



Security Review For Base Dollar



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

Base Dollar

0x37

KupiaSec

Date Audited:

April 20 - April 27, 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: 74c31887701a5e6c051d5140cad4dd19e2f9dd9b

Final Commit: a1f10be21a692aedb69a69f2dfc48fa8739b9506

Files:

- 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
- contracts/src/Zappers/BaseZapper.sol
- contracts/src/Zappers/GasCompZapper.sol
- contracts/src/Zappers/Interfaces/IExchangeHelpers.sol
- contracts/src/Zappers/Interfaces/IExchangeHelpersV2.sol
- contracts/src/Zappers/Interfaces/IExchange.sol
- contracts/src/Zappers/Interfaces/IFlashLoanProvider.sol
- contracts/src/Zappers/Interfaces/IFlashLoanReceiver.sol
- contracts/src/Zappers/Interfaces/ILeverageZapper.sol
- contracts/src/Zappers/Interfaces/IZapper.sol
- contracts/src/Zappers/LeftoversSweep.sol
- contracts/src/Zappers/WETHZapper.sol
- contracts/src/Zappers/WrappedTokenZapper.sol

Final Commit Hash

a1f10be21a692aedb69a69f2dfc48fa8739b9506

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
1	10	12

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: Incorrect TWAP price used for Aerodrome LP tokens [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/98>

Summary

Incorrect TWAP price used for Aerodrome LP tokens

Vulnerability Detail

In BaseDollar, the AeroDrome LP token's price will be calculated based on oracle price and twap price to prevent the possible manipulation.

The twap price will be fetched via the function `pool.quote`. The problem here is that the return value is not the actual twap price. In Aerodrome, the `quote` is used to calculate the output token amount based on one specific input amount. It means that this is one real swap's simulation. And the swap simulation will include some swap slippage here. For example, in WBTC/USDC pool, the twap price is 70000 USDC/WBTC. If users quote 1 WBTC, the actual output USDC amount will be less than 70000 USDC.

The swap slippage may be manipulated by liquidity holders, especially when one pool's liquidity is not efficient. Liquidity holders can decrease/increase the liquidity to manipulate the slippage for this swap simulation. Then the `token1PerToken0` can be manipulated.

```
function _getTwapExchangeRates() internal view returns (ExchangeRate memory
    ↪ exchangeRate) {
@>    try pool.quote(token0, 10 ** token0PoolDecimals, TWAP_GRANULARITY)
        returns (uint256 amountOut)
    {
        // amountOut is in token1 decimals - scale to 18
        exchangeRate.token1PerToken0 = amountOut * 10 ** (18 -
            ↪ token1PoolDecimals);
    } catch {
        if (gasleft() <= gasBefore / 64) revert
            ↪ InsufficientGasForExternalCall();
        exchangeRate.isDown = true;
        return exchangeRate;
    }
}

function sample(
    address tokenIn,
    uint256 amountIn,
    uint256 points,
    uint256 window // window = 1.
```

```

    ) public view returns (uint256[] memory) {
        uint256[] memory _prices = new uint256[](points);

        uint256 length = observations.length - 1;
        uint256 i = length - (points * window);
        uint256 nextIndex = 0;
        uint256 index = 0;

        for (; i < length; i += window) {
            nextIndex = i + window;
            uint256 timeElapsed = observations[nextIndex].timestamp -
                ↪ observations[i].timestamp;
            uint256 _reserve0 = (observations[nextIndex].reserve0Cumulative -
                ↪ observations[i].reserve0Cumulative) /
                timeElapsed;
            uint256 _reserve1 = (observations[nextIndex].reserve1Cumulative -
                ↪ observations[i].reserve1Cumulative) /
                timeElapsed;
            @> _prices[index] = _getAmountOut(amountIn, tokenIn, _reserve0,
            ↪ _reserve1);
            // index < length; length cannot overflow
            unchecked {
                index = index + 1;
            }
        }
        return _prices;
    }
}

```

Impact

LP token's price can be manipulated by Aerodrome's pool's LP holders.

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb-e4000e612d1c3f070e56dd5c5767f3/basedollar/contracts/src/PriceFeeds/AeroLPTokenPriceFeedBase.sol#L110-L119>

Tool Used

Manual Review

Recommendation

Calculate the twap price via the observation in Aerodrome pool directly.

Discussion

geovgy

Can you elaborate on the recommendation?

From the code snippet, `sample` looks to do exactly that which `quote` then averages out for TWAP price.

