FixedPointMathLib.fullMulDiv(
    _params.reserves - _params.BLV.mulWad(_params.circ),
    _params.convexityExp.mulWad(_params.totalSupply),
    •           _params.supply.mulWad(_params.circ).mulWad(WAD)
    +           _params.supply.mulWad(_params.circ)
    );
```

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/90c44036d1e570144a4566d4ab8c7d7f14ea76a5/src/libraries/CurveLib.sol#L109.

I-10 | Unreachable Edge Cases In computeSwap()

Category	Severity	Location	Status
Suggestion	Info	CurveLib.sol: 73-86	Resolved

Description

`CurveLib.computeSwap()` has code that handles specific edge cases - full buys and sells.

```
    // Edge case: buying from zero circulation
    if (_params.circ == 0) {
    return _computeZeroCircSwap(_params, uint256(_deltaCirc));
    ...
    return (int256(receipt), _params.reserves - receipt);
    }
```

However, the invariant curve `K = B × (x/c)ⁿ` is not defined for `c = 0`. Because of that `CurveLib.computeInvariant()` won't allow the pool to ever reach a state where `c = 0`, therefore these codepaths should never be executed.

Recommendation

Reconsider whether the edge cases code is needed.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/e21011f5a7214ea3182a1849523d9651b1f008db/src/libraries/MakerLib.sol#L191.

I-11 | Binary Search Loop Can Break Earlier

Category	Severity	Location	Status
Gas Optimization	Info	BSwap.sol,385,437,464	Resolved

Description

The binary search loops in `_solveBuy()` and `_solveSell()` compare midpoints against desired targets, but they don't `break` in case `midPoint == target`.
This may cause unnecessary iterations and increased gas cost.

Recommendation

Consider breaking if the target is found.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/9106bf33c9ba74fb98e20147f47b0235ece0bf4b/src/components/BSwap.sol#L361.

I-12 | Dead Pool.shareBalance State Field

Category	Severity	Location	Status
Superfluous Code	Info		Resolved

Description

`State.Pool` declares `uint128 shareBalance` at , but this field is never read from or written to anywhere in the codebase. A search for `pool.shareBalance` and assignments to `shareBalance` in `src/` returns no usages.
At the same time, vault share accounting is implemented through `State.meta().allocations[_reserve]` and `BLens.shareBalance()` reads `alloc.vault.balanceOf(address(this))` rather than pool-level storage.
This indicates `Pool.shareBalance` is dead state.

Recommendation

Consider removing the field

Resolution

Baseline: Resolved.

I-13 | K Decreases Due To Computation Inconsistencies

Category	Severity	Location	Status
Rounding	Info	MakerLib.sol: 195	Acknowledged

Description

When `convexityExp` and `BLV` are unchanged, `MakerLib._recordSwap` recomputes the invariant and enforces strict non-decrease.
`ComputeInvariant` depends on `powWad` (approximation via `ln/exp`) plus fixed-point rounding (`divWad`, `divWadUp`, `mulWadUp`, `divWadUp`).
This can introduce small negative numerical drift between the stored invariant and recomputed invariant, causing swap reverts even when the observed delta is tiny relative to invariant magnitude.

Recommendation

Keep the check, but document this as an intentional precision-related limitation.

Resolution

Baseline: Acknowledged.

I-14 | Vaults Are A Security Risk

Category	Severity	Location	Status
Warning	Info	Global	Acknowledged

Description

Users can freely deploy their BTokens and provide `reserves` for backing to the Baseline Relay and start using its features to launch their token.
Currently the executor of the `Relay` contract can choose a vault where the reserves will be deposited in order to generate additional yield. This feature is not riskfree. A compromised executor can provide any vault of their choice, including a fake address which just pulls the funds out of the contract.
Even if the executor acts honestly, if the vault is not properly monitored, the relay contract can lose its reserves, either due to security risks in the vault itself or economic losses.

Recommendation

Be aware of the security risks that come with having the vault design and consider if changes are necessary.

Resolution

Baseline: Acknowledged.

I-15 | Unused previewTakeReserves() Helper

Category	Severity	Location	Status
Superfluous Code	Info	VaultLib.sol: 263	Resolved

Description

The helper `previewTakeReserves` is defined in `VaultLib` but has no call sites in the codebase. It duplicates `convertDepositToAssets` behavior and adds unused surface area, which can confuse future refactors and audits.

Recommendation

Remove `previewTakeReserves` or add a concrete call site that relies on it. If it's intended for future use, add a short comment explaining where/why it should be used.

Resolution

Baseline: Resolved.

I-16 | Outdated Refund Ordering Comment

Category	Severity	Location	Status
Documentation	Info	BCredit.sol: 372	Resolved

Description

The comment in `deleverage()` states that issuing the refund before `unlockCollateral()` prevents it from being treated as yield during `_sync()`. Current yield accounting in `BStaking` is driven by `pendingYield`, `claimableYield`, and `staking.totalStaked`, and is not affected by the order of `giveReserves()` versus `unlockCollateral()`. As a result, the comment does not reflect the present behavior.

```
    // refund should be given before unlocking collateral so it's not treated as yield
during _sync
    if (refund_ > 0) State.pool(_bToken).reserve.giveReserves(msg.sender, refund_, true);
```

Recommendation

Remove the comment.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/d8d6c25ef1f605b0b274c4c02277c28c1cdfc1b7/src/components/BCredit.sol#L393.

I-17 | Invariant Error Lacks Drop Context

Category	Severity	Location	Status
Best Practices	Info		Resolved

Description

`MakerLib._recordSwap()` reverts with `InvariantDecreased()` when `CurveLib.computeInvariant()` returns a smaller value, but the error carries no
values. This makes it difficult to distinguish between precision drift and meaningful invariant drops during fuzzing, monitoring, and debugging.

Recommendation

Add parameters to `InvariantDecreased()` (e.g., `prevInvariant` and `newInvariant`) and revert with `revert InvariantDecreased(prevInvariant,
newInvariant)` so tooling can assess the magnitude and context of the drop.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/1eafb8857d83238187112cd137efd98833057b84/src/libraries/MakerLib.sol#L69.

I-18 | Check Computed Blv Against Book Price

Category	Severity	Location	Status
Best Practices	Info	CurveLib.sol: 268	Resolved

Description

In `computeInitialCurveParams()`, the final validation checks `_initialBLV` against `bookPrice` via `require()`, even though the function may compute a different `BLV_` in the quadratic branch. `BLV_` is still Bounded by `bookPrice` when `_initialActivePrice > bookPrice`, so this is not a functional bug, but the check is semantically tied to the computed output and would be clearer and more future-proof if applied to BLV_ directly.

Recommendation

Replace the final check with `require(BLV_ <= bookPrice, InvalidBLVPrice())`

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/d41af439767db76e0263400a11ffdf503edc87dc/src/libraries/CurveLib.sol#L261.

I-19 | Lack Of BToken Parameters Validation

Category	Severity	Location	Status
Validation	Info	BController.sol: 145-151	Acknowledged

Description

`BController.setBTokenDeployment()` doesn't run the logic in `_validateBTokenParams()`, which means a token with unexpected configuration may be added to the system.

Recommendation

Consider validating the token parameters.

Resolution

Baseline: Acknowledged.

Round Two Findings & Resolutions

ID	Title	Category	Severity	Status
H-01	K Drops Due To Overcharged Fee	Rounding	High	Resolved
H-02	Sell Side Buffer Rounding Breaks Swap	DoS	High	Resolved
H-03	Inverse Pow Rounding Inflates Buy Side Fees	Rounding	High	Resolved
H-04	Convexity Reset Removes Buy Side Price Impact	Math	High	Partially resolved
H-05	Round Trip Swaps Enable Reserves Drain	Math	High	Resolved
M-01	computeSwap Can DoS Swaps	DoS	Medium	Resolved
M-02	computeNextBLV Can DoS Swaps	DoS	Medium	Resolved
M-03	Yield Distribution Can Be Violated	Rewards	Medium	Partially resolved
M-04	Reopen Buy Bypasses Stored Invariant	Logical Error	Medium	Resolved
M-05	Non Additive IL Fee Exploitable	Logical Error	Medium	Partially resolved
M-06	TPS Decay Computed But Never Applied	Logical Error	Medium	Resolved
L-01	Pool Initialization May Fail	Validation	Low	Resolved
L-02	Impossible To Default If Resulting Circ == 0	DoS	Low	Resolved

Round Two Findings & Resolutions

ID	Title	Category	Severity	Status
L-03	Hook Swap Bypasses Reserve Dust Accounting	Logical Error	Low	Resolved
L-04	Hidden Reserves In Hook Swaps	Logical Error	Low	Resolved
I-01	Initial Credits May Be Unclaimable	Warning	Info	Acknowledged
I-02	Pool Creation Cannot Be Paid With Native Tokens	Best Practices	Info	Resolved
I-03	Outdated Comment	Best Practices	Info	Resolved
I-04	Lack Of Validation For creatorFeePct	Validation	Info	Resolved
I-05	Redundant Function Parameter	Superfluous Code	Info	Resolved
I-06	previewRepay() Should Return debtToRepay	Suggestion	Info	Resolved
I-07	Redundant mulWad() In Active Price Computation	Superfluous Code	Info	Resolved
I-08	Unused Errors	Superfluous Code	Info	Resolved
I-09	Comment Mismatch On Premium Threshold	Documentation	Info	Resolved
I-10	Confusing Plural Used For Variable Naming	Best Practices	Info	Resolved
I-11	K Can Overflow	DoS	Info	Resolved

H-01 | K Drops Due To Overcharged Fee

Category	Severity	Location	Status
Rounding	High	MakerLib.sol: 152	Resolved

Description

In response to `L-05` the `feesReceived` variable is rounded up.

```
    feesReceived_ = NormalizeLib.denormalizeWadUp(feesWad, rDec);
```

This resolves the problem of swaps reverting when the swap fee cannot be represented as a whole unit of the reserve token. However, the same feesReceived_ variable is then passed to _recordSwap() and accounted as a fee at the expense of reserves. Example: the reserve token has 8 decimals.

• InvariantDelta = -4000000000000000221497

• FeesWad = 400000000000000218505

• feesReceived_ = 400000000001

• DeltaResWad = `invariantDelta - feesWad` = -440000000000000000440002

The user pays 4400000000001 reserves, of which 400000000001 are fees. The remaining 4000000000000 go to backing reserves. Normalized to 18 decimals, 4000000000000000000 < 4000000000000000221497 (invariantDelta). Once the swap completes, actual totalReserves are lower than expected, so the computed invariant is lower than it should be. This is worse when more than 95% of total supply sits in the pool because there is no liquidity fee to offset the invariant drop. As a result, the curve uses an outdated lastInvariant that is larger than the real invariant and can become blocked.

Recommendation

Consider implementing the original recommendation of `L-05`

Resolution

Baseline: Resolved.

H-02 | Sell Side Buffer Rounding Breaks Swap

Category	Severity	Location	Status
DoS	High	CurveLib.sol	Resolved

Description

Sell side IL fee math can revert for a huge range of sell sizes
computeSwap() calls _computeFee()
_computeFee() sell branch _deltaCirc < 0 contains two requires

```
    require(buffer > _newBuffer, InvalidFeeForSwap());
    require(marginalBufferDelta < bufferDecrease, InvalidFeeForSwap());
```

IL fee bufferDecrease - marginalBufferDelta is combined with swap fee on premium portion on sells
For certain reachable curve states, especially when
BLV is extremely close to book price reserves / circ
The remaining buffer is tiny: buffer = reserves - BLV*circ ≈ 0, and convexity is moderate or high (convexityExp > 2e18, though it can also happen at 2e18).
The sell fee math becomes fragile due to inconsistent rounding between
newBuffer being computed rounded up in computeSwap

```
    uint256 newBuffer = fullMulDivUp(lastInvariant, 1e18, pow(ratio,n));
```

marginalPremium and marginalBufferDelta being computed rounded down or mixed in the SELL branch

```
    uint256 marginalPremium = fullMulDiv(newBuffer, );
    uint256 marginalBufferDelta = absDelta.mulWad(marginalPremium);
```

The branch then enforces strict inequalities:
buffer > newBuffer
marginalBufferDelta < bufferDecrease. When the buffer is tiny, rounding newBuffer up can easily make
newBuffer = buffer, so bufferDecrease = 0 and the call reverts, or bufferDecrease becomes so small that marginalBufferDelta >= bufferDecrease. This creates a minimum sell size, sometimes in the hundreds of thousands of tokens, and it also affects deleverage because borrowers cannot reduce debt by selling collateral. The pool can therefore become effectively non-tradable in one direction for most sizes.
Below is a curve state where:
Selling 100 tokens reverts with InvalidFeeForSwap
Selling 175301 tokens succeeds, meaning sells smaller than about 175k tokens are impossible.

