yAudit

April 2026

Prepared for
Yearn

Audited by
fedebianu
Invader

Liquity V2 Lender Borrower Strategy

Smart Contract Security Assessment

Contents

1 Review Summary **2**
- 1.1 Protocol Overview . . . 2
- 1.2 Audit Scope . . . 2
- 1.3 Risk Assessment Framework . . . 2
  - 1.3.1 Severity Classification . . . 2
- 1.4 Key Findings . . . 3
- 1.5 Overall Assessment . . . 3

2 Audit Overview **3**
- 2.1 Project Information . . . 3
- 2.2 Audit Team . . . 3
- 2.3 Audit Timeline . . . 3
- 2.4 Audit Resources . . . 3
- 2.5 Critical Findings . . . 5
- 2.6 High Findings . . . 5
  - 2.6.1 Withdraw limit overstates liquidity when branch CCR headroom is tight . 5
- 2.7 Medium Findings . . . 6
  - 2.7.1 Swap failure in `_claimAndSellRewards()` blocks `_tend()` deleveraging 6
  - 2.7.2 Standard withdrawals can fail when deleveraging crosses Liquity minimum debt . . . 7
  - 2.7.3 Emergency withdraw can revert atomically when trove closure is unavailable 9
- 2.8 Low Findings . . . 10
  - 2.8.1 `adjustZombieTrove()` underflow when zombie debt exceeds `MIN_DEBT` 10
  - 2.8.2 `adjustZombieTrove()` tight coupling with lender deposit blocks zombie recovery . . . 11
  - 2.8.3 Dust BOLD deposits can revert in nested vault chain, bricking core strategy flows . . . 11
  - 2.8.4 `_tend()` reverts during recovery mode, blocking reward harvesting and rebalancing . . . 12
  - 2.8.5 Hard-coded $1 `BOLD` pricing can break debt buybacks . . . 14
  - 2.8.6 Liquidation-surplus collateral can be underreported until emergency claim, enabling limited cheap-share minting . . . 14
  - 2.8.7 Use of deprecated Chainlink function `latestAnswer()` . . . 16
  - 2.8.8 Shutdown does not stop keeper flows from re-leveraging idle funds . . . 16
- 2.9 Gas Savings Findings . . . 18
- 2.10 Informational Findings . . . 18
  - 2.10.1 Final unwind buyback can leave residual `BOLD` debt after a successful swap 18
  - 2.10.2 APR guardrails are disabled by hard-coded profitability assumptions . . . 19
- 2.11 Final Remarks . . . 20

1 Review Summary

1.1 Protocol Overview

A Yearn V3 tokenized strategy that opens a single Liquity V2 trove against a collateral asset, borrows BOLD, deploys borrowed BOLD into yBOLD and ysyBOLD, and rebalances leverage around a target LTV using Liquity operations and Curve swaps.

