yAudit

March 2026

Prepared for
Flex

Audited by
HHK
adriro

Flex

Smart Contract Security Assessment

Contents

1 Review Summary

1.1 Protocol Overview

Flex is a fixed-rate money market where borrowers choose their own interest rate. Borrowers open Troves by locking collateral, selecting a fixed annual interest rate, and taking borrow-token debt. Lenders deposit borrow tokens into a Yearn V3-style vault and earn interest, upfront fees, and liquidation surplus. Liquidity exits are maintained through redemptions: when lender liquidity is insufficient, the protocol sells lower-rate borrowers' collateral through a Dutch auction and routes proceeds to the withdrawing lender or redeemer.

1.2 Audit Scope

This audit covers 11 smart contracts totaling approximately 1800 lines of code across 7 days of review.

```
src
├── auction.vy
├── dutch_desk.vy
├── factory.vy
├── lender
│   ├── Lender.sol
│   └── LenderFactory.sol
├── oracles
│   └── yvusd_to_usdc_oracle.vy
├── periphery
│   ├── daddy.vy
│   └── leverage_zapper.vy
├── registry.vy
├── sorted_troves.vy
└── trove_manager.vy
```

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	7
Informational	9

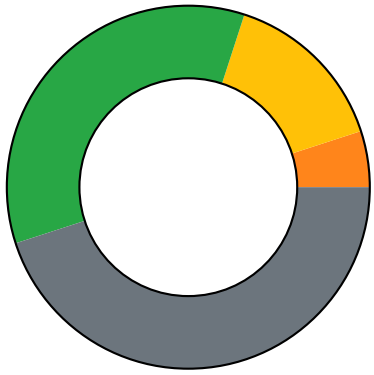

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The architecture is solid overall, but given the complexity and novelty of some features, several issues were uncovered.

2 Audit Overview

2.1 Project Information

Protocol Name: Flex
Repository: https://github.com/flexmeow/flex-contracts
Commit Hash: d1a59ba0d33eacbc5e65ff70f52858afdf01210e

2.2 Audit Team

HHK, adriro

2.3 Audit Timeline

The audit was conducted from March 23 to 31, 2026.

2.4 Audit Resources

- Code repositories and documentation

Category	Mark	Description
Access Control	Average	A high-severity issue was found where an unvalidated arbitrary call in the LeverageZapper allowed trove theft. Core protocol contracts otherwise enforce proper ownership and caller checks.
Mathematics	Average	Multiple issues affecting fee calculations and interest accounting were identified, including stale debt reads and incorrect weighted debt offsets in the upfront fee logic.
Complexity	Average	The protocol spans trove management, Dutch auctions, a Yearn V3-style lending vault, sorted troves, and a leverage zapper, creating meaningful cross-contract and asynchronous flow complexity. Multiple medium-severity findings arose from these interactions, particularly around redemption and trove ordering.
Libraries	Good	The Lender vault integrates with the Yearn V3 TokenizedStrategy framework, and the codebase uses standard ERC-20 interfaces. A minor mismatch with the BaseHealthCheck one-time bypass behavior was identified but does not affect core functionality.
Decentralization	Good	Markets are permissionless and can be deployed by anyone through the factory. A managed registry exists to endorse markets but does not restrict their creation or usage.
Documentation	Average	Documentation is correct in general, but several informational issues suggest that certain edge cases and assumptions, such as async delivery semantics and deployment parameter constraints, would benefit from clearer documentation.
Testing and verification	Good	The codebase includes a well-structured testing suite with good coverage across the core protocol modules.

Table 2: Code Evaluation Matrix

2.5 Critical Findings

None.

2.6 High Findings

2.6.1 Arbitrary `raw_call` in LeverageZapper `_swap` allows trove theft

An attacker can steal any unclaimed trove opened through the LeverageZapper by exploiting the unvalidated `raw_call` in `_swap()`.

Technical Details

`_swap()` executes an arbitrary call with no validation on the target address or call selector:

```
raw_call(swap.router, swap.data)  # no whitelist, no selector check
```

After `open_leveraged_trove()` (as well as trove leverage changes), the zapper owns the newly created trove until the user calls `accept_ownership()`. During this window, an attacker can call `open_leveraged_trove()` with crafted swap data such as `abi.encode(transfer_ownership(victim_trove_id, attacker))`.
The "swap" then executes `transfer_ownership()` on the TroveManager. Since `msg.sender` is the zapper and the zapper is the trove's owner, the call succeeds. The victim's `pending_owner` is overwritten to the attacker, who then calls `accept_ownership()` to steal the trove.
The attacker only needs enough collateral to open a minimum trove to make the overall transaction succeed.

