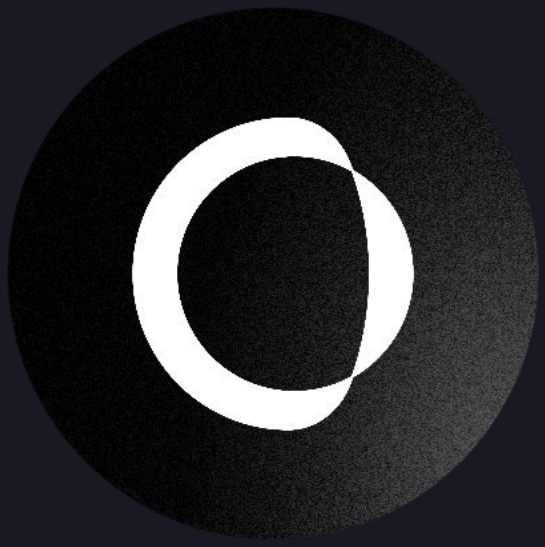
SHERLOCK

Security Review For Usual



Collaborative Audit Prepared For: **Usual**
Lead Security Expert(s): **eeyore**
xiaoming90
Date Audited: **November 4 - November 8, 2025**

Introduction

Fira Lending Market is a suite of Solidity contracts implementing a modular multi-asset lending protocol. It enables users to supply and borrow ERC-20 assets, accrue interest based on utilization, enforce loan-to-value (LTV) and liquidation thresholds, and price assets via on-chain oracles. The system integrates ERC-4626 vaults so that liquidity providers can interact through a vault interface, and all contracts are designed for Ethereum-compatible chains and tested with Foundry.

Scope

Repository: usual-dao/fira-lending-market

Audited Commit: 8075bdecb9c105a1d1368f984549dec908dacca1

Final Commit: f7592c08d66b3b2978bafbf2a4a1abb0868d4789

Files:

- src/interfaces/AggregatorV3Interface.sol
- src/interfaces/IAdaptiveCurveIrm.sol
- src/interfaces/ICallbacks.sol
- src/interfaces/IChainlinkOracleV2Factory.sol
- src/interfaces/IChainlinkOracleV2.sol
- src/interfaces/IERC20.sol
- src/interfaces/IERC4626.sol
- src/interfaces/IFixedRateIrm.sol
- src/interfaces/IIrm.sol
- src/interfaces/ILendingMarket.sol
- src/interfaces/IOracle.sol
- src/interfaces/IPermissionedSisuVaultFactory.sol
- src/interfaces/ISisuVaultFactory.sol
- src/interfaces/ISisuVault.sol
- src/interfaces/IUsd0PP.sol
- src/irms/AdaptiveCurveIrm.sol
- src/irms/FixedRateIrm.sol
- src/lending_market/LendingMarket.sol
- src/lending_market/USLLendingMarket.sol

- src/libraries/ChainlinkDataFeedLib.sol
- src/libraries/ConstantsLib.sol
- src/libraries/ExpLib.sol
- src/libraries/MarketParamsLib.sol
- src/libraries/MathLibInt128.sol
- src/libraries/MathLib.sol
- src/libraries/PendingLib.sol
- src/libraries/periphery/LendingMarketBalancesLib.sol
- src/libraries/periphery/LendingMarketLib.sol
- src/libraries/periphery/LendingMarketStorageLib.sol
- src/libraries/SafeTransferLib.sol
- src/libraries/SharesMathLib.sol
- src/libraries/UtilsLib.sol
- src/libraries/VaultLib.sol
- src/migrator/USLMigrator.sol
- src/oracles/ChainlinkOracleV2Factory.sol
- src/oracles/ChainlinkOracleV2.sol
- src/sisu_vault/PermissionedSisuVaultFactory.sol
- src/sisu_vault/PermissionedSisuVault.sol
- src/sisu_vault/SisuVaultFactory.sol
- src/sisu_vault/SisuVault.sol

Final Commit Hash

f7592c08d66b3b2978bafbf2a4a1abb0868d4789

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.

- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	0	8

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue L-1: Residual USD0 after debt repayment during migration should be swept to the account's owner [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-usual-lending-market-nov-4th/issues/41

This issue has been acknowledged by the team but won't be fixed at this time.

Vulnerability Detail

The MigrateBatchFlow.migrationBatch() function demonstrates how the migration is intended to be executed.

It was observed that after repaying the debt, any remaining USD0 will be swept to the `user` address. Note that the `user` address is a sub-account.

```
File: MigrateBatchFlow.t.sol
110:         // 5) Repay Euler debt and sweep remainder back to user via swapper
 ↪  multicall
111:         bytes[] memory swapperCalls = new bytes[](2);
112:         swapperCalls[0] = abi.encodeWithSignature(
113:             "repay(address,address,uint256,address)", address(usd0),
 ↪  EULER_USD0_DEBT, userDebt, user
114:         ); // @audit-info user => account (Receiver of the repay)
115:         swapperCalls[1] =
 ↪  abi.encodeWithSignature("sweep(address,uint256,address)", address(usd0), 1,
 ↪  user);
```

The following is an extract from Euler's integration guide:

> **If you send ERC-20 tokens directly to a sub-account address, those tokens will be lost.** Typical ERC-20 contracts do not understand the EVC authentication system and cannot recover or recognize tokens sent to sub-accounts. Always interact with ERC-20 tokens using your main address.

In this case, since any remaining USD0 tokens after debt repayment are swept to the sub-account, they will be lost.

Impact

If any residual USD0 tokens remain after the migration, they will be lost. Based on the migration script, the script will always borrow the actual amount of debt owned in Euler from Fira and repay it in Euler. Thus, it is unlikely that any residual USD0 or a meaningful amount of USD0 will remain after the debt repayment. As such, marking this as Informational.

Code Snippet

https://github.com/sherlock-audit/2025-11-usual-lending-market-nov-4th/blob/d38e419bff9e804bdce2bc43bb64e8dae6b2a915/fira-lending-market/test/migrator/MigrateBatchFlow.t.sol#L115

Tool Used

Manual Review

Recommendation

Consider forwarding the remaining USD0 to the account's owner rather than the sub-account (`user`).

Discussion

SHAKOTN

As solidity script won't be used for the actual migration we will ack this and make sure it's implemented properly on the FE side

Issue L-2: Migration might not work for some users if the existing debt exceeds the maximum borrowable amount [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-usual-lending-market-nov-4th/issues/42

This issue has been acknowledged by the team but won't be fixed at this time.

Vulnerability Detail

In the current Euler USL vault, the LTV is set to 86%. Assume the upcoming Fira USL vault's LTV is set to 88% per the test script here.

```
File: BaseMarketSetup.t.sol
21: contract BaseMarketSetup is Test {
..SNIP..
34:     uint256 internal constant LTV = 0.88e18;
35:     uint256 internal constant LLTV = 0.9999e18;
```

In USL, 1 USD0 == 1USD0PP == 1 USD. It was noted that USL launched in February 2025, and the expected launch date for the Fira Lending market is January 2026.

Assume that Bob deposited 1000 USD0PP as collateral and borrowed the maximum possible amount up to the LTV limit in February 2025 when USL is first launched. In this case, Bob borrowed 860 USD0 (1000 * 0.86), so the debt will be 860.

Euler's USL vault has a borrowing rate of 5% per year. In January 2026, around 10 months have passed, and Bob will incur an additional 35 debts due to the interest.

```
860 * (5%/12 months) * 10 months = 35
```

Bob's total debt will be around 895 USD0 (860 + 35).

The `MigrateBatchFlow.migrationBatch()` function demonstrates how the migration should be executed.