1.2 Audit Scope

This audit covers 2 contracts totaling approximately 700 lines of code across 5 days of review.

```
src/
├── BaseLenderBorrower.sol
└── Strategy.sol
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Increased transaction costs
Informational	Code quality and best practice recommendations	Reduced maintainability and readability

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	3
Low	8
Informational	2

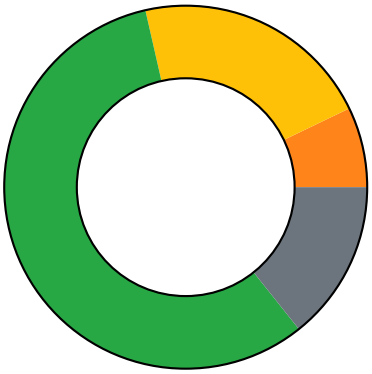

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The codebase is compact and generally well-structured, but it relies on several cross-protocol assumptions across Liquity, Yearn ERC-4626 wrappers, and Curve execution. The main risk areas are stressed-state exit semantics, emergency unwind behavior, and maintenance-path coupling.

2 Audit Overview

2.1 Project Information

Protocol Name: Yearn
Repository: https://github.com/johnnyonline/yv3-liquityv2-lender-borrower-strategy
Commit Hash: 50e86c58bf8f76241503b7987cba5fbe33892a0c
Commit URL: https://github.com/johnnyonline/yv3-liquityv2-lender-borrower-strategy/commit/50e86c58bf8f76241503b7987cba5fbe33892a0c

2.2 Audit Team

fedebianu, Invader

2.3 Audit Timeline

The audit was conducted from March 26 to April 01, 2026.

2.4 Audit Resources

Code repositories

Category	Mark	Description
Access Control	Good	Role separation is clear across management, keeper, and emergency-authorized actors.
Mathematics	Good	Core arithmetic is straightforward.
Complexity	Average	The local codebase is not large, but the strategy combines Liquity leverage management, nested Yearn vault interactions, and Curve execution, which raises integration complexity.
Libraries	Good	The strategy relies on mature OpenZeppelin and Yearn primitives.
Decentralization	Average	The system depends on privileged management, keeper, and emergency roles.
Code Stability	Good	The core architecture is focused and relatively mature, although several findings arise from stressed-state logic, operational assumptions, and edge-case handling.
Documentation	Average	README, comments, and tests provide useful context, but several important assumptions around lender liquidity, price feeds, and emergency semantics are not fully documented.
Monitoring	Average	The strategy benefits from the standard Yearn tokenized-strategy event surface, but the scoped contracts add little strategy-specific signaling for stressed-state transitions, recovery paths, or edge-case failures.
Testing and verification	Good	The fork-based suite covers many happy paths and several stressed flows, but some integration edge cases and degraded-liquidity scenarios were not explicitly exercised.

Table 2: Code Evaluation Matrix

2.5 Critical Findings

None.

2.6 High Findings

2.6.1 Withdraw limit overstates liquidity when branch CCR headroom is tight

The strategy advertises withdrawable liquidity through `maxWithdraw()` and `maxRedeem()` that it cannot actually free because `BaseLenderBorrower.availableWithdrawLimit()` ignores the branch `CCR` constraint enforced by `Strategy._maxWithdrawal()`. Under `TCR < CCR`, a strict `withdraw()` path can revert, while Yearn's default `redeem()` path can instead burn shares, repay strategy debt, and return little or no collateral to the withdrawing user.

Technical Details

The write path is `CCR`-aware:

- `Strategy._maxWithdrawal()` caps collateral release by branch headroom above `CCR`;
- when branch `TCR < CCR`, that headroom becomes `0`.

That collateral lock is imposed by Liquity and is expected behavior at the protocol layer; the strategy cannot and should not bypass it.
The view path is not:

- `BaseLenderBorrower.availableWithdrawLimit()` ignores `Strategy._maxWithdrawal()`;

Yearn's `maxWithdraw()` and `maxRedeem()` trust `availableWithdrawLimit()`. During `BaseLenderBorrower._liquidatePosition()` the strategy first withdraws `BOLD` from the lender and repays debt, then attempts to free collateral. If `_maxWithdrawal()` is `0`, the debt reduction succeeds but the collateral withdrawal does not. A loss-intolerant path such as `withdraw(..., maxLoss)` can therefore revert, while the default three-argument `redeem()` accepts full loss and can complete as a debt-only exit. The repository's own tests already show this stressed-state behavior.

Impact

High. During `TCR < CCR`, a withdrawing user can lose most or all redemption value while the strategy still repays shared debt for the benefit of remaining shareholders and continues to overstate exit liquidity to integrators and UIs.

Recommendation

Make `availableWithdrawLimit()` a conservative upper bound on what can actually be freed. In practice:

- incorporate the same `CCR`-aware cap used by `_maxWithdrawal()`; or
- revert cleanly in states where debt can be repaid but collateral cannot be released.

The strategy should not socialize debt repayment onto the withdrawing user without releasing collateral.

Developer Response

Acknowledged. They can withdraw, but with a loss. Users should use Yearn's `maxLoss`.

2.7 Medium Findings

2.7.1 Swap failure in `_claimAndSellRewards()` blocks `_tend()` deleveraging

Technical Details

`Strategy._tend()` unconditionally calls `_claimAndSellRewards()` before delegating to `BaseLenderBorrower._tend()`:

```
function _tend(uint256 /*_totalIdle*/) internal override {
  _claimAndSellRewards();
  return BaseLenderBorrower._tend(balanceOfAsset());
}
```

`_claimAndSellRewards()` calls `_sellBorrowToken()`, which executes a swap with a `_minAmountOut` slippage check. If the swap returns less than the configured tolerance, the entire `_tend()` call reverts, including the downstream rebalancing logic that may need to deleverage the position.
This coupling is dangerous because swap failures are most likely during volatile market conditions, which are exactly when deleveraging is most critical. A pool imbalance or low-liquidity event prevents the strategy from reducing its leverage, even when the position approaches liquidation thresholds.

Impact

Medium. The coupling between selling and position rebalancing means the strategy can fail to deleverage when the swap fails.

Recommendation

Decouple reward selling from rebalancing. Either wrap `_claimAndSellRewards()` in a try-catch, or skip selling when the position needs rebalancing.

Developer Response

Acknowledged. This sale is critical in the case of a redemption, so it's part of a rebalance. Either way, slippage can be adjusted if it fails.

2.7.2 Standard withdrawals can fail when deleveraging crosses Liquity minimum debt

Technical Details

The strategy's standard withdrawal flow can fail when the next deleveraging step would push the trove's remaining debt below Liquity's `2,000 BOLD` minimum debt threshold but above zero. The protocol's documented behavior prevents partial repayments that would leave debt in the forbidden `0 < debt < MIN_DEBT` range unless the trove is fully closed.
The issue manifests in the withdrawal sequence. When a user requests a withdrawal that requires deleveraging, `BaseLenderBorrower._liquidatePosition()` first computes a target repayment amount by calling `BaseLenderBorrower._calculateAmountToRepay()`. This function derives a partial repayment intended to preserve the strategy's target LTV after the collateral withdrawal, with no awareness of the Liquity minimum debt boundary. The function then withdraws that amount of `BOLD` from the lender stack via `BaseLenderBorrower._withdrawFromLender()` and attempts to repay it by calling `BaseLenderBorrower._repayTokenDebt()`, which delegates to `Strategy._repay()`. The inline comment in `Strategy` explicitly states that `repayBold()` enforces the `MIN_DEBT` rule.
When the computed partial repayment would leave residual debt below `MIN_DEBT`, Liquity's `repayBold()` call does not visibly revert; instead, it adjusts the actual repayment to stop exactly at `MIN_DEBT`. The trove is then left at that minimum debt level. The subsequent call to `Strategy._withdrawCollateral()` can only free a reduced amount of collateral because the strategy cannot further deleverage without crossing into the forbidden range. The tokenized-strategy layer then reverts the entire withdrawal with "too much loss" because the collateral freed is less than what the user requested.
The fallback branch that buys `BOLD` and performs a full unwind is gated by the condition `balanceOfLentAssets() == 0`. This path is not reached in the described scenario because `BOLD` remains deployed in the lender stack when the partial repayment gets pinned at `MIN_DEBT`. The strategy has no logic to detect that the next repayment step crosses `MIN_DEBT` and should therefore switch to a full close path instead of a partial repay.

Impact

Medium. Users requesting withdrawals through the standard path may find their transactions reverting even when the strategy holds sufficient assets to satisfy the withdrawal economically. The failure occurs when the position is near the minimum debt boundary and a partial deleveraging step would cross it. Collateral becomes operationally locked behind this protocol-threshold edge case until a privileged operator uses emergency functions to manually close or repair the trove. The issue does not allow fund theft, but it degrades withdrawal availability during normal deleveraging operations, which are the moments when reliable exits matter most to users. Funds remain recoverable through alternative privileged paths or after manual intervention.

PoC
Add this function to the test file `src/test/poc/MinDebtWithdrawalStuck.t.sol`:

```
function test_poc_partialWithdrawRevertsWhenRepayWouldCrossLiquityMinDebt() public {
  strategistDepositAndOpenTrove(true);
```

```
    // Bring the trove from its initial MIN_DEBT borrow up to the strategy's target LTV.
    vm.prank(keeper);
    strategy.tend();

    uint256 currentCollateral = strategy.balanceOfCollateral();
    uint256 currentDebt = strategy.balanceOfDebt();
    uint256 targetLTV = (strategy.getLiquidateCollateralFactor() * strategy.targetLTVMultiplier()) / MAX_BPS;

    assertGt(currentDebt, MIN_DEBT, "setup: debt must be above MIN_DEBT");
    assertGt(strategy.balanceOfLentAssets(), 0, "setup: lender position must still exist");


    // Choose a partial withdrawal that leaves the position with a positive target debt,
    // but one that is strictly below Liquity's MIN_DEBT.
    uint256 desiredRemainingDebt = MIN_DEBT - 50 ether;
    uint256 desiredRemainingDebtUsd = (desiredRemainingDebt * 1e8) / 1e18;
    uint256 desiredCollateralUsd = (desiredRemainingDebtUsd * 1e18) / targetLTV;
    uint256 assetPrice = uint256(AggregatorInterface(strategy.PRICE_FEED()).latestAnswer());
    uint256 collateralToLeave = (desiredCollateralUsd * 1e18) / assetPrice;
    uint256 withdrawAmount = currentCollateral - collateralToLeave;

    uint256 advertisedMaxWithdraw = strategy.maxWithdraw(strategist, 0);
    assertGt(withdrawAmount, 0, "setup: withdraw amount must be non-zero");
    assertLt(withdrawAmount, currentCollateral, "setup: must be a partial withdraw");
    assertGe(advertisedMaxWithdraw, withdrawAmount, "setup: strategy advertises this as withdrawable");

