



Security Review For Base Dollar



Collaborative Audit Prepared For:
Lead Security Expert(s):

Base Dollar
TIMOH
TessKimy

Date Audited:

June 22 - July 1, 2026

Introduction

Base Dollar is a stablecoin CDP protocol on Base chain. It is a friendly-fork of Liquity V2 with additional features including more collateral types, non-redeemable branches and most notably leveraging Aerodrome LP tokens while earning AERO rewards.

Scope

Repository: basedollar/basedollar

Audited Commit: 7021fce03340c6f42bc587c24749620b16e060b9

Final Commit: fced4598cb8b6c6171457fef4e2e9e1b8843cb5f

Files:

- contracts/script/DeployLiquity2.s.sol
- contracts/src/ActivePool.sol
- contracts/src/AddressesRegistry.sol
- contracts/src/AeroManager.sol
- contracts/src/BoldToken.sol
- contracts/src/BorrowerOperations.sol
- contracts/src/CollateralRegistry.sol
- contracts/src/CollSurplusPool.sol
- contracts/src/DebtInFrontHelper.sol
- contracts/src/DefaultPool.sol
- contracts/src/Dependencies/AddRemoveManagers.sol
- contracts/src/Dependencies/AggregatorV3Interface.sol
- contracts/src/Dependencies/Constants.sol
- contracts/src/Dependencies/FixedPointMathLib.sol
- contracts/src/Dependencies/LiquityBase.sol
- contracts/src/Dependencies/LiquityMath.sol
- contracts/src/Dependencies/Ownable.sol
- contracts/src/ERC20Wrappers/WrappedToken.sol
- contracts/src/GasPool.sol
- contracts/src/HintHelpers.sol
- contracts/src/Interfaces/IActivePool.sol

- `contracts/src/Interfaces/IAddRemoveManagers.sol`
- `contracts/src/Interfaces/IAddressesRegistry.sol`
- `contracts/src/Interfaces/IAeroGauge.sol`
- `contracts/src/Interfaces/IAeroManager.sol`
- `contracts/src/Interfaces/IAeroPool.sol`
- `contracts/src/Interfaces/IAeroVoter.sol`
- `contracts/src/Interfaces/IBoldRewardsReceiver.sol`
- `contracts/src/Interfaces/IBoldToken.sol`
- `contracts/src/Interfaces/IBorrowerOperations.sol`
- `contracts/src/Interfaces/ICollateralRegistry.sol`
- `contracts/src/Interfaces/ICollSurplusPool.sol`
- `contracts/src/Interfaces/ICommunityIssuance.sol`
- `contracts/src/Interfaces/IDebtInFrontHelper.sol`
- `contracts/src/Interfaces/IDefaultPool.sol`
- `contracts/src/Interfaces/IHintHelpers.sol`
- `contracts/src/Interfaces/IInterestRouter.sol`
- `contracts/src/Interfaces/ILiquidityBase.sol`
- `contracts/src/Interfaces/ILQTYStaking.sol`
- `contracts/src/Interfaces/ILQTYToken.sol`
- `contracts/src/Interfaces/IMainnetPriceFeed.sol`
- `contracts/src/Interfaces/IMultiTroveGetter.sol`
- `contracts/src/Interfaces/IPriceFeed.sol`
- `contracts/src/Interfaces/IRedemptionHelper.sol`
- `contracts/src/Interfaces/IRETHPriceFeed.sol`
- `contracts/src/Interfaces/IRETHToken.sol`
- `contracts/src/Interfaces/ISortedTrove.sol`
- `contracts/src/Interfaces/IStabilityPoolEvents.sol`
- `contracts/src/Interfaces/IStabilityPool.sol`
- `contracts/src/Interfaces/ITroveEvents.sol`
- `contracts/src/Interfaces/ITroveManager.sol`
- `contracts/src/Interfaces/ITroveNFT.sol`

- contracts/src/Interfaces/IWETH.sol
- contracts/src/Interfaces/IWrappedToken.sol
- contracts/src/Interfaces/IWSTETHPriceFeed.sol
- contracts/src/Interfaces/IWSTETH.sol
- contracts/src/MultiTroveGetter.sol
- contracts/src/NFTMetadata/MetadataNFT.sol
- contracts/src/NFTMetadata/Utils/baseSVG.sol
- contracts/src/NFTMetadata/Utils/bauhaus.sol
- contracts/src/NFTMetadata/Utils/FixedAssets.sol
- contracts/src/NFTMetadata/Utils/JSON.sol
- contracts/src/NFTMetadata/Utils/SVG.sol
- contracts/src/NFTMetadata/Utils/Utils.sol
- contracts/src/PriceFeeds/AeroLPTokenPriceFeedBase.sol
- contracts/src/PriceFeeds/AeroLPTokenPriceFeed.sol
- contracts/src/PriceFeeds/AEROPriceFeed.sol
- contracts/src/PriceFeeds/cbBTCPriceFeed.sol
- contracts/src/PriceFeeds/cbETHPriceFeed.sol
- contracts/src/PriceFeeds/CompositePriceFeed.sol
- contracts/src/PriceFeeds/MainnetPriceFeedBase.sol
- contracts/src/PriceFeeds/RETHPriceFeed.sol
- contracts/src/PriceFeeds/TokenPriceFeedBase.sol
- contracts/src/PriceFeeds/WETHPriceFeed.sol
- contracts/src/PriceFeeds/WSTETHPriceFeed.sol
- contracts/src/RedemptionHelper.sol
- contracts/src/SortedTrove.sol
- contracts/src/StabilityPool.sol
- contracts/src/TroveManager.sol
- contracts/src/TroveNFT.sol
- contracts/src/Types/BatchId.sol
- contracts/src/Types/LatestBatchData.sol
- contracts/src/Types/LatestTroveData.sol