Impact

High. Direct theft of any leveraged trove that hasn't been claimed by its owner. The attacker pays only the cost of opening a minimum trove. All troves opened through the zapper are vulnerable unless users are able to atomically use the zapper and claim the trove which can be done with smart contract wallet and EIP-7702 compatible wallets but many wallets are still non-compatible and thus vulnerable.

Recommendation

Validate the swap router against a whitelist, or replace `raw_call` with a typed swap interface that cannot target protocol contracts:

```
assert swap.router != params.trove_manager, "!router"
```

Developer Response

Fixed in PR#13.

2.7 Medium Findings

2.7.1 Lender-owned troves bypass lender-driven redemptions

A borrower can open a trove with `owner = lender` while receiving the funds themselves. The trove becomes permanently exempt from lender withdrawal redemptions, breaking the lender's primary exit mechanism.

Technical Details

`open_trove()` accepts an arbitrary `owner` parameter while transferring borrowed tokens to `msg.sender`. An attacker sets `owner = lender_address` and keeps the funds.
When an LP withdraws, `Lender._freeFunds()` calls `TROVE_MANAGER.redeem()` with `msg.sender == lender`. Inside `_redeem()`, the self-redemption check skips any trove where `msg.sender == trove.owner`:

```
1 if msg.sender != trove.owner or is_zombie_trove:
```

Since the attacker set `owner = lender`, this check fails for lender-driven redemptions — the trove is skipped entirely. The docstring explicitly states "The Lender (for withdrawals) can redeem anyone," but the implementation contradicts this.
If attackers position lender-owned troves at the lowest rates (tail of the sorted list), lender withdrawals burn LP shares without kicking any auction. The lender contract has no trove-management functions, so these positions are effectively untouchable via redemption.

Impact

Medium. LP withdrawals that rely on redemption are blocked when enough lender-owned troves sit at the redemption frontier. Shares can be burned without freeing liquidity. The market's primary forced-deleveraging mechanism becomes unreliable. Can only be cleared via liquidation (if the trove goes underwater) or borrower-initiated redemption at a higher rate.

Recommendation

Reject `owner == self.lender` in `open_trove()`.

Developer Response

Fixed in e0ad71d18fc8d220b2d0fe0ff2d63bb411dd6d89.

2.7.2 Borrowers pay interest on debt they haven't received, with no guarantee of full delivery

When lender idle liquidity is insufficient, borrowers are charged full debt immediately but receive borrow tokens asynchronously via Dutch auction. A collateral price drop between debt commitment and auction settlement can leave the borrower permanently short.

Technical Details

In `open_trove()` and `borrow()`, debt (including upfront fee) is written to storage and interest begins accruing before token delivery:

```
# 1. Debt committed to storage, interest starts accruing
self.troves[trove_id] = Trove(debt=debt_amount_with_fee, ...)
self._accrue_interest_and_account_for_trove_change(debt_amount_with_fee, ...)

# 2. Try to deliver tokens — may go async
self._transfer_borrow_tokens(debt_amount, annual_interest_rate, min_borrow_out,
min_collateral_out)
```

When idle liquidity is insufficient, `_transfer_borrow_tokens()` transfers whatever idle exists, then calls `_redeem()` which takes collateral from other troves and kicks a Dutch auction.
The auction starts at or above oracle price and decays. If the collateral price drops between the redemption and auction settlement (e.g., yvUSD `pricePerShare` decreases due to a strategy loss), the auction price may fall below the minimum price before any taker fills it. At that point:

- The auction becomes inactive and must be re-kicked at the new (lower) price
- Even after re-kick, the collateral is now worth less than the debt that was freed, so full proceeds are impossible
- The auction can partially fill — `amount_received` may never reach `maximum_amount`
- The auction has a max length (configurable, e.g., 1 day) — if it expires without being filled, the collateral sits until someone re-kicks

Throughout this entire period, the borrower is paying interest on the full debt amount, including the portion they haven't received and may never fully receive.

Impact

Medium. The borrower bears three compounding costs for the async portion: 1. **Interest on unreceived tokens** — debt accrues from the moment of commitment, not delivery 2. **Delivery delay** — minutes to days or longer, as each re-kick restarts the full auction length and multiple re-kicks may be needed in a declining market 3. **Potential permanent shortfall** — if collateral value drops, auction proceeds < debt freed, and the borrower never receives the full amount
The `min_borrow_out` parameter protects the idle portion but not the auction portion. `min_collateral_out` only checks that collateral was seized from troves, not that the auction will settle at full value.