    uint256 newCollateralUsd = ((currentCollateral - withdrawAmount) * assetPrice) / 1e18;
    uint256 projectedDebtAfterRepay = (((newCollateralUsd * targetLTV) / 1e18) * 1e18) / 1e8;

    assertGt(projectedDebtAfterRepay, 0, "setup: projected residual debt should stay positive");
    assertLt(projectedDebtAfterRepay, MIN_DEBT, "setup: projected residual debt must fall below MIN_DEBT");

    console2.log("advertised maxWithdraw", advertisedMaxWithdraw);
    console2.log("current collateral", currentCollateral);
    console2.log("current debt", currentDebt);
    console2.log("requested partial withdraw", withdrawAmount);
    console2.log("projected residual debt", projectedDebtAfterRepay);

    vm.expectRevert(bytes("too much loss"));
    vm.prank(strategist);
    strategy.withdraw(withdrawAmount, strategist, strategist, 0);

    // The standard user withdrawal path is now stuck even though the strategy remains solvent.
    // Liquity keeps the trove pinned at MIN_DEBT, so the strategy cannot free the full
    // requested collateral and the tokenized-strategy layer reverts the withdrawal.
    assertEq(strategy.balanceOfCollateral(), currentCollateral, "collateral unchanged after revert");
    assertEq(strategy.balanceOfDebt(), currentDebt, "debt unchanged after revert");
    assertGt(strategy.totalAssets(), 0, "funds remain in the strategy after the failed exit");
}
```

The PoC demonstrates that the strategy reports `maxWithdraw = 2 ETH`, but a normal partial withdrawal of `0.964335711307090681 ETH` fails once the implied post-deleverage debt would be `1949.99999998 BOLD`, i.e. below Liquity's `MIN_DEBT`. The withdrawal path reverts,

while collateral and debt remain unchanged in the strategy, showing user exits can get stuck without privileged intervention.

Recommendation

Make the withdrawal path explicitly aware of Liquity's `MIN_DEBT` boundary. Before repaying debt during partial deleveraging, compute the post-repayment residual debt and disallow any branch that would leave `0 < remainingDebt < MIN_DEBT`. In that case, either clamp repayment so the trove remains at or above `MIN_DEBT`, or require a full unwind if enough `BOLD` is available to close the trove entirely. Then apply the same rule to `availableWithdrawLimit()` and `maxWithdraw()` so the strategy does not advertise zero-loss liquidity that the standard path cannot realize.

Developer Response

Acknowledged. True, I forgot to mention that strategy management should hold a baseline amount of assets to prevent that.

2.7.3 Emergency withdraw can revert atomically when trove closure is unavailable

`Strategy._emergencyWithdraw()` overrides the base emergency unwind with an all-or-nothing trove closure path. If `TroveOps.onEmergencyWithdraw()` reaches `closeTrove()` when full closure is unavailable, the entire `emergencyWithdraw()` call reverts and rolls back any lender withdrawal performed earlier in the transaction.

Technical Details

The base implementation in `BaseLenderBorrower._emergencyWithdraw()` is resilient:

- withdraw what is available from the lender;
- repay as much debt as possible;
- withdraw whatever collateral can safely be released.

`Strategy._emergencyWithdraw()` replaces that logic with:

1. optional lender withdrawal of `BOLD`;
2. immediate call to `TroveOps.onEmergencyWithdraw(...)`.

For active troves, `TroveOps.onEmergencyWithdraw()` unconditionally calls `closeTrove()`. That call can revert when:

- the strategy does not hold enough loose `BOLD` to cover the full debt; or
- Liquity blocks closure because branch `TCR < CCR`.

Because the entire flow is atomic, any earlier lender withdrawal is reverted as well. The strategy therefore loses even the partial-unwind behavior that the base contract already provided.

Impact

Medium. When full trove closure is unavailable, `emergencyWithdraw()` reverts instead of partially unwinding, leaving funds trapped in active strategy state and forcing operators into manual recovery during stress.

Recommendation

Do not make the emergency unwind path depend atomically on successful trove closure: if closeTrove is unavailable, preserve any `BOLD` withdrawn from the lender and any debt repayment already performed, and defer collateral release until it becomes possible.

Developer Response

Acknowledged.

2.8 Low Findings

2.8.1 `adjustZombieTrove()` underflow when zombie debt exceeds `MIN_DEBT`

Technical Details

`TroveOps.adjustZombieTrove()` computes the BOLD change needed to bring a zombie trove back to `MIN_DEBT`:

```
MIN_DEBT - _balanceOfDebt, // boldChange
```

This assumes the zombie trove's debt is below MIN*DEBT. However, zombie troves can accrue interest and receive debt redistributions from other troves' liquidations, potentially pushing their debt back above MIN*DEBT while remaining in zombie status.
This permanently blocks zombie recovery via `adjustZombieTrove()`, locking the trove's collateral until an alternative recovery path is used.

Impact

Low. The fix is trivial and means the strategy isn't reliant on alternative manual intervention.

Recommendation

```
MIN_DEBT > _balanceOfDebt ? MIN_DEBT - _balanceOfDebt : 0, // boldChange
```

Developer Response

Fixed in commit 99ba1e8.

2.8.2 `adjustZombieTrove()` tight coupling with lender deposit blocks zombie recovery

Technical Details

`adjustZombieTrove()` is an emergency-authorized function meant to recover a zombified trove after redemption. After adjusting the trove, it unconditionally sweeps all loose BOLD into the lender:

```
function adjustZombieTrove(uint256 _upperHint, uint256 _lowerHint) external
onlyEmergencyAuthorized {
  TroveOps.adjustZombieTrove(BORROWER_OPERATIONS, troveId, balanceOfAsset(),
balanceOfDebt(), _upperHint, _lowerHint);
  _lendBorrowToken(balanceOfBorrowToken());
}
```

If the lender vault has been shut down when the operator attempts zombie recovery, `_lendBorrowToken()` reverts, blocking the entire operation. Emergency recovery functions should be decoupled from non-essential external dependencies — the lend is a nice-to-have optimization, not a prerequisite for trove recovery.

Impact

Low. The scenario requires both a zombified trove and a constrained lender vault simultaneously.

Recommendation

Remove the lend call from the emergency path. Let the next `_tend()` handle lending idle BOLD.

Developer Response

Acknowledged.

2.8.3 Dust BOLD deposits can revert in nested vault chain, bricking core strategy flows

Technical Details

`LenderOps.lend()` chains two ERC-4626 deposits in a single expression:

```
STAKED_LENDER_VAULT.deposit(LENDER_VAULT.deposit(_amount, address(this)), address(this));
```

When the strategy accumulates dust BOLD (e.g., 1 wei from swap rounding or a direct donation), `LENDER_VAULT.deposit(1 wei)` returns 0 shares once the vault's share price has appreciated above 1:1. The Yearn V3 vault's `_convert_to_shares()` rounds down: `1 * total_supply / total_assets` truncates to 0 when

`total_assets > total_supply`. The vault's `_deposit()` then hits the explicit assertion `assert shares > 0, "cannot mint zero"` and reverts.
This is confirmed by the Yearn V3 vault implementation (vault.vy):

```
# _convert_to_shares (line 464)
if assets == max_value(uint256) or assets == 0:
    return assets
# ...
shares: uint256 = numerator / total_assets  # rounds down to 0 for dust

# _deposit (line 652)
assert shares > 0, "cannot mint zero"  # reverts
```

`_leveragePosition()` at BaseLenderBorrower.sol:459-460 calls `_lendBorrowToken(borrowTokenBalance)` whenever `borrowTokenBalance > 0`, but does not check whether that amount would produce nonzero shares in the nested vault chain:

```
uint256 borrowTokenBalance = balanceOfBorrowToken();
if (borrowTokenBalance > 0) _lendBorrowToken(borrowTokenBalance);
```

Both preconditions are guaranteed by the vault code: dust deposits produce 0 shares via truncating division, and the vault explicitly reverts on 0 shares. A 1 wei BOLD donation bricks `_leveragePosition`, `_tend`, and `_harvestAndReport` until the dust is manually removed.

Impact

Low. All core strategy flows that terminate in `_leveragePosition` become unusable. The revert is guaranteed once the vault's share price exceeds 1:1, which is the normal state for any yield-bearing vault.

Recommendation

Preview the deposit output before committing:

```
function lend(uint256 _amount) external {
  uint256 shares = LENDER_VAULT.previewDeposit(_amount);
  if (shares > 0 && STAKED_LENDER_VAULT.previewDeposit(shares) > 0) {
    STAKED_LENDER_VAULT.deposit(LENDER_VAULT.deposit(_amount, address(this)), address(
this));
  }
}
```

Developer Response

Acknowledged. Don't think it's worth the gas.

2.8.4 `_tend()` reverts during recovery mode, blocking reward harvesting and rebalancing

`report()` and a specific `tend()` path can still reach `_leveragePosition(0)` even after the strategy has concluded that no new deployment is allowed.

Technical Details

`BaseLenderBorrower._leveragePosition()` does not use `_amount` as a guard on its borrowing branch. After the initial `_supplyCollateral(_amount)` call, it recomputes the full current position and, whenever `currentLTV < targetLTV`, it derives a fresh `amountToBorrowBT` from total collateral and debt:

- it computes `targetDebtUsd` from current collateral;
- converts the shortfall into `amountToBorrowBT`;
- and calls `_borrow(amountToBorrowBT)` whenever that amount exceeds `minAmountToBorrow`.

The function does not check whether deployment is currently paused. In this strategy, borrowing pause is surfaced through `Strategy._isBorrowPaused()`, and `BaseLenderBorrower.availableDepositLimit()` correctly returns `0` when borrowing is paused. However, passing `0` into `_leveragePosition()` does not stop the borrow branch from executing.
Under branch stress such as `TCR < CCR`, Liquity blocks new debt through `withdrawBold()`. In any active-trove state where the strategy is below target LTV while borrowing is paused, `report()` or any `tend()` path that still reaches `_leveragePosition(0)` can therefore proceed to `_borrow()` and revert instead of simply leaving idle collateral undeployed. `BaseLenderBorrower._tendTrigger()` does gate the regular business-as-usual tend path on `_isBorrowPaused()`. The root cause here is narrower: once a maintenance path has already concluded that the deployable amount is zero, `_leveragePosition(0)` can still recompute and attempt a fresh borrow from the full position.

Impact

Low. When borrowing is paused but the trove remains active and under-levered, `report()` or any `tend()` path that still reaches `_leveragePosition(0)` can revert by attempting a fresh borrow instead of safely skipping redeployment.

Recommendation

Treat a zero deployment amount as a hard stop for new borrowing. In practice, either return early from `BaseLenderBorrower._leveragePosition()` when `_amount == 0` and borrowing is paused, or make the borrow branch explicitly respect the same pause conditions already encoded in `availableDepositLimit()` and `Strategy._isBorrowPaused()`. If the strategy has already concluded that no deployment is allowed, `report()` or the subsequent `tend()` path should skip redeployment rather than attempting a fresh borrow anyway.

Developer Response

Acknowledged.

2.8.5 Hard-coded $1 `BOLD` pricing can break debt buybacks

`Strategy._getPrice()` hard-codes `BOLD` to `$1`. That nominal price is appropriate for Liquity collateral-ratio math, but the strategy also reuses it in the debt buyback path. If `BOLD` trades above peg, `Strategy._buyBorrowToken()` can underestimate how much collateral must be sold and can set an unattainable `minAmountOut`, causing unwind buybacks to fail.

Technical Details

The fixed `BOLD = 1e8` assumption flows into:

- `BaseLenderBorrower._borrowTokenOwedInAsset()`, which estimates how much collateral is needed to cover debt;
- `Strategy._buyBorrowToken()`, which derives expected output and `minAmountOut` for the swap.

In the buyback path, the strategy needs a live market or route quote, not a nominal accounting value. If `BOLD` is above peg for any reason, the expected `BOLD` output is overstated and the swap can underfill the remaining debt.

Impact

Low. If `BOLD` trades above peg, debt buybacks can revert or underfill, leaving debt outstanding.

Recommendation

The strategy should size the collateral to sell from a current market quote for the outstanding BOLD debt, cap spend by the collateral actually available, and verify after execution whether the debt was fully covered. Because this operation is expected to run through a private RPC, using a live quote here is an appropriate way to size the swap correctly and avoid leaving residual BOLD debt due to ordinary slippage or temporary off-peg conditions.

Developer Response

Acknowledged.

2.8.6 Liquidation-surplus collateral can be underreported until emergency claim, enabling limited cheap-share minting

Technical Details

The strategy manages a Liquity trove through a tokenized vault interface. When the trove is liquidated, the `TROVE_MANAGER` closes it with status `closedByLiquidation`, and any remaining collateral after debt repayment is sent to the `COLL_SURPLUS_POOL` for later claim. However, the strategy's accounting flow does not include this claimable surplus in normal reporting.