```
    // All parameters in WAD units
    totalSupply = 1_000_000e18
    pool.totalBTokens supply = 18_698e18 circ = totalSupply - supply = 981_302e18
    ...
    convexityExp = 10e18
    swapFee = 0.01e18
```

In that state
_deltaCirc = -100e18 revert InvalidFeeForSwap
_deltaCirc = -175_301e18 OK
The revert happens because the sell branch tries to compute
IL fee = bufferDecrease - marginalBufferDelta then adds swap fee on premium but the IL fee step is guarded by

```
    require(marginalBufferDelta < bufferDecrease, InvalidFeeForSwap());
```

Rounding then makes that condition fail for a huge range of _deltaCirc.
The buy branch explicitly compensates for rounding by recomputing a different buffer newBufferDown for fee calculation

```
    uint256 newBufferDown = fullMulDiv(lastInvariant, pow(), 1e18);
```

But the sell branch does not perform an analogous rounding-safe recomputation. Instead, it uses newBuffer computed via fullMulDivUp and then enforces the strict require:
https://gist.github.com/GuardianAudits/30d68bde1f917caf36e7f45036e55476

Recommendation

make the sell side fee math use a rounded down buffer, instead of using computeSwap() newBuffer which is fullMulDivUp ceil, this will help, but might be not enough, consider redesigning this area

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/c9b80084a8b56929bd02b102d537fc4c67efa0f1/src/libraries/CurveLib.sol#L166.

H-03 | Inverse Pow Rounding Inflates Buy Side Fees

Category	Severity	Location	Status
Rounding	High	CurveLib.sol	Resolved

Description

Buy-side inverse-pow rounding can turn most of the trade into creator-claimable fees.
In computeSwap(), the main swap math uses

```
    uint256 ratio = x1.divWad(c1);
    uint256 newBuffer = fullMulDivUp(lastInvariant, WAD, powWad(ratio, n));
```

Then the buy branch of _computeFee() recomputes a rounded down buffer using the reciprocal ratio

```
    uint256 newBufferDown = FixedPointMathLib.fullMulDiv(
    _p.lastInvariant,
    int256(c1.divWad(x1)).powWad(_p.convexityExp.toInt256()).toUint256(),
    WAD
    );
```

This is dangerous because for buys where the pool still holds more than half the supply after the trade (x1 > c1), we have:
`x1 / c1 > 1 in computeSwap(), so the main path is fine, but c1 / x1 < 1 inside _computeFee(), where powWad(c1/x1, n) can round to 0.`
Example: if post-trade x1 / c1 = 3 and convexityExp = 50e18:
Then

```
    main path uses 3^50, which is still finite
    fee path uses (1/3)^50 ≈ 1.4e-24
    In WAD math, (1/3)^50 rounds to 0
```

That produces:

```
    newBufferDown = 0;
    bufferIncrease = 0;
```

So buy IL fee becomes

```
    fee ≈ marginalBufferDelta + BLV swap fee
```

Instead of

```
    fee = marginalBufferDelta - trueBufferIncrease + BLV swap fee
```

So the code reclassifies almost the entire premium spend as fee
This lets high convexity pools massively overcharge buys without reflecting that in the visible swapFeePct
Fee handling afterward:

```
    liquidityFee_ = _feesReceived - 50% of _feesReceived;
    if (pool.totalBTokens >= 95% of totalSupply) {
    liquidityFee_ = 0;
    }
    pool.distributeFees(_bToken, _feesReceived - liquidityFee_);
```

When the pool still holds at least 95% of supply, which is common early on, 100% of the inflated fee is distributed rather than retained. A pool creator can then:
1. Create a high convexityExp pool,
2. Keep inventory high (totalBTokens >= 95% totalSupply),
3. Let users buy,
4. Have _computeFee() silently explode the fee,
5. ClaimPoolFees() and siphon those reserves
https://gist.github.com/GuardianAudits/a39a8e6b1afac439eec25ade8c4039ed

Recommendation

we need to not recompute newBufferDown via the reciprocal c1/x1, reuse _newBuffer and round it down
Inside _computeFee() buy branch, replace

```
    uint256 newBufferDown = FixedPointMathLib.fullMulDiv(
    _p.lastInvariant,
    int256(c1.divWad(x1)).powWad(_p.convexityExp.toInt256()).toUint256(),
    WAD
    );
```

with this, same math as computeSwap(), just rounded down

```
    uint256 newBufferDown = FixedPointMathLib.fullMulDiv(
     p.lastInvariant,
```

H-04 | Convexity Reset Removes Buy Side Price Impact

Category	Severity	Location	Status
Math	High	CurveLib.sol	Partially resolved

Description

User triggered convexity relaxation erases buy side price impact, so split buys are massively underpriced
On a buy that pushes the pool to a new all time high book price, _translateConvexity() Path 1 computes a new exponent from the pre trade price

```
    uint256 prevPrice = CurveLib.computeActivePrice(_prev);
    uint256 floor = FixedPointMathLib.fullMulDiv(
    prevPrice - _params.BLV,
    _params.supply.mulWad(_params.circ),
    actualBuffer.mulWad(_params.totalSupply)
    );
```

But computeActivePrice() is

```
    P = BLV + buffer  n  totalSupply / (supply * circ)
```

So after the trade, if we set

```
    n_new = (prevPrice - BLV)  supply  circ / (buffer * totalSupply)
```

Then the new post trade marginal price becomes

```
    P_after = BLV + buffer  n_new  totalSupply / (supply * circ) = prevPrice
```

So the code is literally snapping the post trade price back to the pre trade price
_recordSwap() prices the current buy using the old, steeper curve first, then lowers convexityExp after accounting. That means
1 - Trade 1 is charged under old n
2 - State is then rewritten so the next trade starts from the old price again
3 - Repeating small buys lets the attacker walk through a convex curve at an almost flat marginal price
4 - maxCirc/maxReserves are updated first, so the attacker can staircase this repeatedly until n collapses to 2e18
That's order splitting underpricing issue, example state

```
    BLV = 1
    convexityExp = 6e18
    totalSupply = 1000
    pool inventory / circulating = 500 / 500
    reserves = 1500
    swap fee = 0.3%
```

A single buy of 70 tokens costs about 9367 reserve units. But if the attacker splits it into 7 buys of 10
Total cost is only about 2021
ConvexityExp ratchets from 6 down to about 2.04
The post trade active price stays ~25 for the first several chunks instead of ratcheting up That is not normal convex curve behavior. It means fragmentation itself manufactures a huge discount So attacker can exploit this by
1 - Find a pool with convexityExp > 2e18 and premium gate satisfied
2 - Split a target buy into many small buys
3 - Each buy triggers Path 1 and cheapens the next state
https://gist.github.com/GuardianAudits/afbf268d14324d977507450fb4f7360a

Recommendation

https://gist.github.com/GuardianAudits/cd76e271a4d4083beb5dfd2c4b609dc2

Resolution

Partially resolved

H-05 | Round Trip Swaps Enable Reserves Drain

Category	Severity	Location	Status
Math	High	MakerLib.sol	Resolved

Description

ConvexityExp can be permanently ratcheted downward with tiny sell / buy back round trips,
Even if the attacker returns the pool to essentially the same inventory split
recordSwap() calls translateCurve(_bToken, prev) after every swap when convexityExp > 2e18, _translateCurve() routes to _translateConvexity()
translateConvexity() has two paths (currentBookPrice >= maxBookPrice and < maxBookPrice) and both can only decrease convexityExp
Nothing ever increases convexityExp, so it becomes a path dependent, global one way state
Allowing this attack
1. Repeatedly do a tiny round trip
sellTokensExactIn()
buyTokensExactOut() (buy back)
2. The sell can push price below maxBookPrice (relaxing convexityExp via the below path), and the buy-back can cross back above (relaxing it again via the above path using prevPrice)
3. After each loop, totalBTokens / circ are ~unchanged (fees aside), but convexityExp is permanently lower
4. Repeat until convexityExp collapses toward 2e18 at trivial cost
5. Then execute a single large sellTokens() or deleverage(), with a flatter curve, the protocol pays out materially more reserves than it would on an untouched curve
This differs from splitting a large sell, because here the attacker can pre soften the curve cheaply with micro round trips, end with nearly the same position, and monetize later in one big swap
In the PoC attacker spends ~2 reserves to unlock ~38 extra reserves on the later dump for example, which can be repeated to drain the buffer
https://gist.github.com/GuardianAudits/ed0e25a10793a0f761f7709c72464002

Recommendation

https://gist.github.com/GuardianAudits/cd76e271a4d4083beb5dfd2c4b609dc2

Resolution

Baseline: Resolved.

M-01 | computeSwap Can DoS Swaps

Category	Severity	Location	Status
DoS	Medium	CurveLib.sol	Resolved

Description

In computeSwap()

```
    uint256 ratio = x1.divWadUp(c1);
    require(
    ratio <= WAD || _params.convexityExp.mulWad(uint256(int256(ratio).lnWad())) <= 135e18,
    ...
    int256(ratio).powWad(_params.convexityExp.toInt256()).toUint256()
    );
```

When ratio < 1e18 (x1 < c1), the require does not compute / guard anything, and execution proceeds to the powWad() denominator
powWad() in solady is implemented as

```
    return expWad(( lnWad(x) * y) / WAD);
```

And expWad() explicitly returns 0 for sufficiently negative inputs

```
    if (x <= -41446531673892822313) return r; // r = 0
```

So when ratio < 1, lnWad(ratio) is negative, and convexityExp is large
The exponent (lnWad(ratio) * convexityExp) / WAD can become ≤ -41.4465e18 then expWad() returns 0, therefore powWad(ratio, convexityExp) returns 0 and fullMulDivUp(denominator = 0) reverts
That's a DoS reachable from pool state supply/circ and a high convexityExp
Let n = convexityExp / 1e18
We get the failure when

```
    ln(ratio) * n <= -41.4465
    =
    ratio <= exp(-41.4465 / n)
```

Examples

```
    If n = 50, threshold is ratio <= exp(-0.8289) = 0.436
```

Meaning once post trade x1/c1 drops below ~0.436,
PowWad rounds to zero and the swap math reverts

