Usual: Fira Lending Market

Security Review

Cantina Managed review by:
Sujith Somraaj, Lead Security Researcher
Hake, Associate Security Researcher

November 19, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Usual is a Stablecoin DeFi protocol that redistributes control and redefines value sharing. It empowers users by aligning their interests with the platform's success.

From Nov 10th to Nov 17th the Cantina team conducted a review of fira-lending-market on commit hash f7592c08. The team identified a total of **16** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	3	1	2
Gas Optimizations	2	1	1
Informational	11	6	5
Total	**16**	**8**	**8**

2.1 Scope

The security review had the following components in scope for fira-lending-market on commit hash f7592c08:

```
src
├── interfaces
│   └── IAdaptiveCurveIrm.sol
├── irms
│   ├── AdaptiveCurveIrm.sol
│   └── FixedRateIrm.sol
├── lending_market
│   ├── LendingMarket.sol
│   └── USLLendingMarket.sol
├── libraries
│   └── periphery
│       ├── LendingMarketBalancesLib.sol
│       ├── LendingMarketLib.sol
│       └── LendingMarketStorageLib.sol
├── migrator
│   └── USLMigrator.sol
└── sisu_vault
    ├── PermissionedSisuVault.sol
    ├── PermissionedSisuVaultFactory.sol
    ├── SisuVault.sol
    └── SisuVaultFactory.sol
```

2.2 Cantina Managed Team Statement

Scope Limitation: This audit focused exclusively on the differences between the client's implementation and the Morpho codebase. The review did not encompass the entire system (Morpho codebase plus modifications), and therefore any functionality outside of the specific differences was considered out of scope.

Permissioned Markets Only: This review considered only permissioned markets. Should the client wish to enable permissionless markets, we recommend conducting additional testing and considering a follow-up security review to properly assess potential risks introduced by the dual liquidation path, including front-running conditions, market manipulation risks, and other factors such as market parameter validation and boundary checks.

3 Findings

3.1 Low Risk

3.1.1 Atomic bad debt realization enables share price manipulation attack

Severity: Low Risk

Context: SisuVault.sol#L595-L599

Description: The SisuVault immediately recognizes bad-debt losses from underlying lending markets within the same transaction, causing the vault's share price to drop atomically.

This creates interesting edge cases that flash loan-based attacks could exploit, where attackers can profit from temporary share price manipulation caused by socialized bad debt.

Recommendation: Consider avoiding realizing bad debt automatically. Instead, losses remain within the underlying Lending market and introduce a new variable that tracks them. This prevents share price manipulation attacks to a greater degree.

Usual: Acknowledged.

Cantina Managed: Acknowledged.

3.1.2 Liquidators prioritized over LPs in bad debt liquidations due to rounding, causing unnecessary losses to lenders

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Summary: The `_liquidatePostMaturity()` function uses downward rounding when calculating the value of seized collateral and the amount liquidators must repay. This systematic rounding in favor of liquidators results in LPs bearing a larger share of bad debt than necessary, creating a permanent value leak from the LP pool.

Description: When a position lacks sufficient collateral to cover its full debt plus liquidation incentive, the protocol enters a bad debt scenario. In this case, the liquidator seizes all available collateral and repays only what is economically justified based on that collateral's value. The issue arises from the rounding direction used in these calculations:

```
// Not enough collateral: seize all collateral, liquidator repays only the economically
↪  justified portion
seizedAssets = uint256(position_.collateral);
uint256 seizedValue = seizedAssets.mulDivDown(collateralPrice, ORACLE_PRICE_SCALE);
repaidAssetsPaid = seizedValue.wDivDown(liquidationIncentive);
```

The `seizedValue` calculation uses `mulDivDown`, which rounds down the collateral's value. Subsequently, `repaidAssetsPaid` also uses `wDivDown`, further rounding down the amount the liquidator must pay. This cascading downward rounding means:

1. The estimated value of seized collateral is slightly lower than its actual value.
2. The amount the liquidator pays is reduced accordingly.
3. The calculated bad debt (`badDebtAssets = fullDebtAssets - repaidAssetsPaid`) becomes larger than it should be.
4. LPs absorb more bad debt through the reduction of `totalSupplyAssets`.