The migration will first redeem all 1000 USD0PP collateral from Euler and deposit it to Fira. Then, it will attempt to borrow the amount of USD0 from Fira to repay the user's debt in Euler.

In this case, the user's debt is 895 in Euler. Thus, the migration will attempt to borrow 895 USD0 from Fira. However, since the LTV of Fira USL is 88%, the maximum amount that can be borrowed is only USD 880, and the transaction will revert, halting the migration.

Impact

For some users, the current migration setup will not work.

Code Snippet

https://github.com/sherlock-audit/2025-11-usual-lending-market-nov-4th/blob/d38e419bff9e804bdce2bc43bb64e8dae6b2a915/fira-lending-market/test/migrator/MigrateBatchFlow.t.sol#L104

Tool Used

Manual Review

Recommendation

Although increasing the LTV further can potentially work around this issue, it is not recommended, as it will increase the protocol's risk. The higher the LTV, the less collateral (USD0PP) backing the debt (USD0).

In the migration process, consider adding a step that allows the user to repay a portion of the debts if they exceed the maximum borrowable amount.

Discussion

SHAKOTN

I think there are several workarounds around this.

First is what you've recommended wrt repaying manually debt partially on Euler. I don't think we want to add another step to migration as it will stretch it out even further and we want it to stay as lean as possible

Issue L-3: Authorization given to the migrator is not revoked at the end of the migration process [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-usual-lending-market-nov-4th/issues/43

This issue has been acknowledged by the team but won't be fixed at this time.

Vulnerability Detail

The MigrateBatchFlow.migrationBatch() function demonstrates how the migration is intended to be executed.

Assume that Bob's account has multiple sub-accounts. In this case, the `migrationBatch()` function will be called against each subaccount.

```
File: MigrateKnownUsersFixedRate.t.sol
24:     function testMigrationBatch() public {
25:         for (uint256 i = 0; i < knownUsers.length; i++) {
26:             address user = knownUsers[i];
27:             migrationBatch(user);
28:         }
29:     }
```

At the start of the migration process, it will attempt to execute the `market.setAuthorization()` function to grant the migrator contract permission, as shown in Line 42 below.

```
File: MigrateBatchFlow.t.sol
34:     function migrationBatch(address user) public {
35:         (,, uint256 preBorrowAssets,, uint256 preCollateralAssets) =
↪  market.getUserPosition(params, user);
36:         uint256 preEulerShares = IERC20(EULER_USD0PP_VAULT).balanceOf(user);
37:         assertEq(preCollateralAssets, 0, "collateral supplied");
38:         assertEq(preBorrowAssets, 0, "borrowed assets");
39:         assertGt(preEulerShares, 0, "pre-migration Euler shares should exist");
40:         address owner = evc.getAccountOwner(user);
41:         vm.startPrank(owner);
42:         market.setAuthorization(address(migrator), true);
43:         uint256 userDebt = IBorrowing(EULER_USD0_DEBT).debtOf(user);
```

However, the issue is that at the end of the migration process, the script does not revoke the authorization given to the migrator.

Thus, when the `migrationBatch()` function is executed against Bob's second sub-account, the script will attempt to call `market.setAuthorization()` function again, and it will revert this time due to the check in Line 534, as the authorization is already.

```
File: LendingMarket.sol
532:     /// @inheritdoc ILendingMarketBase
533:     function setAuthorization(address authorized, bool newIsAuthorized)
 ↪  external {
534:         require(newIsAuthorized != isAuthorized[msg.sender][authorized],
 ↪  ErrorsLib.ALREADY_SET);
535:
536:         isAuthorized[msg.sender][authorized] = newIsAuthorized;
537:
538:         emit EventsLib.SetAuthorization(msg.sender, msg.sender, authorized,
 ↪  newIsAuthorized);
539:     }
540:
```

Impact

Migration might revert unexpectedly if a user has multiple subaccounts.