```
    If n = 20, threshold is ratio <= exp(-2.0723) = 0.126
    If n = 10, threshold is ratio <= exp(-4.1446) = 0.0159
```

So in high convexity pools, this can happen at non extreme circulation ratios once the pool reaches a state where for a buy the resulting x1/c1 is below that threshold, then all paths that call computeSwap for buys, buyTokensExactIn / buyTokensExactOut / leverage, will revert, until the state is moved back into a safe area via enough sells increasing x/c

Recommendation

This issue happens because we divide by powWad(ratio,n) when ratio < 1e18
we can do the same math, but remove the need of powWad()
Instead of `newBuffer = K / ( ratio^n )`
we can compute the same thing as
`newBuffer = K * ( invRatio^n )`
Mathematically these are identical

Resolution

Baseline: Resolved.

M-02 | computeNextBLV Can DoS Swaps

Category	Severity	Location	Status
DoS	Medium	CurveLib.sol	Resolved

Description

computeNextBLV() computes

```
    prevCirc = _params.totalSupply - _prevSupply
```

Then uses prevCirc squared in the denominator

```
    uint256 prevCirc = _params.totalSupply - _prevSupply;
    uint256 penalty = fullMulDivUp(, _params.supply.mulW ... pply).mulWad(prevCirc).mulWad(prevCirc) //
prevCirc^2
    );
```

PrevCirc can be 0, whenever the previous pool state had
PrevSupply = totalSupply (pool holds all bTokens), so prevCirc = totalSupply - prevSupply = 0 that can be triggerable DoS for swaps that cross the 95% threshold, because _recordSwap() calls computeNextBLV() whenever
Pool is in the update BLV path, maker.convexityExp = 2e18 and pool ownership is below the safety threshold

```
    if (pool.totalBTokens < pool.totalSupply.mulWad(BUFFER_SAFETY_THRESHOLD)) {
    // if convexityExp == 2e18:
    maker.blvPrice = CurveLib.computeNextBLV(getCurveParams(_bToken), prev.supply, prev.reserves);
    }
```

So If someone sells enough bTokens so the pool ends up with pool.totalBTokens = pool.totalSupply, that makes prevCirc = 0
Next, someone tries to do a single buy large enough that after the buy
pool.totalBTokens < 0.95 * totalSupply (circulating becomes > 5%) That swap triggers the BLV update path and calls computeNextBLV() with prevCirc = 0 And the swap revert
A small buy < 5% can move off circ = 0 and avoid the revert on the next swap But any swap that tries to move from circ = 0 to pool has < 95% of supply in one trade will revert
This affects also leverage() / deleverage() flows because they go through _recordSwap() too

Recommendation

At the start of computeNextBLV()

```
    uint256 prevCirc = _params.totalSupply - _prevSupply;
    // If previous circulation was zero, there is no meaningful penalty term
    // So we can keep BLV unchanged
    if (prevCirc == 0) return _params.BLV;
```

This makes the circ = 0 to big buys transitions safe

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/efa622844950de3fa8a74c67240e68d3afe6e956/src/libraries/CurveLib.sol#L278.

M-03 | Yield Distribution Can Be Violated

Category	Severity	Location	Status
Rewards	Medium	BStaking.sol	Partially resolved

Description

The contract is trying to drip rewards out over time
It uses lastUpdated as the clock to measure how much time passed, and tokensPerSecond as the speed of dripping rewards
_sync() only persists tokensPerSecond and lastUpdated when accumulator changes, but getAccumulator() can and does change tokensPerSecond even when no yield is distributed newYield = 0 That creates a state where
staking.tokensPerSecond gets stuck at an old non zero value, doesn't decay toward 0 when pendingYield = 0
staking.lastUpdated also gets stuck, timeElapsed grows large That happens because sync() discards tokensPerSecond decay unless accumulator changed

```
    (uint256 accumulator, uint256 newYield, uint256 tokensPerSecond) = getAccumulator(_bToken);
    if (accumulator != staking.accumulator) {
    pool.pendingYield -= newYield.toUint128();
    ...
    staking.lastUpdated = block.timestamp.toUint32();
    }
```

So if accumulatorIncrease = 0, happens when pendingYield = 0, then
staking.tokensPerSecond is not updated
staking.lastUpdated is not updated, so timeElapsed keeps growing
The next time pool.pendingYield becomes non zero again,
Yield = min(tokensPerSecond * timeElapsed, pendingYield) collapses to pendingYield, meaning all pending yield becomes distributable in one _sync(), and therefore claimable immediately
This is MEV friendly. Whoever can be the first to trigger _sync() after pendingYield refills gets to decide
Who is staked at the exact instant the entire bucket is pushed into claimableYield/accumulator
This breaks the intended reward drip model
Preconditions for this
1- pendingYield is 0 for a period
2- nobody calls a staking function that runs sync() for some time, so staking.lastUpdated doesn't
Advance and tokensPerSecond decay isn't persisted,
3- later pendingYield becomes > 0, from swaps, borrow, and someone triggers sync()
https://gist.github.com/GuardianAudits/445dde235979efc5e0b7e9fc35769a39

Recommendation

If this is acceptable consider acknowledging it

Resolution

Baseline:
https://github.com/0xBaseline/mercury/blob/1b7518ed8a253384faead6d56c37e84928704a76/src/components/BStaking.sol#L213.

M-04 | Reopen Buy Bypasses Stored Invariant

Category	Severity	Location	Status
Logical Error	Medium	CurveLib.sol	Resolved

Description

In computeZeroCircSwap() When a pool is in circ = 0 and someone reopens it with a buy, the code does

```
    uint256 ratio = _deltaCirc.divWadUp(x1);
    uint256 newBuffer = FixedPointMathLib.fullMulDivUp(
    _p.lastInvariant,
    int256(ratio).powWad(_p.convexityExp.toInt256()).toUint256(),
    WAD
    );
```

If the reopen buy is small enough, ratio < 1, and for sufficiently large convexityExp the WAD fixed point result of

```
    powWad(ratio, convexityExp)
```

Rounds down to 0

In the normal swap path, that same kind of rounding results in the division by zero DoS issue But here, in computeZeroCircSwap, the zero is used as a multiplier, so the trade does not revert. Instead

```
    newBuffer = 0
    bufferReserves = 0
```

And the reopen buy is charged only

```
    payment = _deltaCirc  BLV  (1 + swapFee)
```

So the buyer gets reopened supply at pure BLV + fee, while the invariant implied premium is silently discarded

A user can push a pool into circ = 0 by selling all circulating bTokens through the special c1 = 0 branch in computeSwap after that, recordSwap() does not refresh lastInvariant, because it only recomputes the invariant when

```
    pool.totalBTokens < pool.totalSupply * 95%
```

At circ = 0, the pool holds 100% of supply, so maker.lastInvariant stays as a stale positive K from the pre zero state that stale K is exactly what computeZeroCircSwap() is supposed to use to price the first reopen buy. But the WAD rounding zeroes it out

The rounding condition is

```
    (deltaCirc / x1) ^ convexityExp < 1e-18
```

With convexityExp = 50e18 for example, a reopen buy below about 30.4% rounds to 0 and triggers the issue So in high convexity pools, an attacker can reopen a large chunk of supply at floor pricing

The first reopen buyer can acquire supply far cheaper than the stored invariant requires That matters because those bTokens can then be sold later once the pool is reopened, or used to capture upside while the pool's premium was never paid on entry So the premium encoded in lastInvariant is effectively bypassed

https://gist.github.com/GuardianAudits/4d947e3373396c6e391a01d385fe0ae5

Recommendation

we need to prevent the powWad(ratio, convexityExp) rounding to 0 from silently zeroing newBuffer / premium on reopen buys

Resolution

Baseline: Resolved.

M-05 | Non Additive IL Fee Exploitable

Category	Severity	Location	Status
Logical Error	Medium	CurveLib.sol	Partially resolved

Description

In _computeFee() / computeSwap()
For a sell, the code does

```
    invariantDelta = BLV * Δ + bufferDecrease
    fee = (bufferDecrease - marginalBufferDelta) + premiumSwapFee
    so userOut = invariantDelta - fee
```

That simplifies to

```
    userOut  ≈  Δ    ( BLV + (1 - swapFee)  marginalPremium_after )
```

So the seller is effectively paid the post trade marginal premium across the whole trade, not the integral over the interval
For a buy, the same thing happens in reverse
The buyer pays roughly the post trade marginal premium across the whole trade, plus the BLV fee,
Instead of paying the integral over the interval That means the fee model is path dependent,
One big sell gets paid using the worst marginal price on the path, but many small sells get paid using a sequence of better marginals, so splitting strictly improves execution.
Likewise, one big buy pays the worst marginal price on the path, but many small buys pay a staircase of cheaper marginals, so splitting strictly lowers cost.
Therefore, the IL fee is not enforceable on chain, because a user can always partition the trade
A contract can loop sellTokensExactIn() or buyTokensExactOut() in one transaction and collapse the non linear fee toward zero
https://gist.github.com/GuardianAudits/2debf9a39176280c73d7772b20e4f6c8

Recommendation

Consider replacing _computeFee() with additive fee
https://gist.github.com/GuardianAudits/cd76e271a4d4083beb5dfd2c4b609dc2

Resolution

Partially resolved

M-06 | TPS Decay Computed But Never Applied

Category	Severity	Location	Status
Logical Error	Medium	BStacking.sol	Resolved

Description

In getAccumulator() the code compute

```
    uint256 decayedTps = tokensPerSecond_.mulWad(_decayFactorWad(timeElapsed,
timeToDistribute));
```

But then we update from the old stored TPS

```
    if (targetTps > decayedTps) {
    tokensPerSecond_ += gain;
    } else {
    tokensPerSecond_ -= gain;
    }
```

It should really update from the decayed TPS, not the stale one

Otherwise decay is only used for comparison, but not actually used as the new base rate After long gaps decayedTps can go ~ 0 so gain goes ~ 0, leaving TPS stale because getAccumulator() computes decayedTps but uses tokensPerSecond_ from the pre decay value and uses it for yield, so decay isn't applied to the stored TPS

Recommendation

apply decay to the TPS state not just the comparison

```
    // getAccumulator()
    uint256 decayedTps = tokensPerSecond_.mulWad(_decayFactorWad(timeElapsed,
timeToDistribute));
    tokensPerSecond_ = decayedTps; // use decayed TPS as the new base
```

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/c9b80084a8b56929bd02b102d537fc4c67efa0f1/src/components/BStaking.sol#L258.

L-01 | Pool Initialization May Fail

Category	Severity	Location	Status
Validation	Low	CurveLib.sol: 243	Resolved

Description

The following requirement was added to the first branch in `CurveLib.computeInitialCurveParams()`

```
    require(convexityExp_ > 2e18 && convexityExp_ <= MAX_CONVEXITY, InvalidConvexityExp());
```

Because of rounding during arithmetic operations, the resulting `convexityExp` may end up being exactly `2e18`. Since `2e18` is excluded from the allowed values, the pool initialization will fail.

Recommendation

Consider relaxing the check to allow the convexity to be `2e18`.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/56d7000fbf08ffa2d28c108abc0ae9ce0d968481/src/libraries/CurveLib.sol#L243.

L-02 | Impossible To Default If Resulting Circ == 0

Category	Severity	Location	Status
DoS	Low	BCredit.sol: 275	Resolved

Description

On each call to `defaultSelf()` the `totalBTokens` are increased and therefore the circulating supply decreases. After that, `CurveLib.computeInvariant()` is executed. This call will revert if the resulting circulating tokens are 0, blocking the default from happening.

Recommendation

Consider not recomputing the invariant if the resulting `circ == 0`. In fact, you can modify `computeInvariant()` to return the old `K` if `circ == 0`.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/0b826f2fb5c37277c7779e1325cd175ec52fba00/src/libraries/CurveLib.sol#L46.

L-03 | Hook Swap Bypasses Reserve Dust Accounting

Category	Severity	Location	Status
Logical Error	Low	BHook.sol	Resolved

Description

buyTokensExactIn() explicitly captures the dust difference between user supplied _amountIn and the curve's computed actualCost

