Prepared for
Fira

Audited by
ret2basic.eth
jesjupyter

November 25, 2025

Fira Security Review

Smart Contract Security Assessment

Contents

1 Review Summary **2**
1.1 Protocol Overview . . . 2
1.2 Audit Scope . . . 2
1.3 Risk Assessment Framework . . . 3
1.3.1 Severity Classification . . . 3
1.4 Key Findings . . . 4
1.5 Overall Assessment . . . 4

2 Audit Overview **4**
2.1 Project Information . . . 4
2.2 Audit Team . . . 4
2.3 Audit Timeline . . . 4
2.4 Audit Resources . . . 4
2.5 Critical Findings . . . 6
2.6 High Findings . . . 6
2.7 Medium Findings . . . 6
2.7.1 PermissionedSisuVault allows unauthorized share transfers . . . 6
2.8 Technical Details . . . 6
2.9 Impact . . . 6
2.10 Recommendation . . . 7
2.11 Low Findings . . . 7
2.11.1 Inconsistent Rounding Undercuts Post-Maturity Liquidators . . . 7
2.11.2 No Enforced Safety Buffer Between LTV and LLTV . . . 8
2.12 Gas Savings Findings . . . 9
2.13 Informational Findings . . . 9
2.13.1 Redundant code in _liquidatePostMaturity . . . 9
2.14 Technical Details . . . 9
2.15 Impact . . . 9
2.16 Recommendation . . . 9
2.16.1 SisuVaultPriceFeed hardcoded FLOOR leads to liquidation freeze or fund drain . . . 10
2.17 Technical Details . . . 10
2.17.1 The "Infinite Drain" Scenario (P_real < LTV) . . . 10
2.17.2 The "Liquidation Freeze" Scenario (LTV < P_real < 1.0) . . . 11
2.18 Impact . . . 11
2.19 Recommendation . . . 11
2.20 Appendix: Oracle Configuration and Call Chain . . . 12
2.20.1 Configuration Context . . . 12
2.20.2 Execution Flow . . . 12
2.21 Appendix: Bad Debt Impact Analysis . . . 13
2.21.1 1. Bad Debt Creation (LendingMarket) . . . 13
2.21.2 2. Vault Asset Valuation (SisuVault) . . . 14
2.21.3 3. Share Price Calculation (SisuVault) . . . 14
2.21.4 4. Oracle Floor Trigger (SisuVaultPriceFeed) . . . 14
2.21.5 Potential Redundant Health Check in withdrawCollateral . . . 15
2.21.6 Indistinguishable Zero Result For Unknown Market . . . 16

1 Review Summary

1.1 Protocol Overview

Fira is a lending market protocol forked from Morpho Blue, designed to integrate with the Usual ecosystem. It features permissionless and permissioned lending markets, specialized vaults (SisuVault) for managing collateral, and custom oracle adapters for pricing vault shares. The protocol aims to enable efficient lending and borrowing of USD0 and other assets against SisuVault shares.

