Security Assessment

Final Report

# Silo - Liquidations by Defaulting

February 2026

Prepared for Silo

# Table of Contents

# Project Summary

## Project Scope

| Project Name | Repository Link | Initial Commit Hash | Final Commit Hash | Platform |
|---|---|---|---|---|
| Description | https://github.com/silo-finance/silo-contracts-v3 | f77309ddd7 | 2ad3aec634 | Solidity |

## Project Overview

This document describes the security review of **Description**. The work was undertaken from **January 15th, 2026** to **February 5th, 2026**.

The following contract list is included in our scope:

*silo-core/common/SiloCoreContracts.sol*
*silo-core/contracts/Silo.sol*
*silo-core/contracts/SiloConfig.sol*
*silo-core/contracts/SiloFactory.sol*
*silo-core/contracts/SiloLens.sol*
*silo-core/contracts/hooks/SiloHookV2.sol*
*silo-core/contracts/hooks/SiloHookV3.sol*
*silo-core/contracts/hooks/_common/Whitelist.sol*
*silo-core/contracts/hooks/defaulting/DefaultingRepayLib.sol*
*silo-core/contracts/hooks/defaulting/DefaultingSiloLogic.sol*
*silo-core/contracts/hooks/defaulting/PartialLiquidationByDefaulting.sol*
*silo-core/contracts/hooks/liquidation/PartialLiquidation.sol*
*silo-core/contracts/hooks/liquidation/lib/PartialLiquidationLib.sol*
*silo-core/contracts/incentives/SiloIncentivesController.sol*
*silo-core/contracts/incentives/SiloIncentivesControllerCompatible.sol*

*silo-core/contracts/incentives/SiloIncentivesControllerFactory.sol*
*silo-core/contracts/incentives/base/BaseIncentivesController.sol*
*silo-core/contracts/incentives/base/DistributionManager.sol*
*silo-core/contracts/lib/SiloLendingLib.sol*
*silo-core/contracts/lib/SiloLensLib.sol*
*silo-core/contracts/lib/Views.sol*
*silo-core/contracts/lib/Actions.sol*
*silo-core/contracts/lib/Hook.sol*
*silo-core/contracts/lib/SiloERC4626Lib.sol*
*silo-core/contracts/silo-router/SiloRouterV2Implementation.sol*
*silo-vaults/contracts/SiloVaultsFactory.sol*
*silo-vaults/contracts/libraries/EventsLib.sol*

The team performed a manual audit of all the files in scope**.** During the manual audit, the Certora team discovered bugs in the code, as listed on the following page.

## Protocol Overview

The audit focused on the newly introduced **Liquidation by Defaulting** mechanism, which provides an alternative approach for handling insolvent positions within Silo. This mechanism is designed to address the challenges associated with large-scale liquidations of illiquid collateral assets, where traditional liquidation methods may result in significant slippage and adverse price impact.

Under this mechanism, instead of liquidating collateral on the open market to repay outstanding debt, the protocol directly transfers the collateral tokens to lenders upon default. As a result, the borrower’s debt is effectively forgiven, and lenders receive the collateral asset in lieu of repayment, thereby avoiding forced market sales.

The audit scope also included the deprecation of the existing same-borrow functionality.

In addition, backward compatibility was introduced for the SiloIncentivesController, enabling its use with certain legacy hooks. Improvements were also made to the handling of zero-asset liquidation redemption edge cases through the implementation of a new internal _tryRedeem function, which prevents unintended reverts when redeeming small amounts.

Finally, the maxWithdraw functionality was updated to provide more accurate withdrawal estimates.

# Assessment Methodology

Our assessment approach combines design level analysis with a deep review of the implementation to ensure that a protocol is secure, economically sound, and behaves as intended under realistic conditions.

At the design level, we evaluate the architecture, the economic assumptions behind the protocol, and the safety properties that should hold independently of a specific chain or environment. This process includes reviewing internal and cross protocol interactions, state transition flows, trust boundaries, and any mechanism that could be exploited to extract value, deny service, or alter core system behavior. At this stage, a focused threat modelling exercise helps identify key attack surfaces and adversarial capabilities relevant to the system. Design level issues often relate to incentive structures, governance implications, or systemic behavior that emerges under adversarial conditions.

Implementation analysis focuses on the concrete behavior of the code within the execution model of the target chain. This involves reviewing the correctness of logic, access control, state handling, arithmetic behavior, and the nuanced behaviors of the chain environment. Familiar classes of vulnerabilities such as reentrancy conditions, faulty permission checks, precision issues, or unsafe assumptions often surface at this layer. These findings require context aware reasoning that takes into account both the code and the architectural intent.

To support this analysis, the codebase is examined through repeated manual passes and supplemented by automated tools when appropriate. High-risk logic areas receive deeper scrutiny, invariants are validated against both design intent and actual implementation, and potential vulnerability leads are thoroughly investigated. Automated techniques such as static analysis, fuzzing, or symbolic execution may be used to complement manual review and provide additional insight.