Recommendation

Consider splitting the debt commitment: only record debt for the portion actually delivered from idle liquidity. The async portion should be tracked separately and finalized when the auction settles.
Alternatively, if the current atomic-debt design is kept, the protocol should clearly document the async delivery risk and ensure frontends set `min_borrow_out = debt_amount` by default to force revert when full instant delivery isn't available.

Developer Response

Acknowledged, by design and already documented.

2.7.3 Trove state mutation before `re_insert()` breaks sorted trove ordering

In `adjust_interest_rate()`, the trove's `annual_interest_rate` is written to storage before calling `sorted_troves.re_insert()`. Because `re_insert()` locates the correct insertion position by walking the linked list and comparing interest rates, and the node being re-inserted already contains the *new* rate, providing stale hints can cause the function to conclude the node is already in the right position when it is not.

Technical Details

In `adjust_interest_rate()`, the trove's state, including `annual_interest_rate`, is written to storage via `self.troves[trove_id] = trove` before `sorted_troves.re_insert()` is called. Inside `_re_insert()`, the function first checks whether the provided hint position `(prev_id, next_id)` is valid by calling `_valid_insert_position()`. This validation calls `_trove_annual_interest_rate()`, which reads the trove's rate directly from `trove_manager.troves()` storage. Since the rate has already been updated, the node being moved now reflects the *new* rate during position validation. Furthermore, `_re_insert()` has a guard at line 547 that skips the remove-and-reinsert entirely if `prev_id == trove_id` or `next_id == trove_id`, meaning if hint validation resolves the node's own position as valid, the node is left in place regardless of whether the ordering invariant holds.
Consider a sorted list ROOT → A[3%] → B[2%] → C[1%]. If B's rate is changed to 4%, the trove is first updated in storage so the list temporarily becomes ROOT → A[3%] → B[4%] → C[1%]. When `re_insert()` is called with hints [B, C], the validation logic checks whether B's new rate (4%) falls between B's prev (itself, with rate 4%) and C (rate 1%). This comparison succeeds, so the function treats the node as already in the correct position and returns without moving it. The list ends up with broken ordering: ROOT → A[3%] → B[4%] → C[1%], violating the sorted invariant.

Impact

Medium. The sorted trove invariant is silently broken, which can cause incorrect redemption ordering.

Recommendation

Call `sorted_troves.re_insert()` *before* mutating the trove's `annual_interest_rate` in storage. Alternatively, have the sorted troves implementation remove the node before re-inserting it, so that stale self-comparisons cannot occur.

Developer Response

Fixed in ded618d.

2.8 Low Findings

2.8.1 Missing buffer between entry and liquidation thresholds leaves edge positions exposed to abrupt oracle updates

`open_trove()`, `borrow()`, `remove_collateral()`, and the fee-charging `adjust_interest_rate()` path all allow positions as long as `collateral_ratio >= minimum_collateral_ratio`, while `liquidate_trove()` becomes callable as soon as `collateral_ratio < minimum_collateral_ratio`.
This means there is no buffer between the point where a trove is allowed to enter or increase risk and the point where it becomes liquidatable. If an oracle update sharply reduces the reported collateral value right after a trove is opened or adjusted near the edge, the trove can immediately move into liquidation territory and, in some cases, into the bad-debt path.

Technical Details

Under the intended deployment parameters:

- `safe_collateral_ratio = 120%`
- `minimum_collateral_ratio = 110%`
- `max_penalty_collateral_ratio = 105%`
- `min_liquidation_fee = 0.5%`
- `max_liquidation_fee = 5%`

there is no entry-side breathing room at all, and only 5 CR points between the maximum allowed leverage (`110%`) and the `105%` underwater-after-fee boundary.
As a result, a trove can be opened or adjusted right at `110%` based on the current oracle price and then become liquidatable immediately if the oracle value drops. If the update is large enough, the trove can land below `100% + fee`, where the full-liquidation branch clears the full debt from accounting even though the liquidator repays less than the trove's debt, socializing the shortfall to the lender.
This can also be combined with self-liquidation through a second address: one address opens the trove and receives the borrowed debt tokens, and a second address liquidates after the oracle normalizes, using those borrowed proceeds to fund the liquidation.
With the current yvUSD oracle, which is based on Yearn vault PPS, such a jump should be relatively rare. However, this becomes more relevant when using other oracle setups or more volatile collateral, for example a Chainlink oracle on a volatile asset, where a single update can move materially even without any adversarial manipulation around the deviation threshold.