- contracts/src/Types/TroveChange.sol
- contracts/src/Types/Troveld.sol

Final Commit Hash

[fced4598cb8b6c6171457fef4e2e9e1b8843cb5f](#)

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	5	13

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: Most oracles can't be deployed [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/issues/78>

Summary

MainnetPriceFeedBase.sol inherited by most oracles accepts only 8 decimals, but oracles are meant to be deployed with Api3 which has 18 decimals.

Vulnerability Detail

MainnetPriceFeedBase.sol checks oracle has 8 decimals:

```
constructor(address _ethUsdOracleAddress, uint256 _ethUsdStalenessThreshold,
    ↪ address _borrowOperationsAddress) {
    // Store ETH-USD oracle
    ethUsdOracle.aggregator = AggregatorV3Interface(_ethUsdOracleAddress);
    ethUsdOracle.stalenessThreshold = _ethUsdStalenessThreshold;
    ethUsdOracle.decimals = ethUsdOracle.aggregator.decimals();

    borrowerOperations = IBorrowerOperations(_borrowOperationsAddress);

    assert(ethUsdOracle.decimals == 8);
}
```

Documentation explicitly states that price has 18 decimals:

All Api3 data feeds have 18 decimals. For example, if ETH/USD is 2918.5652133, value will read 2918565213300000000000.

MainnetPriceFeedBase.sol is used by CompositePriceFeed.sol which is used by: RETHPriceFeed, WETHPriceFeed, WSTETHPriceFeed, cbBTCPriceFeed, cbETHPriceFeed.

Deploy script uses Api3 to configure those branches:

```
// Oracle addresses (Base Mainnet - API3)
address ETH_USD_ORACLE_ADDRESS = 0x5b0cf2b36a65a6BB085D501B971e4c102B9Cd473;
address STETH_USD_ORACLE_ADDRESS = address(0); // TODO: Not activated yet
address WSTETH_STETH_ORACLE_ADDRESS = 0xa7C64E79eeee1A4c9B6Ea2976Fa37c276BB1A6cD;
address RETH_ETH_ORACLE_ADDRESS = 0x3Ce8154d55426e8c71F1F0EffDDc6183a92bE45f;
address BTC_USD_ORACLE_ADDRESS = 0xCAC4d304032a46C8D0947396B7cBb07986826A36;
address CBBTC_USD_ORACLE_ADDRESS = 0xa4183Cbf2eE868dDFccd325531C4f53F737FFF68;
address CBETH_ETH_ORACLE_ADDRESS = address(0); // TODO: Not activated yet
address AERO_USD_ORACLE_ADDRESS = 0x201bC62f44f018B70E0c5eC8fF524fB26261959c;
```

Impact

Almost all oracles can't be deployed.

Code Snippet

<https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/blob/main/basedollar/contracts/src/PriceFeeds/MainnetPriceFeedBase.sol#L50>

Tool Used

Manual Review

Recommendation

Accept 18 decimals in `MainnetPriceFeedBase.sol`

Issue M-2: AeroLPTokenPriceFeedBase.getExchangeRates() calculation is mathematically incorrect [RE-SOLVED]

Source: <https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/issues/80>

Summary

Current code assumes each observation period is equal in duration, which makes calculation mathematically wrong.

