Silo Finance

Silo Core

Security Assessment Report

Version: 2.0

April, 2025

Contents

Introduction 2
- Disclaimer . . . 2
- Document Structure . . . 2
- Overview . . . 2

Security Assessment Summary 3
- Scope . . . 3
- Approach . . . 3
- Coverage Limitations . . . 4
- Findings Summary . . . 4

Detailed Findings 5

Summary of Findings 6
- NFTs Minted By `SiloFactory` Can Be Burned By Any User . . . 7
- Time Based Multiplier Can Grow With Little Impact . . . 8
- `LiquidationHelper` May Accumulate Dust Which Can Be Stolen . . . 10
- Bad Debt Accrues Interest In Silo . . . 12
- Potential Unwanted Result Upon Out-Of-Range Input . . . 13
- Pragma Solidity Version Range Allows Breaking Changes . . . 14
- Unnecessary Write Operations In `_createOracles()` . . . 15
- No Emergency Pause Mechanism For Critical Silo Operations . . . 17
- Miscellaneous General Comments . . . 18

A Test Suite 22

B Vulnerability Severity Classification 23

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Silo Finance components. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Silo Finance components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Silo Finance components in scope.

Overview

Silo is a risk-isolated lending market that allows users to deposit tokens to earn interest, or to use them as collateral to then borrow other tokens.

Silo Core comprises of the main smart contracts of the protocol, implementing lending logic, managing and isolating risk, and acting as a vault for assets.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the Silo Finance repository.

The scope of this time-boxed review was strictly limited to the following files and directories at commit 6631f79.

The fixes of the identified issues were assessed at commit 8def80e.

1. `Silo.sol`
2. `SiloConfig.sol`
3. `SiloDeployer.sol`
4. `SiloFactory.sol`
5. `SiloLens.sol`
6. `interestRateModel/*`
7. `lib/*`
8. `liquidation/*`
9. `utils/*`

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support the Solidity component of the review, the testing team also utilised the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Aderyn: `https://github.com/Cyfrin/aderyn`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 9 issues during this assessment. Categorised by their severity:

- Medium: 2 issues.
- Low: 2 issues.
- Informational: 5 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Silo Finance components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
SILO-01	NFTs Minted By `SiloFactory` Can Be Burned By Any User	Medium	Resolved
SILO-02	Time Based Multiplier Can Grow With Little Impact	Medium	Closed
SILO-03	`LiquidationHelper` May Accumulate Dust Which Can Be Stolen	Low	Closed
SILO-04	Bad Debt Accrues Interest In Silo	Low	Closed
SILO-05	Potential Unwanted Result Upon Out-Of-Range Input	Informational	Closed
SILO-06	Pragma Solidity Version Range Allows Breaking Changes	Informational	Resolved
SILO-07	Unnecessary Write Operations In `_createOracles()`	Informational	Resolved
SILO-08	No Emergency Pause Mechanism For Critical Silo Operations	Informational	Closed
SILO-09	Miscellaneous General Comments	Informational	Resolved

SILO-01	NFTs Minted By `SiloFactory` Can Be Burned By Any User		
Asset	`SiloFactory.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `burn()` function lacks access control, allowing anyone to burn deployer NFT tokens, resulting in Silo fees being redirected from the deployer to Silo DAO.

When `SiloFactory` creates a Silo marketplace, it mints an `ERC721` NFT token to the deployer. The owner of this NFT is the recipient of Silo fees. The NFT can be burned to redirect all Silo fees to the Silo DAO.

Anyone can call `SiloFactory.burn()` function to burn the NFT:

```
function burn(uint256 _siloIdToBurn) external virtual {
    _burn(_siloIdToBurn);
}
```

Note, since the Silo DAO is the recipient of the funds, it has the ability to redirect them back to the deployers in an unlikely event of an exploit. Given the low probability of the entire DAO being compromised or acting maliciously, along with the absence of economic incentives for an attacker of carrying out such attack, the overall severity rating of this finding is reduced.

Recommendations

Add access control checks to `SiloFactory.burn()` ensuring that only the token owner can call it.