Impact

Low. This requires the trove to be opened or adjusted very close to the liquidation boundary and depends on a sufficiently abrupt oracle move or oracle mispricing window immediately afterward. The current yvUSD oracle makes this less likely in practice, but the lack of a buffer still reduces the protocol's margin for error and could matter more for other markets or oracle sources.

Recommendation

Introduce a small buffer between the entry threshold and the liquidation threshold, so newly opened or freshly adjusted troves are not sitting directly on the liquidation line.

Also consider adding a narrow same-block protection or delay so a position that was healthy earlier in the block cannot be liquidated immediately after an oracle update in that same block. Finally, exercise caution when selecting oracle sources. The current yvUSD oracle is less likely to produce abrupt jumps, but more reactive oracles and more volatile collateral can make this edge case much more realistic.

Developer Response

Acknowledged. The MCR should be set high enough to account for this.

2.8.2 `setDoHealthCheck(false)` in `deploy()` only disables health check for a single report

The `LenderFactory.deploy()` function calls `setDoHealthCheck(false)` on newly created Lender vaults intending to permanently disable health checks. However, this only disables the health check for the next `harvestAndReport()` call, after which it is automatically re-enabled.

Technical Details

In `BaseHealthCheck._executeHealthCheck()`, when `doHealthCheck` is `false`, the function immediately sets `doHealthCheck = true` and returns without performing the check. This means the flag acts as a one-time bypass rather than a permanent toggle.
As a result, after the first `harvestAndReport()` call on a newly deployed Lender, all subsequent reports will have health checks enforced. If the intent is to permanently disable health checks for Lender vaults, the current approach does not achieve that.

Impact

Low. Health checks will be re-enabled on the second harvest operation, which may cause unexpected `harvestAndReport()` reverts if profit or loss exceeds the default thresholds.

Recommendation

If health checks should be permanently disabled, override `_executeHealthCheck()` in the Lender contract to be a no-op. If the intent is only to skip the first report's health check, update the comment to reflect that behavior.

Developer Response

Fixed in commit 7d3aabe.

2.8.3 Borrowers can receive full debt but zero borrow tokens when redemption loop finds nothing

When a borrower opens a trove or borrows and lender idle liquidity is insufficient, `_transfer_borrow_tokens()` can silently under-deliver if the redemption loop finds no redeemable troves.

Technical Details

The `_redeem()` loop exits early without reverting when no troves qualify for redemption (borrower has the lowest rate, all troves are owned by the borrower, or `_MAX_REDEMPTIONS` is hit). Back in `_transfer_borrow_tokens()`, the only check is on collateral output:

```
collateral_out: uint256 = self._redeem(amount - available_liquidity,
annual_interest_rate)
assert collateral_out >= min_collateral_out, "!min_collateral_out"
```

With default `min_collateral_out = 0`, this passes even if zero collateral was redeemed. The borrower's debt (including upfront fee) is already committed to storage and accruing interest, but they received no borrow tokens for the shortfall portion.

Impact

Low. The borrower ends up with debt they're paying interest on for tokens they never received. Requires low idle liquidity combined with no redeemable troves below the borrower's rate. Users can protect themselves by setting `min_borrow_out = debt_amount`, but the default (0) is unsafe.

Recommendation

Consider reverting when the total delivered (idle + redemption) is less than the requested amount.

Developer Response

Acknowledged. The `min_borrow_out` and `min_collateral_out` parameters exist for this.

2.8.4 `LeverageZapper` misroutes deferred borrow auction proceeds, enabling cross-user theft

When lender idle liquidity is insufficient during a leveraged open or lever-up, auction proceeds are sent to the zapper contract instead of the user, where they can be captured by the next zapper caller.

Technical Details

When `_transfer_borrow_tokens()` calls `_redeem()`, the receiver defaults to `msg.sender` which is the zapper:

```
receiver: address = msg.sender  # = LeverageZapper, not the end user
```

The auction stores `receiver = zapper`. When later filled, proceeds (borrow tokens) land on the zapper with no per-user accounting. The next zapper caller's `_handle_open()` or `_handle_lever_up()` reads the full `balanceOf(self)`, swaps everything including the stranded proceeds, and `_sweep()` sends it all to the new caller.
The original user still owes the full trove debt but never receives the deferred portion.

Impact

Low. Direct fund loss for zapper users during low-liquidity conditions. The deferred borrow proceeds are trapped on the zapper and captured by the next unrelated caller. Users can easily mitigate by setting `min_borrow_out = debt_amount` to revert if full instant delivery isn't available.

