Sherlock AI Monolith Audit Report — April 2026

Sherlock AI — Monolith Protocol Security Review

Date: April 2026 **Tool:** Sherlock AI Automated Analysis **Scope:** Factory.sol, Coin.sol, InterestModel.sol, Vault.sol, Lender.sol

---

Issue #1: Lender.getCollateralPrice: Staleness-driven price decay to 0 does not disable allowLiquidations, enabling collateral theft

Severity: Medium **Issue Validity:** Protocol didn't classify

Description: ## Description

The `getCollateralPrice` function in the Lender contract is responsible for retrieving the price of collateral assets. However, it fails to properly handle scenarios where the price feed becomes stale. Specifically, when the price feed's data becomes stale beyond the `stalenessThreshold` plus `STALENESS_UNWIND_DURATION`, the function sets the price to 1 instead of disabling liquidations. This allows liquidations, write-offs, and redemptions to proceed with a near-zero price, enabling attackers to seize collateral at essentially no cost.

Under normal circumstances, the function retrieves the price from a Chainlink feed. If the feed returns a price of 0 or reverts, the `allowLiquidations` flag is set to false, preventing liquidations. However, when the price decays to 0 due to staleness, the function sets the price to 1 (1 wei in the normalized price format) without disabling liquidations. This oversight allows malicious actors to exploit the system by liquidating positions at an artificially low price.

Attack Path

Consider the following attack scenario:

1. A Chainlink feed stops updating, possibly due to sequencer downtime or feed deprecation. The feed previously returned a valid price for an 18-decimal collateral token.
2. Time passes beyond the `stalenessThreshold` plus `STALENESS_UNWIND_DURATION` (e.g., 25 hours total with no update).
3. The `getCollateralPrice` function returns a price of 1, with `reduceOnly` set to true and `allowLiquidations` set to true. New borrowing is blocked by `reduceOnly`, but liquidations are permitted.
4. An attacker calls `liquidate(borrower, type(uint).max, 0)`. The borrowing power is calculated as approximately 0, making the position liquidatable. The attacker pays a few wei of

Coin and receives all of the borrower's collateral, which could be worth a significant amount (e.g., 100e18 tokens worth $60,000).
5. Alternatively, the attacker calls `writeOff(borrower, attacker)`. The collateral value is calculated as 100 wei, triggering a write-off since the debt exceeds this value. The borrower's debt is socialized to other borrowers, and the attacker receives all collateral for free.
6. Alternatively, the attacker calls `redeem(redeemableBorrower, amountIn, 0)`. With the price set to 1, the amount of collateral received per wei of Coin is enormous. Although the `maxRedeemByCollateral` cap limits the amountIn to a few wei, the attacker still gains a disproportionate amount of collateral.

Impact

This vulnerability allows attackers to seize collateral at a near-zero cost, leading to significant financial losses for the protocol and its users. Liquidations, write-offs, and redemptions proceed with an artificially low price, enabling malicious actors to exploit the system and drain collateral from borrower positions. This can result in a loss of trust in the protocol and potential insolvency if not addressed promptly.

Suggested Fix

Disable liquidations when staleness unwinds price to zero.

```
-         price = price == 0 ? 1 : price;
-         return (price, reduceOnly, allowLiquidations);
+         if (price == 0) {
+             allowLiquidations = false;
+             return (0, reduceOnly, allowLiquidations);
+         }
+         return (price, reduceOnly, allowLiquidations);
```

Issue #2: Lender.writeOff: When totalDebt is zero after decreaseDebt, bad debt is silently destroyed causing protocol insolvency

Severity: Medium **Issue Validity:** Protocol didn't classify

Description: ## Description

The `writeOff` function in the Lender contract is designed to handle situations where a borrower's debt exceeds the value of their collateral by a factor of 100. This function aims to remove the borrower's debt from the system. However, if the borrower is the only or last remaining borrower in the Lender market, a critical issue arises. When the `decreaseDebt` function is called within `writeOff`, it zeros out the borrower's debt shares. If this results in `totalDebt` (the sum of `totalFreeDebt` and `totalPaidDebt`) becoming zero, the redistribution block is skipped entirely due to the condition `if(totalDebt > 0)` on line 398 evaluating to false. Consequently, the borrower's debt, which corresponds to already-minted Coin in circulation, is erased without being redistributed to other borrowers. This creates unbacked Coin in circulation, leading to protocol insolvency.