0x37-web3

```
function _getAmountOut(
    uint256 amountIn,
    address tokenIn,
    uint256 _reserve0,
    uint256 _reserve1
) internal view returns (uint256) {
    if (stable) {
        uint256 xy = _k(_reserve0, _reserve1);
        _reserve0 = (_reserve0 * 1e18) / decimals0;
        _reserve1 = (_reserve1 * 1e18) / decimals1;
        (uint256 reserveA, uint256 reserveB) = tokenIn == token0 ? (_reserve0,
            ↪ _reserve1) : (_reserve1, _reserve0);
        amountIn = tokenIn == token0 ? (amountIn * 1e18) / decimals0 : (amountIn *
            ↪ 1e18) / decimals1;
        uint256 y = reserveB - _get_y(amountIn + reserveA, xy, reserveB);
        return (y * (tokenIn == token0 ? decimals1 : decimals0)) / 1e18;
    } else {
        (uint256 reserveA, uint256 reserveB) = tokenIn == token0 ? (_reserve0,
            ↪ _reserve1) : (_reserve1, _reserve0);
        // (reserveA + amountIn) * (reserveB - amountOut) = reserveA * reserveB
        // rearrange the formula, we have amountOut = (amountIn * reserveB) /
            ↪ (reserveA + amountIn)
        return (amountIn * reserveB) / (reserveA + amountIn);
    }
}
```

This is the implementation for `_getAmountOut` in Aerodrome.

Let's consider below scenario: WBTC/USDC pool. The chainlink price from oracle is 70000 USD. Assume the average wbtc amount is 10 WBTC, and the average USDC amount is 7_000_000 USDC.

- Reserve BTC = 10
- Reserve USDC = 7000_000 x y = k pool.quote(token0, 10 ** token0PoolDecimals, TWAP_GRANULARITY) --> calculate USDC amount based on input 1 WBTC. The output USDC amount is 63,636.4 USDC. So the price we get is far away from 70000.

geovgy

Ok I see now here how `amountIn` does skew the price and understand what you mean with

Issue M-1: Uncollected fees may be locked in Coll-SurplusPool/DefaultPool/SP [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/97>

Summary

Uncollected fees may be locked in CollSurplusPool/DefaultPool/SP

Vulnerability Detail

When one unhealthy trove is liquidated, the trove's collateral may be distributed to liquidators, borrowers themselves(CollSurplusPool), SP holders(SP) and the remaining borrowers(DefaultPool).

In BaseDollar, Aerodrome LP tokens can be taken as the collateral token. These collateral tokens will be transferred to CollSurplusPool/StabilityPool/DefaultPool.

Different with other normal token, AeroDrome LP tokens can claim some fees. When LP tokens are left in CollSurplusPool/StabilityPool/DefaultPool, these LP tokens will keep earning fees. In Aerodrome, LP holders need to trigger `claimFees` to claim un-claimed fees. The problem here is that CollSurplusPool/StabilityPool/DefaultPool miss the related interface, which will cause that some unclaimed fees will be locked in BaseDollar pools.

```
function batchLiquidateTrove(uint256[] memory _troveArray) public override {
    _redistributeDebtAndColl(
        activePoolCached, defaultPoolCached, totals.debtToRedistribute,
        → totals.collToRedistribute
    );
    // When one trove is liquidated, the collateral will be transferred to the coll
    → surplus pool.
    if (totals.collSurplus > 0) {
        activePoolCached.sendColl(address(collSurplusPool), totals.collSurplus);
    }
}
```

Impact

Unclaimed fees in CollSurplusPool/StabilityPool/DefaultPool may be locked in the contract. This may become one high loss when users don't claim their collateral immediately.

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb%20e4000e612d1c3f070e56dd5c5767f3/basedollar/contracts/src/TroveManager.sol#L465-L467>

Tool Used

Manual Review

Recommendation

Add one interface to claim fees.

Discussion

geovgy

This is a great catch. But I believe a `medium` severity level would be more appropriate for this issue since its not a loss of existing funds/collateral, rather just missing out on potential earnings.

Issue M-2: Depeg recover mechanism may not take effect as expected [ACKNOWLEDGED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/99>

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

Summary

Depeg recover mechanism may not take effect as expected

Vulnerability Detail

When the stable coin baseDollar is depegged, the system encourages users to redeem collateral tokens to increase baseDollar's price. In baseDollar, there are some unredeemable troves. Trove branches which use Aerodrome LP tokens as collaterals are not allowed to be redeemed.

The problem here is that if the collateral ratio of LP tokens is too high, users may not redeem enough collateral tokens to increase baseDollar's price. This will cause that depeg recover mechanism does not take effect as expected.

```
function redeemCollateral(uint256 _boldAmount, uint256 _maxIterationsPerCollateral,
    ↪ uint256 _maxFeePercentage)
    external
{
    // Compute redemption amount for each collateral and redeem against the
    ↪ corresponding TroveManager
    for (uint256 index = 0; index < totals.numCollaterals; index++) {
        //uint256 unbackedPortion = unbackedPortions[index];
        if (unbackedPortions[index] > 0) {
            uint256 redeemAmount = _boldAmount * unbackedPortions[index] /
                ↪ totals.unbacked;
            if (redeemAmount > 0) {
                ITroveManager troveManager = getTroveManager(index);
                uint256 redeemedAmount = troveManager.redeemCollateral(
                    msg.sender, redeemAmount, prices[index], redemptionRate,
                    ↪ _maxIterationsPerCollateral
                );
                totals.redeemedAmount += redeemedAmount;
            }

            // Ensure that per-branch redeems add up to `_boldAmount` exactly
            _boldAmount -= redeemAmount;
            totals.unbacked -= unbackedPortions[index];
        }
    }
}
```

```
}  
}
```

Impact

If baseDollar is de-pegged, users cannot increase baseDollar's price via redeeming collateral.