Resolution

As of commit 8def80e, access controls have been added to the `burn()` function:

```
require(msg.sender == _ownerOf(_siloIdToBurn), NotYourSilo());
```

SILO-02	Time Based Multiplier Can Grow With Little Impact		
Asset	`InterestRateModelV2.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

It is possible to build up `Tcrit`, a time based multiplier of the interest rate, at just above the critical rate without incurring punitive interest. Subsequent borrowing will then be an order of magnitude more expensive, despite two almost identical borrowing patterns.

The issue occurs because the value of `Tcrit` builds up over time based on whether the utilisation rate of the vault is above or below the fixed threshold `ucrit`. However, the impact of being above `ucrit` is proportionate to the difference between `ucrit` and `u`, the utilisation rate, as seen on line [312]:

```
_l.rp = _c.kcrit * (decimalPoints + Tcrit) / decimalPoints * (_l.u - _c.ucrit) / decimalPoints;
```

In this code, `_l.u` is the utilisation rate and `_c.ucrit` is the `ucrit` threshold.

However, the impact of `_l.u` being above or below `_c.ucrit` is far more pronounced on the value of `Tcrit`, the time based interest multiplier. It either increases or decreases depending on this one condition:

```
Tcrit = Tcrit + _c.beta * _l.T;
```

```
Tcrit = _max(0, Tcrit - _c.beta * _l.T);
```

Because of the minimal impact on interest but high impact on `Tcrit`, it is possible for the utilisation rate to stay just over the `ucrit` threshold with no significant impact on the interest charged. If the utilisation rate then sharply increases, the difference in the interest charged can be as high as an order of magnitude. This is because of the expression `(decimalPoints + Tcrit)` which would now be a very high number but was previously low impact because of the low value of the expression `(_l.u -_c.ucrit)`.

It is unlikely (although not impossible) that this would be exploited by an attacker, but it could lead to unexpectedly high interest rate surges, possibly leading to surprising and unpredictable liquidations.

Recommendations

This issue could be addressed in the user interface by making clear when `Tcrit` is at a high value and warning borrowers of this fact, however, this would not protect borrowers in a situation where the sudden increase in utilisation is due to a withdrawal.

Consider increasing or decreasing `Tcrit` by an amount that scales with the level above `ucrit` that the utilisation rate is at. For example:

```
Tcrit = Tcrit + _c.beta * _l.T * (_l.u - _c.ucrit);
```

Resolution

The development team acknowledged the issue and resolved no code changes were required at this time, although the calculation of `Tcrit` may be updated in future in a manner similar to that recommended.

SILO-03	`LiquidationHelper` May Accumulate Dust Which Can Be Stolen		
Asset	`LiquidationHelper.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `onFlashLoan()` function has no access control, as a result, it is possible for an attacker to steal dust that has accumulated within the contract.

The only liquidation scenario that is currently supported is liquidations involving the external swap of collateral to debt using a decentralised exchange (DEX).

For swaps involving large amounts, dust can accumulate to a meaningful value, especially over multiple transactions. If multiple liquidation events occur for the collateral asset to debt asset before a liquidation event in the opposite position, this accumulated dust will not be recovered and will remain in the contract's balance.

This is particularly likely if there are more loans for the debt asset compared to the collateral asset within the silo. The accumulated dust of the collateral asset can then be stolen by an attacker as follows:

- The attacker calls `onFlashLoan()` directly, passing the address of the collateral asset (which has accumulated dust) for the `_debtAsset` parameter. The attacker also ensures that the values for `_maxDebtToCover` and `_fee` add up to the total dust amount. The `_data` parameter will contain a malicious value for `_liquidation.hook`.

```
function onFlashLoan(
    address /* _initiator */,
    address _debtAsset,
    uint256 _maxDebtToCover,
    uint256 _fee,
    bytes calldata _data
)
    external
    returns (bytes32)
```