Recommendation

Hardcode `min_borrow_out = debt_amount` in the zapper's calls to `open_trove()` and `borrow()`, ensuring the transaction reverts if full instant delivery isn't possible.
The zapper's atomic swap-and-sweep pattern fundamentally cannot handle deferred delivery.

Developer Response

Acknowledged, already documented.

2.8.5 `_get_upfront_fee()` passes wrong `debt_amount` during interest rate adjustment

In `adjust_interest_rate()`, `_get_upfront_fee()` is called with `new_debt` (the trove's full debt including accrued interest) as the `debt_amount` parameter. However, the function adds `debt_amount` to `self.total_debt` to compute the new total debt. Since `new_debt` is already part of the accounted total debt, this double-counts that portion when computing the average interest rate.

Technical Details

During an interest rate adjustment, `new_debt` equals the trove's existing recorded debt plus accrued interest, this is not new debt being added to the system. When `_get_upfront_fee()` computes `new_total_debt = self.total_debt + debt_amount`, it effectively counts the trove's existing debt twice in the denominator. This inflates `new_total_debt`, which deflates the computed average interest rate and results in a lower upfront fee than intended.
Additionally note that passing 0 here would also be incorrect because the function needs `debt_amount` to compute `debt_amount * avg_interest_rate` for the fee.

Impact

Low. The upfront fee is systematically undercharged during premature rate adjustments, reducing protocol revenue. The magnitude depends on the trove's debt relative to total system debt.

Recommendation

Refactor `_get_upfront_fee()` to accept separate parameters for the fee base amount and the debt delta used to compute the new average rate. For rate adjustments, the debt delta should be zero (no new debt is entering the system) while the fee base should be the trove's current debt.

Developer Response

Fixed in 98d8980 and 2d221ff.

2.8.6 Upfront fee weighted debt calculation does not account for accrued interest

In `_get_upfront_fee()`, the new total weighted debt is calculated as `self.total_weighted_debt + (debt_amount * annual_interest_rate)`. This does not correctly offset the aggregate weighted debt the way `_accrue_interest_and_account_for_trove_change()` does.

Technical Details

For a borrow operation, the correct weighted debt offset should add `(recorded_debt + accrued_interest + new_debt) * rate` and subtract `recorded_debt * rate`. For an interest rate adjustment, it should add `(recorded_debt + accrued_interest) * new_rate` and subtract `recorded_debt * old_rate`. Instead, `_get_upfront_fee()` simply adds `debt_amount * annual_interest_rate` to the current aggregate. This means the accrued interest component is not reflected in the weighted debt, leading to an inaccurate average interest rate used for the upfront fee formula. The actual accounting in `_accrue_interest_and_account_for_trove_change()` handles this correctly, but the preview calculation in `_get_upfront_fee()` does not mirror that logic.

Impact

Low. The upfront fee is slightly miscalculated. The deviation depends on accrued but un-checkpointed interest, which under normal operation is small.

Recommendation

Mirror the weighted debt offset logic from `_accrue_interest_and_account_for_trove_change()` when computing the preview average rate in `_get_upfront_fee()`.

Developer Response

Acknowledged.

2.8.7 Upfront fee calculation uses stale `total_debt` without accruing pending interest

In `_get_upfront_fee()`, `self.total_debt` is read directly without first calling `_sync_total_debt()` to accrue pending aggregate interest. This means the `new_total_debt` and the resulting average interest rate are computed using a stale debt figure.

Technical Details

`_get_upfront_fee()` computes `new_total_debt = self.total_debt + debt_amount`. However, `self.total_debt` has not been synced since the last state-changing operation, so it excludes all interest that has accrued in the interim. The stale total debt leads to an inaccurate average interest rate (`new_total_weighted_debt // new_total_debt`), which in turn miscalculates the upfront fee charged to the borrower. Depending on time elapsed since last sync, the error could be non-trivial, systematically undercharging or overcharging the fee relative to the true average rate.

Impact

Low. The fee deviation is bounded by the interest accrued since the last sync and is unlikely to be large under normal usage patterns.

Recommendation

Call `_sync_total_debt()` at the beginning of `_get_upfront_fee()`, or ensure callers always sync total debt before invoking this function.

Developer Response

Acknowledged.

2.9 Gas Savings Findings

2.9.1 Collateral transfer in `DutchDesk.kick()` takes an unnecessary hop

Technical Details

During auction kicks, collateral hops TroveManager → DutchDesk → Auction in two transfers. The DutchDesk never uses the collateral — it just forwards it. Transferring directly from TroveManager to Auction saves one ERC20 transfer.

Impact

Gas.