Collaboration with the development team plays an important role throughout the audit. This helps confirm expected behaviors, clarify design assumptions, and ensure an accurate understanding of the protocol’s intended operation. All findings are documented with clear reasoning, reproducible examples, and actionable recommendations. A follow up review is conducted to validate the applied fixes and verify that no regressions or secondary issues have been introduced.

# Threat and Security Overview

## Assets

**User collateral** held in Silo markets (all ERC20 collateral tokens deposited).

**Outstanding debt** accounting (borrowed ERC20 debt tokens; principal + accrued interest).

**Liquidation value flow** (seized collateral, repaid debt, liquidation bonuses/fees, defaulting write-offs).

**Share / position value accounting** (user "position value" / shares used for solvency & LTV calculations).

**Protocol solvency state** (bad debt, write-offs, and any implicit protocol loss bucket).

**Reward emissions & yield** distributed to depositors/participants (incentive tokens).

**Protected collateral** is provided by users who do not want their collateral to be used by borrowers.

## Actors

**Borrower** (can become liquidatable; may try to game liquidation/reward mechanics).

**Lender / Depositor** (earns yield/rewards; harmed by dilution or bad-debt leakage).

**Liquidator / Keeper** (executes liquidations; may be MEV/flash-funded; can reward-snipe).

**MEV searcher / Flash liquidity provider** (front-runs liquidations to capture rewards).

**Protocol admin / governance** (sets gauges/notifiers/config; can introduce centralization risk).

**SiloIncentivesController** (reward distribution authority; receives and redistributes rewards).

**Gauge / Notifier contract(s)** (external callback endpoints; can revert/DoS if misconfigured/malicious).

**Hook / Module contracts** (e.g., SiloHookV3, PartialLiquidation) facilitating liquidation paths.

## Trust Assumptions

**Users validate Silo hooks before interaction**

Users are assumed to review and understand any hooks configured on a Silo (e.g., liquidation hooks, incentive hooks) and accept the risk of malicious or value-extracting hook logic.

**Only standard-compliant ERC20 tokens are used**

The system assumes ERC20 tokens strictly follow the ERC20 specification (no fee-on-transfer, rebasing, callback behavior, or non-standard return values).

**Silo market parameters are economically sound at creation**

It is assumed that Silos are initialized with reasonable and safe economic parameters (LTVs, liquidation thresholds, penalties, incentives) and not adversarial or nonsensical configurations.

**Price oracles return correct and manipulation-resistant prices**

Oracles are assumed to provide accurate, timely, and manipulation-resistant price data; oracle compromise or manipulation is considered out of scope.

**Protocol owner / governance acts in good faith**

Admins are assumed not to maliciously configure gauges, hooks, notifiers, controllers, or parameters to censor users, extract value, or brick protocol functionality.

## Attack Vectors

**Flash reward sniping**

Attempted use of flash liquidity combined with liquidation-triggered “immediate distribution” to momentarily qualify for rewards, extract emissions, and exit in the same transaction (MEV-style reward capture).

**Reward dilution or misrouting**

Potential for IncentivesController reward flows to dilute other users’ yield or incorrectly attribute rewards due to liquidation-driven distribution mechanics.

**Self-liquidation reward farming**

Borrower attempts to intentionally trigger or coordinate their own liquidation in order to receive liquidation-triggered rewards and indirectly profit from being liquidated.

**Value leakage via defaulting write-offs**

Risk that the defaulting liquidation path could incorrectly write off collateral or mishandle seized collateral distribution, causing avoidable value loss.

**Untracked protocol loss**

Possibility that bad debt realized during defaulting liquidation is not fully or correctly accounted, leading to hidden insolvency or silent protocol losses.

**Position-value underflow in liquidation economics**

Defaulting liquidation math allowing a user's position value to be reduced beyond the actual bad debt, breaking accounting invariants and obscuring losses.

**Notifier-based denial of service**

Setting a misconfigured or reverting notifier via `setGauge` could block or halt defaulting liquidations if insufficient validation is applied.

**Wrong-silo validation confusion**

`validateControllerForCollateral` referencing an incorrect silo or configuration could enable bypasses or griefing via misapplied validation logic.

**Hook execution in insufficient or transitional state**

Hooks or modules being callable when the system is in an "insufficient" or intermediate state could lead to constraint bypass or inconsistent state transitions.

**Partial liquidation path divergence**

Differences between the normal liquidation flow and SiloHookV3 / PartialLiquidation execution paths could result in edge-case inconsistencies or accounting divergence.

**Pathological _maxDebtToCover values**

Liquidators providing extreme `_maxDebtToCover` values could stress repayment and collateral-seizure boundaries if not tightly constrained.

**Failure masking via try/catch usage**

Use of `try/catch` blocks could suppress critical errors and allow execution to continue in a partially updated or inconsistent state.

**Configuration checks that always pass**

Reliance on config-fetching or validation checks that are effectively always true could mask misconfiguration and enable unsafe LTV or solvency computations.

