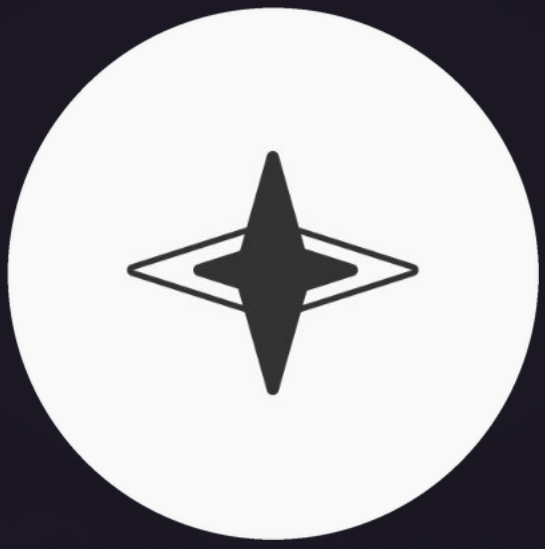
SHERLOCK

Security Review For Vyro Finance



Collaborative Audit Prepared For: **Vyro Finance**
Lead Security Expert(s): **KupiaSec**
santipu_
Date Audited: **February 20 - February 23, 2026**

Introduction

Vyro CDP is a non-custodial borrowing protocol on Unichain that mints vyUSD, an over-collateralized stablecoin backed by ETH, wstETH, and UNI. Borrowers select their own interest rate band based on their risk tolerance, producing market-driven borrowing costs.

Scope

Repository: summerstonexyz/vyro

Audited Commit: de3e50fcbd19afdc0531ffb5e714418ae0607b93

Final Commit: aebcc884dbacbee3f36f3e891dc06f4f88ff698c

Files:

- contracts/src/ActivePool.sol
- contracts/src/AddressesRegistry.sol
- contracts/src/BoldToken.sol
- contracts/src/BorrowerOperationsAdjustFacet.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/LiquityBase.sol
- contracts/src/Dependencies/LiquityMath.sol
- contracts/src/GasPool.sol
- contracts/src/Governance/TimelockedMultiSigGovernor.sol
- contracts/src/Helpers/AdjustHelper.sol
- contracts/src/Helpers/TroveAdjustLib.sol
- contracts/src/HintHelpers.sol
- contracts/src/Interfaces/IActivePool.sol
- contracts/src/Interfaces/IAddRemoveManagers.sol

- contracts/src/Interfaces/IAddressesRegistry.sol
- contracts/src/Interfaces/IBoldRewardsReceiver.sol
- contracts/src/Interfaces/IBoldToken.sol
- contracts/src/Interfaces/IBorrowerOperationsAdjustFacet.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/ILiquityBase.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/ISortedTroves.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/IUNIPriceFeed.sol
- contracts/src/Interfaces/IWETH.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/CompositePriceFeed.sol
- contracts/src/PriceFeeds/MainnetPriceFeedBase.sol
- contracts/src/PriceFeeds/UNIPriceFeed.sol
- contracts/src/PriceFeeds/WETHPriceFeed.sol
- contracts/src/PriceFeeds/WSTETHPriceFeed.sol
- contracts/src/RedemptionHelper.sol
- contracts/src/SortedTroves.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/TroveId.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/LeverageLSTZapper.sol
- contracts/src/Zappers/LeverageWETHZapper.sol
- contracts/src/Zappers/Modules/Exchanges/CurveExchange.sol
- contracts/src/Zappers/Modules/Exchanges/Curve/ICurveFactory.sol
- contracts/src/Zappers/Modules/Exchanges/Curve/ICurvePool.sol
- contracts/src/Zappers/Modules/Exchanges/Curve/ICurveStableswapNGFactory.sol
- contracts/src/Zappers/Modules/Exchanges/Curve/ICurveStableswapNGPool.sol
- contracts/src/Zappers/Modules/Exchanges/HybridCurveUniV3ExchangeHelpers.sol
- contracts/src/Zappers/Modules/Exchanges/HybridCurveUniV3ExchangeHelpersV2.sol
- contracts/src/Zappers/Modules/Exchanges/HybridCurveUniV3Exchange.sol
- contracts/src/Zappers/Modules/Exchanges/UniswapV3/INonfungiblePositionManager.sol
- contracts/src/Zappers/Modules/Exchanges/UniswapV3/IPoolInitializer.sol
- contracts/src/Zappers/Modules/Exchanges/UniswapV3/IQuoterV2.sol
- contracts/src/Zappers/Modules/Exchanges/UniswapV3/ISwapRouter.sol
- contracts/src/Zappers/Modules/Exchanges/UniswapV3/IUniswapV3Factory.sol
- contracts/src/Zappers/Modules/Exchanges/UniswapV3/IUniswapV3Pool.sol
- contracts/src/Zappers/Modules/Exchanges/UniswapV3/UniPriceConverter.sol
- contracts/src/Zappers/Modules/Exchanges/UniV3Exchange.sol
- contracts/src/Zappers/Modules/FlashLoans/BalancerFlashLoan.sol
- contracts/src/Zappers/Modules/FlashLoans/Balancer/vault/IFlashLoanRecipient.sol
- contracts/src/Zappers/Modules/FlashLoans/Balancer/vault/IVault.sol
- contracts/src/Zappers/Modules/Testnet/TestnetExchangeBoldMinter.sol
- contracts/src/Zappers/Modules/Testnet/TestnetExchange.sol
- contracts/src/Zappers/Modules/Testnet/TestnetFlashLoanProvider.sol
- contracts/src/Zappers/WETHZapper.sol