This systematic bias accumulates over multiple liquidations, creating a permanent value transfer from LPs to liquidators.

Recommendation: Consider modifying the rounding direction to protect LPs who are absorbing the bad debt. Since LPs are taking the loss, calculations should round in their favor:

```
  // Not enough collateral: seize all collateral, liquidator repays only the economically
  ↪  justified portion
  seizedAssets = uint256(position_.collateral);
- uint256 seizedValue = seizedAssets.mulDivDown(collateralPrice, ORACLE_PRICE_SCALE);
```

```
- repaidAssetsPaid = seizedValue.wDivDown(liquidationIncentive);
+ uint256 seizedValue = seizedAssets.mulDivUp(collateralPrice, ORACLE_PRICE_SCALE);
+ repaidAssetsPaid = seizedValue.wDivUp(liquidationIncentive);
```

Usual: Fixed in PR 27.

Cantina Managed: Fix verified.

3.1.3 Single-step transfer mechanism could lead to USL liquidation functionality becoming inaccessible

Severity: Low Risk

Context: LendingMarket.sol#L184-L191

Description: The `setUslLiquidator()` function in the LendingMarket contract implements a single-step transfer mechanism for updating the USL liquidator address. This function allows the owner to directly set a new USL liquidator without requiring the new address to explicitly accept the role. The current implementation immediately transfers the USL liquidator role to the specified address:

```
function setUslLiquidator(address newUslLiquidator) external onlyOwner {
    require(newUslLiquidator != address(0), ErrorsLib.ZERO_ADDRESS);
    require(newUslLiquidator != uslLiquidator, ErrorsLib.ALREADY_SET);

    uslLiquidator = newUslLiquidator;

    emit EventsLib.SetUslLiquidator(newUslLiquidator);
}
```

This pattern presents operational risks as an incorrect address could be set accidentally, potentially rendering the USL liquidation functionality inaccessible. Since the USL liquidator role has privileged access through the `onlyUslLiquidator` modifier, setting an incorrect address could disrupt protocol operations that depend on this role.

Recommendation: Consider implementing a two-step transfer pattern for the USL liquidator role, similar to common ownership transfer patterns. This would involve having a pending USL liquidator that must explicitly accept the role:

```
address public pendingUslLiquidator;

function setUslLiquidator(address newUslLiquidator) external onlyOwner {
    require(newUslLiquidator != address(0), ErrorsLib.ZERO_ADDRESS);
    require(newUslLiquidator != uslLiquidator, ErrorsLib.ALREADY_SET);

    pendingUslLiquidator = newUslLiquidator;

    emit EventsLib.PendingUslLiquidator(newUslLiquidator);
}

function acceptUslLiquidatorRole() external {
    require(msg.sender == pendingUslLiquidator, ErrorsLib.NOT_PENDING_USL_LIQUIDATOR);

    uslLiquidator = msg.sender;
    delete pendingUslLiquidator;

    emit EventsLib.SetUslLiquidator(msg.sender);
}
```

This pattern ensures that the new USL liquidator address is valid and under the control of the intended party before the role transfer is completed.

Usual: Acknowledged.

Cantina Managed: Acknowledged.

3.2 Gas Optimization

3.2.1 Multiple early return conditions can be consolidated in `_isHealthyLTV()`

Severity: Gas Optimization

Context: LendingMarket.sol#L748-L749

Description: The `_isHealthyLTV()` function contains two separate early return conditions that can be combined into a single conditional statement:

```
if (marketParams.ltv == 0) return true;
if (position[id][borrower].borrowShares == 0) return true;
```

Having two separate if statements results in:

1. Two conditional jump instructions in the bytecode.
2. Potentially redundant JUMPDEST operations.
3. Slightly higher gas consumption for cases where the first condition is false but the second is true.

While the gas savings are minimal (approximately 3-6 gas per call when the second condition is true), consolidating these conditions improves code efficiency and readability.

Recommendation: Combine the two early return conditions using a logical OR operator:

```
if (marketParams.ltv == 0 || position[id][borrower].borrowShares == 0) return true;
```

Usual: Acknowledged.