**Frontrunning Liquidations**

An attacker may attempt to frontrun liquidation transactions in order to gain an economic advantage. By observing pending liquidation calls in the mempool, a malicious actor could submit a competing transaction with higher priority, potentially capturing liquidation incentives or otherwise altering the expected liquidation outcome.

**Liquidation Denial of Service Due to Reward Distribution Overflow**

A denial-of-service condition may arise during liquidation if reward distribution logic encounters an overflow. In such cases, the liquidation transaction may revert, preventing the position from being liquidated and leaving an insolvent position unresolved.

## Findings Summary

The table below summarizes the findings of the review, including type and severity details.

| Severity | Discovered | Fixed |
|---|---|---|
| Critical | - | - |
| High | - | - |
| Medium | - | - |
| Low | 8 | 2 |
| Informational | 9 | 3 |
| **Total** | 17 | 5 |

## Severity Matrix

| | | | | |
|---|---|---|---|---|
| **Impact** | High | Medium | High | Critical |
| | Medium | Low | Medium | High |
| | Low | Low | Low | Medium |
| | | Low | Medium | High |
| | | | **Likelihood** | |

# Detailed Findings

| ID | Title | Severity | Status |
|---|---|---|---|
| L-01 | Missing _maxDebtToCover Validation in Defaulting Liquidation | Low | Fixed |
| L-02 | Missing Notifier Validation in setGauge Allows Permanent DoS of Defaulting Liquidations | Low | Acknowledged |
| L-03 | is_killed implemented but unused in the latest hook implementations | Low | Acknowledged |
| L-04 | maxLiquidation will return incorrect result in SiloHookV3 | Low | Acknowledged |
| L-05 | When loan is close to bad debt, part of the debt cost can be socialized | Low | Fixed |
| L-06 | Liquidators will receive Silo shares, even though they set _receiveSToken as false | Low | Acknowledged |
| L-07 | Debt silo collateral share token shall not be integrated by other projects | Low | Acknowledged |
| L-08 | Flash Reward Sniping via Liquidation Defaulting Immediate Distribution | Low | Acknowledged |
| I-01 | Redundant uint104 Capping and Casting in Reward Distribution | Informational | Fixed |

| | | | |
|---|---|---|---|
| I-02 | Dead Code Report: Same-Asset Borrowing Deprecation Remnants | Informational | Acknowledged |
| I-03 | Untracked Protocol Losses from Bad Debt in Defaulting Liquidation | Informational | Acknowledged |
| I-04 | Keeper fee inconsistency with the documentation | Informational | Fixed |
| I-05 | Unnecessary check in getAssetsDataForLtvCalculations() that will always return true | Informational | Acknowledged |
| I-06 | validateDefaultingCollateral could check the liquidation discount as well | Informational | Fixed |
| I-07 | Unnecessary logic in _fetchConfigs() | Informational | Acknowledged |
| I-08 | Unnecessary reading from storage | Informational | Acknowledged |
| I-09 | Future IRM models might be susceptible to gas exhaustion attacks | Informational | Acknowledged |
| I-10 | beforeAction repay hook is triggered during an insufficient state | Informational | Acknowledged |
| I-11 | beforeAction repay hook is used in a different context | Informational | Acknowledged |

## Low Severity Issues

### L-01: Missing _maxDebtToCover Validation in Defaulting Liquidation

| Severity: Low | Impact: Low | Likelihood: Low |
|---|---|---|
| Files: PartialLiquidationByDefaulting.sol | Status: Fixed | |

**Description:**

The liquidationCallByDefaulting function accepts a _maxDebtToCover parameter to allow callers to limit the amount of debt they want to liquidate. However, when dust prevention logic in PartialLiquidationExecLib.getExactLiquidationAmounts() forces a full liquidation, the returned repayDebtAssets may exceed _maxDebtToCover.

Unlike the standard liquidationCall in PartialLiquidation.sol, which includes the check:

```
require(repayDebtAssets <= _maxDebtToCover, FullLiquidationRequired());
```

The liquidationCallByDefaulting function lacks this validation. Consequently, the function does not respect the caller's specified limit.

The practical impact is limited since defaulting liquidations do not require the keeper to transfer debt tokens - the debt is written off against protocol collateral. The keeper actually benefits from larger liquidations as they receive more collateral shares. However, this violates the expected semantics of the _maxDebtToCover parameter.

**Recommendations:**

Either add the missing requirement check to ensure repayDebtAssets does not exceed _maxDebtToCover:

```
RevertLib.revertIfError(params.customError);
require(repayDebtAssets <= _maxDebtToCover, FullLiquidationRequired());
```

Or remove the _maxDebtToCover parameter entirely from the function signature, since limiting debt coverage is not meaningful for defaulting liquidations where the keeper does not pay for the debt repayment.