In `Strategy._leveragePosition()`, the function returns immediately when the trove is not active, preventing normal redeployment of deposits. The base class's `BaseLenderBorrower._harvestAndReport()` calculates total assets as `balanceOfAsset() + balanceOfCollateral() - _borrowTokenOwedInAsset()`. For a `closedByLiquidation` trove, `Strategy.balanceOfCollateral()` reads the active trove balance, which is zero after liquidation. The surplus collateral sitting in `COLL_SURPLUS_POOL.getCollateral(address(this))` is only recovered through the emergency withdrawal path in `TroveOps.onEmergencyWithdraw()`, not through standard reports.
Meanwhile, `BaseLenderBorrower.availableDepositLimit()` checks `Strategy._isSupplyPaused()` and `Strategy._isBorrowPaused()` but does not gate on trove status. If the protocol is not shut down, `availableDepositLimit()` can return nonzero even when the trove is inactive. This creates a window where:

1. The trove is liquidated with surplus collateral in the pool.
2. A report runs before shutdown or emergency claim, recording understated `totalAssets`.
3. Deposits remain open because `_isSupplyPaused()` returns false.
4. An allowlisted depositor can mint shares against the understated asset base.

When the surplus is later claimed via emergency withdrawal, the newly realized collateral increases `totalAssets`, but the shares minted during underreporting have already locked in a discounted entry price. The value difference is effectively transferred from existing holders to the new depositor.

Impact

Low. Allowlisted depositors can mint shares at an artificially low price during the reporting window between liquidation and emergency claim. The loss is bounded by the unclaimed surplus collateral for that specific strategy instance. Existing vault participants bear the dilution when the omitted surplus is later realized and the accounting corrects. The TokenizedStrategy framework immediately increases stored `totalAssets` on deposit, so idle deposits themselves are not lost; the misaccounting is limited to the narrow liquidation-surplus scenario.

Recommendation

To address the root cause, when the trove status is `closedByLiquidation`, include `COLL_SURPLUS_POOL.getCollateral(address(this))` in `BaseLenderBorrower._harvestAndReport()` so economically recoverable surplus collateral is reflected in `totalAssets` before new shares are minted. Alternatively, `BaseLenderBorrower.availableDepositLimit()` can return zero whenever the trove is non-active to prevent new deposits while liquidation surplus remains unaccounted for.

Developer Response

Acknowledged.

2.8.7 Use of deprecated Chainlink function `latestAnswer()`

Strategy default price feed and the custom `WSTETHPriceFeed` and `RETHPriceFeed` wrappers read only `latestAnswer()` and do not validate staleness or positivity of the underlying answers. While the comment states that it uses "the same one Liquity uses", the strategy bypasses Liquity's oracle wrapper and its built-in staleness/fallback logic.

Technical Details

`AggregatorInterface` exposes `latestRoundData()`, including `updatedAt`, but the wrapper feeds compose prices using only `latestAnswer()`. As a result:

- stale source data can be trusted indefinitely
- zero answers can collapse pricing
- negative answers can wrap into a very large `uint256` after casting

Those prices are used in LTV checks, `_maxWithdrawal()`, swap minimum-out calculations, and accounting paths.

Impact

Low. Bad oracle data can materially distort strategy behavior:

- leverage and deleverage logic can use the wrong price
- withdrawals can be misquoted
- swap slippage protection can become meaningless
- accounting and safety checks can misfire during stress

Recommendation

Use `latestRoundData()` for Chainlink reads, enforce freshness windows, require positive answers on each underlying feed, and reject invalid composed prices before converting them to `uint256`.

Developer Response

Acknowledged. Liquity already does the checks and will shut down if any of them fail.

2.8.8 Shutdown does not stop keeper flows from re-leveraging idle funds

`shutdownStrategy()` does not actually freeze leverage. After shutdown, keepers can still call `tend()` or `report()` and cause the strategy to redeploy idle collateral and borrow again through `BaseLenderBorrower._tend()` and `BaseLenderBorrower._harvestAndReport()`.

Technical Details

Yearn's base contracts explicitly document that post-shutdown maintenance should avoid redeploying funds. This implementation does not enforce that expectation:

- `BaseLenderBorrower._harvestAndReport()` always calls `_leveragePosition(...)`;
- `BaseLenderBorrower._tend()` also calls `_leveragePosition(...)` unless it exits through the negative-carry branch;
- `Strategy.availableDepositLimit(address(this))` explicitly allows self-deployment even though user deposits are blocked;
- the strategy's pause helpers only consider Liquity branch conditions, not the Yearn shutdown flag.

As long as the trove is still active and the strategy has loose collateral or can first sell surplus `BOLD` into collateral, a keeper can re-risk the position after shutdown.

Impact

Low. Because shutdown does not stop `tend()` and `report()` from re-leveraging idle funds, keepers can recreate debt after an intended unwind and keep users exposed during incident response.

PoC
Add this function to the test file `src/test/Exchange.t.sol`:

```
function test_PoC() public {
  uint256 strategistDeposit = strategistDepositAndOpenTrove(true);
  uint256 idleAmount = 1 ether;

  assertGt(strategistDeposit, 0, "trove should be active");

  vm.prank(emergencyAdmin);
  strategy.shutdownStrategy();
  assertTrue(strategy.isShutdown(), "shutdown precondition not met");

  airdrop(asset, address(strategy), idleAmount);
  uint256 idleBefore = strategy.balanceOfAsset();
  uint256 collateralBefore = strategy.balanceOfCollateral();
  uint256 debtBefore = strategy.balanceOfDebt();

  assertGt(idleBefore, 0, "strategy needs idle collateral after shutdown");
  assertGt(strategy.availableDepositLimit(address(strategy)), 0, "self-call deposit limit remains open after shutdown");

  vm.prank(keeper);
  strategy.tend();

  assertLt(strategy.balanceOfAsset(), idleBefore, "keeper should redeploy idle collateral even after shutdown");
  assertGt(strategy.balanceOfCollateral(), collateralBefore, "shutdown did not stop collateral redeployment");
  assertGt(strategy.balanceOfDebt(), debtBefore, "shutdown did not stop fresh borrowing");
}
```

Recommendation

Short-circuit all redeployment paths when the Yearn shutdown flag is set. Concretely:

- skip `_leveragePosition(...)` in `_harvestAndReport()` after shutdown;
- skip `_leveragePosition(...)` in `_tend()` after shutdown;
- optionally return `0` from `availableDepositLimit(address(this))` while shutdown is active.

Developer Response

Acknowledged.

2.9 Gas Savings Findings

None.

2.10 Informational Findings

2.10.1 Final unwind buyback can leave residual `BOLD` debt after a successful swap

Technical Details

The final unwind branch in `BaseLenderBorrower._liquidatePosition()` is meant to buy enough `BOLD` to clear any remaining debt once lender assets are exhausted and `leaveDebtBehind` is false.
The sizing logic in `Strategy._buyBorrowToken()` does two separate things:

- it sizes the asset input at par by converting `borrowTokenOwedBalance()` into collateral terms via `_fromUsd(_toUsd(...))`;
- it accepts swap outputs down to `allowedSwapSlippageBps / MAX_BPS` of the expected amount through `_minAmountOut`.

That means the strategy can deliberately submit exactly enough collateral to buy `100%` of the remaining debt at par while simultaneously accepting a successful swap that returns less than `100%` of that debt. This issue is independent from `BOLD` depegging: even if `BOLD` is exactly `$1`, any nonzero allowed slippage is enough to create a shortfall.
In the final unwind sequence:

1. `BaseLenderBorrower._liquidatePosition()` calls `Strategy._buyBorrowToken()`;
2. the swap succeeds because it returns at least `_minAmountOut`;
3. `BaseLenderBorrower._repayTokenDebt()` repays only the `BOLD` actually bought;
4. residual debt remains open;
5. the subsequent collateral withdrawal attempt still sees nonzero debt, so the strategy cannot unlock the full intended collateral.

Impact

Informational.

Recommendation

Do not assume that a single par-sized swap will fully clear the residual debt. After each buyback, verify whether the acquired `BOLD` is sufficient to cover the remaining shortfall and either retry with the updated deficit or exit explicitly with residual debt still outstanding. As an optional optimization, the strategy may gross up the input amount by the configured slippage factor, but that should be treated as a tradeoff to reduce retry likelihood rather than as the primary correctness fix.

Developer Response

Acknowledged.

2.10.2 APR guardrails are disabled by hard-coded profitability assumptions

The base lender-borrower framework contains profitability guardrails that should stop new borrowing or trigger deleveraging when carry turns negative. `Strategy` disables those guardrails by hard-coding `getNetBorrowApr()` to `0` and `getNetRewardApr()` to `1`, so leverage decisions no longer reflect real borrow cost or reward conditions.

Technical Details

`BaseLenderBorrower` uses the APR getters in two critical places:

- `_leveragePosition()` cancels new borrowing when borrow APR exceeds reward APR;
- `_tend()` deleverages when carry is negative.

In this strategy:

- `getNetBorrowApr(uint256)` always returns `0`;
- `getNetRewardApr(uint256)` always returns `1`.

That makes both checks permanently evaluate as ”profitable”. The repository also includes `StrategyAprOracle`, which computes a real borrow-vs-reward spread, but that oracle is not used by the strategy’s on-chain decision logic. By disabling the base APR guardrails, the strategy can keep borrowing and avoid deleveraging during negative-carry conditions, socializing avoidable fee drag and performance loss to users.

Impact

Informational.

Recommendation

Document the design choice.

Developer Response

Acknowledged.

2.11 Final Remarks

The strategy is strongest when Liquity branch conditions are healthy, the lender stack behaves as expected, and keeper maintenance can rebalance positions without hitting stressed-state protocol restrictions. Operational assumptions become much more important near CCR, MIN_DEBT, emergency unwind paths, and buyback flows that depend on external swap execution. The main issues identified cluster around stressed-state exit semantics and maintenance-path edge cases.