Cantina Managed: Acknowledged.

3.2.2 Inefficient empty bytes literal `bytes(string(""))` instead of `hex""`

Severity: Gas Optimization

Context: SisuVault.sol#L412, SisuVault.sol#L810

Description: SisuVault uses an inefficient encoding for empty bytes: `bytes(string(""))` instead of the more gas-efficient `hex""`. Replace all instances of `bytes(string(""))` with `hex""` to slighly save some gas.

Usual: Fixed in PR 25.

Cantina Managed: Fix verified.

3.3 Informational

3.3.1 Market creation can be griefed through front-running (Client reported issue)

Severity: Informational

Context: USLLendingMarket.sol#L15-L21

Summary: The `createMarket()` function in the LendingMarket contract allows anyone to create a market with arbitrary `MarketConstants`. Since the market ID is generated solely from `MarketParams` and not from `MarketConstants`, an attacker can front-run legitimate market creation transactions to create the same market with malicious constants, effectively griefing the protocol's intended market configuration.

Description: The vulnerability exists in the public `createMarket()` function which can be called by anyone:

```
function createMarket(MarketParams memory marketParams, MarketConstants memory
↪  constants) external virtual {
    _createMarket(marketParams, constants);
}
```

The market ID is generated from the `MarketParams` struct using the `id()` function (from `MarketParamsLib`), which creates a deterministic identifier based on the market parameters. However, the `MarketConstants` struct, which contains critical configuration like `liquidationIncentive`, is not included in the ID calculation.

This creates an attack vector where:

1. An attacker monitors the mempool for legitimate `createMarket()` transactions.
2. Upon seeing one, the attacker front-runs it with the same `MarketParams` but different (potentially harmful) `MarketConstants`.
3. The attacker's transaction creates the market first with incorrect constants.
4. The legitimate transaction fails with `MARKET_ALREADY_CREATED` error.

The impact is that markets can be created with unintended liquidation incentives or other constants, potentially making the market unusable or creating unfavorable conditions for users.

Recommendation: The protocol team has already implementing the correct fix by restricting `createMarket()` to only be callable by the owner:

```
- function createMarket(MarketParams memory marketParams, MarketConstants memory
↪  constants) external virtual {
+ function createMarket(MarketParams memory marketParams, MarketConstants memory
↪  constants) external virtual onlyOwner {
      _createMarket(marketParams, constants);
  }
```

This ensures that only trusted parties can create markets with the appropriate constants, preventing griefing attacks while maintaining the protocol's intended market configuration control.

Usual: Fixed in PR 19.

Cantina Managed: Fix verified.

3.3.2 `PermissionedSisuVault` is not fully 4626 compliant

Severity: Informational

Context: PermissionedSisuVault.sol#L11

Description: The PermissionedSisuVault contract implements access controls via the onlyPermissioned modifier on the core ERC-4626 functions (deposit, mint, withdraw, redeem), restricting these operations to a single permissioned address.

However, the contract fails to override the corresponding max* query functions (maxDeposit, maxMint, maxWithdraw, maxRedeem) to reflect these permission restrictions.

Per EIP-4626:

- `maxDeposit()` MUST return the maximum amount of assets that can be deposited, and MUST return 0 if deposits are entirely disabled.
- `maxMint()` MUST return the maximum amount of shares that can be minted, and MUST return 0 if minting is entirely disabled.
- `maxWithdraw()` MUST return the maximum assets withdrawable, and MUST return 0 if withdrawals are entirely disabled.
- `maxRedeem()` MUST return the maximum shares redeemable, and MUST return 0 if redemptions are entirely disabled.

Recommendation: Override all four max* functions in `PermissionedSisuVault.sol` to return 0 for unpermissioned callers.

Usual: Acknowledged.

Cantina Managed: Acknowledged.

3.3.3 Sisu vault creation fails with zero timelock

Severity: Informational

Context: SisuVault.sol#L128

Description: The `SisuVault.sol` contract enforces a minimum timelock of 1 day (MIN_TIMELOCK) during vault initialization, preventing the creation of vaults with a zero timelock. This validation occurs in the constructor through the `_checkTimelockBounds()` function.