**Customer Response:**

Fixed in PR-1769

**Fix Review:**

Fixed

## L-02: Missing Notifier Validation in setGauge Allows Permanent DoS of Defaulting Liquidations

| Severity: Low | Impact: Low | Likelihood: Low |
|---|---|---|
| Files:<br>GaugeHookReceiver.sol | Status: Acknowledged | |

**Description:**

The "Liquidation by Defaulting" mechanism relies on the Hook contract calling immediateDistribution on the SiloIncentivesController to distribute seized collateral shares to lenders. The immediateDistribution function is restricted by the onlyNotifier modifier.

The setGauge function in GaugeHookReceiver validates that the gauge's SHARE_TOKEN matches the expected token but fails to verify that the gauge's NOTIFIER is set to the hook address itself. If an incentives controller is deployed with a different notifier (e.g., the share token itself, which is common in standard Aave-like deployments), the hook will lack the required permissions.

When a liquidation attempts to call _liquidateByDistributingCollateral, the call to immediateDistribution will revert. Since this is a mandatory step in the defaulting process, this misconfiguration results in a permanent Denial of Service (DoS) for all defaulting liquidations in that market. In V3 markets where this is the only liquidation method, it leads to guaranteed insolvency.

This is a governance footgun since the deployer will notice liquidations reverts potentially weeks/months after deployment.

**Recommendations:**

Add validation in setGauge to ensure the controller's NOTIFIER is the hook receiver:

```
require(_gauge.NOTIFIER() == address(this), InvalidNotifier());
```

**Customer Response:**

Acknowledged (we limit changes)

## L-03: is_killed implemented but unused in the latest hook implementations

| Severity: Low | Impact: Low | Likelihood: Low |
|---|---|---|
| Files:<br>SiloIncentivesControllerCompatible.sol | Status: Acknowledged | |

**Description:**

The SiloIncentivesControllerCompatible implements the following function:

```
    function is_killed() external view returns (bool) {
        return _isKilled;
    }
```

However, when a gauge is removed, the GaugeHookReceiver::removeGauge will not check if the gauge is active or killed:

```
  /// @inheritdoc IGaugeHookReceiver
    function removeGauge(IShareToken _shareToken) external virtual onlyOwner {
        ISiloIncentivesController configuredGauge =
configuredGauges[_shareToken];

        require(address(configuredGauge) != address(0), GaugeIsNotConfigured());

        delete configuredGauges[_shareToken];

        emit GaugeRemoved(address(_shareToken));
    }
```

While previous hook receivers performed the following check:

```
    require(configuredGauge.is_killed(), CantRemoveActiveGauge());
```

**Recommendations:**

Consider adding additional is_killed validation.

**Customer Response:**

*“We're not using it in the current contract versions, we're only using it in already deployed old contracts. This implementation is to be backwards compatible. We cannot remove it.”*

## L-04: maxLiquidation will return incorrect result in SiloHookV3

| Severity: Low | Impact: Low | Likelihood: High |
|---|---|---|
| Files: PartialLiquidation.sol | Status: Acknowledged | |

**Description:**

The SiloHookV3 is going to enable only liquidations by defaulting. This type of liquidations require a higher ltv for a liquidation to occur:

```
function liquidationCallByDefaulting(address _borrower, uint256 _maxDebtToCover)
       public
       virtual
       nonReentrant
       onlyAllowedOrPublic
       returns (uint256 withdrawCollateral, uint256 repayDebtAssets)
   {
       collateralConfig.lt += LT_MARGIN_FOR_DEFAULTING;
```

As a result, maxLiquidation() may return positive amounts, even though liquidation is impossible.

**Recommendations:**

Consider implementing a maxDefaultingLiquidation function.

**Customer Response:**

*“This is the consequence of our choice for MVP. We chose to do as few changes as possible. Acknowledged.”*

## L-05: When loan is close to bad debt, part of the debt cost can be socialized

| Severity: Low | Impact: Low | Likelihood: Low |
|---|---|---|
| Files:<br>PartialLiquidationLib.sol<br>PartialLiquidationByDefaulting.sol | Status: Fixed | |

**Description:**

When the collateral available from a borrower is less than debt_to_repay * liquidation_fee, the available collateral is returned instead of the expected amount (in PartialLiquidationLib.sol, calculateCollateralToLiquidate(), line 212). This collateral will be split in liquidationCallByDefaulting()(lines 106-122) by the regular ratio, i.e. 1+f*(1-kf):f*kf (using _getKeeperAndLenderSharesSplit() terminology). So, if debt*(1+f*(1-kf))>collateral, the lenders will lose value from the defaulting liquidation.

This also contrasts with the developer comment that describes KEEPER_FEE (in PartialLiquidationByDefaulting.sol), which implies the keeper should only be paid after the lenders get their funds back:

```
    /// @dev The portion of total liquidation fee proceeds allocated to the
keeper. Expressed in 18 decimals.
    /// For example, liquidation fee is 10% (0.1e18), and keeper fee is 20%
(0.2e18),
    /// then 2% liquidation fee goes to the keeper and 8% goes to the protocol.
```

Note that this occurs before bad debt (we need LTV of 1/(1+f*(1-kf))<1), but doesn't happen immediately when we are unable to cover the expected liquidation bonus (1/(1+f*(1-kf))>1/(1+f)).