Recommendation

Consider implementing the suggested changes.

Developer Response

Acknowledged. Keeping the two-hop pattern to preserve separation of concerns between Trove Manager and Auction.

2.9.2 Redundant `_sync_total_debt()` call in `_redeem()`

The `_redeem()` function calls `_sync_total_debt()` at its start, but `_accrue_interest_and_account_for_trove_change()` is called at the end of the function which internally calls `_sync_total_debt()` again, resulting in a redundant update to `total_debt` and `last_debt_update_time`.

Technical Details

`_sync_total_debt()` writes the updated `total_debt` and `last_debt_update_time` to storage. When `_accrue_interest_and_account_for_trove_change()` runs at the end of `_redeem()`, it calls `_sync_total_debt()` again. Since both calls happen within the same transaction, the second call will compute zero pending interest (same `block.timestamp`) and overwrite `total_debt` with the same value before applying the trove changes.

Impact

Gas Savings.

Recommendation

Remove the standalone `self._sync_total_debt()` call at the top of `_redeem()`. The subsequent call via `_accrue_interest_and_account_for_trove_change()` handles debt synchronization.

Developer Response

Fixed in commit 02b498b.

2.9.3 `_sync_total_debt()` gas improvements

Technical Details

1. The function `_accrue_interest_and_account_for_trove_change()` begins by calling `_sync_total_debt()`, which accrues pending interest and writes the updated value to `self.total_debt` in storage. Subsequently, the function calculates the net debt change from the trove operation and writes `self.total_debt` a second time. Because both writes target the same storage slot within a single transaction, the first SSTORE is effectively wasted. The final value could be computed and written once.
2. Redundant writes on same-block calls. `_sync_total_debt()` writes `total_debt` and `last_debt_update_time` even when `block.timestamp == last_debt_update_time` and pending interest is zero. Multiple operations in the same block waste gas and could early return.

Impact

Gas Savings.

Recommendation

1. Refactor `_sync_total_debt()` to return the synced value without writing to storage, and let `_accrue_interest_and_account_for_trove_change()` perform a single write after applying all adjustments.
2. Refactor `_sync_total_debt()` to return early when already updated in the same block.

Developer Response

Acknowledged.

2.9.4 Redundant `trove_id != 0` check in `liquidate_trove()`

In `liquidate_trove()`, there is an `assert trove_id != 0` check before loading the trove. This is redundant because the function subsequently asserts that `trove.status` is `ACTIVE` or `ZOMBIE`, which already implicitly rejects trove ID 0 (since the zero-ID trove will have an uninitialized status).

Technical Details

The `liquidate_trove()` function contains an early `assert trove_id != 0` guard. However, the function also checks `trove.status` against `ACTIVE` or `ZOMBIE` after loading the trove from storage. Since trove ID 0 is never initialized, its status will be the default value, which will fail the status assertion.

Impact

Gas Savings.

Recommendation

Remove the `assert trove_id != 0` check.

Developer Response

Removed here 6b4e0b3.

2.10 Informational Findings

2.10.1 Factory validation permits auction start prices below oracle value

`Factory._validate_params()` enforces `starting_price_buffer_percentage >= minimum_price_buffer_percentage` but allows both to be below `1e18`. Markets can be deployed with auctions that start below oracle fair value.

Technical Details

In `DutchDesk._get_prices()`:

```
starting_price = kick_amount * collateral_price * starting_price_buffer_pct // _WAD // precision
```

If `starting_price_buffer_pct < 1e18`, the auction starts below oracle value. Takers can immediately call `take()` and extract the spread between purchase price and oracle value with no waiting.
This converts the Dutch auction from a price-discovery mechanism into deterministic value extraction — redemption receivers (lender withdrawers, shortfall borrowers) receive less than oracle-equivalent value, and the spread goes to the taker.

Impact

Informational. Requires deployment misconfiguration. If deployed, every redemption in the affected market leaks value to takers. Real impact depends on whether the market is endorsed and attracts usage.

Recommendation

Enforce `>= 1e18` for both buffer parameters in `_validate_params()`:

```
assert params.starting_price_buffer_percentage >= _WAD, "!start_price_buffer"
assert params.re_kick_starting_price_buffer_percentage >= _WAD, "!re_kick_price_buffer"
```

Developer Response

Fixed in commit 93b1e6243535edf98665840d0b086aad137a236c.

2.10.2 Missing deployment parameter validation can brick markets

Neither `factory.deploy()` nor the individual `initialize()` functions validate parameters. Misconfigured markets are permanently broken.