Recommendation: Consider allowing the creation of vaults with zero timelock (recommended for flexibility).

```
+ if(initialTimelock != 0) _checkTimelockBounds(initialTimelock);
- _checkTimelockBounds(initialTimelock);
```

Usual: Acknowledged.

Cantina Managed: Acknowledged.

3.3.4 Missing validation for `maturityGracePeriod`

Severity: Informational

Context: LendingMarket.sol#L259-L263

Description: The `maturityGracePeriod` parameter in `MarketConstants` is not validated during market creation, allowing it to be set to any value from 0 to `type(uint128).max`. This can lead to either immediate liquidations without a grace period or indefinitely delayed liquidations that prevent proper risk management for expired USD0++ collateral positions.

Recommendation: Add validation bounds for maturityGracePeriod during market creation:

```
// Add to ConstantsLib.sol
/// @dev The minimum maturity grace period (e.g., 6 hours)
uint256 constant MIN_MATURITY_GRACE_PERIOD = 6 hours;

/// @dev The maximum maturity grace period (e.g., 30 days)
uint256 constant MAX_MATURITY_GRACE_PERIOD = 30 days;


// Add validation for maturityGracePeriod
require(
 constants.maturityGracePeriod >= MIN_MATURITY_GRACE_PERIOD,
 ErrorsLib.MATURITY_GRACE_PERIOD_TOO_LOW
);

require(
  constants.maturityGracePeriod <= MAX_MATURITY_GRACE_PERIOD,
  ErrorsLib.MATURITY_GRACE_PERIOD_TOO_HIGH
);
```

Usual: Acknowledged.

Cantina Managed: Acknowledged.

3.3.5 Unused constant `LIQUIDATION_INCENTIVE_OFFSET`

Severity: Informational

Context: LendingMarketStorageLib.sol#L44

Description: The `LIQUIDATION_INCENTIVE_OFFSET` constant is defined but never used in any getter function. Since both `maturityGracePeriod` and `liquidationIncentive` are packed in a single slot, there's no need for this constant.

Recommendation: Consider removing the unused constant and rename `MATURITY_GRACE_PERIOD_OFFSET` to `MATURITY_GRACE_PERIOD_AND_LIQUIDATION_INCENTIVE_OFFSET`.

Usual: Fixed in PR 23.

Cantina Managed: Fix verified.

3.3.6 Restrict creating markets with `ltv > lltv`

Severity: Informational

Context: LendingMarket.sol#L249

Description: The `_createMarket()` function does not validate the relationship between LTV and LLTV. When LLTV > LTV, the LTV parameter becomes completely useless because the borrow function enforces both checks, and the lower value is the actual limit.

This could create a lot of confusion:

- Market creators might set LLTV, thinking it's the liquidation threshold, not realizing it also limits borrowing.
- Users see two different limits and don't understand why they can't borrow up to the higher one.

Recommendation: Consider adding a validation to ensure LTV is always less than or equal to LLTV during market creation.

Usual: Fixed in PR 22.

Cantina Managed: Fix verified.

3.3.7 Normal liquidation can frontrun USL liquidation, causing loss to USL liquidator

Severity: Informational

Context: LendingMarket.sol#L498-L503, LendingMarket.sol#L639

Description: When a position is both unhealthy (LTV > LLTV) and post-maturity, it can be liquidated through two different paths with different liquidation incentives:

1. `liquidate()` - Normal liquidation with dynamic incentive (typically 5% for 86% LLTV).
2. `liquidateExpiredUSL()` - Post-maturity liquidation with fixed incentive (typically 7.5% in tests but can go higher till 15%).

This creates MEV/frontrunning opportunities where the path with lower incentive benefits the borrower but denies profit to the other liquidator.