Furthermore, this liquidation may not be available before this situation (debt*(1+f)>collateral) occurs, as the following validation in validateSiloInitData() for defaulting liquidation will be irrelevant due to higher lt:

```
 require(_initData.lt1 + _initData.liquidationFee1 <= _100_PERCENT,
ISiloFactory.InvalidLt());
```

And further validation in validateDefaultingCollateral() will also not fully handle this case:

```
require(config0.lt + LT_MARGIN_FOR_DEFAULTING < _DECIMALS_PRECISION,
InvalidLTConfig0());
```

**Recommendations:**

Consider using the following validation for the defaulting:

```
require(lt + LT_MARGIN_FOR_DEFAULTING + liquidationFee <= 100%)
```

Additionally, determine if keeper should receive a premium, if there is no fee that was seized

**Customer Response:**

*"This is a bad debt scenario and debt should be socialized to lenders. Bad debt starts when collateral - fee < debt. This is by design."*

Fixed in PR-1774

**Fix Review:**

Fixed

## L-06: Liquidators will receive Silo shares, even though they set _receiveSToken as false

| Severity: Low | Impact: Low | Likelihood: Medium |
|---|---|---|
| Files: PartialLiquidation.sol | Status: Acknowledged | |

**Description:**

Liquidations used to revert due to ReturnZeroAssets error, when the seized collateral shares were redeemed for 0 collateral assets.

This is the reason for the introduction of _tryRedeem(), which is going to handle such cases by directly transferring the share tokens in case of such reverts:

```
try ISilo(_silo).redeem({
           _shares: _shares,
           _receiver: msg.sender,
           _owner: address(this),
           _collateralType: _collateralType
       }) returns (uint256 assets) {
           withdrawCollateral = assets;
       } catch (bytes memory e) {
           if (_isToAssetsConvertionError(e)) {
               IERC20(_shareToken).transfer(msg.sender, _shares);
           } else {
               RevertLib.revertBytes(e, string(""));
           }
       }
```

However, this behaviour is not consistent with the input parameters, enabling liquidators to receive Silo share tokens when they put _receiveSToken as false. This would require all liquidators to have functionality to handle Silo swap tokens.

In maxLiquidate, the function may need to return _receiveSToken if the amount of shares that is being released cannot be redeemed for at least one asset.

**Recommendations:**

Consider documenting that even though _receiveSToken is false, users may still receive swap tokens, so liquidators handle them as well.

**Customer Response:**

*"They will receive shares; however, these shares cannot be translated to assets. The only option here is to either not send anything or send shares. The liquidator doesn't actually lose anything here, and it's not required for the liquidator to take care of these shares. However, we cannot leave the shares for the borrower because it's wrong. Maybe the liquidator already did some liquidations or has shares, and now when they have more, they will be able to withdraw it as assets. This is the best way."*

## L-07: Debt silo collateral share token shall not be integrated by other projects

| Severity: Low | Impact: Low | Likelihood: Low |
|---|---|---|
| Files:<br>PartialLiquidationByDefaulting.sol | Status: Acknowledged | |

**Description:**

The debt Silo collateral share token will be debased during a liquidation in _deductDefaultedDebtFromCollateral():

```
 function deductDefaultedDebtFromCollateral(uint256 _assetsToRepay) external
virtual {
        ISilo.SiloStorage storage $ = SiloStorageLib.getSiloStorage();

        bool success;
        uint256 totalCollateralAssets =
$.totalAssets[ISilo.AssetType.Collateral];

        // if underflow happens, $.totalAssets[ISilo.AssetType.Collateral] is set
to 0 and success is false
        (success, $.totalAssets[ISilo.AssetType.Collateral]) =
totalCollateralAssets.trySub(_assetsToRepay);
        uint256 deductedFromCollateral = _assetsToRepay;
```

This means that after each liquidation by default, the price of the collateral share token of the debt Silo will instantly decrease. This could allow for an attacker to perform manipulations such as selling their swap token to a victim at the pre-liquidation price, and liquidating in the same transaction as the sale.

**Recommendations:**

Consider adding a registry or another way for users to easily identify non-standard tokens by Silo

**Customer Response:**

*"3rd party integrations should be carefully reviewed. Acknowledged."*

## L-08: Flash Reward Sniping via Liquidation Defaulting Immediate Distribution

| Severity: Low | Impact: Low | Likelihood: High |
|---|---|---|
| Files: PartialLiquidationByDefaulting.sol | Status: Acknowledged | |

**Description:**

During liquidationCallByDefaulting, the protocol redistributes seized collateral (including the liquidation discount) directly to debtSilo.collateralShare holders via immediateDistribution():

```
function _liquidateByDistributingCollateral(
...
    ) internal virtual {
        ISiloIncentivesController controllerCollateral =
validateControllerForCollateral(_debtSilo);

        // distribute collateral shares to lenders
        if (_withdrawSharesForLenders > 0) {
            IShareToken(_shareToken)
                .forwardTransferFromNoChecks(_borrower,
address(controllerCollateral), _withdrawSharesForLenders);
            controllerCollateral.immediateDistribution(_shareToken,
_withdrawSharesForLenders);
        }
```

Because rewards are distributed immediately and proportionally to current share balances, an attacker can temporarily inflate their share ownership using a flash loan, capture an outsized portion of the liquidation rewards, and then exit in the same transaction.