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb%20e4000e612d1c3f070e56dd5c5767f3/basedollar/contracts/src/CollateralRegistry.sol#L155-L164>

Tool Used

Manual Review

Recommendation

Add one safety check for the unredeemable collateral ratio, e.g. the ratio for unredeemable collateral cannot exceed 50%.

Issue M-3: stEthPerToken interface does not exist in WSTETH on base chain [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/100>

Summary

stEthPerToken interface does not exist in WSTETH on base chain

Vulnerability Detail

In WSTETHPriceFeed, the wstETH's price is calculated via $\text{USD_per_STETH} * \text{STETH_per_WSTETH}$. The STETH_per_WSTETH is fetched from the interface of stEthPerToken in wstETH contract.

The problem here is that wstETH contract's implementation on base chain is different with the wstETH contract's implementation on Ethereum chain. In wstETH contract on base chain, there is not one interface stEthPerToken.

```
function _getCanonicalRate() internal view override returns (uint256, bool) {
    uint256 gasBefore = gasleft();
    try IWSTETH(rateProviderAddress).stEthPerToken() returns (uint256
        ↪ stEthPerWstEth) {
        // If rate is 0, return true
        if (stEthPerWstEth == 0) return (0, true);

        return (stEthPerWstEth, 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 reverted for another reason, return true
        return (0, true);
    }
}
```

Impact

WstETH price cannot be calculated correctly.

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddbe4000e612d1c3f070e56dddd5c5767f3/basedollar/contracts/src/PriceFeeds/WSTETHPriceFeed.sol#L80-L88>

Tool Used

Manual Review

Recommendation

Consider to fetch `STETH_per_WSTETH` from chainlink oracle.

Issue M-4: pendingNewClaimFee not cleared on immediate fee decrease — prior pending increase survives apparent reduction [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/105>

Summary

When the governor immediately decreases `claimFee`, `updateClaimFee()` applies the decrease but does not clear `pendingNewClaimFee`. A previously queued high-fee increase remains valid and can be activated after its 7-day timelock, despite the visible fee reduction making users believe the threat was neutralized.

Vulnerability Detail

```
// basedollar/contracts/src/AeroManager.sol
function updateClaimFee(uint256 newFee) external onlyGovernor {
    require(newFee != claimFee, "...same fee...");
    _requireClaimFeeLimit(newFee);
    if (newFee > claimFee) {
        pendingNewClaimFee = newFee;
        pendingNewClaimFeeTimestamp = block.timestamp;
        emit ClaimFeeUpdatePending(...);
    } else {
        claimFee = newFee;          // immediate decrease
        emit ClaimFeeUpdated(...);
        // ← pendingNewClaimFee NOT cleared
    }
}
```

Sequence: (1) Queue 40% increase at T. (2) Immediately lower to 5% – `ClaimFeeUpdated(10%→5%)` emitted, community calms down. (3) `pendingNewClaimFee=40%` still valid. (4) At T+7d, call `acceptClaimFeeUpdate()` → `claimFee=40%`.

Impact

Borrowers lose AERO rewards at a fee rate (40%) they believed was cancelled. The decrease+pending-increase sequence defeats the purpose of the 7-day timelock.

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb%20e4000e612d1c3f070e56dd5c5767f3/basedollar/contracts/src/AeroManager.sol#L325-L329>

Tool Used

Manual Review

Recommendation

```
} else {
    uint256 oldFee = claimFee;
    claimFee = newFee;
    emit ClaimFeeUpdated(oldFee, newFee);
+   if (pendingNewClaimFee > 0) {
+       pendingNewClaimFee = 0;
+       pendingNewClaimFeeTimestamp = 0;
+       emit ClaimFeeUpdateCancelled(pendingNewClaimFee, newFee);
+   }
}
```

Issue M-5: setAeroManagerAddress() in ActivePool is permanently uncallable — AeroManager irrevocably bound [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/106>

Summary

ActivePool.setAeroManagerAddress() requires msg.sender == aeroManagerAddress. AeroManager has no function that calls this. AeroManager renounces ownership in setAddressses(). The migration path is permanently unreachable; a compromised or buggy AeroManager can never be replaced.

Vulnerability Detail

```
// basedollar/contracts/src/ActivePool.sol
constructor(...) {
    // ...
    if (_isAeroLPCollateral) {
        collToken.approve(aeroManagerAddress, type(uint256).max); // unlimited,
        → irrevocable
    }
}

function setAeroManagerAddress(address _newAeroManagerAddress) external {
    _requireCallerIsAeroManager(); // only AeroManager can call this - but
    → AeroManager never does
    collToken.approve(aeroManagerAddress, 0);
    collToken.approve(_newAeroManagerAddress, type(uint256).max);
    aeroManagerAddress = _newAeroManagerAddress;
}
```

AeroManager contains zero calls to setAeroManagerAddress. setAddressses() ends with _renounceOwnership(). There is therefore no in-protocol path to call setAeroManagerAddress. Additionally, collToken.approve(aeroManagerAddress, type(uint256).max) in the ActivePool constructor gives AeroManager unlimited LP token pull rights that can never be revoked.