Attack Path

Consider the following scenario:

1. A Lender market has a single borrower who has minted 100,000e18 Coin against their collateral.
2. The price of the collateral crashes, resulting in the borrower's debt (100,000e18) exceeding the collateral value multiplied by 100 (e.g., collateral worth less than 1,000e18).
3. An external actor calls `writeOff(borrower, attacker)`. The `decreaseDebt` function zeros out the borrower's debt shares.
4. Since there are no other borrowers, `totalFreeDebt + totalPaidDebt` equals zero.
5. The condition `if(totalDebt > 0)` on line 398 evaluates to false, causing the redistribution block to be skipped.
6. As a result, 100,000e18 Coin remains in circulation, held by whoever the borrower sold it to, but there is zero debt backing it in this Lender market.
7. The protocol becomes insolvent by 100,000e18 Coin for this market.

Impact

This issue leads to protocol insolvency for the affected market. The unbacked Coin remains in circulation, creating a situation where the protocol cannot cover the outstanding Coin with existing collateral or debt. This insolvency undermines the financial stability of the protocol and could lead to a loss of confidence among users and investors. The problem is particularly acute in single-collateral protocols where the last borrower defaults, as there are no other borrowers to redistribute the debt to, highlighting a limitation of the socialization model used in the protocol.

Suggested Fix

Revert writeOff when totalDebt is zero to avoid unbacked Coin.

```
-         uint256 totalDebt = totalFreeDebt + totalPaidDebt;
-         if (totalDebt > 0) {
+         uint256 totalDebt = totalFreeDebt + totalPaidDebt;
+         require(totalDebt > 0, "Cannot write off last borrower");
+         if (totalDebt > 0) {
              // existing redistribution logic
          }
```

Issue #3: Factory.getFeeOf: Cannot set custom fee of zero, always falls through to global feeBps

Severity: Low/Info **Issue Validity:** Protocol didn't classify

Description: ## Description

The `getFeeOf` function in the Factory contract is designed to return a custom fee for a specific lender if one is set. However, it only returns the custom fee if `customFeeBps[_lender] > 0`. If the factory operator explicitly sets `customFeeBps[lender] = 0` using `setCustomFeeBps(lender, 0)`, the function defaults to returning the global `feeBps` instead. This logic flaw prevents the factory

operator from exempting a specific lender from the global fee by setting their fee to zero. The zero value is used both as a sentinel for "not set" and as a valid fee, leading to this issue.

Attack Path

Consider the following scenario:

1. The Factory contract has a global feeBps set to 500, representing a 5% fee.
2. The factory operator wants to exempt a specific lender from this global fee as part of a promotional arrangement.
3. The operator calls `setCustomFeeBps(lenderAddress, 0)` to set the lender's fee to zero.
4. During the next `accrueInterest()` call, the Lender contract calls `factory.getFeeOf(address(this))`.
5. The `getFeeOf` function checks `customFeeBps[lenderAddress]`, which is 0, causing the condition `customFee > 0` to evaluate as false.
6. The function defaults to returning the global `feeBps` of 500 instead of the intended custom fee of 0.
7. As a result, `cachedGlobalFeeBps` is set to 500, and the lender continues to pay the global fees.

Impact

This issue results in the inability of the factory operator to exempt specific lenders from the global fee by setting their fee to zero. While this does not lead to a direct loss of user funds, it causes unfair treatment of lenders who are supposed to be exempt from fees. The impact is limited to operator inconvenience and the inability to implement certain promotional arrangements, rather than any critical financial loss or security breach.

Suggested Fix

Add explicit flag to distinguish unset custom fees from zero to allow exemptions.

```
-    mapping(address => uint256) public customFeeBps;
+    mapping(address => uint256) public customFeeBps;
+    mapping(address => bool) private isCustomFeeSet;
@@
-    function setCustomFeeBps(address _lender, uint256 _fee) external onlyOperator {
-        customFeeBps[_lender] = _fee;
-    }
+    function setCustomFeeBps(address _lender, uint256 _fee) external onlyOperator {
+        customFeeBps[_lender] = _fee;
+        isCustomFeeSet[_lender] = true;
+    }
@@
-    function getFeeOf(address _lender) external view returns (uint256) {
-        uint256 customFee = customFeeBps[_lender];
-        return customFee > 0 ? customFee : feeBps;
-    }
+    function getFeeOf(address _lender) external view returns (uint256) {
+        if (isCustomFeeSet[_lender]) {
+            return customFeeBps[_lender];
```

```
+        }
+        return feeBps;
+    }
```

Issue #4: Lender.setLocalReserveFeeBps: Missing beforeDeadline modifier allows operator to change fees after immutability

Severity: Medium **Issue Validity:** Protocol didn't classify

Description: ## Description

The `setLocalReserveFeeBps` function in the Lender contract allows the operator to change the local reserve fee percentage, but it lacks the `beforeDeadline` modifier. This omission permits the operator to modify the fee even after the immutability deadline has passed, which breaks the immutability guarantee for borrowers. Normally, all parameter-changing setter functions, such as `setHalfLife`, `setTargetFreeDebtRatio`, `setRedeemFeeBps`, and `setMaxBorrowDeltaBps`, enforce the `beforeDeadline` modifier to prevent changes after the deadline. However, `setLocalReserveFeeBps` only has the `onlyOperator` modifier, creating an inconsistency that allows the operator to change the local reserve fee percentage (up to 10%) even after the immutability deadline.