Vulnerability Detail

Let's take a look on how it calculates TWAP price from observations. It calculates average price for each period, then calculates final average price from all periods:

```
function getExchangeRates(uint256 amountIn, uint256 granularity) public view
↳ returns (uint256, uint256) {
    uint256 amount0In = amountIn * 10 ** token0PoolDecimals;
    uint256 amount1In = amountIn * 10 ** token1PoolDecimals;
    (uint256[] memory _prices0, uint256[] memory _prices1) = _sample(amount0In,
↳ amount1In, granularity);
    uint256 priceAverageCumulative0;
    uint256 priceAverageCumulative1;
    uint256 _length = _prices0.length;
    for (uint256 i = 0; i < _length; i++) {
@>         priceAverageCumulative0 += _prices0[i];
@>         priceAverageCumulative1 += _prices1[i];
    }
@>     return (priceAverageCumulative0 / granularity, priceAverageCumulative1 /
↳ granularity);
}

function _sample(
    uint256 amount0In,
    uint256 amount1In,
    uint256 points
) internal view returns (uint256[] memory, uint256[] memory) {
    uint256[] memory _prices0 = new uint256[](points);
    uint256[] memory _prices1 = new uint256[](points);

    uint256 length = pool.observationLength() - 1;
    uint256 i = length - points;
    uint256 index = 0;
```



```

        for (; i < length; i += 1) {
            IAeroPool.Observation memory nextObs = pool.observations(i + 1);
            IAeroPool.Observation memory currentObs = pool.observations(i);
            uint256 timeElapsed = nextObs.timestamp - currentObs.timestamp;
@>            uint256 _reserve0 = (nextObs.reserve0Cumulative -
↪ currentObs.reserve0Cumulative) /
@>                timeElapsed;
@>            uint256 _reserve1 = (nextObs.reserve1Cumulative -
↪ currentObs.reserve1Cumulative) /
@>                timeElapsed;
            _prices0[index] = _getAmountOut(amount0In, pool.token0(), _reserve0,
↪ _reserve1);
            _prices1[index] = _getAmountOut(amount1In, pool.token1(), _reserve0,
↪ _reserve1);
            // index < length; length cannot overflow
            unchecked {
                index = index + 1;
            }
        }
        return (_prices0, _prices1);
    }
}

```

During second time it just divides by granularity, i.e. assumes that all periods have same duration. But that is wrong, pool only guarantees same minimum duration for each period (30 minutes).

Simple example: 120 min period with price 100 and 12 min period with price 200. Current code calculates price 150, but it's obviously incorrect.

Impact

TWAP calculation is not correct and it doesn't correctly weight prices based on elapsed time. It also causes unbounded time for twap calculation because we don't know how regular the pool is called. It's originally configured to calculate TWAP for 4 hours but it can even calculate with last 12 hours. Impact depends on activity of the pool.

Code Snippet

<https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/blob/main/basedollar/contracts/src/PriceFeeds/AeroLPTokenPriceFeedBase.sol#L169>

Tool Used

Manual Review

Recommendation

Consider changing calculation so it uses formula $\text{price} = \frac{\text{sum}(\text{priceMultipliedByItsTimeElapsed})}{\text{totalTimeElapsed}}$

Discussion

geovgy

Interesting finding. The functions `getExchangeRates` and `_sample` follow the same calculations as `quote` ([source](#)) and `sample` ([source](#)) from Aerodrome. Only differences being slippage removed in `_getAmountOut` and calculating for both tokens rather than just one for efficiency.

The recommended change makes sense. However, would this change open up an exploitable price mismatch between Base Dollar and Aerodrome given this is the current way they calculate it?

TIMOH593

@geovgy From what I see, `quote()` and `sample()` are view helpers which are not used by Aerodrome on-chain. So there is no impact from Aerodrome side.

@DemoreXTess please recheck

DemoreXTess

Rechecked. Finding is still valid. Aerodrome's sample works points based rather than time based. That's why it doesn't match with our intention.

We can check timestamps of points and stop for loop when we reach our intended window.

I think that's better because in current version we don't know how much time we used to calculate TWAP.

geovgy

Ok got it. Thanks.

Issue M-3: cbBTC branch can't be used [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/issues/83>

Summary

cbBTC branch is deployed with cbBTC token which has 8 decimals. However Liquity assumes collateral has 18 decimals.

Vulnerability Detail

Deploy script `DeployLiquity2.s.sol` specifies cbBTC address and passes as collateral:

```
// Collateral token addresses (Base Mainnet)
address CBBTC_ADDRESS = 0xcbB7C0000aB88B473b1f5aFd9ef808440eed33Bf;
...
// cbBTC
vars.collaterals[3] = IERC20Metadata(CBBTC_ADDRESS);
```

Liquity has assumption that collateral has 18 decimals, so it messes up internal calculations. For example it underestimates CR by $1e10$ times.

Impact

cbBTC branch can't be used, `openTrove()` always reverts. That's because protocol calculates CR $1e10$ times lower than it should because of 10 decimals difference.

Code Snippet

<https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/blob/main/basedollar/contracts/script/DeployLiquity2.s.sol#L95>

Tool Used

Manual Review

Recommendation

Use 18 decimals wrapper over cbBTC.

Issue M-4: AeroLPTokenPriceFeed redemption price selection does not actually maximize LP value [RE-SOLVED]

Source: <https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/issues/85>

Summary

`_selectPrices` tries to maximize the LP token value during redemptions by taking the max of one token price and the min of the other. The LP price formula depends on both token prices together, so pushing one price up and the other down does not maximize the LP value. The intended protection against value leakage on redemptions is not achieved by this choice.