Final Commit Hash

aebcc884dbacbee3f36f3e891dc06f4f88ff698c

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	3

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: Single owner can permanently DoS governance by canceling any operation unilaterally [RESOLVED]

Source: https://github.com/sherlock-audit/2026-02-vyro-finance-audit/issues/109

Summary

Any single owner in the `TimelockedMultiSigGovernor` can unilaterally cancel any pending operation, including operations to remove them from the owner set. This breaks the multisig security model and allows a single rogue signer to permanently block governance.

Vulnerability Detail

The `cancelOperation` function at `TimelockedMultiSigGovernor` only requires the caller to be one owner:

```
function cancelOperation(bytes32 operationId) external onlyOwner {
    Operation storage op = _operations[operationId];
    if (op.target == address(0)) revert OperationMissing(operationId);
    if (op.executed) revert OperationCompleted(operationId);
    if (op.canceled) revert OperationCompleted(operationId);

    op.canceled = true;
    emit OperationCanceled(operationId, msg.sender);
}
```

Cancellation is permanent, once `op.canceled` is set to true, it can never be undone, and the operation can never be executed.

The problem is that removing an owner requires going through the full operation lifecycle (`submitOperation` → `approveOperation` → `timelock` → `executeOperation`), since `updateOwners` is gated by `onlySelf`. But a malicious owner can simply front-run the execution (or cancel at any point before it) by calling `cancelOperation` on the `updateOwners` operation that targets them.

The other owners can re-submit the same operation, but the malicious owner will just cancel it again. This creates an unbreakable cycle; the malicious owner can never be removed because the mechanism to remove them is the same mechanism they can grief.

This also extends beyond owner removal. The rogue signer can cancel any pending operation: parameter updates, timelock changes, upgrades, or any external call the multisig governs. A single compromised or malicious key is enough to completely shut down the governance process indefinitely.

Impact

A single malicious or compromised owner can permanently DoS the entire governance system. No operation can ever be executed if that owner keeps canceling them. Since owner changes also go through the same path, there is no way to remove the bad actor, making the situation irrecoverable without redeploying.

Code Snippet

https://github.com/sherlock-audit/2026-02-vyro-finance-audit/blob/main/vyro/contracts/src/Governance/TimelockedMultiSigGovernor.sol#L183

Tool Used

Manual Review

Recommendation

There are different approaches to mitigate this issue, here we present two of them:

- Option A – Introduce a Guardian role. Add a dedicated guardian address (set at initialization, updatable via `onlySelf`) that is the only account authorized to cancel operations. This separates the cancel privilege from regular signers, so a rogue owner cannot grief the governance flow. The guardian would typically be a separate trusted entity or a secondary multisig with a different signer set.
- Option B – Restrict cancellation to the proposer. Store `msg.sender` as the submitter in the Operation struct during `submitOperation`, and require `msg.sender == op.submitter` in `cancelOperation`. This way, only the owner who originally submitted the operation can cancel it. A malicious owner can only cancel their own proposals, not operations submitted by others, so the remaining honest signers can always push through an `updateOwners` call to remove them.