- The execution of `_liquidation.hook.liquidationCall()` on line [92] would call a malicious contract that simply returns random values for `_withdrawCollateral` and `_repayDebtAssets`. These variables are not used within the function so the values are irrelevant.

```
(
    _withdrawCollateral, _repayDebtAssets
) = _liquidation.hook.liquidationCall({
    _collateralAsset: _liquidation.collateralAsset,
    _debtAsset: _debtAsset,
    _user: _liquidation.user,
    _maxDebtToCover: _maxDebtToCover,
    _receiveSToken: false
});
```

- On line [101] `flashLoanWithFee` will be equal to `_maxDebtToCover + _fee`, which is not just the value of the accumulated dust amount, but also the contract's balance for the collateral token.

```
uint256 flashLoanWithFee = _maxDebtToCover + _fee;
```

- The attacker sets `_liquidation.collateralAsset` equal to the `_debtAsset` parameter which is the collateral token. This ensures that the `if` statement on line [103] executes instead of the `else` statement, ensuring that a swap operation does not take place. The `balance` variable will return the contract's balance for the collateral token, which is also equal to `flashLoanWithFee`. This will result in the `_transferToReceiver()` operation passing zero, which ensures that `TOKENS_RECEIVER` does not collect any tokens.

```
if (_liquidation.collateralAsset == _debtAsset) {
    uint256 balance = IERC20(_liquidation.collateralAsset).balanceOf(address(this));
    // bad debt is not supported, we will get underflow on bad debt
    _transferToReceiver(_liquidation.collateralAsset, balance - flashLoanWithFee);
}
```

- Finally, on line [123] the contract grants approval to the attacker for `flashLoanWithFee`, thus allowing them to remove the accumulated dust amount from the contract's balance.

```
IERC20(_debtAsset).approve(msg.sender, flashLoanWithFee);
```

Recommendations

To address this, the function needs to implement access control to ensure that only the flash loan lender is able to call it.

This can be done by creating a `transient` variable to store the address of the flash loan lender using the `flashLoanFrom` parameter in `executeLiquidation()`. This variable could then be used to verify that `msg.sender` is the intended caller. This follows the EIP-3156 flash loan specification for the borrower contract implementation to ensure access control.

To ensure safety, the `executeLiquidation()` function would also have to be updated. This is because this function allows the caller to pass in the value for `_flashLoanFrom` which could point to a malicious contract. This malicious contract would pass the access control checks recommended here and execute the attack mentioned earlier.

As a result, since the flash loan lender is the silo, the `executeLiquidation()` function should restrict the `_flashLoanFrom` parameter to the addresses of the silo. This will ensure that only a legitimate flash loan lender can call the `onFlashLoan()` function.

Resolution

The development team pointed out that this contract is not part of the main protocol and as such will have limited utilisation. Additionally, there may be a revised version available with a dust rescue function.

SILO-04	Bad Debt Accrues Interest In Silo		
Asset	`SiloLendingLib.sol`, `PartialLiquidationLib.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

In the Silo design, bad debt is not socialised among collateral providers. Liquidators are not guaranteed to repay the bad debt as there is no incentive for them to fully cover it.

However, bad debt within the Silo accrues interest over time, further increasing its value. As utilization rates rise, interest rates also increase, compounding the bad debt. In markets with high utilisation, the rate of compounding accelerates even more, increasing the problem. Over time, this could cause both the bad debt and market utilization to reach extremely high levels.

Recommendations

Address bad debt through separate accounting and modify the interest calculation logic to prevent interest from accruing on bad debt.

Resolution

The development team acknowledged the issue and resolved no code changes were required at this time.

SILO-05	Potential Unwanted Result Upon Out-Of-Range Input
Asset	`Hook.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The function `shareTokenTransfer()` accepts an arbitrary `uint256` value as an input (`_tokenType`). This can result in unintended behavior if `_tokenType > 2` .

Recommendations

The testing team understands that this unusual pattern may be intentional to accommodate `IShareToken.HookSetup.tokenType` . However, since the token type is known beforehand, it would be more reliable to use a type such as `ISilo.AssetType` to prevent incorrect calculations.