1.2 Audit Scope

This audit covers 24 smart contracts totaling approximately 2700 lines of code across 4 days of review. The codebase is a fork of Morpho and auditors were very familiar with Morpho, so the audit was faster than normal.

```
src
├── irms
│   ├── AdaptiveCurveIrm.sol
│   └── FixedRateIrm.sol
├── lending_market
│   ├── LendingMarket.sol
│   └── USLLendingMarket.sol
├── libraries
│   ├── ChainlinkDataFeedLib.sol
│   ├── ConstantsLib.sol
│   ├── ErrorsLib.sol
│   ├── EventsLib.sol
│   ├── ExpLib.sol
│   ├── MarketParamsLib.sol
│   ├── MathLib.sol
│   ├── MathLibInt128.sol
│   ├── PendingLib.sol
│   ├── SafeTransferLib.sol
│   ├── SharesMathLib.sol
│   ├── UtilsLib.sol
│   ├── VaultLib.sol
│   └── periphery
│       ├── LendingMarketBalancesLib.sol
│       ├── LendingMarketLib.sol
│       └── LendingMarketStorageLib.sol
├── migrator
│   └── USLMigrator.sol
├── oracles
│   ├── ChainlinkOracleV2.sol
│   ├── ChainlinkOracleV2Factory.sol
│   └── SisuVaultPriceFeed.sol
└── sisu_vault
    ├── PermissionedSisuVault.sol
```

```
├── PermissionedSisuVaultFactory.sol
├── SisuVault.sol
└── SisuVaultFactory.sol
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

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

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	2
Informational	4

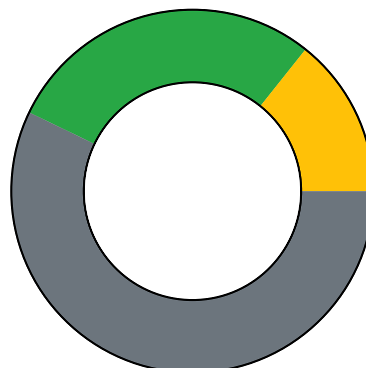

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The Fira codebase is solid in general. We uncovered a rounding issue in liquidator profit (medium severity) and a transfer/transferFrom override issue (medium severity), no other major issues discovered during the audit.

2 Audit Overview

2.1 Project Information

Protocol Name: Fira
Repository: https://github.com/usual-dao/fira-lending-market
Commit Hash: 542c7f985877b97228bf5321ca8a68a1b53a40b0
Commit URL: https://github.com/usual-dao/fira-lending-market/tree/542c7f985877b97228bf5321ca8a68a1b53a40b0

2.2 Audit Team

ret2basic.eth, jesjupyter

2.3 Audit Timeline

The audit was conducted from November 19 to 24, 2025.

2.4 Audit Resources

Code repositories and documentation Additional resources: whitepaper, previous audits, etc.

Category	Mark	Description
Access Control	Good	Role-based access control is present (Owner, USL Liquidator)
Mathematics	Good	The core mathematical libraries inherited from Morpho Blue are robust.
Complexity	Good	The system's modularity (separating IRMs, Oracles, and Markets) follows original Morpho design.
Libraries	Good	The protocol correctly utilizes standard libraries (OpenZeppelin, Morpho) for core functionality, reducing the attack surface for standard operations.
Decentralization	Good	The protocol relies on centralized roles (`owner`, `uslLiquidator`) and permissioned components (`PermissionedSisuVault`).
Code Stability	Good	The codebase is a mix of stable, audited code (Morpho) and new, experimental features.
Documentation	Good	Code is generally commented and a audit scope document was provided before the audit.
Monitoring	Good	Standard event emission is present for key actions.
Testing and verification	Good	The project includes a comprehensive Foundry test suite.

Table 2: Code Evaluation Matrix

2.5 Critical Findings

None.

2.6 High Findings

None.

2.7 Medium Findings

2.7.1 PermissionedSisuVault allows unauthorized share transfers

The `PermissionedSisuVault` contract is designed to restrict access to a single `permissionedAddress`. It achieves this by overriding the ERC4626 entry and exit functions (`deposit`, `mint`, `withdraw`, `redeem`) and applying the `onlyPermissioned` modifier. However, the contract inherits from `SisuVault` (and subsequently `ERC20`) but does not override `transfer` or `transferFrom`. As a result, standard ERC20 token transfers are unrestricted.

```
// src/sisu_vault/PermissionedSisuVault.sol

contract PermissionedSisuVault is SisuVault {
    // ... overrides for deposit, mint, withdraw, redeem ...