Vulnerability Detail

For redemptions within the deviation threshold the code maximizes `token1` and minimizes `token0`.

```
if (_isRedemption && withinThreshold) {
    // For redemptions within threshold: maximize LP value to prevent value leakage
    // max token1 price, min token0 price -> higher LP value
    token1Price = LiquidityMath._max(token1MarketPrice, token1OraclePrice);
    token0Price = LiquidityMath._min(token0MarketPrice, token0OraclePrice);
} else {
    // For borrows (or redemptions outside threshold): minimize LP value
    token1Price = LiquidityMath._min(token1MarketPrice, token1OraclePrice);
    token0Price = LiquidityMath._max(token0MarketPrice, token0OraclePrice);
}
```

These prices feed the LP price formula. For a volatile pair the price is

```
uint256 p = FixedPointMathLib.sqrt(price0 * 1e16 * price1);
return (2 * p * k) / (1e8 * total_supply);
```

The LP price scales with the square root of `price0` times `price1`. The stable pair formula is also symmetric and increasing in both token prices. Because the value depends on the product of the two prices, raising `token1` while lowering `token0` can leave the product roughly unchanged or even lower it. To actually maximize the LP value within the allowed price set you would take the higher option for both `token0` and `token1`. The comment that maxing one and minning the other gives a higher LP value does not hold for this formula, so the redemption branch does not produce the maximal LP value it intends to.

Impact

On redemptions within the deviation threshold the LP price used can be lower than the true maximum of the allowed price range. Since the stated purpose of the redemption branch is to maximize LP value to prevent value leakage, the chosen prices can let a redeemer receive collateral valued below the intended conservative maximum, which is a value leakage on redemptions against LP collateral. The size of the effect is bounded by the deviation threshold.

Recommendation

For the redemption case select the higher of market and oracle price for both token0 and token1 so the LP value is genuinely maximized, and for the borrow case select the lower of both. This matches the symmetric way both token prices enter the LP price formula.

Issue M-5: RETHPriceFeed doesn't work and can't be deployed [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/issues/95>

Summary

On Base chain RETH is bridged token, not Rocket Pool itself. RETHPriceFeed calls function `reth.getExchangeRate()`, but there is no such function in bridged token. Deployment requires working feed, but it doesn't work.

Vulnerability Detail

```
constructor(
    address _ethUsdOracleAddress,
    address _rEthEthOracleAddress,
    address _rEthTokenAddress,
    uint256 _ethUsdStalenessThreshold,
    uint256 _rEthEthStalenessThreshold,
    address _borrowerOperationsAddress
)
    CompositePriceFeed(_ethUsdOracleAddress, _rEthTokenAddress,
        ↪ _ethUsdStalenessThreshold, _borrowerOperationsAddress)
{
    ...
@>    _fetchPricePrimary(false);
    ...
}

function _fetchPricePrimary(bool _isRedemption) internal override returns
    ↪ (uint256, bool) {
    assert(priceSource == PriceSource.primary);
    (uint256 ethUsdPrice, bool ethUsdOracleDown) =
        ↪ _getOracleAnswer(ethUsdOracle);
    (uint256 rEthEthPrice, bool rEthEthOracleDown) =
        ↪ _getOracleAnswer(rEthEthOracle);
@>    (uint256 ethPerReth, bool exchangeRateIsDown) = _getCanonicalRate();
    ...
}

function _getCanonicalRate() internal view override returns (uint256, bool) {
    uint256 gasBefore = gasleft();

@>    try IRETHToken(rateProviderAddress).getExchangeRate() returns (uint256
    ↪ ethPerReth) {
```

```

        // If rate is 0, return true
        if (ethPerReth == 0) return (0, true);

        return (ethPerReth, false);
    } catch {
        // Require that enough gas was provided to prevent an OOG revert in the
        ↪ external call
        // causing a shutdown. Instead, just revert. Slightly conservative, as
        ↪ it includes gas used
        // in the check itself.
        if (gasleft() <= gasBefore / 64) revert
            ↪ InsufficientGasForExternalCall();

        // If call to exchange rate reverts, return true
        return (0, true);
    }
}

```

According to deploy script, RETH is bridged token:

```
address RETH_ADDRESS = 0xB6fe221Fe9EeF5aBa221c348bA20A1Bf5e73624c;
```

<https://basescan.org/address/0xB6fe221Fe9EeF5aBa221c348bA20A1Bf5e73624c#readContract>

Impact

RETHPriceFeed always fallbacks to 0 price, basically RETH can't be priced. This check is triggered in constructor, so it can't be deployed.

Code Snippet

<https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/blob/main/basedollar/contracts/src/PriceFeeds/RETHPriceFeed.sol#L93-L106>

Tool Used

Manual Review

Recommendation

Looks like there is no way to get canonical rate on Base, so consider other ways to price.