For a market with LLTV = 86% (like a typical USL market):

Method	User Loses (collateral)	Liquidator Pays (loan)	Net Loss to User
liquidate (5%)	105 USD	100 USD	5 USD value
liquidateExpiredUSL (10%)	110 USD	100 USD	10 USD value

Proof of Concept: The following proof of concept demonstrates this issue for a market with 86% LLTV. Under same scenario, the usual liquidation route resulted in collateral savings for the user.

```
pragma solidity ^0.8.0;

import {Test} from "forge-std/Test.sol";

import "src/lending_market/USLLendingMarket.sol";
import "src/irms/FixedRateIrm.sol";
import "src/interfaces/ILendingMarket.sol";
import "src/mocks/ERC20Mock.sol";
import "src/mocks/OracleMock.sol";

import "forge-std/console.sol";

contract CollateralToken is ERC20Mock {
    function getEndTime() external view returns (uint256) {
```

```
        return 0;
    }
}

contract AuditTest is Test {
    using MarketParamsLib for MarketParams;

    USLLendingMarket public lendingMarket;
    FixedRateIrm public irm;
    OracleMock public oracle;

    ERC20Mock public loanToken;
    CollateralToken public collateralToken;

    MarketParams public params;

    address owner = makeAddr("OWNER");

    address lender = makeAddr("LENDER");
    address borrower = makeAddr("BORROWER");
    address liquidator = makeAddr("LIQUIDATOR");

    uint256 LLTV = 0.86 ether; // 60%
    uint256 LTV = 0.86 ether; // 80%

    function setUp() external {
        loanToken = new ERC20Mock();
        vm.label(address(loanToken), "LoanToken");

        collateralToken = new CollateralToken();
        vm.label(address(collateralToken),  "CollateralToken");

        oracle = new OracleMock();
        oracle.setPrice(1e36);

        lendingMarket = new USLLendingMarket(owner);
        irm = new FixedRateIrm();

        params.loanToken = address(loanToken);
        params.collateralToken = address(collateralToken);
        params.oracle = address(oracle);
        params.irm = address(irm);
        params.lltv = LLTV;
        params.ltv = LTV;

        vm.startPrank(owner);
        lendingMarket.enableIrm(address(irm));

        lendingMarket.enableLltv(LLTV); // 80%
        lendingMarket.enableLtv(LTV);
        lendingMarket.setUslLiquidator(address(liquidator));
        irm.setBorrowRate(params.id(), 0.5e16 / uint256(365 days));

        MarketConstants memory constants;
        constants.maturityGracePeriod = uint128(1 days);
        constants.liquidationIncentive = uint128(0.15e18); // 15%

        /// MAX_LIQUIDATION_INCENTIVE_FACTOR = 1.15e18
        /// WAD = 1e18
        lendingMarket.createMarket(
            params,
            constants
        );
        vm.stopPrank();
    }
```

```
function test_liquidate() external {
    vm.startPrank(lender);
    loanToken.setBalance(lender, 100e18);
    loanToken.approve(address(lendingMarket), 100e18);

    lendingMarket.supply(
        params,
        100e18,
        0,
        lender,
        hex""
    );
    vm.stopPrank();


    vm.startPrank(borrower);
    collateralToken.setBalance(borrower, 100e18);
    collateralToken.approve(address(lendingMarket), 100e18);

    lendingMarket.supplyCollateral(
        params,
        100e18,
        borrower,
        hex""
    );

    lendingMarket.borrow(
        params,
        86e18,
        0,
        borrower,
        borrower
    );
    vm.stopPrank();

    vm.startPrank(liquidator);
    loanToken.setBalance(liquidator, 100e18);
    loanToken.approve(address(lendingMarket), 100e18);

    vm.warp(block.timestamp + 1 days);
    oracle.setPrice(0.9e36);

    (, uint256 p,) = lendingMarket.position(params.id(), borrower);

    lendingMarket.liquidate(
        params,
        borrower,
        0,
        p,
        hex""
    );

    console.log(loanToken.balanceOf(address(lendingMarket)));
    (,,uint256 collateralLeft) = lendingMarket.position(params.id(), borrower);
    console.log("collateralLeft in liquidate():", collateralLeft);
    vm.stopPrank();
}

function test_liquidate_usl() external {
     vm.startPrank(lender);
    loanToken.setBalance(lender, 100e18);
    loanToken.approve(address(lendingMarket), 100e18);

    lendingMarket.supply(
        params,
        100e18,
```

```
            0,
            lender,
            hex""
        );
        vm.stopPrank();


        vm.startPrank(borrower);
        collateralToken.setBalance(borrower, 100e18);
        collateralToken.approve(address(lendingMarket), 100e18);

        lendingMarket.supplyCollateral(
            params,
            100e18,
            borrower,
            hex""
        );

        lendingMarket.borrow(
            params,
            86e18,
            0,
            borrower,
            borrower
        );
        vm.stopPrank();

        vm.startPrank(liquidator);
        loanToken.setBalance(liquidator, 100e18);
        loanToken.approve(address(lendingMarket), 100e18);

        vm.warp(block.timestamp + 1 days);
        oracle.setPrice(0.9e36);

        lendingMarket.liquidateExpiredUSL(
            params,
            borrower,
            hex""
        );

        console.log(loanToken.balanceOf(address(lendingMarket)));
        (,,uint256 collateralLeft) = lendingMarket.position(params.id(), borrower);
        console.log("collateralLeft in liquidate usl():", collateralLeft);
        vm.stopPrank();

    }
}
```