    // MISSING: overrides for transfer / transferFrom
}
```

If the `permissionedAddress` mints shares to a receiver (e.g., a partner or a specific treasury wallet), that receiver can freely transfer those shares to any other address.

2.8 Technical Details

The `PermissionedSisuVault` inherits from `SisuVault`, which in turn inherits from `ERC4626` (and `ERC20`).

1. **Inheritance Chain**: `PermissionedSisuVault` -> `SisuVault` -> `ERC4626` -> `ERC20`.
2. **Overrides**: The contract correctly overrides `deposit`, `mint`, `withdraw`, and `redeem` to enforce the `onlyPermissioned` modifier.
3. **Missing Overrides**: The standard ERC20 functions `transfer` and `transferFrom` are **not** overridden.
4. **Result**: Once shares are minted to an address (even if that address was initially approved/permissioned), that address can call `transfer` to move the shares to *any* other address, bypassing the permissioned gateway.

2.9 Impact

This oversight allows for the circumvention of the "permissioned" nature of the vault regarding share ownership.

1. **Unauthorized Ownership:** An authorized share holder can transfer shares to an unauthorized or blacklisted entity.
2. **Invariant Violation:** The audit scope document states "Only Usual should be allowed to supply USD0 for borrowing". If shares (which represent the supplied capital) can be transferred to third parties without restriction, the protocol loses control over who effectively owns the supplied capital.

While the unauthorized holder cannot call `withdraw` (as that is restricted to `permissionedAddress`), they legally/technically own the claim on the underlying assets.

2.10 Recommendation

Override `transfer` and `transferFrom` to enforce access control, or disable transfers entirely if the shares are intended to be non-transferable (soulbound) to ensure only the `permissionedAddress` (or receivers explicitly minted to) can hold them.
If the intention is to allow transfers only between whitelisted entities, logic should be added to check both `from` and `to` against a whitelist or restrict initiation to `permissionedAddress`.

Developer Response

Acknowledged - won't fix. Operationally, we only expect the owner to hold shares in the vault and not to transfer any tokens, so we don't see any need to implement any precaution here.

2.11 Low Findings

2.11.1 Inconsistent Rounding Undercuts Post-Maturity Liquidators

Standard liquidation intentionally rounds against liquidators (favoring borrowers) when converting debts and collateral. Post-maturity liquidation inverts those choices to encourage liquidators. However, `_liquidatePostMaturity` still floors the liquidation incentive multiplication, so even in the "liquidator-favored" path, part of the incentive is rounded away.

Technical Details

Normal liquidation uses all "round down" conversions:

```
seizedAssets = repaidShares.toAssetsDown(...)
    .wMulDown(liquidationIncentiveFactor)
    .mulDivDown(ORACLE_PRICE_SCALE, collateralPrice
```

That bias keeps borrowers from overpaying. Conversely, `_liquidatePostMaturity` flips the steps: debt is rounded up, collateral requirement rounded up—to benefit the liquidator. Yet the incentive multiplication still uses `wMulDown`:

```
uint256 fullDebtAssets = borrowShares.toAssetsUp(...);
uint256 liquidationIncentive = WAD + marketConstants[id].liquidationIncentive;
uint256 liquidationValueFull = fullDebtAssets.wMulDown(liquidationIncentive); // <= uses `wMulDown`
uint256 requiredCollateral = liquidationValueFull.mulDivUp(...);
```

Because `fullDebtAssets` is already rounded up, this extra floor trims the liquidation bonus before the collateral conversion, contradicting the intent to favor liquidators in this path.

Impact

Liquidators executing post-maturity liquidations receive slightly less collateral than the configured incentive promises.

Recommendation

Round the incentive multiplication upward in `_liquidatePostMaturity` so every step aligns with the "favor liquidator" intent.

Developer Response

Acknowledged - fixed in https://github.com/usual-dao/fira-lending-market/pull/31

2.11.2 No Enforced Safety Buffer Between LTV and LLTV

No Enforced Safety Buffer Between LTV and LLTV
Market creation currently accepts `ltv == lltv`, which undermines the intended buffer between the borrow limit (`LTV`) and liquidation threshold (`LLTV`). When `LTV` equals `LLTV`, any account that reaches the borrow limit becomes liquidatable immediately, negating the protective window LTV is supposed to provide.

Technical Details

`_createMarket` only checks `ltv <= lltv` (or allows `ltv == 0`) before storing the market configuration:

```
require(isLtvEnabled[marketParams.ltv] || marketParams.ltv == 0, ErrorsLib.
LTV_NOT_ENABLED);
require(marketParams.ltv <= marketParams.lltv || marketParams.ltv == 0, ErrorsLib.
LTV_EXCEEDS_LLTV);
```

Because equality is permitted, a market can configure ltv to match lltv. For markets that intend to run with an LTV buffer(`ltv != 0`), reaching the borrow limit should still be below the liquidation threshold, giving borrowers time to deleverage. Allowing equality removes that buffer and makes the LTV limit meaningless—users hitting the supposed "safe" limit can be liquidated immediately.

Impact

With an `LTV buffer =0` , user may be liquidated as soon as they reach the advertised borrow cap.

Recommendation

During market creation and updates, enforce a strict inequality whenever LTV is enabled: `require(marketParams.ltv < marketParams.lltv || marketParams.ltv == 0, …)` .
Markets that truly want no buffer can continue using `ltv = 0` , while buffered markets are guaranteed to keep LTV safely below LLTV.

Developer Response

Acknowledged - fixed in https://github.com/usual-dao/fira-lending-market/pull/29

2.12 Gas Savings Findings

None.

2.13 Informational Findings

2.13.1 Redundant code in _liquidatePostMaturity

The `_liquidatePostMaturity` function in `LendingMarket.sol` contains a redundant `UtilsLib.min` check inside an `if` block that already guarantees the condition.

2.14 Technical Details

In `src/lending_market/LendingMarket.sol`, the `_liquidatePostMaturity` function has the following logic:

```
        if (uint256(position_.collateral) >= requiredCollateral) {
            // Enough collateral to cover full debt at incentive: seize exact required (rounded up)
            seizedAssets = UtilsLib.min(requiredCollateral, uint256(position_.collateral));
            repaidAssetsPaid = fullDebtAssets;
        }
```

The `if` condition `uint256(position_.collateral) >= requiredCollateral` ensures that `requiredCollateral` is less than or equal to `position_.collateral`. Therefore, `UtilsLib.min(requiredCollateral, uint256(position_.collateral))` will always return `requiredCollateral`. The use of `UtilsLib.min` is unnecessary and adds gas overhead.

2.15 Impact

Gas inefficiency. The logic remains correct, but the redundant call consumes extra gas.

2.16 Recommendation

Remove the `UtilsLib.min` call and assign `requiredCollateral` directly to `seizedAssets`.

```
        if (uint256(position_.collateral) >= requiredCollateral) {
            // Enough collateral to cover full debt at incentive: seize exact required (rounded up)
            seizedAssets = requiredCollateral;
            repaidAssetsPaid = fullDebtAssets;
        }
```

Developer Response

Acknowledged - won't fix

2.16.1 SisuVaultPriceFeed hardcoded FLOOR leads to liquidation freeze or fund drain

The `SisuVaultPriceFeed` contract, intended to serve as a Chainlink-compatible oracle for `SisuVault` shares, implements a hardcoded floor price of 1.0 (in underlying asset decimals).