Issue M-2: Changing SP yield split retroactively applies to all unminted pending interest [RESOLVED]

Source: https://github.com/sherlock-audit/2026-02-vyro-finance-audit/issues/112

Summary

`CollateralRegistry::setSPYieldSplit` updates the yield split ratio without first minting the accrued pending interest. This causes all previously accumulated interest to be distributed using the new ratio instead of the ratio that was in effect when it accrued.

Vulnerability Detail

When governance calls `setSPYieldSplit`, the new value takes effect immediately:

```
function setSPYieldSplit(uint256 _newSplit) external override onlyGovernor {
    if (_newSplit > DECIMAL_PRECISION) revert InvalidSPYieldSplit();
    spYieldSplit = _newSplit;
    emit SPYieldSplitUpdated(_newSplit);
}
```

The problem is that interest is not minted continuously. It accumulates as "pending" interest based on elapsed time since `lastAggUpdateTime`, and only gets minted and split when a user operation triggers `_mintAggInterest`:

```
function _mintAggInterest(uint256 _upfrontFee) internal returns (uint256
↪  mintedAmount) {
    mintedAmount = calcPendingAggInterest() + _upfrontFee;

    // Mint part of the BOLD interest to the SP and part to the router for LPs.
    if (mintedAmount > 0) {
        uint256 spYield = collateralRegistry.spYieldSplit() * mintedAmount /
        ↪  DECIMAL_PRECISION;
        uint256 remainderToLPs = mintedAmount - spYield;

        boldToken.mint(address(interestRouter), remainderToLPs);

        if (spYield > 0) {
            boldToken.mint(address(stabilityPool), spYield);
            stabilityPool.triggerBoldRewards(spYield);
        }
    }

    lastAggUpdateTime = block.timestamp;
}
```

The split ratio is read from `collateralRegistry.spYieldSplit()` at mint time and applied to the entire `mintedAmount`, which includes all interest accumulated since `lastAggUpdateTime`. If the split was changed between the last mint and the current one, all that historical interest gets retroactively redistributed under the new ratio.

Example: Suppose `spYieldSplit` is 50% and interest has been accumulating for a week with no user operations. Governance then changes the split to 10%. The next user operation triggers `_mintAggInterest`, and the entire week's worth of interest (which accrued under a 50% split) is now distributed as 10% to SP and 90% to LPs. SP depositors lose the yield they had effectively already earned, and LP holders receive a windfall. The reverse scenario (increasing the split) would harm LPs in favor of SP depositors.

The severity increases with the size of the pending interest at the time of the split change, which depends on how long it has been since the last minting event. On low-activity branches, this gap can be substantial.

Impact

SP depositors or LP holders receive incorrect yield amounts. Depending on the direction of the split change, one side gets overpaid at the expense of the other. The magnitude scales with the amount of unminted pending interest at the time of the change, which can be significant on branches with infrequent operations.

Code Snippet

https://github.com/sherlock-audit/2026-02-vyro-finance-audit/blob/main/vyro/contracts/src/CollateralRegistry.sol#L569

Tool Used

Manual Review

Recommendation

`setSPYieldSplit` should flush all pending interest across every active branch before updating the split. This ensures that interest accrued under the old ratio is minted and distributed with the old ratio, and only future interest uses the new one. This can be done by calling `mintAggInterest` on each branch's `ActivePool` before writing the new `spYieldSplit` value.

Issue M-3: `removeCollateral` incorrectly requires zero open troves, preventing branch sunsetting [RESOLVED]

Source: https://github.com/sherlock-audit/2026-02-vyro-finance-audit/issues/113

Summary

`CollateralRegistry::removeCollateral` enforces that the branch has zero open troves before it can be deactivated. This defeats the intended sunsetting design, where removal should come first to stop new activity and incentivize users to close or redeem their positions, with `cleanRemovedCollateral` performing the final zero troves check afterwards.