```
    uint256 actualCost = (-userDeltaReserves).toUint256();
    uint256 dustFee = _amountIn > actualCost ? _amountIn - actualCost : 0;
    if (msg.sender != address(this)) {
    pool.reserve.takeReserves(msg.sender, dustFee);
    FeeLib.distributeFees(pool, _bToken, dustFee);
    }
```

In the Uniswap-v4 hook BHook.beforeSwap, the hook calls BSwap() as a self-call, so inside BSwap we have msg.sender == address(this). That means the dustFee capture block is skipped So the protocol can end up with extra reserve tokens actually received (the exact-in amount) that are not reflected in per-pool accounting (pool.totalReserves, fee distribution, et.) because the dust capture path never ran

Recommendation

Replace the dust block in buyTokensExactIn with

```
    uint256 dustFee = _amountIn > actualCost ? _amountIn - actualCost : 0;
    if (dustFee > 0) {
    State.Pool storage pool = State.pool(_bToken);
    ...
    FeeLib.distributeFees(pool, _bToken, dustFee);
    }
```

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/179c7d8b07deeec24a9e44a4c6e1ea51a411c9ab/src/components/BSwap.sol#L95.

L-04 | Hidden Reserves In Hook Swaps

Category	Severity	Location	Status
Logical Error	Low	BHook.sol,166	Resolved

Description

In hook exact-output sells, BHook self-calls sellTokensExactOut, so msg.sender = address(this) skips the dustFee clawback + distributeFees block. But swapTokens / _updateAccounting() still updates pool.totalReserves using the full userDeltaReserves (the curve's actual received amount), while BHook.beforeSwap only transfers the requested _amountOut,
Leaving actualReceived - _amountOut sitting in the Relay untracked

Recommendation

Replace the dust block in sellTokensExactOut with

```
    uint256 dustFee = actualReceived > _amountOut ? actualReceived - _amountOut : 0;
    if (dustFee > 0) {
    State.Pool storage pool = State.pool(_bToken);
    ...
    FeeLib.distributeFees(pool, _bToken, dustFee);
    }
```

Resolution

Baseline:
https://github.com/0xBaseline/mercury/blob/179c7d8b07deeec24a9e44a4c6e1ea51a411c9ab/src/components/BSwap.sol#L95.

I-01 | Initial Credits May Be Unclaimable

Category	Severity	Location	Status
Warning	Info	BCredit.sol	Acknowledged

Description

New `getBorrowForCollateral()` checks were added when a pool with credit position is created and when user claims. However, `getBorrowForCollateral()` performs math operations which include rounding and because of that `f(x) == f(x1 + x2)`, where `x = x1 + x2` is not guaranteed.

This can lead to valid states causing unclaimable credits. For example:

- `BLV = 0.05 ether`
- `InitialCollateral = 1000 ether`
- `InitialDebt = 50 ether`

The pool will be created successfully because `initialDebt == maxDebt`. However because of rounding issues some of the users may not be able to claim their part.

Recommendation

Run a local simulation of the maximum allowed debt per each user before creating a pool with credit positions. It's important to not use `getBorrowForCollateral()` for this simulation because the pool is not created and it values will be 0.

Resolution

Baseline: Acknowledged.

I-02 | Pool Creation Cannot Be Paid With Native Tokens

Category	Severity	Location	Status
Best Practices	Info	BFactory.sol	Resolved

Description

`BFactory.createPool()` uses `handleIncoming` to charge the user the needed amount of reserve tokens. Usually, this function supports payment with native token to be converted to its wrapped version, but because `createPool()` is not payable, such payments are impossible.

Recommendation

If this feature is desired, consider adding `payable` to `createPool()`. Otherwise, acknowledge the issue

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/179c7d8b07deeec24a9e44a4c6e1ea51a411c9ab/src/components/BFactory.sol#L145.

I-03 | Outdated Comment

Category	Severity	Location	Status
Best Practices	Info	BFactory.sol: 181C1-182C56	Resolved

Description

The comment at `BFactory.sol#L181-182` is outdated as there is no vault anymore.

```
    // use the vault-credited amount (after rounding) for totalReserves
    pool.totalReserves = totalReserves.toUint128();
```

Recommendation

Remove the comment.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/179c7d8b07deeec24a9e44a4c6e1ea51a411c9ab/src/components/BFactory.sol#L173.

I-04 | Lack Of Validation For creatorFeePct

Category	Severity	Location	Status
Validation	Info	BController.sol: 110	Resolved

Description

`BController.modifyCreatorFeePct()` allows the executor to change the the creator fee for a given pool, but does not validate the new fee is in range (`<= 1e18`).

Recommendation

Consider adding the same validation as in `BFactory.createPool()`

```
    require(params.creatorFeePct <= 1e18, InvalidCreatorFee());
```

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/179c7d8b07deeec24a9e44a4c6e1ea51a411c9ab/src/components/BController.sol#L111.

I-05 | Redundant Function Parameter

Category	Severity	Location	Status
Superfluous Code	Info	BCredit.sol: 544	Resolved

Description

The internal `BCredit._previewRepay()` function defines a `bToken` parameter that's never used.

Recommendation

Remove the `bToken` parameter.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/179c7d8b07deeec24a9e44a4c6e1ea51a411c9ab/src/components/BCredit.sol#L544.

I-06 | previewRepay() Should Return debtToRepay

Category	Severity	Location	Status
Suggestion	Info	BCredit.sol: 536	Resolved

Description

The debt repayment flow was reworked to repay the exact amount sent by the user and revert if it's more than the active debt.
The public `previewRepay()` function now returns only `collateralRedeemed_`, but it still caps the debt that will be repaid to the maximum available amount.

```
    _reservesIn = FixedPointMathLib.min(_reservesIn, account.debt);
```

This may mislead offchain integrators that use this function.

Recommendation

Consider returning `_reservesIn` as a second parameter so it's known how much of the debt was repaid.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/179c7d8b07deeec24a9e44a4c6e1ea51a411c9ab/src/components/BCredit.sol#L537.

I-07 | Redundant mulWad() In Active Price Computation

Category	Severity	Location	Status
Superfluous Code	Info	CurveLib.sol: 62	Resolved

Description

`CurveLib.computeActivePrice()` computes the premium to be added to `BLV`:

```
    uint256 premium = FixedPointMathLib.fullMulDiv(
    _params.reserves - _params.BLV.mulWad(_params.circ),
    _params.convexityExp.mulWad(_params.totalSupply),
    _params.supply.mulWad(_params.circ).mulWad(WAD)
    );
```

On the last line of the code snippet, we can see `mulWad(WAD)`. Because `mulWad` is equivalent to `x * y / WAD`, the end result will be `x * WAD / WAD = x`, which makes the operation unnecessary.

Recommendation

Delete the last `mulWad`

```
    uint256 premium = FixedPointMathLib.fullMulDiv(
    _params.reserves - _params.BLV.mulWad(_params.circ),
    _params.convexityExp.mulWad(_params.totalSupply),
    -           _params.supply.mulWad(_params.circ).mulWad(WAD)
    +          _params.supply.mulWad(_params.circ)
    );
```

Resolution

Baseline: Resolved.

I-08 | Unused Errors

Category	Severity	Location	Status
Superfluous Code	Info	BFactory.sol: 38-40	Resolved

Description

The following errors in `BFactory` are not used:
- `InsufficientPoolBTokens()`
- `InvalidBTokenDecimals()`
- `InvalidReserveDecimals()`

Recommendation

Remove the errors

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/179c7d8b07deeec24a9e44a4c6e1ea51a411c9ab/src/components/BFactory.sol#L38.

I-09 | Comment Mismatch On Premium Threshold

Category	Severity	Location	Status
Documentation	Info	MakerLib.sol: 273	Resolved

Description

In `MakerLib._translateConvexity()` the comment says “premium > 100% (activePrice > 2 × BLV)”, but the code returns only when `premium < WAD`, so relaxation happens when `premium >= WAD`, i.e., `activePrice >= 2 × BLV` (subject to rounding). This is a comment-logic mismatch that can mislead reviewers about the exact gating condition.

Recommendation

Align the comment with the code behavior.

Resolution

Baseline: Resolved.

I-10 | Confusing Plural Used For Variable Naming

Category	Severity	Location	Status
Best Practices	Info	BCredit.sol: 259	Resolved

Description

The credit account of the user in `defaultSelf()` is named `accounts` instead of `account` which may be confusing for code readers.

```
    State.CreditAccount storage accounts = credit.accounts[msg.sender];
```

Recommendation

Consider renaming the variable to `account`.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/179c7d8b07deeec24a9e44a4c6e1ea51a411c9ab/src/components/BCredit.sol#L259.

I-11 | K Can Overflow

Category	Severity	Location	Status
DoS	Info	Global	Resolved

Description

Currently, `K` is stored as `uint128.` This may not be enough for pools with larger supply, which will cause overflow issues.

Recommendation

Consider whether switching to `uint256` is favorable.

Resolution

Baseline: The issue was resolved in commit 260c036.

Round Three Findings & Resolutions

ID	Title	Category	Severity	Status
C-01	Permanent BLV Breakage	Logical Error	Critical	Resolved
C-02	Pending Commit Can Be Permanently Poisoned	Logical Error	Critical	Resolved
H-01	Active Price Mismatch Censors Exact Buys	DoS	High	Resolved
M-01	Stale Invariant Doesn't Account For Fees	Logical Error	Medium	Resolved
M-02	Mixed Direction Block Pricing Not Correct	Logical Error	Medium	Acknowledged
M-03	Swaps Revert Due To Fee Subtraction Underflow	DoS	Medium	Resolved
M-04	Lower Domain Failure In _tryRecomputeMaxReserves	DoS	Medium	Resolved
M-05	Sell Sign Inversion From Invariant Rounding	DoS	Medium	Partially resolved
L-01	Dripping Assymmetry When Pending Yield Is 0	Gaming	Low	Resolved
L-02	BLens May Return Stale Curve Parameters	Logical Error	Low	Resolved
I-01	BCredit.claim() Integration Notes	Documentation	Info	Resolved
I-02	Not All Tokens Can Be Bought	Informational	Info	Acknowledged

C-01 | Permanent BLV Breakage

Category	Severity	Location	Status
Logical Error	● Critical	.	Resolved

Description

computeNextBLV() has a dangerous denominator that is built with two nested floor normalizations

```
    uint256 penaltyDenom = fullMulDiv(
    fullMulDiv(_params.supply, _params.supply, 1e18),
    fullMulDiv(prevCirc, prevCirc, 1e18),
    1e18
    );
```

So penaltyDenom can become 0 even when both params.supply > 0 and prevCirc > 0, once that happens, the later fullMulDivUp reverts

```
    fullMulDivUp( , penaltyNumer, penaltyDenom)
```

The condition is

```
    currentSupply * previousCirculation < 1e27 in WAD units,
```

For 18-decimal tokens, that is about:

```
    current inventory * previous circulation < 1e-9 token²
```

Example on a 18-dec pool

```
    totalSupply =  10_000e18
    block-start prevCirc =  0.00001e18 tokens
    block-end supply =  0.00002e18 tokens
```

Then inside computeNextBLV()