Impact

Any post-deployment AeroManager vulnerability becomes permanent with no emergency response. Full LP collateral TVL across all AeroLP branches is permanently

exposed.

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb-e4000e612d1c3f070e56dddd5c5767f3/basedollar/contracts/src/ActivePool.sol#L359>
<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb-e4000e612d1c3f070e56dddd5c5767f3/basedollar/contracts/src/AeroManager.sol#L121>

Tool Used

Manual Review

Recommendation

Add a `migrateActivePool(address activePool, address newAeroManager)` external only Governor function in `AeroManager` that calls `IActivePool(activePool).setAeroManagerAddress(newAeroManager)`. Alternatively, change `setAeroManagerAddress` to be callable by a governor with an appropriate timelock.

Issue M-6: aeroTokenAddress migration permanently locks all accumulated claimableRewards [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/107>

Summary

After `acceptAeroTokenAddressUpdate()` replaces `aeroTokenAddress` with a new token, `claimRewards()` forever transfers the NEW token – which has zero balance – while the OLD token accumulates in the contract with no rescue path.

Vulnerability Detail

`claimableRewards[user]` is credited via `distributeAero()` in units of the token current at that time. `claimRewards()` always reads the live `aeroTokenAddress`:

```
// basedollar/contracts/src/AeroManager.sol
function acceptAeroTokenAddressUpdate() external onlyGovernor {
    // ...
    aeroTokenAddress = pendingAeroTokenAddress; // no migration of claimableRewards
    // ...
}

function claimRewards(address user) external nonReentrant {
    uint256 amount = claimableRewards[user];
    require(amount > 0, "AeroManager: No rewards to claim");
    claimableRewards[user] = 0;
    IERC20(aeroTokenAddress).safeTransfer(user, amount); // always uses current
    ↪ address
    emit RewardsClaimed(user, amount);
}
```

After `acceptAeroTokenAddressUpdate()` switches `aeroTokenAddress = NEW_AERO`, the contract holds only OLD_AERO (zero NEW_AERO balance). Every `claimRewards` call reverts. No `withdrawToken()`, `rescueToken()`, or emergency sweep exists anywhere in `AeroManager`.

Impact

Permanent loss of all pending AERO rewards for every borrower at the time of a token switch. The full OLD_AERO balance is permanently locked in the contract.

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb%20e4000e612d1c3f070e56dd5c5767f3/basedollar/contracts/src/AeroManager.sol#L308>

Tool Used

Manual Review

Recommendation

Before switching `aeroTokenAddress`, snapshot pending `claimableRewards` and retain the old address in a legacy mapping so users can claim in the old token during a migration window. Alternatively, add an `emergencySweep(address token, address to)` callable only by governor.

Issue M-7: cbETHPriceFeed constructor always reverts due to incorrect decimal assumption [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/108>

Summary

The `cbETHPriceFeed` constructor hardcodes a requirement that the `cbETH/ETH` Chainlink oracle returns 8 decimals. On Base, the `cbETH/ETH` feed of chainlink/api3 oracle returns **18 decimals**. The constructor will always revert, making the `cbETH` collateral branch permanently undeployable.

Vulnerability Detail

In `cbETHPriceFeed.sol`, the constructor fetches the oracle's decimal precision on-chain and then asserts it equals exactly 8:

```
cbEthEthOracle.decimals = cbEthEthOracle.aggregator.decimals();  
  
require(cbEthEthOracle.decimals == 8, "cbETHPriceFeed: cbETH-ETH oracle must have 8  
↪ decimals");
```

- the decimal of `cbETH/ETH` price feed on Base is 18. check [here](#).
- All Api3 data feeds have 18 decimals: check [here](#).

So it always reverts.

Impact

The `cbETH` collateral branch can never be deployed. The constructor reverts unconditionally on Base mainnet. Any deployment attempt for the `cbETH` branch fails at the `cbETHPriceFeed` construction step, blocking the entire branch's `AddressesRegistry` setup.

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb4000e612d1c3f070e56ddd5c5767f3/basedollar/contracts/src/PriceFeeds/cbETHPriceFeed.sol#L21>

Tool Used

Manual Review

Recommendation

```
-    require(cbEthEthOracle.decimals == 8, "cbETHPriceFeed: cbETH-ETH oracle must
↪    have 8 decimals");
+    require(
+        cbEthEthOracle.decimals == 8 || cbEthEthOracle.decimals == 18,
+        "cbETHPriceFeed: cbETH-ETH oracle must have 8 or 18 decimals"
+    );
```


Issue M-8: `closeTroveFromCollateral` is permanently non-functional [ACKNOWLEDGED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/110>

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

Summary

The `WrappedTokenZapper` constructor has the exchange approval line explicitly commented out. `receiveFlashLoanOnCloseTroveFromCollateral` unconditionally calls `exchange.swapToBold()`, which requires pulling wrapped tokens from the zapper via `ERC20 transferFrom`. With zero allowance, every `closeTroveFromCollateral` call reverts inside the flash loan callback.