Vulnerability Detail

The `removeCollateral` function requires that the branch's `TroveManager` has no open troves:

```
  function removeCollateral(uint256 _index) external override onlyGovernor {
      uint256 branchId = _branchIdAtIndex(_index);
      CollateralConfig storage config = collateralById[branchId];

      if (config.activeIndex == type(uint256).max) revert InvalidBranchId();
      address troveManagerAddr = address(config.troveManager);
      if (troveManagerAddr.code.length != 0) {
>>          if (config.troveManager.getTroveIdsCount() != 0) revert TrovesRemain();
      }

      // ...
  }
```

The intended lifecycle is a two step process. First, governance calls `removeCollateral` to deactivate the branch, which removes it from `activeCollateralIds`. This signals to borrowers that the branch is being sunset and they should close or redeem their positions. Second, once all troves are closed, governance calls `cleanRemovedCollateral` to finalize the removal and revoke the branch's permissions with the BOLD token.

The function `cleanRemovedCollateral` already has the correct zero troves check:

```
  function cleanRemovedCollateral(uint256 _branchId) external override onlyGovernor
  ↪ {
      // ...

      address troveManagerAddr = address(config.troveManager);
      if (troveManagerAddr.code.length != 0) {
>>        if (config.troveManager.getTroveIdsCount() != 0) {
```

```
            revert TrovesRemain();
        }
    }

    // ...
}
```

Having the same check duplicated in `removeCollateral` creates a deadlock: governance cannot deactivate the branch while troves exist, but users have no signal or incentive to close their troves because the branch is still fully active. In practice, it's nearly impossible to reach zero troves organically on a live branch since there is no mechanism to force all borrowers to close simultaneously.

This becomes critical when a collateral token starts showing risk (e.g. depegging, loss of liquidity, oracle degradation). Governance needs to act quickly to remove the branch from the active set, but the zero troves check makes that impossible until every single borrower has voluntarily exited.

Impact

Governance cannot sunset a risky collateral branch while it has open troves, which is precisely the scenario where removal is most urgent. A collateral that becomes unsafe remains fully active in the protocol (included in redemptions, accepting new borrows) until every trove is manually closed, exposing the stablecoin to undercollateralization risk.

Code Snippet

https://github.com/sherlock-audit/2026-02-vyro-finance-audit/blob/main/vyro/contracts/src/CollateralRegistry.sol#L155

Tool Used

Manual Review

Recommendation

Remove the `getTroveIdsCount() == 0` check from `removeCollateral`. The zero troves requirement should only be enforced in `cleanRemovedCollateral`, which is the finalization step meant to run after all positions have been closed.

Issue M-4: `addManager` Role Not Properly Revoked [RESOLVED]

Source: https://github.com/sherlock-audit/2026-02-vyro-finance-audit/issues/114

Summary

The addManager role is not effectively revoked when the TroveNFT ownership changes, which may cause the manager to unintentionally add collateral benefiting the new owner.

Vulnerability Detail

When the ownership of a TroveNFT changes, the addManager removeManager roles are expected to be invalidated. However, even if the ownership is transfered, the addManager role is not invalideated.

```
150:         address addManager = _isAddManagerActiveForOwner(_troveId, _owner,
↪  ownershipChangeCount)
151:             ? _addManagerOf[_troveId]
152:             : address(0);
```

If the manager calls addCollateral() while the ownership transfer is processed in the same block, the added collateral will ultimately belong to the new owner. As a result, the manager may unintentionally deposit collateral for the benefit of the new owner, even though ownership has already changed.

Impact

The manager may suffer a loss of funds by adding collateral that ultimately benefits the new owner after a TroveNFT ownership transfer executed.

Code Snippet

https://github.com/sherlock-audit/2026-02-vyro-finance-audit/blob/main/vyro/contracts
allowbreak src
allowbreak Dependencies
allowbreak AddRemoveManagers.sol#L152

Tool Used

Manual Review

Recommendation

```
150:         address addManager = _isAddManagerActiveForOwner(_troveId, _owner,
↪  ownershipChangeCount)
151:             ? _addManagerOf[_troveId]
-152:             : address(0);
+152:             : _owner;
```

Issue M-5: Troves on removed branches cannot be redeemed [RESOLVED]

Source: https://github.com/sherlock-audit/2026-02-vyro-finance-audit/issues/116

Summary

When governance removes a collateral branch via `removeCollateral`, normal redemptions become unavailable. This leaves no mechanism to externally wind down trove positions on a branch that may have been removed precisely because its collateral has become risky.