This allows value extraction from long-term liquidity providers without taking on market or liquidation risk.

**Recommendations:**

Consider documenting this potential behaviour.

**Customer Response:**

*"Sniper must take a stake in Silo in order to snipe rewards.  Acknowledged."*

## Informational Severity Issues

### I-01: Redundant uint104 Capping and Casting in Reward Distribution

**Files:**

PendleRewardsClaimer.sol

**Description:**

The code caps rewardAmount to type(uint104).max and casts the result to uint104 before calling immediateDistribution():

```
uint256 amountToDistribute = Math.min(rewardAmount, type(uint104).max);
controller.immediateDistribution(rewardTokens[i], uint104(amountToDistribute));
```

This protection was originally implemented because immediateDistribution accepted a uint104 parameter. However, the function signature has since been updated to accept uint256

**Recommendations:**

Remove the uint104 capping and casting, and pass rewardAmount directly.

**Customer Response:**

Fixed in PR-1763

**Fix Review:**

Fixed

## I-02: Dead Code Report: Same-Asset Borrowing Deprecation Remnants

**Description:**

### 1. Deprecated Hook Constants

**Title:** Unused Hook Bitmap Constants

**Locations:**

Hook.sol

**Description:**

Two hook constants remain defined but are never used in any hook matching or action logic:

```
uint256 internal constant BORROW_SAME_ASSET = 2 ** 3; // deprecated
uint256 internal constant SWITCH_COLLATERAL = 2 ** 8; // deprecated
```

These constants occupy bitmap positions (bit 3 and bit 8) that could potentially be reclaimed. While marked with // deprecated comments, they still contribute to code complexity and could confuse future developers.

**Recommendation:**

Remove both constants entirely or replace with explicit comments explaining the reserved bit positions to prevent future reuse conflicts:

```
// uint256 internal constant _RESERVED_BIT_3 = 2 ** 3; // formerly BORROW_SAME_ASSET
// uint256 internal constant _RESERVED_BIT_8 = 2 ** 8; // formerly SWITCH_COLLATERAL
```

### 2. Unused Struct Definition

**Title:** Orphaned SwitchCollateralInput Struct

**Location:**

- `silo-core/contracts/lib/Hook.sol:179-181`

**Description:**

The struct SwitchCollateralInput is defined but never instantiated or used in any active code path:

```
struct SwitchCollateralInput {
    address user;
}
```

This struct was originally used to pass data to switch collateral hooks, which are now deprecated.

**Recommendation:**

Remove the struct definition entirely as it serves no purpose and increases bytecode size.

### 3. Unused Decoder Function

**Title:** Orphaned switchCollateralDecode Function

**Location:**

- `silo-core/contracts/lib/Hook.sol:646-659`

**Description:**

The function switchCollateralDecode is a 14-line internal function that decodes packed hook data for the deprecated switch collateral feature:

```
function switchCollateralDecode(bytes memory packed)
    internal
    pure
    returns (SwitchCollateralInput memory input)
{
    address user;
    assembly {
        let pointer := PACKED_ADDRESS_LENGTH
        user := mload(add(packed, pointer))
    }
    input = SwitchCollateralInput(user);
}
```

This function is never called anywhere in the codebase and depends on the orphaned SwitchCollateralInput struct.

**Recommendation:**

Remove the function entirely. It contributes to bytecode size and maintenance burden without providing any functionality.

## 4. Deprecated Interface Method

**Title:** Retained maxBorrowSameAsset Interface Declaration

**Locations:**

- `silo-core/contracts/interfaces/ISilo.sol:366`
- `silo-core/contracts/Silo.sol:488-490`

**Description:**

The interface declares and the contract implements a deprecated view function:

```
// Interface
function maxBorrowSameAsset(address _borrower) external view returns (uint256
maxAssets);

// Implementation
function maxBorrowSameAsset(address) external pure virtual returns (uint256) {
    return 0;
}
```

Unlike borrowSameAsset() which reverts, this function silently returns 0. While not a security issue, it maintains an unnecessary public API surface.

**Recommendation:**

Consider one of the following:

- **Option A (Breaking):** Remove from interface and make implementation revert with `Deprecated()`
- **Option B (Non-breaking):** Keep as-is but add explicit `@deprecated` NatSpec documentation to discourage usage

## 5. Deprecated Config Interface Method

**Title:** Retained setThisSiloAsCollateralSilo Interface Declaration

**Locations:**

- `silo-core/contracts/interfaces/ISiloConfig.sol:143`
- `silo-core/contracts/SiloConfig.sol:143-145`

**Description:**

The interface declares a method that always reverts:

```
// Interface
function setThisSiloAsCollateralSilo(address _borrower) external returns (bool
collateralSiloChanged);

// Implementation
function setThisSiloAsCollateralSilo(address) external virtual returns (bool) {
    revert Deprecated();
}
```

The interface still documents the method without indicating deprecation.