Technical Details

Misconfiguration	Effect
Token decimals > 18	Auction scalers round to 0, bricking all auctions
`step_decay_rate >= 10000`	Ray multiplier underflows, bricking all auctions
`safe_cr <= minimum_cr`	Liquidation formula underflows, reverting partial liquidations
`safe_cr <= 100% + max_fee`	Liquidation formula divides by zero
`minimum_cr <= max_penalty_cr`	Fee interpolation divides by zero
Borrow token < 4 decimals	`one_hundredth_pct` rounds to 0, zeroing all fees

Impact

Informational. Requires deployment misconfiguration. Once deployed, the market is permanently bricked.

Recommendation

Add parameter assertions in `factory.deploy()` and `Auction.initialize()`.

Developer Response

Fixed in commit 2c04c01.

2.10.3 LeverageZapper has no on-chain slippage protection on swaps

`_swap()` executes swap calldata as-is with no on-chain minimum output check. Slippage protection depends entirely on the off-chain constructed calldata encoding a `minOut` in the router call.

Technical Details

After each swap in `_handle_open()`, the zapper reads `balanceOf(self)` and proceeds with whatever amount was received, no assertion that the output meets a minimum threshold.

Impact

Informational. No funds at risk as long as the swap receiver ensures slippage, but the function could use proper documentation or additional check.

Recommendation

Add a post-swap minimum output check inside `_swap()` or add natspec to the function.

Developer Response

Fixed in commit c7db341.

2.10.4 LeverageZapper uses crvUSD flash loans requiring extra swaps

The `LeverageZapper` flash-borrows crvUSD and swaps it to the collateral/borrow token via a DEX aggregator. For USDC-denominated markets, this adds unnecessary swap hops and slippage.

Technical Details

The crvUSD flash lender at `0x26dE...` offers zero-fee flash loans. However, the two swap round-trips (crvUSD→collateral and borrow→crvUSD) incur DEX fees and slippage that may exceed what a direct flash loan in the borrow token would cost.

Impact

Informational. Users pay unnecessary swap fees and slippage on the two crvUSD conversion hops. The cost compounds on both the collateral and debt swap legs of every leveraged operation.

Recommendation

Consider using Morpho flash loans denominated in the borrow token directly (e.g., USDC). This eliminates the swap fees.

Developer Response

Fixed in commit c2f6929.

2.10.5 Asynchronous redemption in `_freeFunds()` is interpreted as a loss by `TokenizedStrategy`

When a withdrawal exceeds the Lender's idle balance, `_freeFunds()` calls `TROVE_MANAGER.redeem()` which kicks a Dutch auction. Because auction proceeds are delivered asynchronously, the borrow tokens are not present in the contract at the time `TokenizedStrategy` checks the result, causing it to interpret the shortfall as a realized loss.

Technical Details

`TokenizedStrategy` calls `_freeFunds(_amount)` during `withdraw()` / `redeem()` and then measures the contract's asset balance to determine how much was actually freed. Since `TROVE_MANAGER.redeem()` initiates an auction that settles later, the idle balance does not increase during the call. The strategy therefore reports a loss equal to the auctioned portion. This has several practical consequences:

1. The default `withdraw()` sets `maxLoss = 0`, meaning any withdrawal requiring auction-based redemption will revert. The `maxWithdraw()` view does not account for this, so it may advertise a withdrawable amount that is not actually withdrawable without specifying a non-zero `maxLoss`.
2. Integrators relying on ERC-4626 standard `withdraw()` / `redeem()` will experience unexpected reverts or apparent losses without understanding the auction mechanism.
3. The accounting (totalAssets, totalShares) is decremented correctly, so no funds are actually lost, but the UX and integration surface is misleading.

Impact

Informational. No funds are lost, but the behavior diverges from ERC-4626 expectations. Integrators that do not use the non-standard `maxLoss` variant will encounter reverts when idle liquidity is insufficient.

Recommendation

Document this async withdrawal behavior prominently. Consider having `maxWithdraw()` return only the idle balance to avoid advertising amounts that require auction settlement. Recommend integrators use the `maxLoss`-aware `withdraw()` / `redeem()` variants to control the portion of funds routed via auctions.

Developer Response

Acknowledged. Should be documented already in the strategy's `_freeFunds()` function.

2.10.6 `_redeem()` does not check if the sorted troves list is empty before fetching the last element

In `_redeem()`, when no zombie trove exists, the function calls `sorted_troves.last()` without first checking if the list is empty. If the list is empty, this returns the root/zero sentinel value, and the subsequent loop iteration would operate on an invalid trove.