```
// src/oracles/SisuVaultPriceFeed.sol

function _pricePerShare() internal view returns (int256) {
    uint256 oneShare = 10 ** uint256(VAULT.decimals());
    uint256 assets = VAULT.convertToAssets(oneShare);
    if (assets < FLOOR) {
        return int256(FLOOR); // FLOOR is 1.0
    }
    return int256(assets);
}
```

If the `SisuVault` suffers losses (e.g., due to bad debt socialization in the `LendingMarket`), the actual exchange rate of shares-to-assets will drop below 1.0. However, the oracle will continue to report a price of 1.0.

2.17 Technical Details

This section details the specific conditions required for an attacker to profit from the bug.

2.17.1 The "Infinite Drain" Scenario (P_real < LTV)

The most severe impact occurs when the real share price drops below the LTV ratio (e.g., 0.88).
Prerequisites:

- **LTV**: 88% (0.88).
- **Oracle Price**: Fixed at 1.0 (due to bug).
- **Real Share Price**: Drops to 0.80 (e.g., due to a 20% loss in SisuVault).

Attack Loop:

1. **Flash Loan**: Attacker borrows 10,000,000 `USD0`.
2. **Mint Cheap Shares**:
    - Attacker deposits 10M `USD0` into `SisuVault` <- attacker will lose this asset but will gain more profit.
    - Since real price is 0.80, they receive 10M / 0.80 = 12.5M Shares.
    - Note: The `SisuVault` itself (ERC4626) correctly calculates the depressed share price (0.80) during deposits/withdrawals. The bug is isolated to the `SisuVaultPriceFeed` oracle, which incorrectly reports 1.0 to the `LendingMarket`.
3. **Inflated Borrow**:

- Vault automatically supplies 12.5M Shares to `LendingMarket` via SisuVault supply queue.
- Oracle values collateral at 12.5M * 1.0 = 12.5M USD.
- Borrow Power = 12.5M * 88% (LTV) = 11M USD0.
- Note: Borrowing is limited by `LTV` (0.88), not `LLTV` (0.9999), as `LendingMarket` enforces the stricter limit for new borrows.

4. **Profit & Exit**:

   - Attacker borrows 11M `USD0` <- this is what attacker gained.
   - Attacker repays the 10M Flash Loan.
   - **Net Profit**: 11M - 10M = 1M `USD0`.
   - **Cost**: 0 (excluding gas).

5. **Repeat**: The attacker can repeat this until the `LendingMarket` is drained of all `USD0` liquidity.

2.17.2 The "Liquidation Freeze" Scenario (LTV < P_real < 1.0)

If the price is between 0.88 and 1.0 (e.g., 0.9), the attacker cannot drain it instantly because `LTV` restricts borrowing. However, the protocol enters a "zombie" state:

1. **Rational Liquidators Strike**: Liquidators normally keep the protocol healthy.
2. **Unprofitable Liquidation**:

   - If a position needs liquidation, the Oracle says shares are worth 1.00.
   - The protocol asks the liquidator to pay ~$0.925 debt in exchange for 1 share (valued at $1.00 by Oracle).
   - The liquidator receives 1 share, but it's actually only worth $0.9 on the market.
   - **Result**: Liquidators lose money if they liquidate. They will stop liquidating.

3. **Escalation**: Bad debt accumulates unchecked until the price drops below 0.88, triggering the "Infinite Drain" scenario above.

2.18 Impact

If `SisuVault` shares are used as collateral in the `LendingMarket` (or any other protocol relying on this feed):

1. **Insolvency Masking:** The system will fail to recognize that the collateral has lost value.
2. **Arbitrage / Exploitation:** An attacker can buy/mint `SisuVault` shares at their true depressed value (e.g., 0.9 assets) and borrow against them at the floored oracle value (1.0 assets).
3. **Protocol Draining:** This effectively allows borrowing more than the collateral is worth, draining the lending pool and leaving lenders with bad debt.

2.19 Recommendation

Remove the hardcoded floor logic. The oracle should report the true `convertToAssets` value to accurately reflect the collateral's value, including any losses.

```
function _pricePerShare() internal view returns (int256) {
    uint256 oneShare = 10 ** uint256(VAULT.decimals());
    uint256 assets = VAULT.convertToAssets(oneShare);
-   if (assets < FLOOR) {
-       return int256(FLOOR);
-   }
    return int256(assets);
}
```

Addressing Manipulation Concerns:
We understand that the FLOOR design was added likely for avoiding price oracle manipulation. After fixing this issue, we also recommend to add additional circuit breaker functionalities to halt the oracle during extreme market conditions.

2.20 Appendix: Oracle Configuration and Call Chain

2.20.1 Configuration Context

The `ChainlinkOracleV2` contract supports two modes for pricing vault-like assets:

1. **Direct Vault Mode:** `BASE_VAULT` is set to the vault address. The oracle calls `convertToAssets` directly on the vault.
2. **Feed Mode:** `BASE_VAULT` is set to `address(0)`, and `BASE_FEED_1` is set to a contract that implements the Chainlink interface.

`SisuVaultPriceFeed` is specifically designed for **Feed Mode**. It wraps the vault's `convertToAssets` logic into a Chainlink-compatible interface (`latestRoundData`). When deployed, the system is configured as follows:

- `ChainlinkOracleV2.BASE_VAULT` = `address(0)`
- `ChainlinkOracleV2.BASE_FEED_1` = `address(SisuVaultPriceFeed)`

2.20.2 Execution Flow

When the `LendingMarket` values collateral, it triggers the following sequence:

1. **LendingMarket Request**: The `LendingMarket` needs to check the value of collateral (e.g., during `liquidate` or `borrow`). It calls `price()` on the configured oracle.

```
// src/lending_market/LendingMarket.sol
uint256 collateralPrice = IOracle(marketParams.oracle).price();
```

2. **Oracle Adapter**: The `marketParams.oracle` is a `ChainlinkOracleV2`. It calculates the price by querying its configured feeds. Since `BASE_VAULT` is `address(0)`, it relies entirely on the feed price.