Vulnerability Detail

The `WrappedTokenZapper` constructor has the exchange approval line explicitly commented out.

```
// Approve Coll to exchange module (for closeTroveFromCollateral)
// _wrappedToken.approve(address(_exchange), type(uint256).max); // Not using
↪ exchange at the moment
```

The `receiveFlashLoanOnCloseTroveFromCollateral` function still calls `exchange.swapToBold` unconditionally:

```
exchange.swapToBold(_effectiveFlashLoanAmount, trove.entireDebt); // requires
↪ approval that doesn't exist
```

Every call to `closeTroveFromCollateral` for `cbBTC`, `cbETH`, and any other wrapped-token branch reverts due to zero allowance.

Impact

The `closeTroveFromCollateral` function is entirely non-functional

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb4e4000e612d1c3f070e56dd5c5767f3/basedollar/contracts/src/Zappers/WrappedTokenZapper.sol#L41-L42>

Tool Used

Manual Review

Recommendation

```
- // _wrappedToken.approve(address(_exchange), type(uint256).max); // Not using  
  ↪ exchange at the moment  
+ _wrappedToken.approve(address(_exchange), type(uint256).max);
```

Issue M-9: AeroLP Stable LP Token Price Calculation Integer Overflow — Price Feed Permanent DoS [RE-SOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/113>

Summary

`_calculate_stable_lp_token_price()` computes $p_0 = k * \text{mulWadDown}(a, b)$ where k is cubic in pool reserves and `mulWadDown(a, b)` is cubic in token prices. For a cbETH/ETH stable pair with $\approx 100,000$ tokens per side at $\sim \$3,000$ each, this product overflows `uint256`, causing every `fetchPrice()` call to revert and permanently DoS-ing the AeroLP branch.

Vulnerability Detail

The stable LP price formula:

```
/// @notice Calculates K for the stable invariant  $x^3 * y + y^3 * x = k$ 
...
uint256 a = FixedPointMathLib.rpow(price0, 3, 1e18); // price03 / 1e36
uint256 b = FixedPointMathLib.rpow(price1, 3, 1e18); // price13 / 1e36
uint256 p0 = k * FixedPointMathLib.mulWadDown(a, b); // OVERFLOWS for large k
```

- wETH/msETH pool:

$\text{reserve0} = 100,000e18, \text{reserve1} = 100,000e18, \text{price0} = 3000e18, \text{price1} = 3000e18$
 $k = r_0^3 * r_1 / 1e36 + r_1^3 * r_0 / 1e36 = 1e15 * 1e5 * 1e18 * 2 = 2e38$
 $a = 27e9 * 1e18 = 2.7e28$
 $b = 27e9 * 1e18 = 2.7e28$
 $p_0 = k * (a * b / 1e18) = 2e38 * 7.29e38 = 14.58e76 > 2^{256} \approx 1.158e77$

Impact

Any AeroLP stable pair where both tokens have ETH-level prices ($\approx \$3,000$) and the pool holds $\approx 100,000$ tokens per side will have every `fetchPrice()` call revert.

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb e4000e612d1c3f070e56dd5c5767f3/basedollar/contracts/src/PriceFeeds/AeroLPTokenPriceFeedBase.sol#L226-L250>

Tool Used

Manual Review

Recommendation

```
function _calculate_stable_lp_token_price(
    uint256 total_supply,
    uint256 price0, // must be in 18 decimals
    uint256 price1, // must be in 18 decimals
    uint256 reserve0,
    uint256 reserve1
) internal pure returns (uint256) {
    uint256 k = _getK(reserve0, reserve1);
    //fair_reserves = ( (k * (price0 ** 3) * (price1 ** 3)) )^(1/4) / ((price0
    ↪ ** 2) + (price1 ** 2));
    uint256 a = FixedPointMathLib.rpow(price0, 3, 1e18); //keep same decimals
    ↪ as chainlink
    uint256 b = FixedPointMathLib.rpow(price1, 3, 1e18);
    uint256 c = FixedPointMathLib.rpow(price0, 2, 1e18);
    uint256 d = FixedPointMathLib.rpow(price1, 2, 1e18);

-    uint256 p0 = k * FixedPointMathLib.mulWadDown(a, b); //2*18 decimals

-    uint256 fair = p0 / (c + d); // number of decimals is 18
+    uint256 fair;
+    if (type(uint256).max / k >= FixedPointMathLib.mulWadDown(a, b)) {
+        uint256 p0 = k * FixedPointMathLib.mulWadDown(a, b); //2*18 decimals
+        fair = p0 / (c + d); // number of decimals is 18
+    } else {
+        uint256 p0 = FixedPointMathLib.mulWadDown(a, b) * 1e18 / (c + d);
+        fair = k * p0 / 1e18;
+    }
    // each sqrt divides the num decimals by 2. So need to replenish the
    ↪ decimals midway through with another 1e18
    uint256 frth_fair = FixedPointMathLib.sqrt(
        FixedPointMathLib.sqrt(fair * 1e18) * 1e18
    ); // number of decimals is 18

    return 2 * ((frth_fair * 1e18) / total_supply);
}
```

Issue M-10: Improper oracle staleness threshold security check [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/120>

Summary

Improper oracle staleness threshold security check

Vulnerability Detail

In baseDollar, API3 is used as the oracle source. When one response is fetched from API3, there will be one security validation to make sure that the returned oracle price is not staled.