Code Snippet

https://github.com/sherlock-audit/2025-11-usual-lending-market-nov-4th/blob/d38e419bff9e804bdce2bc43bb64e8dae6b2a915/fira-lending-market/src/lending_market/LendingMarket.sol#L534

https://github.com/sherlock-audit/2025-11-usual-lending-market-nov-4th/blob/d38e419bff9e804bdce2bc43bb64e8dae6b2a915/fira-lending-market/test/migrator/MigrateBatchFlow.t.sol#L42

Tool Used

Manual Review

Recommendation

Consider the following solutions:

1. Check if the authorization is already set before calling the `market.setAuthorization()` function
2. Revoke the authorization to the migrator at the end of each migration process. Since the migration is completed, there is no valid reason for the migrator to have authorization over the user's account.

Discussion

SHAKOTN

As script is just a demonstration to how migration will work, we ack this but won't fix it since the code in tests won't be used for migration. However, we will note this and make sure it's implemented properly on the front end bundler

Issue L-4: Market creation can be disrupted by frontrunning with incorrect constants [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-usual-lending-market-nov-4th/issues/44

Vulnerability Detail

The LendingMarket.createMarket() function allows anyone to create a market by providing `MarketParams` and `MarketConstants`. However, the market ID is generated solely from the `MarketParams` and does not include the `MarketConstants`.

```
File: LendingMarket.sol
213:     function createMarket(MarketParams memory marketParams, MarketConstants
 ↪  memory constants) external {
214:         Id id = marketParams.id();
215:         require(isIrmEnabled[marketParams.irm], ErrorsLib.IRM_NOT_ENABLED);
216:         require(isLltvEnabled[marketParams.lltv], ErrorsLib.LLTV_NOT_ENABLED);
217:         require(isLtvEnabled[marketParams.ltv] || marketParams.ltv == 0,
 ↪  ErrorsLib.LTV_NOT_ENABLED);
218:         require(market[id].lastUpdate == 0, ErrorsLib.MARKET_ALREADY_CREATED);
```

The market ID generation in `MarketParamsLib.sol` only hashes the `MarketParams`:

```
File: MarketParamsLib.sol
15:     function id(MarketParams memory marketParams) internal pure returns (Id
 ↪  marketParamsId) {
16:         assembly ("memory-safe") {
17:             marketParamsId := keccak256(marketParams,
 ↪  MARKET_PARAMS_BYTES_LENGTH)
18:         }
19:     }
```

The `MarketConstants` struct contains:

- `maturityGracePeriod` - determines when positions can be liquidated post-maturity
- `liquidationIncentive` - determines the liquidator's profit margin

Since the constants are stored separately but not part of the ID, an attacker can frontrun legitimate market creation with the same `MarketParams` but with incorrect `MarketConstants`. Once created, the market cannot be recreated with the correct constants because the check `require(market[id].lastUpdate == 0, ErrorsLib.MARKET_ALREADY_CREATED)` will fail.

Impact

Anyone can frontrun market creation with incorrect `MarketConstants` (wrong `liquidationIncentive` or `maturityGracePeriod`), preventing the intended market configuration from being deployed.

While there is no direct benefit and such an event may be unlikely, if it occurs, it would force a complete contract redeployment to allow for correct market creation with the intended parameters.

Code Snippet

https://github.com/sherlock-audit/2025-11-usual-lending-market-nov-4th/blob/d38e419bff9e804bdce2bc43bb64e8dae6b2a915/fira-lending-market/src/lending_market/LendingMarket.sol#L214

https://github.com/sherlock-audit/2025-11-usual-lending-market-nov-4th/blob/d38e419bff9e804bdce2bc43bb64e8dae6b2a915/fira-lending-market/src/libraries/MarketParamsLib.sol#L15-L19

Tool Used

Manual Review

Recommendation

Consider including constants in the market ID generation.

Discussion

SHAKOTN

While this is a possibility, we've decided not to do any changes here.