```
// src/oracles/ChainlinkOracleV2.sol
return SCALE_FACTOR.mulDiv(
    BASE_VAULT.getAssets(...) * BASE_FEED_1.getPrice() * ...,
    ...
);
```

3. **Feed Library**: The `ChainlinkDataFeedLib` calls `latestRoundData` on the feed.

```
// src/libraries/ChainlinkDataFeedLib.sol
function getPrice(AggregatorV3Interface feed) internal view returns (uint256) {
    (, int256 answer,,,) = feed.latestRoundData();
    return uint256(answer);
}
```

4. **SisuVaultPriceFeed Execution**: The `SisuVaultPriceFeed` (configured as `BASE_FEED_1`) executes `latestRoundData`, which calls `_pricePerShare`.

```
// src/oracles/SisuVaultPriceFeed.sol
function latestRoundData() external view returns (...) {
    return (..., _pricePerShare(), ...);
}
```

5. **Floor Application**: `_pricePerShare` checks the vault's assets and applies the hardcoded floor if the share value has dropped below 1.0.

```
// src/oracles/SisuVaultPriceFeed.sol
if (assets < FLOOR) { return int256(FLOOR); }
```

This chain confirms that `LendingMarket` operations are directly affected by the incorrect price reported by `SisuVaultPriceFeed`.

2.21 Appendix: Bad Debt Impact Analysis

This appendix details how "bad debt" in the `LendingMarket` propagates to the `SisuVault` share price, triggering the floor bug in `SisuVaultPriceFeed`.

2.21.1 1. Bad Debt Creation (LendingMarket)

When a liquidation occurs and the collateral value is insufficient to cover the debt (plus the liquidation incentive), the protocol "socializes" the loss. This is done by reducing the total assets available to lenders (`totalSupplyAssets`) without burning the corresponding shares (`totalSupplyShares`).

```
// src/lending_market/LendingMarket.sol

function liquidate(...) external returns (...) {
    // ... (calculation of badDebtAssets) ...

    // Socialize bad debt by writing down LP assets
    if (badDebtAssets > 0) {
        market_.totalSupplyAssets -= badDebtAssets.toUint128();
    }
}
```

Result: The ratio `totalSupplyAssets / totalSupplyShares` decreases.

2.21.2 2. Vault Asset Valuation (SisuVault)

The `SisuVault` calculates its total assets by summing the value of its positions in the `LendingMarket`. It uses `expectedSupplyAssets` to convert its shares back to assets based on the current market exchange rate.

```
1 // src/sisu_vault/SisuVault.sol

3 function totalAssets() public view override returns (uint256 assets) {
4     for (uint256 i; i < withdrawQueue.length; ++i) {
5         assets += LENDING_MARKET.expectedSupplyAssets(_marketParams(withdrawQueue[i]),
  address(this));
6     }
7 }
```

The `expectedSupplyAssets` function (in `LendingMarketBalancesLib`) applies the exchange rate:

```
1 // src/libraries/periphery/LendingMarketBalancesLib.sol

3 function expectedSupplyAssets(...) internal view returns (uint256) {
4     // ...
5     (uint256 totalSupplyAssets, uint256 totalSupplyShares,,) = expectedMarketBalances(...
  );

7     // supplyShares * (totalSupplyAssets / totalSupplyShares)
8     return supplyShares.toAssetsDown(totalSupplyAssets, totalSupplyShares);
9 }
```

Result: Since `totalSupplyAssets` was reduced by bad debt, the `SisuVault`'s calculated `totalAssets` decreases proportionally.

2.21.3 3. Share Price Calculation (SisuVault)

The `SisuVault` is an ERC4626 vault. Its share price is determined by the ratio of `totalAssets` to `totalSupply` (vault shares).

```
1 // src/sisu_vault/SisuVault.sol

3 function _convertToAssets(...) internal view override returns (uint256) {
4     // ...
5     // shares * (totalAssets / totalSupply)
6     return shares.mulDiv(newTotalAssets + 1, newTotalSupply + 10 ** _decimalsOffset(),
  rounding);
7 }
```

Result: A drop in `totalAssets` (caused by bad debt) directly reduces the `convertToAssets` value (the share price).

2.21.4 4. Oracle Floor Trigger (SisuVaultPriceFeed)

Finally, the oracle reads this depressed share price. If the bad debt is significant enough to push the price below 1.0, the hardcoded floor activates, masking the insolvency.