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

The problem here is that in API3's doc(<https://docs.api3.org/dapps/integration/contract-integration.html>), the returned timestamp is not the block timestamp at the time of the update, but the reported system(i.e. off-chain). So it's possible that the return timestamp is one future timestamp. Then the price staleness will cause the whole transaction reverted. This might cause that unhealthy positions cannot be liquidated in time.

Impact

Price check will cause operations reverted.

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb4000e612d1c3f070e56dd5c5767f3/basedollar/contracts/src/PriceFeeds/MainnetPriceFeedBase.sol#L111-L118>

Tool Used

Manual Review

Recommendation

According to API3's doc, it's suggested using below check:

```
require(timestamp + 24 hours > block.timestamp)
```

Issue L-1: getDebtBetweenInterestRates/getDebtBetweenInterestRateAndTrove does not support unredeemable trove branches [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/101>

Summary

getDebtBetweenInterestRates/getDebtBetweenInterestRateAndTrove does not support unredeemable trove branches

Vulnerability Detail

DebtInFrontHelper is one helper contract which is used calculate debt-in-front precisely.

The problem here is that in function

getDebtBetweenInterestRates/getDebtBetweenInterestRateAndTrove, getTroveManager is used to fetch the related trove manager. It means that the unredeem trove branch's manager cannot be fetched by getTroveManager.

```
function getDebtBetweenInterestRates(
    uint256 _collIndex,
    uint256 _interestRateLo, // inclusive
    uint256 _interestRateHi, // exclusive
    uint256 _excludedTroveId,
    uint256 _hintId,
    uint256 _numTrials
) external view returns (uint256 debt, uint256 blockTimestamp) {
    return
        _getDebtBetween(_collIndex, _interestRateLo, _interestRateHi,
            ↪ _excludedTroveId, false, _hintId, _numTrials);
}

function getDebtBetweenInterestRateAndTrove(
    uint256 _collIndex,
    uint256 _interestRateLo, // inclusive
    uint256 _interestRateHi, // exclusive
    uint256 _troveIdToStopAt, // excluded
    uint256 _hintId,
    uint256 _numTrials
) external view returns (uint256 debt, uint256 blockTimestamp) {
    return
        _getDebtBetween(_collIndex, _interestRateLo, _interestRateHi,
            ↪ _troveIdToStopAt, true, _hintId, _numTrials);
}
```

Impact

Users cannot calculate the debt via this help contract.

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb%20e4000e612d1c3f070e56dd5c5767f3/basedollar/contracts/src/DebtInFrontHelper.sol#L94-L116>

Tool Used

Manual Review

Recommendation

Add support for un-redeemable collateral trove branches.

Issue L-2: Missing sequence uptime check for Chainlink Oracle [ACKNOWLEDGED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/102>

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

Summary

Missing sequence uptime check for Chainlink Oracle

Vulnerability Detail

In BaseDollar, collateral token's price is calculated via Chainlink oracle. According to the Chainlink doc(<https://docs.chain.link/data-feeds/l2-sequencer-feeds>), when Chainlink is used on one L2 chain, L2 sequencer uptime feeds check is needed.

This check can prevent creating unfair advantage for liquidators to liquidate unhealthy trove positions.

```
function _getCurrentChainlinkResponse(AggregatorV3Interface _aggregator)
    internal
    view
    returns (ChainlinkResponse memory chainlinkResponse)
{
    try _aggregator.latestRoundData() returns (
        uint80 roundId, int256 answer, uint256, /* startedAt */ uint256 updatedAt,
        ↪ uint80 /* answeredInRound */
    ) {
        // If call to Chainlink succeeds, return the response and success = true
        chainlinkResponse.roundId = roundId;
        chainlinkResponse.answer = answer;
        chainlinkResponse.timestamp = updatedAt;
        chainlinkResponse.success = true;

        return chainlinkResponse;
    }
}
```

Impact

Users with sufficient technical expertise can still interact directly with the network through the underlying rollup contracts on L1. For example, liquidators may gain advantage to liquidate unhealthy positions.

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb%20e4000e612d1c3f070e56dd5c5767f3/basedollar/contracts/src/PriceFeeds/MainnetPriceFeedBase.sol#L86-L96>

Tool Used

Manual Review

Recommendation

Add L2 sequencer uptime check for Chainlink oracle.

Issue L-3: Inconsistency between the implementation and the comment [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/103>

Summary

Inconsistency between the implementation and the comment

Vulnerability Detail

In BaseDollar, borrowers which use Aerodrome LP tokens as collateral, can earn some rewards from the Aerodrome gauges. The protocol team can claim one part of the claimed fee.

The maximum aero manager fees in implementation is 50%. However, according to the comment here, the max aero manager fee is 20%. It's suggested to make the implementation and the comment consistent.

```
uint256 constant MAX_AERO_MANAGER_FEE = 50 * _1pct; // 20%
```

Impact

This is not one security issue. Just one coding practice suggestion.

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb4000e612d1c3f070e56dd5c5767f3/basedollar/contracts/src/Dependencies/Constants.sol#L124>

Tool Used

Manual Review

Recommendation

Update the implementation or the comment to make them consistent.