**Recommendation:**

Add @deprecated NatSpec to the interface declaration, or remove entirely if no external integrations depend on it.

## Summary

| Item | Location | Lines |
|---|---|---|
| BORROW_SAME_ASSET constant | Hook.sol:188 | 1 |
| SWITCH_COLLATERAL constant | Hook.sol:193 | 1 |
| SwitchCollateralInput struct | Hook.sol:179-181 | 3 |
| switchCollateralDecode() function | Hook.sol:646-659 | 14 |
| maxBorrowSameAsset()<br>interface/implementation | ISilo.sol:366,<br>Silo.sol:488-490 | 4 |
| setThisSiloAsCollateralSilo() interface | ISiloConfig.sol:143 | 1 |

**Total Dead Code:** ~24 lines

**Customer Response:**

*“We removed functionality, but we want current codes to be fully backwards compatible, so you don't have to downgrade to the oldest version to manage old contracts. We cannot remove.”*

## I-03: Untracked Protocol Losses from Bad Debt in Defaulting Liquidation

**Files:**

DefaultingSiloLogic.sol

**Description:**

In deductDefaultedDebtFromCollateral, if the debt to be repaid (_assetsToRepay) exceeds the available collateral assets, the function attempts to cover the deficit using daoAndDeployerRevenue. However, if the excess debt is greater than the available revenue, the trySub operation returns false (or the revenue is just zeroed out), and the remaining excess debt is effectively ignored ("swallowed").

The debt is fully repaid in the subsequent logic, but the asset side (Collateral + Revenue) is reduced by a smaller amount. This breaks the accounting equation Assets = Liabilities + Equity.

**Recommendations:**

Track the unrecoverable loss explicitly to maintain accounting integrity.

**Customer Response:**

Acknowledged

## I-04: Keeper fee inconsistency with the documentation

**Description:**

In the documentation the following is mentioned regarding the keeper fee:

If the liquidation fee is set to **10%**, and the position being liquidated has **$1,000 in debt**, then **$1,100 worth of share tokens** are used. This surplus must now be split between:

- 🧑‍🌾 **Lenders (75%)** → receive $1,075 in value via the Incentive Controller
- 🛠️ **Liquidator (25%)** → receives $25 in shared tokens directly

However, currently the keeper fee in the protocol is set to:

```
    uint256 public constant KEEPER_FEE = 0.2e18;
```

**Recommendations:**

Consider using a consistent value for the keeper fee across the documentation and the code.

**Customer Response:**

Fixed in in PR-1762

**Fix Review:**

Fixed

## I-05: Unnecessary check in getAssetsDataForLtvCalculations() that will always return true

**Files:**

SiloSolvencyLib.sol

**Description:**

The getAssetsDataForLtvCalculations will perform the following:

```
  if (_collateralConfig.token != _debtConfig.token) {
            // When calculating maxLtv, use maxLtv oracle.
            (ltvData.collateralOracle, ltvData.debtOracle) = _oracleType ==
ISilo.OracleType.MaxLtv
                ? (ISiloOracle(_collateralConfig.maxLtvOracle),
ISiloOracle(_debtConfig.maxLtvOracle))
                : (ISiloOracle(_collateralConfig.solvencyOracle),
ISiloOracle(_debtConfig.solvencyOracle));
        }
```

However, since the sameBorrow functionality has been removed, this check will always be true, therefore, it is unnecessary.

**Recommendations:**

Consider removing the redundant check and always executing the logic.

**Customer Response:**

Acknowledged

## I-06: validateDefaultingCollateral could check the liquidation discount as well

**Files:**

PartialLiquidationByDefaulting.sol

**Description:**

validateDefaultingCollateral() is intended to make sure that the Silo configuration is one sided and perform validations on the parameters. However, it does not check if the liquidationFee of the debtSilo. Since the defaulting liquidation markets must be one-way, the liquidation fee should be 0.

**Recommendations:**

Consider enforcing additional validation the liquidation fee of the debt Silo configuration

**Customer Response:**

Fixed in PR-1766

**Fix Review:**

Fixed

## I-07: Unnecessary logic in _fetchConfigs()

**Files:**

PartialLiquidationByDefaulting.sol

**Description:**

_fetchConfigs() will perform the following:

```
    if (collateralConfig.silo != debtConfig.silo) {
            ISilo(collateralConfig.silo).accrueInterest();
            collateralConfig.callSolvencyOracleBeforeQuote();
            debtConfig.callSolvencyOracleBeforeQuote();
        }
```

However, it is unnecessary to accrue the interest in the collateral Silo, as defaulting liquidation functionality is only intended for one-way Silo configs, which indicates that it will not be possible to borrow from the collateral Silo, making the utilization constantly 0.