Reasoning is simple: if anyone to ever front run us creating markets, we can simply recreate the market by adjusting LTV or LLTV parameter in the MarketParams by 1 wei and it will be a different market, and I don't think anyone will grief it over and over spending gas for nothing just so we will run out of LTV and LLTV values(with 1 wei change)

Issue L-5: Unnecessary `UtilsLib.min` function [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-usual-lending-market-nov-4th/issues/45

This issue has been acknowledged by the team but won't be fixed at this time.

Vulnerability Detail

The `UtilsLib.min` in Line 611 is unnecessary as there will never be an instance where `position_.collateral` is smaller than `requiredCollateral` due to the `uint256(position_.collateral) >= requiredCollateral` condition being `True`.

```
File: LendingMarket.sol
579:     function _liquidatePostMaturity(
..SNIP..
606:         uint256 seizedAssets;
607:         uint256 repaidAssetsPaid;
608:         uint256 badDebtShares;
609:         if (uint256(position_.collateral) >= requiredCollateral) {
610:             // Enough collateral to cover full debt at incentive: seize exact
 ↪  required (rounded up)
611:             seizedAssets = UtilsLib.min(requiredCollateral,
 ↪  uint256(position_.collateral));
612:             repaidAssetsPaid = fullDebtAssets;
613:         } else {
```

Impact

N/A. Optimization and Refactoring only.

Code Snippet

Tool Used

Manual Review

Recommendation

Consider implementing the following changes:

```
        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));
+           seizedAssets = requiredCollateral;
            repaidAssetsPaid = fullDebtAssets;
        } else {
```

Discussion

SHAKOTN

Acknowledged

Issue L-6: `getUserPosition()` underreports balances by skipping accrued interest [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-usual-lending-market-nov-4th/issues/46

Vulnerability Detail

The LendingMarket.getUserPosition() view is meant to provide migration helpers with up-to-date balances. But it only converts the stored share balances using the persisted totals, never simulating the elapsed interest since `_accrueInterest` last ran.

```
File: LendingMarket.sol
109:     function getUserPosition(MarketParams memory marketParams, address user)
...
127:         supplyShares = userPosition.supplyShares;
128:         supplyAssets =
 ↪  supplyShares.toAssetsDown(marketStorage.totalSupplyAssets,
 ↪  marketStorage.totalSupplyShares);
...
130:         borrowShares = userPosition.borrowShares;
131:         borrowAssets =
 ↪  borrowShares.toAssetsUp(marketStorage.totalBorrowAssets,
 ↪  marketStorage.totalBorrowShares);
```

The contract updates `totalSupplyAssets` / `totalBorrowAssets` (and `totalSupplyShares` when fees mint LP shares) inside `_accrueInterest`, which is called by state-changing flows (supply, withdraw, borrow, repay, etc.). When no transaction touches the market, the stored totals omit all passive growth from the IRM rate and the fee recipient's dilution of share supply. Because `getUserPosition` does not simulate that accrual, it returns pre-interest values.

Impact

`getUserPosition` exposes supply and borrow asset balances that ignore accrued interest.

Code Snippet

https://github.com/sherlock-audit/2025-11-usual-lending-market-nov-4th/blob/d38e419bff9e804bdce2bc43bb64e8dae6b2a915/fira-lending-market/src/lending_market/LendingMarket.sol#L127-L133

Tool Used

Manual Review

Recommendation

Consider mirroring the virtual-accrual logic used by `expectedMarketBalances` (e.g., factor it into a shared `_previewAccrueInterest` helper that calls `borrowRateView`) so `getUserPosition` returns interest-adjusted balances, or clearly document that the function reports pre-interest values.

Issue L-7: Sisu Vault's Price Per Share will decrease if a high liquidation incentive is configured [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-usual-lending-market-nov-4th/issues/47

This issue has been acknowledged by the team but won't be fixed at this time.