Issue L-4: Suggest using BaseDollar as TroveNFT's name and symbol [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/104>

Summary

Suggest using BaseDollar as TroveNFT's name and symbol

Vulnerability Detail

In BaseDollar, when users borrow baseDollar, one NFT will be minted for this borrow position. Here baseDollar uses the same name and symbol with Liquity. This may raise some confusion for borrowers.

Suggest to initialize the NFT's name and symbol with baseDollar.

```
    constructor(IAddressesRegistry _addressesRegistry)
        ERC721(
@>            string.concat("Liquity V2 - ", _addressesRegistry.collToken().name()),
            string.concat("LV2_", _addressesRegistry.collToken().symbol())
        )
    {
        troveManager = _addressesRegistry.troveManager();
        collToken = _addressesRegistry.collToken();
        metadataNFT = _addressesRegistry.metadataNFT();
        boldToken = _addressesRegistry.boldToken();
    }
```

Impact

This is not one security issue. Just some code improvement suggestion.

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb e4000e612d1c3f070e56ddddd5c5767f3/basedollar/contracts/src/TroveNFT.sol#L21-L31>

Tool Used

Manual Review

Recommendation

Suggest to initialize the NFT's name and symbol with baseDollar.

Issue L-5: AEROPriceFeed constructor parameter `_owner` should be `_borrowerOperationsAddress` [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/109>

Summary

AEROPriceFeed names its first constructor parameter `_owner`, but passes it directly to `TokenPriceFeedBase` as `_borrowerOperationsAddress`. The name is a bit misleading.

Vulnerability Detail

`TokenPriceFeedBase` stores the first argument as `borrowerOperations` and calls `borrowerOperations.shutdownFromOracleFailure()` when the oracle fails.

The parameter name `_owner` implies an ownership role that does not exist anywhere in the contract.

Impact

Code improvement suggestion.

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb4000e612d1c3f070e56dd5c5767f3/basedollar/contracts/src/PriceFeeds/AEROPriceFeed.sol#L9-L10>

Tool Used

Manual Review

Recommendation

Rename the parameter to match the base class and every other feed in the codebase:

```
- constructor(address _owner, address _aeroUsdOracleAddress, uint256
↳ _aeroUsdStalenessThreshold)
-     TokenPriceFeedBase(_owner, _aeroUsdOracleAddress,
↳ _aeroUsdStalenessThreshold)
```

```
+   constructor(address _borrowerOperationsAddress, address _aeroUsdOracleAddress,  
↪   uint256 _aeroUsdStalenessThreshold)  
+       TokenPriceFeedBase(_borrowerOperationsAddress, _aeroUsdOracleAddress,  
↪       _aeroUsdStalenessThreshold)
```

Issue L-6: WrappedToken.withdrawTo permanently locks dust due to integer division truncation [ACKNOWLEDGED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/111>

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

Summary

`withdrawTo` burns the full wrapped amount but transfers `amount / 10**_decimalDiff` underlying tokens, silently discarding any remainder via integer division. For `cbBTC` where `_decimalDiff = 10`, remainders up to $1e10 - 1$ wrapped wei are burned with zero underlying returned, permanently locking value in the contract with no sweep function to recover it.

Vulnerability Detail

For ETH-related tokens, `_decimalDiff` is 0, whereas for `cbBTC` it is 10, which can result in leftover dust amounts. In the case of `cbBTC`, 20,000 wei exceeds \$10 in value.

```
function withdrawTo(address account, uint256 amount) public virtual override
↳ returns (bool) {
    _burn(_msgSender(), amount);
    underlying().safeTransfer(account, amount / 10**_decimalDiff);
}
```

Impact

lock of funds

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb4000e612d1c3f070e56dd5c5767f3/basedollar/contracts/src/ERC20Wrappers/WrappedToken.sol#L42-L45>

Tool Used

Manual Review

Recommendation

Introduce a sweep function.

Issue L-7: NatSpec comment for updateDebtLimit misrepresents the 2× cap enforcement condition [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/112>

Summary

The NatSpec on updateDebtLimit states the limit is "limited to increasing by 2x at a time" unconditionally, but the actual code adds a second condition: increases beyond 2× are also permitted if the new value does not exceed getInitalDebtLimit(). The comment omits this bypass entirely.

Vulnerability Detail

The implementation is:

```
require(  
    _newDebtLimit <= currentDebtLimit * 2  
    || _newDebtLimit <= getTroveManager(_indexTroveManager).getInitalDebtLimit(),  
    "..."  
);
```

The `|| _newDebtLimit <= getInitalDebtLimit()` branch allows the governor to jump to any value up to the initial limit in a single step, regardless of the 2× cap. If the initial limit was set high and later reduced due to a risk event, the governor can immediately restore it to the original high value, bypassing the incremental cap entirely. The NatSpec makes no mention of this escape hatch.