Vulnerability Detail

Removing a branch via `removeCollateral` pulls it from the `activeCollateralIds` array. Since `_getRedeemableCollateralIds` iterates over this array, normal redemptions through `CollateralRegistry.redeemCollateral` will no longer target that branch.

The natural fallback would be urgent redemptions, which are designed to allow fee-free 1:1 (plus bonus) redemptions to quickly unwind a branch. However, `urgentRedemption` is gated by `_requireIsShutDown()`:

```
function _requireIsShutDown() internal view {
    if (shutdownTime == 0) {
        revert NotShutDown();
    }
}
```

`shutdownTime` is only set when `BorrowerOperations` triggers a shutdown due to TCR falling below the critical threshold, or due to oracle failure. `removeCollateral` does not set `shutdownTime` and does not interact with the `TroveManager` at all, so `urgentRedemption` reverts on a removed but not shutdown branch.

The result is that after removal, troves on that branch exist in a limbo state: no one can redeem against them through any path. The only way positions get closed is if borrowers voluntarily do so, which may never happen, especially if they have no incentive to repay.

This directly contradicts the design intent, where branch removal should accelerate position unwinding. Instead, it makes it slower than if the branch were still active (where at least normal redemptions would work).

Impact

A removed collateral branch has no redemption path. If governance removed the branch because the collateral became risky (e.g. depegging, losing liquidity), the protocol

remains fully exposed to that collateral with no external mechanism to reduce it. This can lead to undercollateralization of the stablecoin if the collateral value continues to deteriorate while troves sit idle.

Code Snippet

https://github.com/sherlock-audit/2026-02-vyro-finance-audit/blob/main/vyro/contracts/src/TroveManager.sol#L958

Tool Used

Manual Review

Recommendation

Enable urgent redemptions when a branch has been removed. This can be done by having `removeCollateral` call `shutdown()` on the branch's `TroveManager`, or by updating the guard in `urgentRedemption` to also pass when the branch is no longer in the active set.

Issue L-1: Unnecessary BOLD approval to `CollateralRegistry` in `RedemptionHelper` [RESOLVED]

Source: https://github.com/sherlock-audit/2026-02-vyro-finance-audit/issues/110

Summary

`RedemptionHelper` sets a max approval of BOLD to the `CollateralRegistry` in its constructor, but the registry never pulls BOLD via `transferFrom`; it burns it directly from `msg.sender`.

Vulnerability Detail

In the constructor, `RedemptionHelper` grants an infinite BOLD allowance to the `CollateralRegistry`:

```
    constructor(ICollateralRegistry _collateralRegistry) {
        collateralRegistry = _collateralRegistry;
        boldToken = _collateralRegistry.boldToken();

        // Allow the registry to pull BOLD for redemptions without repeated
        ↪  approvals
>>      IERC20(address(boldToken)).forceApprove(address(_collateralRegistry),
↪   type(uint256).max);
    }
```

However, `CollateralRegistry.redeemCollateral` never calls `transferFrom`. It burns BOLD directly from `msg.sender` (the `RedemptionHelper`) at the end of the redemption:

```
if (totals.redeemedAmount > 0) {
    boldToken.burn(msg.sender, totals.redeemedAmount);
}
```

`BoldToken.burn` is a privileged function gated by `_requireCallerIsCRorBOorTMorSP`, and it doesn't check or consume any ERC-20 allowance.

Impact

Wasted gas on a `forceApprove` call during deployment. No functional impact.

Code Snippet

https://github.com/sherlock-audit/2026-02-vyro-finance-audit/blob/main/vyro/contracts/src/RedemptionHelper.sol#L34

Tool Used

Manual Review

Recommendation

Remove the unnecessary `forceApprove` call from the `RedemptionHelper` constructor.