Resolution

The development team acknowledged the issue and resolved no code changes were required at this time.

SILO-06	Pragma Solidity Version Range Allows Breaking Changes
Asset	`RevertLib.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The contract `RevertLib` specifies the pragma range as `pragma solidity >=0.7.6 <=0.9.0;`. This range includes potential breaking changes that could lead to unintended behavior in the contract.

Recommendations

Consider using `pragma solidity ^0.8.28` as in other contracts.

Resolution

As of commit 8def80e, the pragma was changed to `pragma solidity ^0.8.28`.

SILO-07	Unnecessary Write Operations In `_createOracles()`
Asset	`SiloDeployer.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `_createOracle()` function performs unnecessary write operations on `_siloInitData` as follows:

```
function _createOracles(ISiloConfig.InitData memory _siloInitData, Oracles memory _oracles) internal {
        _siloInitData.solvencyOracle0 = _siloInitData.solvencyOracle0 != address(0)
            ? _siloInitData.solvencyOracle0
            : _createOracle(_oracles.solvencyOracle0);

        _siloInitData.maxLtvOracle0 = _siloInitData.maxLtvOracle0 != address(0)
            ? _siloInitData.maxLtvOracle0
            : _createOracle(_oracles.maxLtvOracle0);

        _siloInitData.solvencyOracle1 = _siloInitData.solvencyOracle1 != address(0)
            ? _siloInitData.solvencyOracle1
            : _createOracle(_oracles.solvencyOracle1);

        _siloInitData.maxLtvOracle1 = _siloInitData.maxLtvOracle1 != address(0)
            ? _siloInitData.maxLtvOracle1
            : _createOracle(_oracles.maxLtvOracle1);
    }
```

Here, `_siloInitData.solvencyOracle0`, `_siloInitData.maxLtvOracle0`, `_siloInitData.solvencyOracle1`, and `_siloInitData.maxLtvOracle1` are written back to themselves if their values are non-zero which is inefficient.

A similar pattern exists for `totalCollateralAssets` on line [**121**] of `SiloSolvencyLib.sol`.

Recommendations

The function could be refactored to reduce this inefficiency if implemented as follows:

```
function _createOracles(ISiloConfig.InitData memory _siloInitData, Oracles memory _oracles) internal {
        if(_siloInitData.solvencyOracle0 == address(0)){
           _siloInitData.solvencyOracle0 = _createOracle(_oracles.solvencyOracle0);
        }
        if(siloInitData.maxLtvOracle0 == address(0)){
            siloInitData.maxLtvOracle0 = _createOracle(_oracles.maxLtvOracle0);
        }
        if(_siloInitData.solvencyOracle1 == address(0)){
            _siloInitData.solvencyOracle1 = _createOracle(_oracles.solvencyOracle1);
        }
        if(siloInitData.maxLtvOracle1 == address(0)){
            siloInitData.maxLtvOracle1 = _createOracle(_oracles.maxLtvOracle1);
        }
  }
```

Resolution

As of commit 8def80e, the function was changed to the recommended pattern.

SILO-08	No Emergency Pause Mechanism For Critical Silo Operations
Asset	`Actions.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Silo is designed to support two assets in the market. Users can provide one asset as collateral to borrow the other. However, the `_MAX_LTV0` parameter is immutable. Additionally, there is no pausing mechanism for borrowing or other critical operations.

If one of the assets in the Silo market gets hacked (similar to the infinite BNB token minting exploit), it takes time for the actual market value of the token to decrease.

However, attackers can exploit this time lag by minting large amounts of the token and supplying it as collateral in the silo to borrow all of the other asset in the market, thereby draining it. Once the hacked token's value drops, the protocol would have significant bad debt and become insolvent.

Recommendations

To prevent such attacks, the vault deployer should have emergency functions to pause borrowing or set the max LTV to 0, thereby avoiding such attacks.

Resolution

The development team acknowledged the issue and resolved no code changes were required at this time.