Technical Details

When `zombie_trove_id` is zero, `_redeem()` falls through to `trove_to_redeem = staticcall sorted_troves.last()`. If the sorted troves list is empty, `last()` returns the root node (ID 0). The loop then reads `self.troves[0]`, which is an uninitialized trove with zero debt and a zero interest rate.

Impact

Informational. In practice the list should never be empty during a redemption since debt must exist for a redemption to be triggered. However, adding an explicit empty-list check would make the precondition clear and prevent unexpected behavior.

Recommendation

Add a check after fetching the last trove ID to ensure the list is non-empty, e.g., `assert trove_to_redeem != 0, "!empty"`, or return early with zero collateral redeemed.

Developer Response

Acknowledged. The loop will break immediately on `redeemer_annual_interest_rate <= trove.annual_interest_rate`.

2.10.7 Low-decimal borrow tokens cause `one_hundredth_pct` and `min_annual_interest_rate` to round to zero

In `TroveManager.initialize()`, the derived precision constants `one_pct` and `one_hundredth_pct` are computed via integer division from `borrow_token_precision`. For tokens with fewer than 4 decimals, `one_hundredth_pct` rounds to zero, and for tokens with fewer than 2 decimals, `one_pct` rounds to zero.

Technical Details

The precision constants are computed as:

```
one_pct = borrow_token_precision // 100
one_hundredth_pct = one_pct // 100
```

For a token with 2 decimals (`borrow_token_precision = 100`), `one_pct = 1` and `one_hundredth_pct = 0`. This causes `min_liquidation_fee` and `max_liquidation_fee` (which are multiples of `one_hundredth_pct`) to be set to zero, effectively disabling liquidation fees. Similarly, `min_annual_interest_rate = one_pct // 2` would round to zero for tokens with fewer than 3 decimals, allowing near-zero interest rates.

Impact

Informational. The protocol is expected to use standard 18-decimal ERC-20 tokens as borrow tokens, so this is unlikely to occur in practice. However, if a low-decimal token were ever onboarded, core economic parameters would silently break.

Recommendation

Add a minimum decimals check in `initialize()` (e.g., `assert borrow_token_decimals >= 4`) or document the assumption that the borrow token must have sufficient decimal precision.

Developer Response

Fixed in commit 2c04c01.

2.10.8 Unused `borrow_token` field in `InitializeParams` struct in `DutchDesk`

The `InitializeParams` struct in `dutch_desk.vy` includes a `borrow_token` field that is never referenced during `initialize()` or anywhere else in the contract.

Technical Details

The `InitializeParams` struct defines `borrow_token: address` at line 59, but the `initialize()` function never reads or stores this value. All other fields in the struct are used to configure the contract's state. The unused field increases calldata size for every deployment and may mislead integrators into thinking the Dutch Desk tracks or validates the borrow token.

Impact

Informational.

Recommendation

Remove the `borrow_token` field from the `InitializeParams` struct and from any deployment scripts that populate it.

Developer Response

Fixed in commit 6b3d0c8.

2.10.9 Zapper trove ID collisions cause reverts for concurrent users

When two users call `open_leveraged_trove()` with the same `owner_index` in the same block, the second transaction reverts.

Technical Details

Trove IDs are derived as `keccak256(abi_encode(msg.sender, owner_index))`. For direct `open_trove()` calls, `msg.sender` differs per user so collisions are impossible. But through the zapper, `msg.sender` is always the zapper contract. Two users picking the same `owner_index` produce the same trove ID, and the second tx reverts on the `assert self.troves[trove_id].status == empty(Status)` check.

Impact

Informational. Degrades UX, especially if frontends use predictable values as `owner_index`.

Recommendation

Derive `owner_index` per user inside the zapper to namespace trove IDs:

```
owner_index: uint256 = convert(keccak256(abi_encode(data.owner, data.owner_index)),
uint256)
```

Developer Response

Acknowledged. Frontend uses random `owner_index` values.

2.11 Final Remarks

The Flex codebase provides leveraged exposure to yield-bearing vaults such as yvUSDC by combining Yearn strategies as the lending layer with a Liquity V2-inspired trove architecture. Since the codebase is written in Vyper and diverges from Liquity V2 in several areas, much of the code is new and has not been battle-tested. Existing code warrants careful scrutiny where it interfaces with newer components. This resulted in several medium and above severity findings. Most issues were promptly addressed by the team, demonstrating strong responsiveness. Given the complexity inherent to this type of protocol, the auditors strongly recommend a follow-up audit before going to production, along with a capped rollout.

yAudit

Flex

Completed 2026-03-31