**Recommendations:**

Consider removing the redundant functionality

**Customer Response:**

Acknowledged

## I-08: Unnecessary reading from storage

**Files:**

PartialLiquidationByDefaulting.sol

**Description:**

In liquidationCallByDefaulting() the configurations are initially fetched from memory in the following:

```
    (ISiloConfig.ConfigData memory collateralConfig, ISiloConfig.ConfigData
memory debtConfig) =

            _fetchConfigs(siloConfigCached, _borrower);
```

However, after that the debtConfig.collateralShareToken is again going to be read from storage in validateControllerForCollateral():

```
function validateControllerForCollateral(address _silo)...

     (, address collateralShareToken,) = siloConfig.getShareTokens(_silo);
```

This would unnecessarily increase the gas cost.

**Recommendations:**

Consider changing the function signature to accept the share token directly

**Customer Response:**

Acknowledged

## I-09: Future IRM models might be susceptible to gas exhaustion attacks

**Files:**

SiloLendingLib.sol

**Description:**

When accrueInterest() is invoked, it wraps the external call to the IRM using try-catch. This is done so that assets cannot be held hostage if the IRM is functioning as expected:

```
     function getCompoundInterestRate(
             address _interestRateModel,
             uint256 _totalCollateralAssets,
             uint256 _totalDebtAssets,
             uint64 _lastTimestamp
         ) internal returns (uint256 rcomp) {
             try

IInterestRateModel(_interestRateModel).getCompoundInterestRateAndUpdate(
                     _totalCollateralAssets,
                     _totalDebtAssets,
                     _lastTimestamp
                 )
                 returns (uint256 interestRate)
             {
                 rcomp = interestRate;
             } catch {
                 // do not lock silo on interest calculation
                 emit IInterestRateModel.InterestRateModelError();
             }
```

If the call fails, the rcomp is going to be equal to 0.

This will trigger the following block updating the timestand and return early:

```
                if (rcomp == 0) {
                $.interestRateTimestamp = uint64(block.timestamp);
                return 0;
            }
```

When an external call is made, 63/64 of the remaining gas is allocated for that call, if the call reverts due to OOG, the transaction has remaining 1/64, which may be enough to successfully finish the call.

While current interest rate models do not consume enough gas for such attack to occur, implementing a new, more gas-expensive IRM may enable an attacker to forcefully revert the try block, due to OOG error, but carry on with the transaction, updating the interestRateTimestamp, without accruing interest.

**Recommendations:**

Consider enforcing additional validation on the gas provided by the user, so that attackers cannot ever perform such attacks. Additionally it is recommended to add tests for existing IRM models to showcase that this attack is not feasible.

**Customer Response:**

*“Added to known issues to keep this in mind for future IRMs.”*

## I-10: beforeAction repay hook is triggered during an insufficient state

**Files:**

PartialLiquidationByDefaulting.sol

**Description:**

In liquidationCallByDefaulting() global reentrancy guard is disabled after the deduction of collateral:

```
 _deductDefaultedDebtFromCollateral(debtConfig.silo, repayDebtAssets);

        siloConfigCached.turnOffReentrancyProtection();
```

After that a repayment is triggered, which will invoke the beforeAction repay hook:

```
  if (_shareStorage.hookSetup.hooksBefore.matchAction(Hook.REPAY)) {

            bytes memory data = abi.encodePacked(_assets, _shares, _borrower,
_repayer);


IHookReceiver(_shareStorage.hookSetup.hookReceiver).beforeAction(address(this),
Hook.REPAY, data);

        }
```

This hook may allow reentrancy during the following state:

- DebtSilo.collateralShareToken.totalAssets has been decreased, however the debtSilo.debtShareToken.totalAssets is still inclusive of the repaid debt.
    - Any function fetching the utilization may be manipulated using a spiked utilization
- The borrower's position has 0 collateral but some debt.
    - A double liquidation is prevented due to  the nonReentrant modifier.

During this state, users can still borrow, update the interest and invoke other functions.

**Recommendations:** Consider keeping the global reentrancy modifier turned on.

**Customer Response:**

Acknowledged

## I-11: beforeAction repay hook is reused in different context

**Files:**

PartialLiquidationByDefaulting.sol

**Description:**

In liquidationCallByDefaulting() the following hook is invoked:

After that a repayment is triggered, which will invoke the beforeAction repay hook:

```
  if (_shareStorage.hookSetup.hooksBefore.matchAction(Hook.REPAY)) {

            bytes memory data = abi.encodePacked(_assets, _shares, _borrower,
_repayer);


IHookReceiver(_shareStorage.hookSetup.hookReceiver).beforeAction(address(this),
Hook.REPAY, data);

        }
```

However, a defaulting liquidation repayment is unlike the normal repayment, so this hook may have an irrelevant implementation.

**Recommendations:** Consider removing the hook call.

**Customer Response:**

Acknowledged

# Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

# About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.