```
    FullMulDiv(supply, supply, 1e18) = 4e8
    FullMulDiv(prevCirc, prevCirc, 1e18) = 1e8
    outer FullMulDiv(4e8, 1e8, 1e18) = 0
```

So the next commit hits a zero denominator and reverts. The failure mode is:
The poisoning swap in one block succeeds. In the next block, the first swap tries to preview/commit deferred maker state.
previewDeferredMakerState() calls computeNextBLV() in the convexityExp = 2e18 path below the 95% threshold.
BSwap buy/sell, BHook.beforeSwap, and BCredit leverage/deleverage are then DoSed because every future interaction retries the same poisoned commit first. The pool stays stuck until privileged intervention.
There is no public recovery path that clears blockPricing or overwrites maker state, so the pool stays bricked across later blocks until a contract upgrade.
https://gist.github.com/GuardianAudits/ca56bc481412b7389e46bb27169d82fa

Recommendation

```
    uint256 penaltyDenom = FixedPointMathLib.fullMulDiv(
    FixedPointMathLib.fullMulDiv(_params.supply, _params.supply, 1e18),
    FixedPointMathLib.fullMulDiv(prevCirc, prevCirc, 1e18),
    1e18
    );
    if (penaltyDenom == 0) return _params.BLV;
```

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/cbf1b44c312fe610a26bd65c4369e751703c1241/src/libraries/CurveLib.sol#L314-L320.

C-02 | Pending Commit Can Be Permanently Poisoned

Category	Severity	Location	Status
Logical Error	● Critical	7e851cfcdc8b861580c488e87ca3ccbf	Resolved

Description

Pending block commit can be permanently poisoned by an unbounded powWad, the path is

```
swapTokens() -> _advanceBlockPricing() -> _commitDeferredMakerState() ->
previewDeferredMakerState() -> _previewTranslateConvexity() ->
previewRecomputeMaxReserves()
```

Inside _previewRecomputeMaxReserves()

```
uint256 positionRatio = _next.maxCirc.mulWad(_params.supply)
.divWad(_params.circ.mulWad(minSupply));
uint256 impliedBuffer = _buffer.mulWad(
int256(positionRatio).powWad(int256(_nNew)).toUint256()
);
```

There is no overflow check before powWad(positionRatio, nNew)
This executes only on a below ATH buy block when convexityExp > 2e18 and the pool is still below the 95% inventory threshold. So when
The pool previously reached a large maxCirc,
Later sells pushed current circ back down, then someone does a small buy while still below the old ATH
The key is that the protocol swap guards only bound the exponent for the current trade ratio supply/circ or circ/supply inside computeSwap But _previewRecomputeMaxReserves() uses a different base

```
positionRatio = (maxCirc  supply) / (circ  (totalSupply - maxCirc))
```

That base can be much larger than any ratio checked during swaps
A reachable shape is

```
previous ATH:  maxCirc = 0.99 * totalSupply
current state after sells:  circ = 0.06 * totalSupply
current convexityExp = 20e18
```

Then current swap state ratio is only about 0.94 / 0.06 = 15.7, which is well inside the normal computeSwap safety bound, but

```
positionRatio = (0.99  0.94) / (0.06  0.01) = 1551
```

For a tiny below ATH buy, _previewTranslateConvexity() computes nFloor from price continuity, and as the buy size goes to zero, nFloor approaches the current convexityExp. So nFloor can stay around 19–20e18
At that point

```
ln(1551) * 19 > 135
```

So powWad(positionRatio, nFloor) overflow / revert
The triggering buy succeeds in its own block, because the maker update is deferred Then on the next block, every future swap starts with _advanceBlockPricing()
advanceBlockPricing() must commit the old pending block,
The commit reenters the same reverting _previewRecomputeMaxReserves() path, so the pending block is never cleared That means the pool gets stuck with a poisoned State.blockPricing entry. After that all of these revert for that pool, Swaps, Hook swaps, Leverage, deleverage
The pool stays bricked across later blocks until doing a contract upgrade
https://gist.github.com/GuardianAudits/7e851cfcdc8b861580c488e87ca3ccbf

Recommendation

https://gist.github.com/GuardianAudits/e0b059fe396b5f3ecf3e329b8d5a674f

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/cbf1b44c312fe610a26bd65c4369e751703c1241/src/libraries/MakerLib.sol#L356-L385.

H-01 | Active Price Mismatch Censors Exact Buys

Category	Severity	Location	Status
DoS	High	.	Resolved

Description

quoteBuyExactOut() mixes two different states in the same calculation

AmountIn comes from quoteSwap(), which uses _previewBlockPricing() and prices off the frozen block start snapshot but initialPrice comes from computeActivePrice(getCurveParams()), which can reflect the live pool state in the current block

On buys below BUFFER_SAFETY_THRESHOLD, _updateAccounting() immediately keeps part of the fee inside pool.totalReserves, raising the live active price that means after one buy, the live initialPrice can be higher than the frozen marginal price used for the quote when that happens, slippage = (effectivePrice - initialPrice) / initialPrice in quoteBuyExactOut() underflows and reverts

```
    uint256 initialPrice = CurveLib.computeActivePrice(p);
    uint256 effectivePrice = amountIn / amountOut;
    slippage_ = (effectivePrice - initialPrice).divWad(initialPrice);
```

This also breaks exact input buys, because _solveBuy() depends on _quoteBuyCost(), which calls quoteBuyExactOut() So buyTokensExactIn() can be DoSed

Recommendation

Compute InitialPrice from the frozen snapshot

Resolution

Baseline: Resolved.

M-01 | Stale Invariant Doesn't Account For Fees

Category	Severity	Location	Status
Logical Error	Medium	MakerLib.sol: 295-299	Resolved

Description

During each swap, the swap fee is added to the pool's liquidity only if the circulating supply is high enough

```
    if (uint256(pool.totalBTokens) <
uint256(pool.totalSupply).mulWad(BUFFER_SAFETY_THRESHOLD)) {
    liquidityFee_ = _feesReceived - `FEE_DISTRIBUTION_SHARE.mulWad(_feesReceived);
    }
```

Even though the curve parameters are frozen for the duration of the block, `totalPoolTokens` is updated after every swap. At the start of the next block, the `invariant` will be refreshed only if the new curve's circulating is not above the `BUFFER_SAFETY_THRESHOLD`. Previously, this logic was executed after every swap, which ensured there would be no fees distributed and therefore the invariant won't change. However, with the new design there may be fees generated in the block before the curve ended up in the `BUFFER_SAFETY` region. The fees will not be accounted in the pricing mechanism because the invariant will remain stale.

This results in users getting better execution prices, and in some cases the calculation may result in user receiving both `bToken` and `reserve` during a swap. This will be caught by the validation in `MakerLib()` and users will have their transactions reverted with `InvalidSwapDirection()`

Recommendation

Consider updating the invariant if a new valid value can be computed. Keep in mind this can result in the curve changing shape even in the buffer threshold area.

https://gist.github.com/GuardianAudits/81cd73fa047b1fc4476a087ee75f38c6

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/cbf1b44c312fe610a26bd65c4369e751703c1241/src/libraries/MakerLib.sol#L269.

M-02 | Mixed Direction Block Pricing Not Correct

Category	Severity	Location	Status
Logical Error	Medium	BlockPricingLib.sol	Acknowledged

Description

BlockPricingLib says swaps should be charged on a frozen block start curve as
`Q(cumulativeAfter) - Q(cumulativeBefore)`

But the implementation does not track one cumulative signed supply delta

It tracks two separate one sided deltas, blockBuyDeltaCirc and blockSellDeltaCirc and cumulativeQuoteBounds() uses only the accumulator that matches the current trade direction That means a buy after earlier sells in the same block is priced as if those sells never happened

As well a sell after earlier buys in the same block is priced as if those buys never happened

Even though _updateAccounting() is mutating pool.totalBTokens and pool.totalReserves after every swap So _cumulativeQuoteBounds() does this logically

For a buy, price from `buyCum -> buyCum + delta`

For a sell, price from `-sellCum -> -(sellCum + delta)`

A buy only sees earlier buys, and a sell only sees earlier sells

In a frozen curve cumulative pricing, the next trade must use the net signed block flow

Recommendation: TBD

Recommendation

Resolution

Baseline: acknowledged

M-03 | Swaps Revert Due To Fee Subtraction Underflow

Category	Severity	Location	Status
DoS	Medium	BlockPricingLib.sol: 148-166	Resolved

Description

`BlockPricingLib._quoteDeltaFromSnapshot()` is computing the fee to be paid by the user as the delta between `deltaUserWad` and `deltaInvariantWad` depending on the trade direction
The two delta variables are computed as it follows:

```
    int256 deltaUserWad = afterUserWad - beforeUserWad;
    int256 deltaInvariantWad = afterInvWad - beforeInvWad;
```

Let's say `f(x) = fee of Q(x)`. Due to buffer roundings and `zeroFloorSubs`, a possible state is `f(cumulativeAfter) <= f(cumulativeBefore)`. Then the relationship between the two delta variables is: `deltaUserWad > deltaInvariantWad` When the code reaches the `if/else` block it will try to compute `feesReceived_`, but the subtraction will result in an underflow and the swap will fail

```
    if (deltaUserWad < 0) {
    // BUY: ceil user payment, ceil curve need, fee is residual
    uint256 userPayNative = NormalizeLib.denormalizeWadUp(uint256(-deltaUserWad), rDec);
    ...
    feesReceived_ = curveReleaseNative - userReceiveNative;
    }
```

Recommendation

If this behavior is not acceptable, you can cap the fee at 0
https://gist.github.com/GuardianAudits/14f0e340c04a97546e3b92efc044953e

Resolution

Baseline:
https://github.com/0xBaseline/mercury/blob/e9d290ecd4f9e7b93c3c9a1b2cc31d522161aec0/src/libraries/BlockPricingLib.sol#L157.

M-04 | Lower Domain Failure In _tryRecomputeMaxReserves

Category	Severity	Location	Status
DoS	Medium	CurveLib.sol	Resolved

Description

We only defends the positionRatio > 1 overflow side

```
    if (
    positionRatio > WAD &&
    _nNew.mulWad(uint256(int256(positionRatio).lnWad())) > 135e18
    ) {
    return false;
    }
```

But the function does not defend the positionRatio = 0 or tiny positive < 1 side before calling

```
    uint256 growth = int256(positionRatio).powWad(int256(_nNew)).toUint256();
```

That matters because positionRatio is built with two floor divisions

```
    num = fullMulDiv(_next.maxCirc, _params.supply, WAD);
    den = fullMulDiv(_params.circ, minSupply, WAD);
    positionRatio = fullMulDiv(num, WAD, den);
```

So in below ATH high convexity pools, positionRatio can floor all the way to 0 when historical maxCirc is tiny relative to current circ
Example Inside _tryRecomputeMaxReserves

```
    num = floor(1 * 9400e18 / 1e18) = 9400
    den = floor(600e18 * (10000e18 - 1) / 1e18) ≈ 6_000_000e18
    positionRatio = floor(9400 * 1e18 / 6_000_000e18) = 0
```

So the next block deferred commit hits powWad(0, _nNew) and reverts
The impact is the same poisoned deferred state pattern but with lower likelihood

Recommendation

Keep _tryRecomputeMaxReserves fail closed on the lower side too