```
1 // src/oracles/SisuVaultPriceFeed.sol

3 function _pricePerShare() internal view returns (int256) {
4     // ...
5     uint256 assets = VAULT.convertToAssets(oneShare);
6     if (assets < FLOOR) {
7         return int256(FLOOR); // Returns 1.0 even if real price is 0.9
8     }
9     return int256(assets);
10 }
```

Developer Response

Won't fix. The current implementation is by design. The SisuVault share is intended to serve as collateral for USD0, analogous to the USD0++ vault share mechanism in Euler. The protocol will employ robust off-chain automation to mitigate potential consequences. Furthermore, given the reliance on static oracles, a share price drop below 1.0 USD0 would indicate a fundamental system compromise. Consequently, while this issue might be deemed critical in a standard context, it is classified as info within the specific operational context of the Usual ecosystem.

2.21.5 Potential Redundant Health Check in withdrawCollateral

`withdrawCollateral` and `borrow` enforces both `_isHealthyLTV` and `_isHealthy` even when an LTV ceiling is configured. Because `ltv ≤ lltv` by construction, passing the LTV check(if the `LVT` is enabled) implies the position is already below the liquidation threshold, so the second check becomes redundant and wastes gas.

Technical Details

The function requires both guards:

```
1         require(_isHealthyLTV(marketParams, id, onBehalf), ErrorsLib.
  INSUFFICIENT_COLLATERAL_LTV);
2         require(_isHealthy(marketParams, id, onBehalf), ErrorsLib.
  INSUFFICIENT_COLLATERAL);
3         require(market[id].totalBorrowAssets <= market[id].totalSupplyAssets, ErrorsLib.
  INSUFFICIENT_LIQUIDITY);
```

For markets with `marketParams.ltv != 0`, `_isHealthyLTV` already ensures the collateralization ratio stays beneath `ltv`, which is guaranteed to be at most `lltv`. Therefore the `_isHealthy` call duplicates work (including oracle lookups) and increases gas cost. Only when LTV is disabled (`ltv == 0`) is the LLTV check necessary.

Impact

No safety issue, but every `withdrawCollateral` / `borrow` call pays extra gas and oracle reads for markets that set an LTV limit

Recommendation

Branch on the market configuration:

```
if (marketParams.ltv != 0) {
    require(_isHealthyLTV(...), ErrorsLib.INSUFFICIENT_COLLATERAL_LTV);
} else {
    require(_isHealthy(...), ErrorsLib.INSUFFICIENT_COLLATERAL);
}
```

Developer Response

Acknowledged - Won't fix

2.21.6 Indistinguishable Zero Result For Unknown Market

`getUserPosition` returns all-zero balances when called with `marketParams` that do not correspond to an existing market. Off-chain consumers therefore cannot tell whether the user truly has no position or whether the queried market was never created.

Technical Details

The `getter` derives the `market id`, fetches the market balances, and reads the stored position without verifying that the market exists:

```
function getUserPosition(MarketParams memory marketParams, address user)
    external
    view
    returns (...)
{
    Id id = marketParams.id();
    (uint256 totalSupplyAssets, uint256 totalSupplyShares, uint256 totalBorrowAssets,
uint256 totalBorrowShares) =
        LendingMarketBalancesLib.expectedMarketBalances(ILendingMarket(address(this)),
marketParams);

    Position storage userPosition = position[id][user];

    supplyShares = userPosition.supplyShares;
    supplyAssets = supplyShares.toAssetsDown(totalSupplyAssets, totalSupplyShares);

    borrowShares = userPosition.borrowShares;
    borrowAssets = borrowShares.toAssetsUp(totalBorrowAssets, totalBorrowShares);

    collateralAssets = userPosition.collateral;
}
```

If no market was created for that id, both `market[id]` and `position[id][user]` are their zero-initialized values, so the getter returns `(0,0,0,0,0)`, which is exactly the same output as a valid but empty position.

Recommendation

Consider mirroring the checks used in state-changing functions by requiring `market[id].lastUpdate != 0`

Developer Response

Acknowledged - fixed in https://github.com/usual-dao/fira-lending-market/pull/30