Issue L-1: Incorrect comment in AeroLPTokenPriceFeedBase [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/issues/79>

Vulnerability detail

Comment takes 4th root only in numerator, while code takes root on final result, i.e. both numerator and denominator

```
//fair_reserves = ( (k * (price0 ** 3) * (price1 ** 3)) )^(1/4) / ((price0 ** 2) +  
↪ (price1 ** 2));
```

Code Snippet

<https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/blob/main/basedollar/contracts/src/PriceFeeds/AeroLPTokenPriceFeedBase.sol#L339>

Tool Used

Manual Review

Recommendation

Update to `//fair_reserves = ( (k * (price0 ** 3) * (price1 ** 3)) / ((price0 ** 2) + (price1 ** 2)) )^(1/4);`

Issue L-2: Fresh pool can't be used by AeroLPTokenPriceFeedBase [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/issues/81>

This issue has been acknowledged by the team but won't be fixed at this time.

Vulnerability Detail

It fetches observations from the last 4 hours. It will revert in case there is not enough observations in pool:

```
uint256 length = pool.observationLength() - 1;
uint256 i = length - points;
uint256 index = 0;

for (; i < length; i += 1) {
    IAeroPool.Observation memory nextObs = pool.observations(i + 1);
    IAeroPool.Observation memory currentObs = pool.observations(i);
```

Code Snippet

<https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/blob/main/basedollar/contracts/src/PriceFeeds/AeroLPTokenPriceFeedBase.sol#L194-L200>

Tool Used

Manual Review

Recommendation

Acknowledge, there is nothing to do with it.

Issue L-3: Governor should close and claim from old gauges before updating aeroTokenAddress in AeroManager [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/issues/82>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Old rewards can't be claimed after updating aeroTokenAddress, that's why admin should make sure all are claimed before performing update.

Vulnerability Detail

Claim reverts if there is different reward token. After update aeroTokenAddress points to new version, while old gauges point to old version - so claim for old gauges reverts:

```
function claim(address gauge) external nonReentrant {
    require(_registeredGauges[gauge], "AeroManager: Gauge is not registered");
    uint256 currentEpoch = currentEpochs[gauge];
    require(!epochClosed[gauge][currentEpoch], "AeroManager: Current epoch is
    ↪ already closed");
    @> require(IAeroGauge(gauge).rewardToken() == aeroTokenAddress, "AeroManager:
    ↪ Reward token does not match");
```

Impact

Reward from old gauges can't be claimed after aeroTokenAddress update.

Code Snippet

<https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/blob/main/basedollar/contracts/src/AeroManager.sol#L307-L311>

Tool Used

Manual Review

Recommendation

Make sure to claim all rewards before the update.

Issue L-4: AeroManager claim fee decreases take effect immediately with no delay [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/issues/84>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

updateClaimFee applies a delay only when the fee is raised. A fee decrease is written to storage right away with no timelock. This is an asymmetry in the fee change controls and lets the governor lower the treasury cut instantly while raising it is delayed.

Vulnerability Details

```
function updateClaimFee(uint256 newFee) external onlyGovernor {
    require(newFee != claimFee, "AeroManager: New fee is the same as the current
    ↪ fee");
    _requireClaimFeeLimit(newFee);
    if (newFee > claimFee) {
        pendingNewClaimFee = newFee;
        pendingNewClaimFeeTimestamp = block.timestamp;
        emit ClaimFeeUpdatePending(claimFee, newFee, pendingNewClaimFeeTimestamp,
        ↪ claimFeeChangeDelayPeriod);
    } else {
        if (pendingNewClaimFee > 0) {
            _cancelClaimFeeUpdate();
        }
        uint256 oldFee = claimFee;
        claimFee = newFee; // immediate
        emit ClaimFeeUpdated(oldFee, newFee);
    }
}
```

When newFee is greater than the current fee the change is staged as pending and must wait claimFeeChangeDelayPeriod before acceptClaimFeeUpdate can finalize it. When newFee is lower the else branch sets claimFee directly with no delay. So a decrease is effective on the next claim immediately.

A decrease lowers the treasury cut and increases the amount that stays for distribution to borrowers, so a decrease is the more borrower friendly direction. Still the lack of any delay on one side is an inconsistent control surface, and combined with the fact that the fee at claim time applies to already accrued rewards, an immediate decrease right before a large claim changes the split for rewards that were accruing under the higher fee.

Impact

The governor can instantly reduce the treasury fee and, because the fee is applied at claim time to the full accrued amount, can change the treasury versus borrower split for already accrued rewards without any waiting period. There is no loss to borrowers from a decrease but the timelock protection that exists for increases does not exist for decreases.

Recommendation

Apply the same delay period to decreases as to increases so every fee change is staged and timelocked, or document that decreases are intentionally immediate and accept the asymmetry.

Issue L-5: TroveManager state variable initialDebtLimit is misspelled [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/issues/86>

Summary

The public state variable initialDebtLimit is missing a letter and should be initialDebtLimit. Because it is public it generates an auto getter with the misspelled name, so the typo is part of the external interface and anything reading it must use the misspelled name.

```
uint256 public debtLimit;  
uint256 public initialDebtLimit; // should be initialDebtLimit
```

The variable is declared public so the compiler creates a getter named initialDebtLimit. Off chain code, scripts, frontends and any other contract that reads this value have to use the misspelled identifier. Renaming it later is a breaking change to the ABI surface, so it is best fixed before deployment.