```
    uint256 positionRatio = FixedPointMathLib.fullMulDiv(num, WAD, den);
    if (positionRatio == 0) return false;
    if (positionRatio > uint256(type(int256).max)) return false;
    ...
    uint256 growth = int256(positionRatio).powWad(int256(_nNew)).toUint256();
    if (growth == 0) return false;
```

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/6746e481bb97ad4617fbca7aa7638121713aa7ad/src/libraries/MakerLib.sol#L447.

M-05 | Sell Sign Inversion From Invariant Rounding

Category	Severity	Location	Status
DoS	Medium	CurveLib.sol	Partially resolved

Description

When the pool is in the supply < circ regime, the protocol compresses the buffer into lastInvariant with one rounding direction, then later expands it back with the opposite rounding direction

computeInvariant() stores K using

```
    invRatio = circ.divWad(supply) (round down)
    K = buffer.divWadUp(powWad(invRatio, n))
```

computeSwap() later reconstructs the post trade buffer using

```
    invRatio = c1.divWadUp(x1) (round up)
    newBuffer = fullMulDivUp(K, powWad(invRatio, n), WAD)
```

That floor to ceil asymmetry is not an inverse map. In high convexity / low buffer supply < circ states, a sell can reconstruct a newBuffer that is too large once that happens, the sell path can do something impossible

PriceAfter >= priceBefore on a sell and in worse cases newReserves > oldReserves, so invariantDelta becomes negative on a sell That bad sign then blows up the sell fee path, because _computeFee() assumes sell side

_invariantDelta > 0 and does uint256(_invariantDelta)

State example In the poc

```
    totalSupply = 1_000_000e18
    totalBTokens (supply) = 200_000e18
    circ = 800_000e18
    ...
    convexityExp = 30e18
    lastInvariant = 174
```

Now if we sell 1e18 bToken

```
    priceBefore = 0.1375e18
    priceAfter = 0.1376068710996149e18  <- price rises on a sell
    invariantDelta = -0.47073133411459367e18  <- negative on a sell
```

In that same state, sells of 0.1, 0.5, 1, and 2 tokens all hit the negative invariantDelta region

The first positive output sell only appears around 5 tokens, and even there the sell still raises price

_computeFee() sell branch casts uint256(_invariantDelta) on a negative int that wraps to a huge uint then fee_.toInt256() reverts So once a pool enters this region, legitimate sell side flows could become uncallable

https://gist.github.com/GuardianAudits/9156c953cdbc2d0a553b71dbdeb631a5

Recommendation

In the computeSwap() x1 < c1 branch

```
    uint256 invRatio = c1.divWad(x1);  // not divWadUp
```

This matches the supply < circ rounding used when computeInvariant() stores K, so sells stop overstating newBuffer But do it for sell only, because the x1 < c1 branch is used by buys too

```
    uint256 invRatio = _deltaCirc < 0 ? c1.divWad(x1) : c1.divWadUp(x1);
```

This might not completely mitigate it, but it will make the state harder to reach,

Also consider acknowledging if the Impact is acceptable

Resolution in commit b8f2f67, Partially resolved

L-01 | Dripping Assymmetry When Pending Yield Is 0

Category	Severity	Location	Status
Gaming	Low	BStaking.sol: 231-234	Resolved

Description

`BStaking._sync()` refreshes `tokensPerSecond` and `lastUpdated` when there is no accumulator change because of 0 `pendingYield`

```
    else if (pool.pendingYield == 0 || staking.totalStaked == 0) {
    staking.tokensPerSecond = tokensPerSecond.toUint128();
    staking.lastUpdated = block.timestamp.toUint32();
    }
```

Once new yield is generated, the instantly distributable amount will depend on whether `_sync()` was called in the 0 yield period. If no interaction triggers `_sync()` when yield is 0, `tokensPerSecond` will still be stuck, allowing a big part of the yield (if not the whole) to be distributed as soon as it enters the system.

Recommendation

You can execute a 0 deposit in `FeeLib.distributeFees()` if the yield was previously 0. This will trigger a `_sync()` right before the new yield arrives and `tokensPerSecond` will be properly updated.

```
    if (remaining > 0){
    if (_pool.pendingYield == 0) {
    BStaking(address(this)).deposit(_bToken, address(this), 0);
    }
    _pool.pendingYield += remaining;
    }
```

Resolution

Baseline: The issue was resolved in commit be9e279.

L-02 | BLens May Return Stale Curve Parameters

Category	Severity	Location	Status
Logical Error	Low	BLens.sol	Resolved

Description

The `BLens` functions `blvPrice()`, `convexityExponent()` and `lastInvariant` read the parameters' value from the committed state. This will result in integrators getting stale data when there is a pending state to be committed.

Recommendation

Consider calling `MakerLib.getCurveParams()` and reading each value from it.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/cbf1b44c312fe610a26bd65c4369e751703c1241/src/components/BLens.sol#L315-L330.

I-01 | BCredit.claim() Integration Notes

Category	Severity	Location	Status
Documentation	Info	BStaking.sol: 122	Resolved

Description

`BCredit.claim()` allows anyone to initiate the claim for a given user, providing `asNative`. This can result in locked funds if integrators are not aware of it. For example, if they are using a contract that operates with only one type of token, anyone can trigger a claim with the inverse `asNative` value to lock the funds.

Recommendation

Document this, so integrators can handle both types of rewards.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/602b4a6a9e1db9327356b5c9652f16f93032cbdc/src/components/BStaking.sol#L119.

I-02 | Not All Tokens Can Be Bought

Category	Severity	Location	Status
Informational	Info	CurveLib.sol: 111	Acknowledged

Description

If a swap tries to buy all the tokens from the pool, `x1` will be calculated as 0.

```
    uint256 x1 = (_params.supply.toInt256() - _deltaCirc).toUint256();
```

Then it will be used as a denominator to compute `invRatio`, which will result in a revert and inability to buy all tokens from the pool.

```
    uint256 invRatio = c1.divWadUp(x1);
```

Even if a trade leaves 1 wei sitting in the pool, the resulting value of `invRatio` will be huge and the transaction will revert when performing next calculations. This creates a limit of how much tokens can be bought.

Recommendation

Be aware of this behavior of the pool.

Resolution

Baseline: Acknowledged.

Round Four Findings & Resolutions

ID	Title	Category	Severity	Status
M-01	Reallocated Fees Commit Divergency	DoS	Medium	Resolved
L-01	Undistributed Fees When Circ Reaches Zero	Logical Error	Low	Resolved
L-02	Buffer Threshold Precision Mismatch	Math	Low	Resolved
L-03	Fee Rounding Favors Buyer Over Protocol	Rounding	Low	Resolved
L-04	Stale BLV In BCredit And quoteLeverage	Logical Error	Low	Acknowledged
L-05	computeNextBLV() Overflow DOSes Pool	DoS	Low	Resolved
I-01	Integration Note Completeness	Documentation	Info	Resolved

M-01 | Reallocated Fees Commit Divergency

Category	Severity	Location	Status
DoS	Medium	MakerLib.sol: 272-293	Resolved

Description

When the pool is in the safety region, _commitDeferredMakerState() computes expected reserves from the implied buffer, then compares that value with stored totalReserves and distributes the excess as fees. This creates several problems:

The fee removal logic is not replayed in `_previewDeferredMakerState()`. This leads to all quoting functions reading an inflated value for `totalReserves`. When a quote calls `CurveLib.computeSwap`, the `invariantDelta` value will also be inflated.

```
    invariantDelta_ = _params.reserves.toInt256() - newReserves;
```

This can result in a positive `invariantDelta_` for buys, DoS-ing the `BSwap.buyTokensExactIn()` flow. For sells, the inflated `invariantDelta_` value will result in wrong fee being calculated when the `zeroFloorSub` case is not entered. Because `totalReserves` is artificially increased until the fee is settled, `GuardLib.ensureSolvent()` becomes less restrictive and the risk of the protocol being left in a non-solvent state increases.

```
    function ensureSolvent(BToken _bToken) internal view {
    require(State.pool(_bToken).totalReserves >= State.credit(_bToken).totalDebt,
GuardLib_Insolvent());
    }
```

This discrepancy will also affect third-party integrators that read from `BLens`.

The way the fee surplus is calculated based on the implied buffer and implied reserves is also risky because these calculations are never exact.

Taking fees out of the pool also creates a new attack vector: stakers can intentionally push the pool into the safety regime to get those fees distributed to them. This is even easier for block builders, who can order transactions in their favor with minimal capital.

Recommendation

If the staker risk is acceptable and you plan to stick to the current design, consider tracking potential fees in a separate variable, not in `totalReserves`. Then correct the preview logic to account for that variable as well.

Resolution

Baseline: The issue was resolved in PR#172.

L-01 | Undistributed Fees When Circ Reaches Zero

Category	Severity	Location	Status
Logical Error	Low	MakerLib.sol: 276	Resolved

Description

When `_updateAccounting()` charges a liquidity fee, it retains a portion of swap fees in `totalReserves`. The surplus-distribution branch in `_commitDeferredMakerState()` is responsible for distributing these retained fees by reverse-engineering the surplus from the curve invariant. However, this branch is guarded by `params.circ > 0` because the implied-reserves calculation requires `supply.divWad(circ)`, which would revert on division by zero. If multiple sells occur in the same block where earlier sells generate liquidity fees (pool below `BUFFER_SAFETY_THRESHOLD`) and a later sell pushes `totalBTokens == totalSupply` (circulating supply to zero), the liquidity fees from the earlier sells remain in `totalReserves` but the surplus-distribution branch is skipped due to `circ == 0`. The curve adjustment branch in `_previewDeferredMakerState()` also does not fire since `supply == totalSupply` is well above the 95% threshold.

The retained liquidity fees are never distributed to stakers, protocol, or creator via `distributeFees()`. When trading resumes and a buy makes circ non-zero again, the next commit refreshes K from the current state, absorbing the undistributed fees into the new invariant permanently.

Additionally, the surplus recomputation is inherently imprecise. `LastInvariant` is itself a rounded value, and the recovery path applies further rounding at each step — `divWad()` on the supply/circ ratio, `powWad()` for the exponentiation, `fullMulDivUp()` for the implied buffer, and `mulWadUp()` for the BLV component. Each layer compounds the inaccuracy, meaning the recomputed surplus can understate the actual retained fee even when circ is non-zero.

Recommendation

Track retained liquidity fees in a dedicated accumulator (e.g. `pool.pendingSurplus`) at the time they are charged in `_updateAccounting()`. In `_commitDeferredMakerState()`, distribute the accumulated value directly via `distributeFees()` and reset the accumulator. This eliminates the dependency on curve math for fee recovery and avoids both the `circ == 0` edge case and the cumulative rounding inaccuracy.

Resolution

Baseline: The issue was resolved in PR#172.

L-02 | Buffer Threshold Precision Mismatch

Category	Severity	Location	Status
Math	Low	MakerLib.sol: 313	Resolved

Description

The `BUFFER_SAFETY_THRESHOLD` check uses native bToken precision in `_updateAccounting()` and `_commitDeferredMakerState()`, but wad (18-decimal) precision in `_previewDeferredMakerState()`.

`_updateAccounting()` and `_commitDeferredMakerState()` compare raw `pool.totalBTokens` against `pool.totalSupply.mulWad(BUFFER_SAFETY_THRESHOLD)`. `_previewDeferredMakerState()` compares `currentParams.supply` against `currentParams.totalSupply.mulWad(BUFFER_SAFETY_THRESHOLD)`, where both operands were normalized to 18 decimals via `NormalizeLib.normalizeWad()`.

For bTokens with fewer than 18 decimals, `mulWad()` truncation at native precision produces a strictly lower threshold than the wad-precision check. At the boundary where `totalBTokens == floor(totalSupply * 0.95e18 / 1e18)`, the native `>=` check in `_commitDeferredMakerState()` evaluates to true and the wad `<` check in `_previewDeferredMakerState()` also evaluates to true. Both branches in `_commitDeferredMakerState()` execute despite being mutually exclusive. When both branches fire, the curve-adjustment branch in `_previewDeferredMakerState()` runs first, recomputing the invariant K using `currentParams.reserves` which includes any liquidity fees added to `totalReserves` by `_updateAccounting()`. The new K absorbs these fees. Then the surplus-distribution branch in `_commitDeferredMakerState()` reads the freshly stored K, computes `impliedReserves`, and finds no surplus because K already accounts for the fee.

The result is that liquidity fees collected during swaps below the threshold get permanently locked into the curve invariant rather than being distributed to stakers, protocol, and creator via `distributeFees()`. In correct behavior (safety region, K frozen), the old K would not account for the fee, yielding a distributable surplus.

This affects pools with `bTokenDecimals < 18` when `totalBTokens` lands exactly at the native threshold boundary.

Recommendation

Use the same precision for all three threshold checks. Make `_previewDeferredMakerState()` use native precision, matching `_updateAccounting()` and `_commitDeferredMakerState()`. Read the pool state directly instead of relying on the wad-normalized `currentParams`:

```
    State.Pool storage pool = State.pool(_bToken);
    if (uint256(pool.totalBTokens) < uint256(pool.totalSupply).mulWad(BUFFER_SAFETY_THRESHOLD)) {
```

Resolution

Baseline: The issue was resolved in commit 995abfc.

L-03 | Fee Rounding Favors Buyer Over Protocol

Category	Severity	Location	Status
Rounding	Low	CurveLib.sol: 167	Resolved

Description

In `_computeFee()`, `marginalPremium` is computed using floor rounding throughout — `fullMulDiv()` for the outer division, and `mulWad()` for both the numerator (`convexityExp * totalSupply`) and denominator (`c1 * x1`) — regardless of trade direction:

```
    uint256 marginalPremium = FixedPointMathLib.fullMulDiv(
    _newBuffer,
    _p.convexityExp.mulWad(_p.totalSupply),
    c1.mulWad(x1)
    );
```

For sells this is partially correct — floor rounding on the numerator `mulWad()` and outer `fullMulDiv()` understates `marginalReceipt`, overstates the fee, and the protocol captures the dust upfront. However, the denominator `c1.mulWad(x1)` also floors, which makes the fraction larger and partially counteracts the understatement — ideally it should use `mulWadUp()` for sells to consistently round down.

For buys, the uniform floor rounding understates `marginalCost`, which understates the fee. Since `userDelta_ = invariantDelta_ - fee_`, the understated fee makes `userDelta_` less negative — the buyer pays less while fee recipients receive less.

The comment claims "~1 WAD wei dust stays in pool reserves rather than being extracted as fee," but this is inaccurate:

1. For buys, the dust leaks to the user, not to the pool.

2. The pool's reserves are determined by `invariantDelta_` (fixed by ceiled `newBuffer`), so they are unaffected by fee rounding in either direction.

Additionally, the block pricing mechanism in `BlockPricingLib` differences two cumulative `computeSwap()` calls. If the first buy's cumulative endpoint rounds down, the second buy's baseline shifts, causing it to overpay. With floor rounding, if no subsequent swap occurs, the dust from the understated fee is simply never collected. With ceil rounding, the protocol would have already captured it from the first buyer.

Recommendation

Round `marginalPremium` up for buys and down for sells, applying consistent rounding to all intermediate operations:

```
    uint256 marginalPremium;
    if (_deltaCirc > 0) {
    // BUY: round up to not understate fee
    ...
    );
    }
```

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/master/src/libraries/CurveLib.sol#L165-L184.

L-04 | Stale BLV In BCredit And quoteLeverage

Category	Severity	Location	Status
Logical Error	Low	BCredit.sol: 456 BLens.sol	Acknowledged

Description

`BCredit` and `BLens.quoteLeverage()` read `blvPrice` directly from committed storage via `State.maker(_bToken).blvPrice` instead of previewing the deferred state through `MakerLib.getCurveParams()`. The committed `blvPrice` is only updated when `_commitDeferredMakerState()` runs, which is triggered by swaps via `_advanceBlockPricing()`. In a new block with uncommitted state from the previous block's trades, any call to these functions before a swap sees a stale BLV.

In `BCredit`, three functions are affected: `getBorrowForCollateral()`, `getMaxBorrow()`, and `_previewBorrow()`. All compute max debt capacity as `blv.mulWad(collateralWad)`. Since BLV is monotonically increasing, the stale value is always lower, meaning users receive less borrowing power than they should.

`BLens.quoteLeverage()` has an additional inconsistency: it calls `activePrice(_bToken)` which correctly previews the deferred state through `MakerLib.getCurveParams()`, but then subtracts the stale `maker.blvPrice` from it. The expression `activePrice(_bToken) - maker.blvPrice` mixes a live value with a stale one. On top of that, it calls `BCredit.getBorrowForCollateral()` which itself uses the stale BLV, so `borrowAmount` is already undervalued before the leverage math begins.

The inaccuracy could cause failed transactions when a user acts on a stale quote that underestimates their capacity.

Recommendation

Read BLV through `MakerLib.getCurveParams()` which previews the deferred state. In `BCredit`, replace `State.maker(_bToken).blvPrice` with `MakerLib.getCurveParams(_bToken).BLV` in `getBorrowForCollateral()`, `getMaxBorrow()`, and `_previewBorrow()`. In `BLens.quoteLeverage()`, replace the direct `State.maker(_bToken)` read with the curve params from `getCurveParams()` to keep BLV consistent with the already-live `activePrice()`.

Resolution

Baseline: Acknowledged.

L-05 | computeNextBLV() Overflow DOSes Pool

Category	Severity	Location	Status
DoS	Low	CurveLib.sol: 315	Resolved

Description

In `computeNextBLV()`, the `penaltyDenom` calculation squares an intermediate value:

```
    uint256 penaltyDenom = FixedPointMathLib.fullMulDiv(
    FixedPointMathLib.fullMulDiv(_params.supply, prevCirc, 1e18),
    FixedPointMathLib.fullMulDiv(_params.supply, prevCirc, 1e18),
    1e18
    );
```

This computes `(supply * prevCirc / 1e18)² / 1e18`. When both `supply` and `prevCirc` are large enough (token supplies exceeding ~1e33 at 18 decimals), the result overflows `uint256` and `fullMulDiv()` reverts with `FullMulDivFailed()`.

Since `computeNextBLV()` is called during `_commitDeferredMakerState()`, which runs inside `_advanceBlockPricing()` at the start of every swap in a new block, the revert permanently bricks the pool — pending state from the previous block can never be committed, and all subsequent swaps revert.

Recommendation

If the use case allows, cap the accepted token `totalSupply` to `type(uint108).max` in `setBTokenDeployment()` or `createPool()`. With uint108-bounded values, `penaltyDenom` stays within `2^252`, safely below `type(uint256).max`.

Additionally, `penaltyNumer` is currently rounded down via `fullMulDiv()`, which makes the `penalty` smaller than the exact value. Since `maxBLV = bookPrice - penalty`, a smaller penalty allows a higher `maxBLV`, letting BLV rise slightly past the K-preserving threshold. For correctness, `penaltyNumer` should be rounded up (using `fullMulDivUp()`) to maximize the penalty and keep `maxBLV` conservative. Note that rounding up the intermediate calculations increases the overflow risk at the boundary, which is another reason to reduce the accepted supply range from uint128 to uint108.

Resolution

Baseline:
https://github.com/0xBaseline/mercury/blob/52469512105af39cf3d1ab810b01c8ced1bacbc5/src/components/BFactory.sol#L253.

I-01 | Integration Note Completeness

Category	Severity	Location	Status
Documentation	Info	BStaking.sol: 120-121	Resolved

Description

The following note was added to `BStaking.claim()` in response to I-01

```
    /// @dev Since this function is permissionless all integrating contracts need to handle
    ///      native token transfers or risk losing funds.
```

However, it goes both ways. A contract that works only with native tokens can have its rewards claimed with `asNative = false`, resulting in stuck rewards.

Recommendation

Modify the note to mention both types of rewards.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/1ad34294b8590506cec9dc3cd571e3e7d74e3302/src/components/BStaking.sol#L120.

Round Five Findings & Resolutions

ID	Title	Category	Severity	Status
L-01	Safety Threshold Precision Mismatch	Rounding	Low	Resolved
L-02	View Reserves Inflated By Unstripped Surplus	Unexpected Behavior	Low	Resolved
I-01	Quote Slippage Is Cumulative, Not Per-trade	Informational	Info	Resolved

L-01 | Safety Threshold Precision Mismatch

Category	Severity	Location	Status
Rounding	Low	MakerLib.sol: 621	Resolved

Description

Similar to Buffer threshold precision mismatch, MakerLib._getCommittedCurveParams() uses WAD precision when calculating whether the pool is in safety regime in order to add the pending surplus.

```
    if (
    pool.pendingSurplus > 0 &&
    params_.supply < params_.totalSupply.mulWad(BUFFER_SAFETY_THRESHOLD)
    ...
    );
    }
```

This is different from the checks in the rest of the codebase - all of them use native token decimals. It's possible for `totalBTokens` to be treated as "in safety" by the native check, while the WAD check considers it "below safety."

This causes `_getCommittedCurveParams()` to add `pendingSurplus` to `params_.reserves`, inflating the curve params passed to `_previewDeferredMakerState()` and `getCurveParams()`. The inflated reserves view leaks into `getCurveParams()` and `getSnapshotCurveParams()`, which could produce incorrect quotes, potentially also impacting swaps through `BSwap` solver paths.

Recommendation

Use the native tokens decimal precision.

```
    if (
    pool.pendingSurplus > 0 &&
    •           params_.supply < params_.totalSupply.mulWad(BUFFER_SAFETY_THRESHOLD)
    ...
    );
    }
```

Resolution

Baseline: The issue was resolved in PR#696c458286b7a7bd785825325ef476a49cd84079.

L-02 | View Reserves Inflated By Unstripped Surplus

Category	Severity	Location	Status
Unexpected Behavior	Low	MakerLib.sol: 598	Resolved

Description

`_getCommittedCurveParams()` returns live `totalReserves` without simulating the batching-asymmetry surplus stripping that `_commitDeferredMakerState()` performs at L301-321. When both buys and sells occur in the same block while in safety (pooled >= 95%), the independent block-batched pricing creates a reserve surplus due to curve convexity — actual `totalReserves` exceed the K-implied reserves. This surplus is stripped at commit time, but view functions that read `_getCommittedCurveParams()` see the inflated value.

Additionally, the pendingSurplus injection at L619-626 runs unconditionally for any call where the pool is below safety, including intra-block calls. That means getCurveParams() previews liquidity fees rolling into reserves mid-block even though the actual rollover happens only at commit time, and the final safety state may differ by then.

Affected downstream consumers:

- `getCurveParams()` (L555) — returns inflated reserves in both same-block and cross-block paths
- `activePrice()` via `BLens` (L316) — computes price from inflated reserves
- `swapTokens()` Swap event (L148-152) — emits `activePrice` from `getCurveParams()` with inflated reserves
- `_previewBlockPricing()` cross-block path (L507-510) — uses inflated `committed.reserves` via `curveParamsFromDeferredState()`

Same-block swap pricing and quoting are not affected because `_previewBlockPricing()` uses the frozen snapshot via `applyPoolSnapshot()` (L501), which reads `startReserves/startSupply` rather than committed reserves.

Recommendation

Limit reserve adjustments in `_getCommittedCurveParams()` to the cross-block window where committed state is fully consistent (K, reserves, supply all from the same commit point). Add batching-asymmetry surplus stripping for the in-safety case, and move `pendingSurplus` injection into the same cross-block guard:

```
    function _getCommittedCurveParams(BToken _bToken) private view returns (CurveParams memory params_) {
    State.Pool storage pool = State.pool(_bToken);
    State.Maker storage maker = State.maker(_bToken);
    ...
    +    }
    }
```

This scopes adjustments to cross-block views only. Intra-block, `getCurveParams().reserves` still reflects raw `totalReserves` including any surplus accumulated so far, since stripping mid-block would mix stale K with live reserves. Actual swap pricing is unaffected (frozen snapshot), and the view self-corrects once the next block's commit runs.

Resolution

Baseline: The issue was resolved in commit be30c2b.

I-01 | Quote Slippage Is Cumulative, Not Per-trade

Category	Severity	Location	Status
Informational	Info	BSwap.sol: 232	Resolved

Description

All four quote functions (`quoteBuyExactOut()`, `quoteBuyExactIn()`, `quoteSellExactIn()`, `quoteSellExactOut()`) compute slippage as `(effectivePrice - initialPrice) / initialPrice`, where `initialPrice` is derived from `getSnapshotCurveParams()` — the frozen block-start curve. The `effectivePrice` is the average price the caller would pay, priced on the cumulative curve that accounts for all prior same-block trades. Because `initialPrice` is the block-start snapshot price rather than the marginal price at the caller's cumulative position, the returned slippage measures deviation from the start-of-block price, not the caller's personal price impact. For the first trade in a block both values coincide. For every subsequent trade, the reported slippage includes the price movement caused by earlier same-block trades.

Recommendation

Document that the slippage returned by quote functions represents the cumulative block slippage, not the individual caller's price impact. Integrators who need per-trade slippage should compute the marginal price at the current cumulative position and measure their effective price against that instead.

Resolution

Baseline: https://github.com/0xBaseline/mercury/blob/e821189589e6d3aa2f809df0b638c5bf3b04ab87/src/components/BSwap.sol#L186.

Round Six Findings & Resolutions

ID	Title	Category	Severity	Status
M-01	Missed Removal Leave Hidden Extractable Reserves	Logical Error	Medium	Resolved
M-02	Snapshot Sell Miscalculate Circulation State	Logical Error	Medium	Resolved
M-03	Same-block Terminal Exit Pays Below BLV Floor	Logical Error	Medium	Resolved
I-01	Rebalance Unlocks Surplus From Initial Credit	Informational	Info	Acknowledged

M-01 | Missed Removal Leave Hidden Extractable Reserves

Category	Severity	Location	Status
Logical Error	Medium	MakerLib.sol	Resolved

Description

In the next block commit path for pools at or above the 95% safety threshold, commitDeferredMakerState() is supposed to remove any safety region surplus from pool.totalReserves and reclassify it as fees

```
    CurveParams memory params = _getCommittedCurveParams(_bToken);
    if (params.reserves > impliedReserves) {
    pool.totalReserves -= surplusNative;
    pool.distributeFees(_bToken, surplusNative);
    }
```

But getCommittedCurveParams() is not raw storage. When block pricing is stale and the pool is in safety, it already applies the same skim in memory:

```
    if (params_.reserves > impliedReserves) {
    params_.reserves -= normalizedSurplus;
    }
```

So commitDeferredMakerState() asks for committed params but receives preview-skimmed params. As a result, by the time it checks

```
    if (params.reserves > impliedReserves)
```

The surplus has already been removed from params.reserves in memory, so the storage-side branch often never executes. That means

```
    pool.totalReserves -= surplusNative;
    pool.distributeFees(_bToken, surplusNative);
```

This does not run, and the real surplus remains stranded inside pool.totalReserves.
Exploit flow
1 - A block ends with the pool still in safety and with totalReserves > impliedReserves
2 - On the next block, advanceBlockPricing() calls commitDeferredMakerState()
3 - The commit fails to remove the surplus because getCommittedCurveParams() already skimmed it in memory
4 - BlockPricingLib.initialize() uses that skimmed view, so the new pricing snapshot is based on lower reserves
5 - Storage still keeps the higher real pool.totalReserves
6 - The attacker buys bTokens against the lower skimmed snapshot, so they do not pay for the surplus
7 - They buy enough to move the pool below the 95% threshold
8 - On the following block, the below safety commit no longer applies the safety skim, so the raw higher pool.totalReserves is merged back into live curve state
9 - The attacker sells back on the now surplus backed curve and extracts the previously stranded reserves

Recommendation

https://gist.github.com/GuardianAudits/8cd0ceaee080304ec4c6558cdf3221aa

Resolution

Baseline: Resolved.

M-02 | Snapshot Sell Miscalculate Circulation State

Category	Severity	Location	Status
Logical Error	Medium	BlockPricingLib.sol	Resolved

Description

The pool takes a snapshot at the start of the block of how many tokens are circulating outside the pool And how much reserve is in the pool then, inside the same block If someone buys tokens, that buy is tracked in a buy bucket, If someone sells tokens, that sell is tracked in a sell bucket when the pool later prices a sell, it only looks at the sell bucket, it ignores the earlier buy in the same block

There is a special branch in the code for the case when this is the final sell and circulation became exactly zero

```
    if (c1 == 0) {
    uint256 blvValue = _params.BLV.mulWad(_params.circ);
    uint256 receipt = blvValue.mulWad(WAD - (_params.swapFee*2));
    return (receipt.toInt256(), _params.reserves - receipt, _params.reserves.toInt256());
    }
```

The pool then pay the seller a final exit amount and classify almost all the remaining reserves as fees That logic is only correct if there are truly no tokens left outside the pool But the pool can think, a sell took circulation from 100 down to 0 when the real live state like "circulation was 100, then someone bought 5, so it became 105, and after selling 100 it is still 5" Just for example If circulation is not zero, there are still people outside the pool holding tokens that should still have backing So due the issue that branch can run even though other holders still exist

This could unlock a payout path that should only exist when nobody else is left

Recommendation

https://gist.github.com/GuardianAudits/c752944102d6de66f83020736794dab3

Resolution

Baseline: Resolved.

M-03 | Same-block Terminal Exit Pays Below BLV Floor

Category	Severity	Location	Status
Logical Error	Medium	CurveLib.sol: 87	Resolved

Description

The terminal-exit branch in `CurveLib.computeSwap()` returns a fixed cumulative receipt of $BLV \cdot C_0 \cdot (1 - 2 \cdot swapFee)$ when cumulative sells reach `snapshot.circ`. The inline comment states "User only receives BLV value (floor price), buffer becomes protocol fee" — implying BLV is the guaranteed floor.

In `BlockPricingLib._quoteDeltaFromSnapshot()`, this terminal-exit cumulative is split between in-block sellers via differencing:

```
seller_B_receipt = BLV·C₀·(1-2·swapFee) - seller_A_cumulative
```

Let S be the tokens Seller A already sold. Seller A's cumulative receipt is computed normally on the curve and equals $BLV \cdot S$ + `bufferShareCaptured`, where `bufferShareCaptured` ≥ 0 is the proportional buffer share the curve gives intermediate sellers (always > 0 in any pool with non-zero buffer).

Two failure modes depending on Seller A's cumulative:

• `seller_A_cumulative` > $BLV \cdot S$ (essentially any sell capturing buffer share): residual > 0 but Seller B's per-token receipt drops below BLV (violates floor invariant)

• `seller_A_cumulative` > $BLV \cdot C_0 \cdot (1-2 \cdot swapFee)$ (large first sell): `deltaUserWad` < 0, the denormalize step routes through the buy branch where `denormalizeWadUp(uint256(-deltaInvariantWad), ...)` underflows → tx reverts

POC numbers (BLV=2, C_0=100, B_0=300, swapFee=0.3%, terminal-exit ceiling=198.8):

Scenario	A's cumulative	Outcome
A sells 5 → 38.3	38.3	B sells 95, receives 160.5 (1.69/token, below BLV=2); shortfall ≈28 ether (~96% structural, ~4% fees)
A sells 50 → 249	249	B's tx reverts (direction flip)

The post-fix `_ensureSellWithinSameBlockCapacity` cap (`maxSellDelta = C₀ - S - B`) only addresses the buy-then-terminal-exit case (B > 0). When B = 0, the cap allows cumulative sells to reach C_0 exactly, so both failure modes still trigger.

Impact: any seller pushing cumulative sells to snapshot.circ after another in-block seller will either receive less than the documented BLV floor or have their transaction reverted. That violates the floor invariant and creates a same-block griefing/DoS vector against exit traffic.

Recommendation

In the following gist you will find a recommended code fix, which doesn't apply `swapFee` for the final sell. If keeping the fee is the desired behavior, `_applySellFloor()` can be modified to account for it.

https://gist.github.com/GuardianAudits/c585959c08ed97bc0987f6a48d3424bb

Example implementation:
https://github.com/GuardianOrg/mercury-team1-1768594959614/commit/da92381268010d0dc8b28b41546db00ac41d45e6

Resolution

Baseline: The issue was resolved in PR#178.

I-01 | Rebalance Unlocks Surplus From Initial Credit

Category	Severity	Location	Status
Informational	Info	BCredit.sol: 478	Acknowledged

Description

The new `_rebalanceCollateral()` helper invoked at the end of `_borrow()`, `_repay()`, `leverage()`, and `deleverage()` recomputes locked collateral as `collateral · debt/(maxBorrow+fee)` and unlocks the surplus. This unifies the locking model: locked collateral always equals the minimum required to back the current debt.

A side effect that's easy to miss: when a pool is created with an over-collateralized initial credit position (`initialCollateral > requiredCollateral(initialDebt)` per merkle leaf), the surplus is no longer bonded to the position. After claiming, a user can trigger the rebalance with any state-changing call (e.g. `Borrow(1 wei)`) and immediately withdraw the surplus through `BStaking.withdraw()`, while keeping the cheap debt position open.

Pre-fix, unlocking surplus required actually repaying debt — `_previewRepay` redeems collateral in proportion to repaid debt, so the surplus was implicitly bonded to the debt's lifecycle.

Concrete example: pool created with `initialCollateral=50, initialDebt=5, BLV=2`. After claim, the position is collateral=50, debt=5, locked=50. The minimum required to back debt=5 is `50 · (5 / maxBorrow) ≈ 2.5`. A single `borrow(1)` call triggers rebalance and unlocks ~47.5 of collateral, withdrawable in the same tx.

This may be fine if deployers create over-collateralized initial positions purely as a safety margin / UX default. But if any deployer relies on the over-collateralization as a long-term commitment or "stake bond" against the claimant, the new fix silently invalidates that assumption.

Recommendation

Document that initial credit position collateral is no longer "locked beyond debt requirement". Deployers who want a permanent collateral lock unrelated to the debt position must enforce it through other mechanisms (e.g. an external time-lock, a separate stake position, or by sizing `initialDebt` close to the maxBorrow of `initialCollateral`).

Resolution

Baseline: Acknowledged.

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.

About Guardian

Founded in 2022 by DeFi experts, Guardian is a leading audit firm in the DeFi smart contract space. With every audit report, Guardian upholds best-in-class security while achieving our mission to relentlessly secure DeFi.

To learn more, visit https://guardianaudits.com

To view our audit portfolio, visit https://github.com/guardianaudits

To book an audit, message https://t.me/guardianaudits