Vulnerability Detail

If the liquidation incentive is set to max (1.15), the lender (which is the SisuVault backed by Usual DAO) will always incur bad debt when `liquidateExpiredUSL()` is executed.

Assume that a borrow position with 880 USD0 debt and 1000 USD0++ collateral is created on the first day of the launch (1 January 2026 00:00:00 GMT+0).

USD0PP will mature at 1844335800 (11 June 2028 11:30:00 GMT+0), and the position can be force liquidated 1 day after maturity by the USL liquidator.

The following attempts to compute the user's debt one day after the maturity.

```
// Removed vault.deposit(INITIAL_DEPOSIT, permissioned);
// Added vault.deposit(880e18, permissioned);
// Change BORROW_RATE from 1585489599 (5%) to 158548959 (0.5%)
function testInterestAtEnd() public {
    vm.warp(1767225600); // 1 January 2026 00:00:00 GMT+0
    address borrower2 = address(0xB0B2);
    Id id = params.id();

    vm.startPrank(borrower2);
    deal(address(usd0pp), borrower2, 1000e18);
    vm.startPrank(borrower2);
    usd0pp.approve(address(market), type(uint256).max);
    market.supplyCollateral(params, 1000e18, borrower2, hex"");
    market.borrow(params, 880e18, 0, borrower2, borrower2);
    vm.stopPrank();
    (,, uint256 borrowAsset,,) = market.getUserPosition(params, borrower2);
    console.log("User's borrowAsset = ", borrowAsset);

    market.accrueInterest(params);
    (uint128 totalSupplyAssets1,,uint128 totalBorrowAssets1,,,) = market.market(id);
    console.log("Market's totalSupplyAssets1 = ", totalSupplyAssets1);
    console.log("Market's totalBorrowAssets1 = ", totalBorrowAssets1);

    vm.warp(1844335800 + 1 days); // 11 June 2028 11:30:00 GMT+0 + 1 days
    market.accrueInterest(params);
    (,, uint256 borrowAsset2,,) = market.getUserPosition(params, borrower2);
```

```
    console.log("User's borrowAsset post maturity + 1 day = ", borrowAsset2);
    (uint128 totalSupplyAssets2,,uint128 totalBorrowAssets2,,,) = market.market(id);
    console.log("Market's totalSupplyAssets post maturity + 1 day = ",
    ↪  totalSupplyAssets2);
    console.log("Market's totalBorrowAssets post maturity + 1 day = ",
    ↪  totalBorrowAssets2);
}
```

```
Logs:
  User's borrowAsset =  880000000000000000000
  Market's totalSupplyAssets1 =  880000000000000000000
  Market's totalBorrowAssets1 =  880000000000000000000
  User's borrowAsset post maturity + 1 day =  890836890334205686080
  Market's totalSupplyAssets post maturity + 1 day =  890836890334205686080
  Market's totalBorrowAssets post maturity + 1 day =  890836890334205686080
```

Thus, one day after USD0PP maturity, the position's debt will be 890.8368903 USD0.

The max `repaidAssetsPaid` is 869.4 USD0 (1000 USD0/1.15), and the lender (SisuVault)'s `totalSupplyAssets` will decrease by 21.27167291 USD0 (890.8368903 - 869.4) due to bad debt, which is not covered by the liquidation. This will, in turn, lead to a decrease in PPS after the `liquidateExpiredUSL` is executed.

It was observed that if the liquidation incentive is set between 1.1226 and 1.1500, the vault will lose some assets (also leading to a decrease in PPS) after the `liquidateExpiredUSL()` is executed due to bad debt.

However, there is no actual loss here as the Usl Liquidator belongs to the Usual, and the profits earned by the liquidator will eventually be routed back to the Usual DAO.

Refer to this spreadsheet for the calculation.

Impact of the flooring of the vault's PPS to 1.0 It was observed that the planned `SisuVaultPriceFeed` intends to floor the price per share (PPS) at 1.00.