Impact

No effect on contract behavior or funds. It is a naming defect that leaks into the external interface and can cause confusion or integration mistakes for anyone reading the value.

Recommendation

Rename initialDebtLimit to initialDebtLimit and update all references before deployment so the public getter has the correct name.

Issue L-6: AeroManager claim fee changes apply to rewards that were already earned in the gauge [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/issues/87>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The claim fee is read at the moment claim is called, not at the moment the rewards accrued in the gauge. A fee change therefore applies to AERO that was already earned but not yet pulled in, so the fee taken on a claim can differ from the fee that was in effect while those rewards were accruing.

Vulnerability Details

The fee is applied inside claim using the current claimFee value.

```
uint256 _claimFee = _getClaimFee(claimedAmount);
IERC20(aeroTokenAddress).safeTransfer(treasuryAddress, _claimFee);
```

```
function _getClaimFee(uint256 amount) internal view returns (uint256) {
    return amount * claimFee / _100pct;
}
```

claimedAmount is the AERO accrued in the gauge since the last claim. That amount can represent a period during which the fee was different. When the fee is changed through updateClaimFee and acceptClaimFeeUpdate the new fee is taken on the next claim against the full accrued balance, including the part that accrued before the change. So rewards earned under an old fee are charged at the new fee whenever the claim happens after the change.

The code comment acknowledges this and notes it is hard to fix because rewards accrue continuously across many gauges and there is no per second fee accounting.

Impact

Borrowers can receive a different net reward than the fee in effect while their rewards accrued would imply, because the fee at claim time is what is charged on the whole accrued amount. The direction of the effect depends on whether the fee was raised or lowered before the claim. There is no loss beyond this timing mismatch and the fee is still capped by MAX_AERO_MANAGER_FEE.

Recommendation

If exact fairness is required, claim and snapshot rewards from each gauge right before any fee change takes effect so the old fee applies to rewards accrued under it. If that is too heavy to maintain across many gauges, document clearly that the fee applies at claim time to the full accrued amount so the behavior is expected.

Issue L-7: AeroManager closeCurrentEpoch has no cooldown between epoch closes [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/issues/88>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

closeCurrentEpoch lets the governor seal the current epoch at any time. There is no minimum time between epoch closes so the governor can close epochs in quick succession. A cooldown would give the system a predictable reward accounting window and reduce the chance of closing an epoch before rewards for that period have been claimed.

Vulnerability Details

```
function closeCurrentEpoch(address gauge) external onlyGovernor {
    uint256 currentEpoch = currentEpochs[gauge];
    require(!epochClosed[gauge][currentEpoch], "AeroManager: Current epoch is
    ↪ already closed");
    epochClosed[gauge][currentEpoch] = true;
    emit EpochClosed(gauge, currentEpoch);
}
```

Once an epoch is closed claim can no longer add rewards to that epoch because claim requires the current epoch to be open.

```
function claim(address gauge) external nonReentrant {
    require(_registeredGauges[gauge], "AeroManager: Gauge is not registered");
    uint256 currentEpoch = currentEpochs[gauge];
    require(!epochClosed[gauge][currentEpoch], "AeroManager: Current epoch is
    ↪ already closed");
    ...
}
```

The epoch only advances later inside distributeAero. Because closeCurrentEpoch can be called whenever the governor wants there is no enforced spacing between epochs. If an epoch is closed earlier than expected the rewards that accrued in the gauge but were not yet pulled in with claim for that epoch will land in the next epoch instead. A cooldown that ties epoch closing to a known cadence would make the timing of reward buckets more predictable and harder to get wrong.

Impact

No direct loss of funds. The governor controls the timing of reward buckets with no enforced spacing, so an epoch can be closed at an unexpected time and rewards that were not yet claimed for that epoch shift into the next one. This is an operational and accounting clarity concern.

Recommendation

Track the timestamp of the last epoch close per gauge and require a minimum cooldown to pass before the next close. This gives each epoch a predictable minimum length.

Issue L-8: AeroManager distributeAero pays out the whole epoch in a single transaction [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/issues/89>

Summary

distributeAero credits every recipient of a closed epoch in one transaction and requires the full reward amount to be allocated in that same call. When a gauge has many borrowers the recipients array can become large enough that the call runs into the block gas limit, which blocks distribution and the epoch advance for that gauge. This is already a known issue but fix is also very simple and affordable.