SILO-09	Miscellaneous General Comments
Asset	All contracts
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Possible Zero Address For Immutable Variables**

   Related Asset(s): LiquidationHelper.sol

   The constructor initialises the values of `NATIVE_TOKEN` , `EXCHANGE_PROXY` , and `TOKENS_RECEIVER` . However, there are no checks to ensure that their respective parameter values are non-zero before assignment, especially since they are all immutable.

   Consider adding a check to ensure that the parameter values passed by the constructor for each of the aforementioned immutable variables are non-zero.

2. **TODO Comment In Production Code**

   Related Asset(s): DexSwap.sol

   There is a TODO comment present on line [18] of `DexSwap.sol`.

   Address and remove all TODO comments found in the codebase.

3. **Mismatch Between Comment And Code**

   Related Asset(s): SiloERC4626Lib.sol and DexSwap.sol

   In `SiloERC4626Lib.sol` the comment on line [30] mentions that the deposit limit for the vault is `type(uint128).max` . However, the code implementation has this value set to `type(uint256).max` instead. Also, in `DexSwap.sol` on line [31] the comment here mentions that the `fillQuote()` function must attach `ETH` equal to the value field from the API response. This does not appear to be implemented however as the function is not marked `payable` .

   Consider updating the comment to match the actual implementation.

4. **Code Consistency**

   Related Asset(s): SiloConfig.sol

   The code on line 103 assigns the value of `_configData0.silo` to `_COLLATERAL_SHARE_TOKEN0` , while `_PROTECTED_COLLATERAL_SHARE_TOKEN0` is assigned the value of `_configData0.protectedShareToken` .

   To improve code consistency, consider assigning the value of `_configData0.collateralShareToken` instead of `_configData0.silo` to `_COLLATERAL_SHARE_TOKEN0` . Since `_configData0.silo` and `_configData0.collateralShareToken` have identical values, the result of the operation will remain the same.

   A similar issue occurs on line [125]. Consider applying the same change there for consistency.

5. **Missing Parentheses**

   Related Asset(s): Hook.sol

   The code on line [220] is missing parentheses. Parentheses improve readability and ensure correct evaluation.

   Consider changing the code to the following:

   ```
   return (_action & _expectedHook) == _expectedHook;
   ```

6. **Lack Of NatSpec Comments**

   Related Asset(s): Actions.sol

   The library `Actions` does not include NatSpec comments describing the functionalities, inputs, and outputs of each function.

   Consider adding comprehensive NatSpec comments to this library, as well as other libraries and contracts, to serve as a reference for developers.

7. **Unnecessary Use of Parentheses**

   Related Asset(s): Rounding.sol

   The code on lines [**8-33**] uses parentheses for `Math.Rounding.Floor` and `Math.Rounding.Ceil`, which are unnecessary.

   Consider removing the parentheses to simplify the code.

8. **Invalid SPDX License**

   Related Asset(s): ShareCollateralTokenLib.sol

   The contract `ShareCollateralTokenLib` specifies `SPDX-License-Identifier: UNLICENSED` which is an invalid SPDX license.

   Consider using a valid SPDX license identifier such as `BUSL-1.1` that is used in other contracts.

9. **Repetitive Conversion Of Asset Type**

   Related Asset(s): SiloERC4626Lib.sol

   The code on lines 56-68 contains the following repetitive command:

   ```
   ISilo.AssetType(uint256(_collateralType))
   ```

   Consider caching the `AssetType` into a variable as this saves around 100 gas.

   A similar issue can also be found on lines [**101-120**] where `ISilo.AssetType(uint256(_args.collateralType))` is repeated several times.

10. **Inaccurate Comparison On Function `isBelowMaxLtv()`**

    Related Asset(s): SiloSolvencyLib.sol

    The function `isBelowMaxLtv()` compares the `ltv` with `_collateralConfig.maxLtv` using a less then or equal to operator on line [**75**].

    ```
    return ltv <= _collateralConfig.maxLtv;
    ```

    The code above returns true if `ltv == _collateralConfig.maxLtv`, which will be inconsistent with the function name because the `ltv` is not below `maxLtv` but equals to `maxLtv`.