Impact

Code improvement suggest

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb-e4000e612d1c3f070e56dddd5c5767f3/basedollar/contracts/src/CollateralRegistry.sol#L380-L383>

Tool Used

Manual Review

Recommendation

Update the NatSpec to accurately reflect both conditions:

```
- /// limited to increasing by 2x at a time, maximum. Decrease by any amount.  
+ /// Decreases are unrestricted. Increases must satisfy at least one of:  
+ /// (1) the new limit does not exceed 2× the current limit, or  
+ /// (2) the new limit does not exceed the initial deployment limit.
```

Issue L-8: claim() accepts any arbitrary gauge address — epoch accounting pollution and reward misappropriation [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/114>

Summary

claim(address gauge) is permissionless with no whitelist check against registered active pools. Combined with governor-only distributeAero(), AERO from unregistered gauges can be credited to arbitrary recipients with no LP-stake relationship.

Vulnerability Detail

The only validation in claim():

```
// basedollar/contracts/src/AeroManager.sol
require(IAeroGauge(gauge).rewardToken() == aeroTokenAddress, "AeroManager: Reward
↪ token does not match");
```

There is no require(registeredGauges[gauge]). The activePools[] mapping (which gates stake/withdraw) is never cross-referenced. A governor-colluding attacker can: deploy FakeGauge with rewardToken()=aeroTokenAddress, call claim(FakeGauge), close/distribute epochs to arbitrary addresses, and starve legitimate borrowers by never closing real registered-gauge epochs.

Impact

A malicious gauge can be utilized.

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb4e4000e612d1c3f070e56dd5c5767f3/basedollar/contracts/src/AeroManager.sol#L254>

Tool Used

Manual Review

Recommendation

Maintain `registeredGauges[gauge]=true` populated in `_addActivePool`. Add `require(registeredGauges[gauge], "Gauge not registered")` at the start of `claim()`.

Issue L-9: Missing sanity check in the updateCollateralGovernor function [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/115>

Summary

Missing sanity check in the updateCollateralGovernor function

Vulnerability Detail

In the updateCollateralGovernor function, sanity check is missing.

```
/// @notice Update the collateral governor
/// @param _newCollateralGovernor The new collateral governor address
function updateCollateralGovernor(address _newCollateralGovernor) external
↳ onlyGovernor {
    address oldGovernor = collateralGovernor;
```

Impact

Code improvement suggest

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb4000e612d1c3f070e56dd5c5767f3/basedollar/contracts/src/CollateralRegistry.sol#L474-L475>

Tool Used

Manual Review

Recommendation

```
+     require(_newCollateralGovernor != address(0), "CR: Governor cannot be zero
↳ address");
+     require(_newCollateralGovernor != collateralGovernor, "CR: Already
↳ governor");
```

Issue L-10: Use the `_calcUpfrontFeeNonRedeemable` function for all non-redeemable helper functions [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/116>

Summary

The `_calcUpfrontFeeNonRedeemable` function is never used.

Vulnerability Detail

The `_calcUpfrontFeeNonRedeemable` function is never used. All nonredeemable helper functions use the `_calcUpfrontFee` function instead of this function even though the `_calcUpfrontFee` function and this function are same.

```
function _calcUpfrontFeeNonRedeemable(uint256 _debt, uint256 _avgInterestRate)
↳ internal pure returns (uint256) {
    return _debt * _avgInterestRate * UPFRONT_INTEREST_PERIOD / ONE_YEAR /
↳ DECIMAL_PRECISION;
}
```

Impact

Code improvement suggest

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb4000e612d1c3f070e56dd5c5767f3/basedollar/contracts/src/HintHelpers.sol#L303-L305>

Tool Used

Manual Review

Recommendation

It would be better for non redeemable helper functions to use the `_calcUpfrontFeeNonRedeemable` function instead of the `_calcUpfrontFee` function.

Issue L-11: The AeroManager does not use 3-day proposeGovernor/acceptGovernor pattern [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/117>

Summary

The `setGovernor` function of AeroManager does not use 3-day `proposeGovernor/acceptGovernor` pattern.

Vulnerability Detail

The `setGovernor` function sets the governor directly without any delay.

```
function setGovernor(address _governor) external onlyGovernor {
    governor = _governor;
    emit GovernorUpdated(_governor);
}
```

Impact

Setting governor without delay could be harmful.

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb-e4000e612d1c3f070e56dddd5c5767f3/basedollar/contracts/src/AeroManager.sol#L151-L154>

Tool Used

Manual Review

Recommendation

Use 3-day `proposeGovernor/acceptGovernor` pattern.

Issue L-12: There is no function to set treasuryAddress [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/issues/118>

Summary

There is no function to set treasuryAddress in the AeroManager.

Vulnerability Detail

AeroManager declares treasuryAddress and only sets it in the constructor. There is no function to set the treasuryAddress.

```
aeroTokenAddress = _aeroTokenAddress;  
governor = _governor;  
treasuryAddress = _treasuryAddress;
```

Impact

If the treasury address ever becomes compromised, blacklisted or deprecated, claim() will revert and the protocol loses its ability to harvest AERO through that function.

Code Snippet

<https://github.com/sherlock-audit/2026-04-base-dollar-apr-20th-2026/blob/7b134ddb e4000e612d1c3f070e56dddd5c5767f3/basedollar/contracts/src/AeroManager.sol#L87>

Tool Used

Manual Review

Recommendation

It is recommended to add a function to set the treasuryAddress.

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.