Recommendation: Block normal liquidation post-maturity entirely.

```
// In liquidate() function
require(
  !_isPostMaturity(marketParams, id),
  ErrorsLib.USE_POST_MATURITY_LIQUIDATION
);

function _isPostMaturity(MarketParams memory params, Id id) internal {
  try IUsd0PP(marketParams.collateralToken).getEndTime() returns (uin256 endTime) {
      if(block.timestamp >=
         endTime + marketConstants[id].maturityGracePeriod
       ) revert(ErrorsLib.USE_POST_MATURITY_LIQUIDATION);
  } catch {
   /// no op
  }
}
```

> Additional Note: The converse of this issue is also true. If the Post-maturity liquidation incentive is lower than the dynamic incentive, users will suffer more from front-running, which can be grievous.

Usual: Acknowledged.

Cantina Managed: Acknowledged.

3.3.8 Inconsistent implementation between `_previewAccrueInterest()` and `expectedMarketBalances()` violates documented equivalence

Severity: Informational

Context: LendingMarket.sol#L138

Description: The `_previewAccrueInterest()` function explicitly claims in its documentation to mirror `expectedMarketBalances()`:

```
/// @dev Computes interest-adjusted market totals without mutating state.
/// Mirrors the logic of `LendingMarketBalancesLib.expectedMarketBalances` using
↪  borrowRateView.
function _previewAccrueInterest(MarketParams memory marketParams, Id id)
```

However, the implementations diverge in subtle ways, causing them to return different results for the same inputs. In `_previewAccrueInterest()` function:

```
Market memory m = market[id];

uint256 totalSupplyAssets = m.totalSupplyAssets;   // ← Widens to uint256
uint256 totalSupplyShares = m.totalSupplyShares;   // ← Widens to uint256
uint256 totalBorrowAssets = m.totalBorrowAssets;   // ← Widens to uint256
uint256 totalBorrowShares = m.totalBorrowShares;   // ← Widens to uint256

// ...
totalBorrowAssets += interest;                      // ← Direct uint256 addition
totalSupplyAssets += interest;                      // ← Direct uint256 addition

if (m.fee != 0) {
  uint256 feeShares = feeAmount.toSharesDown(totalSupplyAssets - feeAmount,
  ↪  totalSupplyShares);
  totalSupplyShares += feeShares;                 // ← Direct uint256 addition
}

return (totalSupplyAssets, totalSupplyShares, totalBorrowAssets, totalBorrowShares);
```

In `expectedMarketBalances()` function:

```
Market memory market = lendingMarket.market(id);

// Works directly with Market struct fields (uint128)

// ...
market.totalBorrowAssets += interest.toUint128();  // ← Explicit cast to uint128
market.totalSupplyAssets += interest.toUint128();  // ← Explicit cast to uint128

if (market.fee != 0) {
  uint256 feeShares = feeAmount.toSharesDown(market.totalSupplyAssets - feeAmount,
  ↪  market.totalSupplyShares);
  market.totalSupplyShares += feeShares.toUint128();  // ← Explicit cast to uint128
}

return (market.totalSupplyAssets, market.totalSupplyShares, market.totalBorrowAssets,
↪  market.totalBorrowShares);
```

Recommendation: Consider adapting `_previewAccrueInterest()` function to truly mirror `expectedMarketBalances()` function.