Vulnerability Details

```
function distributeAero(address gauge, AeroRecipient[] memory recipients) external
↳ onlyGovernor {
    uint256 currentEpoch = currentEpochs[gauge];
    require(epochClosed[gauge][currentEpoch], "AeroManager: Current epoch is not
↳ closed yet to distribute rewards");
    require(recipients.length > 0, "AeroManager: No recipients");

    uint256 rewardAmount = claimedAeroPerEpoch[currentEpoch][gauge];
    uint256 remainingRewardAmount = rewardAmount;
    for (uint256 i; i < recipients.length; i++) {
        require(recipients[i].amount <= remainingRewardAmount, "AeroManager: Total
↳ amount exceeds reward amount");
        remainingRewardAmount -= recipients[i].amount;
        _claimableRewards[recipients[i].borrower][aeroTokenAddress] +=
↳ recipients[i].amount;
    }
    require(remainingRewardAmount == 0, "AeroManager: Reward amount not fully
↳ distributed");
    emit AeroDistributed(gauge, recipients.length, rewardAmount, currentEpoch);
    currentEpoch++;
    currentEpochs[gauge] = currentEpoch;
}
```

Impact

For a gauge with a large borrower set the distribution call can become too expensive to fit in a block. When that happens the closed epoch cannot be distributed and the gauge

epoch cannot advance, which freezes reward distribution for that gauge until the recipient set can somehow be made smaller.

Recommendation

Split distribution into multiple transactions. Allow the governor to credit recipients in batches, track the running total that has been allocated for the epoch, and advance the epoch only in a separate finalize step once the full amount has been distributed.

Issue L-9: AeroLPTokenPriceFeedBase does not check the Chainlink L2 sequencer uptime feed on Base [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/issues/90>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`_isValidChainlinkPrice` validates success, staleness and a positive answer but it does not check the Chainlink L2 sequencer uptime feed. The system is deployed on Base which is an L2. If the Base sequencer has been down and just came back the price feed can accept a price from a feed that has not had a fresh on chain update during the outage.

Vulnerability Details

```
function _isValidChainlinkPrice(ChainlinkResponse memory chainlinkResponse, uint256
↳ _stalenessThreshold)
    internal
    view
    returns (bool)
{
    return chainlinkResponse.success && chainlinkResponse.timestamp +
↳ _stalenessThreshold > block.timestamp
    && chainlinkResponse.answer > 0;
}
```

The check confirms the call succeeded, that the answer is not older than the staleness threshold, and that the answer is positive. On an L2 like Base, Chainlink recommends also reading the sequencer uptime feed and rejecting prices when the sequencer has recently been down, because while the sequencer is down on chain price updates stop and the last answer can become unreliable while still passing a staleness check that uses the L2 block timestamp. This feed validation does not include that sequencer check.

Impact

During and shortly after a Base sequencer outage the price feed can treat a stale or unreliable Chainlink answer as valid. The LP token price and the underlying token prices are used for borrowing, liquidation and redemption, so accepting a price under these conditions can lead to operations executing against an inaccurate collateral value around sequencer downtime.

Recommendation

Add a Chainlink L2 sequencer uptime feed check to the price validation. Read the sequencer feed, require the sequencer to be up, and require that a grace period has passed since it last came back online before accepting Chainlink answers.

Issue L-10: AeroManager assumes every gauge exposes stakingToken [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/issues/91>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`_addActivePool` validates a gauge by calling `stakingToken` on it and comparing the results. This assumes every gauge that will ever be registered exposes this view function and returns the expected address. A future gauge implementation that does not expose this getter.

Vulnerability Details

```
IAeroGauge gauge = IAeroGauge(ap.aeroGaugeAddress());
require(gauge.stakingToken() == address(ap.collToken()), "AeroManager: Staking
↳ token does not match");
require(gauge.rewardToken() == aeroTokenAddress, "AeroManager: Reward token does
↳ not match");
```

This requires call function that must exist on the gauge. If a newer gauge type does not implement `stakingToken` the calls revert and the active pool cannot be added. The check also hard binds the gauge reward token to the current `aeroTokenAddress` at registration time.

Impact

If Aerodrome introduces a gauge type that does not expose this getter in the same shape, the corresponding active pool cannot be registered through `addActivePool` or `setAddresses`, which blocks onboarding that collateral. This is a forward compatibility limitation rather than a vulnerability in the current set of gauges.

Issue L-11: AeroManager claim fee rounds down in favor of the borrower side [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/issues/92>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`_getClaimFee` computes the treasury fee with integer division which rounds down. The dropped remainder stays with the reward pool that is later distributed to borrowers rather than going to the treasury. On each claim the treasury receives slightly less than the exact fee.

Vulnerability Details

```
function _getClaimFee(uint256 amount) internal view returns (uint256) {  
    return amount * claimFee / _100pct;  
}
```

Integer division truncates toward zero so any fractional part of `amount times claimFee` divided by `_100pct` is lost from the fee. In claim the fee is sent to the treasury and the remainder stays for borrower distribution.

```
uint256 _claimFee = _getClaimFee(claimedAmount);  
IERC20(aeroTokenAddress).safeTransfer(treasuryAddress, _claimFee);  
uint256 rewardAmount = claimedAmount - _claimFee;
```

Because `rewardAmount` is `claimedAmount` minus the rounded down fee, the truncated remainder is always added to `rewardAmount`. So rounding always favors the distribution pool and slightly underpays the treasury. The per claim amount is at most one unit of AERO but it happens on every claim across every gauge.