Attack Path

Consider this attack scenario: A Lender contract is deployed with `immutabilityDeadline` set to `block.timestamp + 365 days` and `feeBps` initialized to 0. Users borrow against this contract, trusting that protocol parameters cannot change after the immutability deadline. Once the deadline passes, the operator can either call `enableImmutabilityNow()` or simply wait for the deadline, causing all other setters to revert. However, the operator can still call `setLocalReserveFeeBps(1000)` to set `feeBps` to 10% (the maximum allowed), which succeeds because the function lacks the `beforeDeadline` modifier. On the next `accrueInterest()` call, 10% of all interest is diverted to `accruedLocalReserves` instead of going to vault stakers. The operator can then pull these reserves via `pullLocalReserves()`, extracting value that borrowers and stakers did not consent to.

Impact

This vulnerability allows the operator to divert a portion of the interest to themselves after the immutability deadline, effectively stealing yield from borrowers and stakers. The immutability guarantee is a core trust assumption for users, and its violation undermines confidence in the protocol. The operator can extract up to 10% of the interest generated by active paid debt, which borrowers and stakers did not agree to, resulting in a medium severity impact due to the potential financial loss and breach of trust.

Suggested Fix

Add the beforeDeadline modifier to setLocalReserveFeeBps to enforce immutability.

```
-    function setLocalReserveFeeBps(uint16 newFeeBps) external onlyOperator {
+    function setLocalReserveFeeBps(uint16 newFeeBps) external onlyOperator beforeDeadline {
```

Issue #5: Lender.writeOff: Missing access control allows unauthorized writeOff calls, bypassing partial liquidation and misappropriating collateral

Severity: Medium **Issue Validity:** Protocol didn't classify

Description: ## Description

The `writeOff` function in the `Lender` contract is declared as `external` without any access control modifiers, allowing any external caller to invoke it directly. This function is intended to be called only from the `liquidate` function, which first performs a partial debt repayment before calling `writeOff`. However, due to the lack of access control, an attacker can call `writeOff` directly, specifying an arbitrary address to receive the borrower's remaining collateral. This bypasses the partial liquidation process and results in the entire debt being socialized among other borrowers, while the attacker receives the collateral.

Under normal circumstances, the `liquidate` function would repay a portion of the borrower's debt proportional to their collateral value before calling `writeOff` to socialize the remaining debt. However, the direct invocation of `writeOff` skips this repayment step, leading to a larger amount of bad debt being distributed among other borrowers.

Attack Path

Consider a scenario where a borrower's position deteriorates to a point where their debt is 1,000,000e18 and their collateral value is 9,000e18, just crossing the 100x writeOff threshold due to a price decline. A legitimate liquidator would typically submit a `liquidate` transaction, which would repay approximately 8,181e18 of the debt, take all collateral, and then call `writeOff` to socialize the remaining 991,819e18 debt.

However, an attacker can front-run this liquidation by calling `writeOff` directly, specifying their own address to receive the collateral. This results in the entire 1,000,000e18 debt being socialized among other borrowers, and the attacker receiving all the collateral worth approximately 9,000e18. The legitimate liquidator's transaction would then revert because the borrower now has zero debt and zero collateral, making the position non-liquidatable.

Impact

This vulnerability allows an attacker to misappropriate collateral by bypassing the partial liquidation process, resulting in a larger amount of bad debt being socialized among other borrowers. While the financial gain for the attacker is limited due to the collateral value being less than 1% of the debt in a 100x underwater position, the main harm is the griefing impact on other borrowers who absorb more bad debt than they would have under the normal liquidation flow. This creates an unfair distribution of debt and undermines the integrity of the liquidation process.

Suggested Fix

Restrict writeOff to internal calls to prevent bypassing liquidation logic.

```
-    function writeOff(address borrower, address to) external {
+    function writeOff(address borrower, address to) external {
+        require(msg.sender == address(this), "writeOff can only be called internally");
         // existing writeOff logic
     }
```