If the liquidation incentive is set between 1.1364 and 1.1500, the PPS of the SisuVault will fall below 1.0 after the `liquidateExpiredUSL()` takes place.

As a result, there will be a desync between the price feed result (floored at 1.0) and the actual SisuVault's PPS (below 1.0).

```
contract SisuVaultPriceFeed {
..SNIP..
    /// @notice Computes price per 1 share in units of underlying assets (with
    ↪  underlying decimals).
    /// @return The price per share.
    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);
    }
```

Impact

If the off-chain infrastructure relies on the price feed's PPS, it might lead to unexpected results if there is a desync between the price feed result (floored at 1.0) and the actual SisuVault's PPS (below 1.0).

In addition, it can break the invariant of the Usual Treasury itself. For a brief moment, between when bad debt from liquidation is accounted for in the Treasury's total collateral assets and when the liquidator's address replenishes the funds, the invariant that USD0 is fully backed by its RWA can be broken.

Code Snippet

https://github.com/sherlock-audit/2025-11-usual-lending-market-nov-4th/blob/d38e419bff9e804bdce2bc43bb64e8dae6b2a915/fira-lending-market/src/lending_market/USLLendingMarket.sol#L21

Tool Used

Manual Review

Recommendation

Take note of the impact of `liquidateExpiredUSL()` on the SisuVault when configured in the liquidation incentive. Certain value of liquidation incentive will lead to a decrease in the vault's PPS after `liquidateExpiredUSL()` is executed.

If the liquidation incentive has to be set above 1.136, the off-chain parties must be aware that there may be a desync between the price feed result (floored at 1.0) and the actual SisuVault's PPS (below 1.0) after `liquidateExpiredUSL()` is executed.

Discussion

SHAKOTN

Ack

Issue L-8: Factory vault creation reverts because factory is not the owner [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-usual-lending-market-nov-4th/issues/48

Vulnerability Detail

`PermissionedSisuVaultFactory.createPermissionedSisuVault()` deploys a new `PermissionedSisuVault` and then immediately calls `setPermissionedAddress` on it.

```
File: src/sisu_vault/PermissionedSisuVaultFactory.sol
62:         permissionedSisuVault.setPermissionedAddress(permissionedAddress);
```

However, `PermissionedSisuVault.setPermissionedAddress()` is restricted by `onlyOwner`, and ownership is assigned to the `initialOwner` argument inside the `SisuVault` constructor.

```
File: src/sisu_vault/PermissionedSisuVault.sol
30:     function setPermissionedAddress(address _permissionedAddress) external
 ↪  onlyOwner {
31:         if (_permissionedAddress == address(0)) revert ErrorsLib.ZeroAddress();
32:         permissionedAddress = _permissionedAddress;
33:         emit SetPermissionedAddress(_permissionedAddress);
34:     }
```

Consequently, the factory is never the owner of the vault it just deployed (unless `initialOwner` is incorrectly set to the factory address itself). When the factory tries to call `setPermissionedAddress`, the `onlyOwner` modifier reverts. The factory therefore cannot create permissioned vaults under normal usage.

Impact

`createPermissionedSisuVault()` will revert for every call where `initialOwner` is not the factory address.

Code Snippet

https://github.com/sherlock-audit/2025-11-usual-lending-market-nov-4th/blob/d38e419bff9e804bdce2bc43bb64e8dae6b2a915/fira-lending-market/src/sisu_vault/PermissionedSisuVaultFactory.sol#L62-L62

https://github.com/sherlock-audit/2025-11-usual-lending-market-nov-4th/blob/d38e419bff9e804bdce2bc43bb64e8dae6b2a915/fira-lending-market/src/sisu_vault/PermissionedSisuVault.sol#L30-L34

Tool Used

Manual Review

Recommendation

Skip setting the permissioned address inside the factory and let the vault owner configure it after deployment, or temporarily transfer vault ownership to the factory so it can set the address before handing ownership back.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.