Impact

The treasury receives marginally less than the configured fee on each claim and the borrower distribution pool receives marginally more. The amount per claim is tiny, at most rounding dust, but it is a consistent direction.

Recommendation

If the treasury should never be underpaid, round the fee up instead of down, for example by computing the fee as the ceiling of $\text{amount} \times \text{claimFee} \div 100$. If the dust is acceptable, leave it as is and treat the rounding direction as intended.

Issue L-12: AeroLPTokenPriceFeedBase stable getAmountOut scales the result back down to token decimals [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/issues/93>

Summary

In the stable pair branch of `_getAmountOut` the computation is done at 18 decimals and then the result is scaled back down to the output token decimals before returning. The caller later scales it up to 18 decimals again. For a low decimal token this down then up scaling drops low order digits and loses precision for no reason.

Vulnerability Details

```
if (isStablePair) {
    _reserve0 = _reserve0 * (10 ** (18 - token0PoolDecimals));
    _reserve1 = _reserve1 * (10 ** (18 - token1PoolDecimals));
    (uint256 reserveA, uint256 reserveB) = tokenIn == token0 ? (_reserve0,
        ↪ _reserve1) : (_reserve1, _reserve0);
    amountIn = tokenIn == token0 ? amountIn * (10 ** (18 - token0PoolDecimals)) :
        ↪ amountIn * (10 ** (18 - token1PoolDecimals));
    uint256 y = (amountIn * _d(reserveB, reserveA)) / _d(reserveA, reserveB);
    return (y * 10 ** (tokenIn == token0 ? token1PoolDecimals :
        ↪ token0PoolDecimals)) / 1e18;
    // scaled back down to token decimals here
}
```

Inside the branch reserves and amountIn are scaled up to 18 decimals and y is computed at 18 decimals. The return statement multiplies y by 10 raised to the output token decimals and divides by 1e18, which converts the 18 decimal value back to the output token native decimals. The callers of this function, `getExchangeRates` and `_getTwapExchangeRates`, then multiply the sampled price by 10 raised to 18 minus the token decimals to bring it back to 18 decimals. So the value is reduced to token decimals here and immediately raised back to 18 decimals later. For a token with few decimals such as 6 the intermediate truncation to 6 decimals removes precision that could have been kept by staying at 18 decimals throughout.

Impact

The TWAP based exchange rate loses precision for pools that contain a low decimal token. The lost precision is small but it feeds the LP token price used for borrowing and

redemption, so it slightly reduces the accuracy of the collateral valuation for those pools. There is no direct loss but the price is less precise than it needs to be.

Recommendation

Keep the value at 18 decimals through the stable branch and let the existing caller scaling handle the final decimals, rather than scaling down to token decimals here and back up afterward.

Issue L-13: BorrowerOperations pulls collateral with a raw transferFrom and does not check the return value [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-06-base-dollar-update-june-22nd-2026/issues/94>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`_pullCollAndSendToActivePool` moves collateral from the borrower with a raw `transferFrom` call and ignores the return value. Tokens that do not return a bool, such as USDT, and tokens that return false instead of reverting are not handled safely here. The contract then records the collateral as received whether or not the transfer actually succeeded.

Vulnerability Details

```
function _pullCollAndSendToActivePool(IActivePool _activePool, uint256 _amount)
↳ internal {
    // Send Coll tokens from sender to active pool
    collToken.transferFrom(msg.sender, address(_activePool), _amount);
    // Make sure Active Pool accountancy is right
    _activePool.accountForReceivedColl(_amount);
}
```

`collToken` is a plain `IERC20` and the call uses `transferFrom` directly with no `SafeERC20` wrapper.

```
IERC20 internal immutable collToken;
```

There are two known problems with this pattern. First, some tokens like USDT do not return a bool from `transferFrom`, so a strict `IERC20` call can revert on decoding for those tokens depending on compiler behavior. Second, a token that returns false on failure rather than reverting would let execution continue, and the next line calls `accountForReceivedColl` with the full amount, so the active pool would credit collateral that was not actually received.

The current intended collateral set is WETH, the staked ETH types, cbBTC, cbETH, AERO and AERO LP, which are standard returning tokens, so this is a concern mainly for any future non standard collateral.

Impact

If a non standard ERC20 is ever used as collateral the deposit path can either revert unexpectedly or, for a token that returns false on failure, record collateral that was never received. For the currently planned collateral tokens this does not trigger, so the risk depends on future collateral choices.

Recommendation

Use SafeERC20 safeTransferFrom for the collateral pull so non standard tokens are handled and a failed transfer reverts before accountForReceivedColl runs.

Discussion

geovgy

We originally used SafeERC20 but sadly had to remove it from BorrowerOperations to fit within the contract size limit. This will be part of our due diligence when considering future collateral assets to add.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.