Issue L-2: Hardcoded collateral gas compensation cap is too low for the UNI branch [RESOLVED]

Source: https://github.com/sherlock-audit/2026-02-vyro-finance-audit/issues/111

Summary

`COLL_GAS_COMPENSATION_CAP` is hardcoded to 2 ether as a global constant inherited from `Liquity`, where all collateral tokens (`WETH`, `RETH`, `WSTETH`) have similar USD value. Vyro Finance supports collateral tokens with vastly different prices (e.g. `UNI`), making a single fixed cap inadequate.

Vulnerability Detail

The collateral gas compensation cap is defined as a global constant:

```
// Fraction of collateral awarded to liquidator
uint256 constant COLL_GAS_COMPENSATION_DIVISOR = 200; // dividing by 200 yields 0.5%
uint256 constant COLL_GAS_COMPENSATION_CAP = 2 ether; // Max coll gas compensation
↪  capped at 2 ETH
```

This cap is consumed in `TroveManager._getCollGasCompensation`:

```
function _getCollGasCompensation(uint256 _coll) internal pure returns (uint256) {
    // _entireDebt should never be zero, but we add the condition defensively to
    ↪  avoid an unexpected revert
    return LiquityMath._min(_coll / COLL_GAS_COMPENSATION_DIVISOR,
    ↪  COLL_GAS_COMPENSATION_CAP);
}
```

In Liquity V2, this works fine because `WETH`, `RETH`, and `WSTETH` all trade in a similar range, so a cap of 2 tokens is reasonable across all branches. In this system, however, collateral tokens like `UNI` (~3)wouldhaveacapworth 6, while for `WETH` (~2000), thesamecapisworth 4000. This discrepancy means the cap is too restrictive for cheap tokens such as UNI, leading to underpaying liquidators in the `UNI` branch.

Impact

Liquidators receive inconsistent gas compensation across branches. For low-value collateral tokens such as UNI, the cap will be reached too easily, reducing the incentive to liquidate.

Code Snippet

https://github.com/sherlock-audit/2026-02-vyro-finance-audit/blob/main/vyro/contracts/src/Dependencies/Constants.sol#L41

Tool Used

Manual Review

Recommendation

Store the collateral gas compensation cap as a per-branch configurable parameter in each `TroveManager` (or its `AddressesRegistry`), set at deployment and updatable by governance. This allows each branch to use a cap that reflects the actual value of its collateral token.

Issue L-3: Tracking the Nonce Is Sufficient to Detect TroveNFT Ownership Changes [RESOLVED]

Source: https://github.com/sherlock-audit/2026-02-vyro-finance-audit/issues/115

Summary

The `_removeManagerOwnerOf` variable is not necessary. Since we only need to verify whether the owner of the TroveNFT has changed, tracking the nonce (_removeManagerOwnershipChangeCountOf) should be sufficient.

```
95:     function _setRemoveManagerOwner(uint256 _troveId, address _borrower)
↪  internal {
96:         _removeManagerOwnerOf[_troveId] = _borrower;
97:         _removeManagerOwnershipChangeCountOf[_troveId] =
↪  troveNFT.ownershipChangeCountOf(_troveId);
98:     }
```

Code Snippet

https://github.com/sherlock-audit/2026-02-vyro-finance-audit/blob/main/vyro/contracts/src/Dependencies/AddRemoveManagers.sol#L96

Tool Used

Manual Review

Recommendation

```
95:     function _setRemoveManagerOwner(uint256 _troveId, address _borrower)
↪  internal {
-96:         _removeManagerOwnerOf[_troveId] = _borrower;
97:         _removeManagerOwnershipChangeCountOf[_troveId] =
↪  troveNFT.ownershipChangeCountOf(_troveId);
98:     }

196:     function _isRemoveManagerActiveForOwner(uint256 _troveId, address _owner,
↪  uint256 _ownershipChangeCount)
197:         internal
198:         view
199:         returns (bool)
200:     {
-201:         return _removeManagerOwnerOf[_troveId] == _owner
```

```
-202:             && _removeManagerOwnershipChangeCountOf[_troveId] ==
 ↪  _ownershipChangeCount;
+             return _removeManagerOwnershipChangeCountOf[_troveId] ==
 ↪  _ownershipChangeCount;
203:     }
```

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.