Usual: Fixed in PR 21.

Cantina Managed: Fix verified.

3.3.9 Inline documentation improvements

Severity: Informational

Context: PermissionedSisuVaultFactory.sol#L12, PermissionedSisuVaultFactory.sol#L14, PermissionedSisuVault.sol#L8, PermissionedSisuVault.sol#L30

Description: The following documentation improvements could be made to avoid confusion and improve clarity:

1. `PermissionedSisuVaultFactory.sol`:

```
- /// @title SisuVaultFactory
+ /// @title PermissionedSisuVaultFactory


- /// @notice This contract allows to create Sisu vaults, and to index them easily.
+ ///@notice This contract allows to create Permissioned Sisu vaults, and to index them
↪  easily.
```

2. `PermissionedSisuVault.sol`:

```
- /// @title SisuVault
+ /// @title PermissionedSisuVault
```

Missing documentation for `setPermissionedAddress()` function and SetPermissionedAddress event.

Usual: Fixed in PR 24.

Cantina Managed: Fix verified.

3.3.10 Incorrect rounding in bad debt calculation leads to understated bad debt shares event emission

Severity: Informational

Context: LendingMarket.sol#L658-L660

Description: The `_liquidatePostMaturity()` function uses incorrect rounding when calculating bad debt shares, resulting in understated bad debt amounts on `EventsLib.Liquidate` event emission. This occurs because `repaidAssetsPaid` is converted to shares using `toSharesUp()` instead of `toSharesDown()`, causing the resulting bad debt shares to be smaller than they should be.

Recommendation: Modify the bad debt shares calculation to use toSharesDown() instead of toSharesUp():

```
  badDebtShares =
-     borrowShares - repaidAssetsPaid.toSharesUp(market_.totalBorrowAssets,
↪  market_.totalBorrowShares);
+     borrowShares - repaidAssetsPaid.toSharesDown(market_.totalBorrowAssets,
↪  market_.totalBorrowShares);
```

This ensures that the repaid portion is conservatively rounded down, resulting in the correct (potentially slightly higher) bad debt shares calculation that fully accounts for all bad debt in the system.

Usual: Fixed in PR 26.

Cantina Managed: Fix verified.

3.3.11 Requiring positive liquidation incentive as additional protection against bad debt socialization

Severity: Informational

Context: LendingMarket.sol#L259-L261

Description: The current implementation allows `liquidationIncentive` to be set to zero during market creation, potentially exposing the protocol to increased bad debt risk. While the check ensures the liquidation incentive does not exceed the maximum allowed value, it does not enforce a minimum positive value.

In the `_createMarket()` function:

```
require(
    constants.liquidationIncentive <= MAX_LIQUIDATION_INCENTIVE_FACTOR - WAD,
    ErrorsLib.LIQUIDATION_INCENTIVE_NOT_SET
);
```

When `liquidationIncentive` is zero, liquidators would only recover the debt amount without any profit margin. This removes the economic incentive for liquidations, particularly problematic when gas costs are high, leading to a decrease in willingness for liquidators to participate.

The lack of timely liquidations increases the likelihood of positions becoming undercollateralized beyond recovery, leading to bad debt that must be socialized among liquidity providers.

Recommendation: Consider adding a minimum liquidation incentive requirement to ensure liquidators maintain sufficient economic motivation:

```
  require(
      constants.liquidationIncentive <= MAX_LIQUIDATION_INCENTIVE_FACTOR - WAD,
      ErrorsLib.LIQUIDATION_INCENTIVE_NOT_SET
  );
+ require(
+     constants.liquidationIncentive > 0,
+     ErrorsLib.LIQUIDATION_INCENTIVE_TOO_LOW
+ );
```

This additional validation would ensure that liquidators always have a positive incentive to maintain protocol health, reducing the risk of bad debt accumulation that would need to be socialized.

Usual: Acknowledged.

Cantina Managed: Acknowledged.