    Consider updating the code on line [**75**] to better reflect the function name.

    ```
    return ltv < _collateralConfig.maxLtv;
    ```

11. **Caching Value During Initialisation Phase**

    Related Asset(s): ShareCollateralToken.sol

    The function `decimals()` performs a computation by calling `ShareTokenLib.decimals()` and adding the return value to `SiloMathLib._DECIMALS_OFFSET`. This process can be optimised by caching the decimals value through an initialisation function. By doing so, the computation is performed once during initialisation, and the cached result can be accessed as needed.

    Consider caching the decimals value in an initialisation function to improve gas efficiency.

12. **Potentially Inaccurate `hooksBefore`**
Related Asset(s): GaugeHookReceiver.sol
The calls to `_setHookConfig()` on lines [**65 and 86**] assume that `hooksBefore` has never been configured. However, this assumption may be incorrect if the `setGauge()` function is called a second time. Consequently, the `hooksBefore` data stored in `SiloHookReceiver._setHookConfig()` and the emitted variable `_hooksBefore` could be inaccurate.
It is also worth noting that `HOOKS_BEFORE_NOT_CONFIGURED` is a `uint24`, whereas the input `_hooksBefore` in `SiloHookReceiver._setHookConfig()` is a `uint256`. This results in a data type mismatch.
To improve data accuracy, retrieve `hooksBefore` by calling `SiloHookReceiver._getHooksBefore()` and use it to replace `HOOKS_BEFORE_NOT_CONFIGURED`. Additionally, change the data type of the second input parameter in `SiloHookReceiver._setHookConfig()` from `uint256` to `uint24`.
13. **Data Type Mismatch**
Related Asset(s): SiloHookReceiver.sol
The function `_getHooksBefore()` returns `hooksBefore`, which is of type `uint256`. However, `_hookConfig[_silo].hooksBefore` is of type `uint24`. The assignment code on line [**25**] `hooksBefore = _hookConfig[_silo].hooksBefore` has a data type mismatch issue.
A similar problem also occurs in the function `_getHooksAfter()`, where `hooksBefore` is of type `uint256`, but `_hookConfig[_silo].hooksAfter` is of type `uint24`.
Consider updating `hooksBefore` and `hooksAfter` from `uint256` to `uint24`.
14. **Typos**
Related Asset(s): InterestRateModelV2.sol, SiloMathLib.sol
    - In `interestRateModel/InterestRateModelV2.sol` line [**66**], "uitn256" should read "uint256".
    - In `interestRateModel/InterestRateModelV2.sol` line [**461**], "bee able to" should read "be able to".
    - In `lib/SiloMathLib.sol` line [**67**], "save to uncheck" should read "safe to uncheck".

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team's responses to the raised issues above are as follows.

1. **Possible Zero Address For Immutable Variables**
Related Asset(s): LiquidationHelper.sol
The zero address checks were added.
2. **`TODO` Comment In Production Code**
Related Asset(s): DexSwap.sol
The TODO comment has been removed.

3. **Mismatch Between Comment And Code**

   Related Asset(s): SiloERC4626Lib.sol and DexSwap.sol

   The comments were updated to match the code.

4. **Code Consistency**

   Related Asset(s): SiloConfig.sol

   The development team acknowledged the issue and resolved no code changes were required at this time.

5. **Missing Parentheses**

   Related Asset(s): Hook.sol

   The recommended parentheses were added.

6. **Lack Of NatSpec Comments**

   Related Asset(s): Actions.sol

   Comprehensive NatSpec comments were added to the file.

7. **Unnecessary Use of Parentheses**

   Related Asset(s): Rounding.sol

   The parentheses were removed.

8. **Invalid SPDX License**

   Related Asset(s): ShareCollateralTokenLib.sol

   The file was updated to SPDX license `GPL-2.0-or-later`.

9. **Repetitive Conversion Of Asset Type**

   Related Asset(s): SiloERC4626Lib.sol

   The variable `ISilo.AssetType collateralType` was introduced.

10. **Inaccurate Comparison On Function `isBelowMaxLtv()`**

    Related Asset(s): SiloSolvencyLib.sol

    The development team acknowledged the issue and resolved no code changes were required at this time.

11. **Caching Value During Initialisation Phase**

    Related Asset(s): ShareCollateralToken.sol

    The development team acknowledged the issue and resolved no code changes were required at this time.

12. **Potentially Inaccurate `hooksBefore`**

    Related Asset(s): GaugeHookReceiver.sol

    The recommended code changes were implemented.

13. **Data Type Mismatch**

    Related Asset(s): SiloHookReceiver.sol

    The development team acknowledged the issue and resolved no code changes were required at this time.

14. **Typos**

    Related Asset(s): InterestRateModelV2.sol, SiloMathLib.sol

    The typos were not updated at the time of retesting.

All relevant changes were observed to be implemented in commit 8def80e.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `forge` framework was used to perform these tests and the output is given below.

```
$ forge test --mt sigp --ffi

Ran 1 test for test/tests-local/Silo/InterestRateModelV2.sigp.t.sol:InterestRateModelV2TestSigp
[PASS] test_sigp_IRM_buildUpTcritSilently() (gas: 110497)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.68ms (330.51µs CPU time)

Ran 14 tests for test/tests-local/lib/Hook.sigp.t.sol:HookSigpTest
[PASS] test_sigp_afterBorrowDecode(uint256,uint256,address,address,address,uint256,uint256) (runs: 257, µ: 11040, ~: 11040)
[PASS] test_sigp_afterDepositDecode(uint256,uint256,address,uint256,uint256) (runs: 257, µ: 8398, ~: 8398)
[PASS] test_sigp_afterFlashLoanDecode(address,address,uint256,uint256) (runs: 257, µ: 7696, ~: 7696)
[PASS] test_sigp_afterRepayDecode(uint256,uint256,address,address,uint256,uint256) (runs: 257, µ: 9728, ~: 9728)
[PASS] test_sigp_afterTokenTransferDecode(address,address,uint256,uint256,uint256,uint256) (runs: 257, µ: 9662, ~: 9662)
[PASS] test_sigp_afterTransitionCollateralDecode(uint256,address,uint256) (runs: 257, µ: 6360, ~: 6360)
[PASS] test_sigp_afterWithdrawDecode(uint256,uint256,address,address,address,uint256,uint256) (runs: 257, µ: 11083, ~: 11083)
[PASS] test_sigp_beforeBorrowDecode(uint256,uint256,address,address,address) (runs: 257, µ: 8960, ~: 8960)
[PASS] test_sigp_beforeDepositDecode(uint256,uint256,address) (runs: 257, µ: 6317, ~: 6317)
[PASS] test_sigp_beforeFlashLoanDecode(address,address,uint256) (runs: 257, µ: 6653, ~: 6653)
[PASS] test_sigp_beforeRepayDecode(uint256,uint256,address,address) (runs: 257, µ: 7741, ~: 7741)
[PASS] test_sigp_beforeTransitionCollateralDecode(uint256,address) (runs: 257, µ: 5324, ~: 5324)
[PASS] test_sigp_beforeWithdrawDecode(uint256,uint256,address,address,address) (runs: 257, µ: 8961, ~: 8961)
[PASS] test_sigp_switchCollateralDecode(address) (runs: 257, µ: 4330, ~: 4330)
Suite result: ok. 14 passed; 0 failed; 0 skipped; finished in 18.19ms (180.04ms CPU time)

Ran 1 test for test/tests-local/lib/SiloMathLib.sigp.t.sol:SiloMathLibSigpTest
[PASS] test_sigp_convertToShares(uint256,uint256,uint256) (runs: 257, µ: 28299, ~: 27877)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 24.73ms (23.81ms CPU time)

Ran 1 test for test/tests-local/Silo/SiloFactoryTest.sigp.t.sol:SiloFactoryTestSigp
[PASS] test_sigp_anyoneCanBurnCreatedSiloToken() (gas: 53649)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 98.69ms (190.90µs CPU time)

Ran 14 tests for test/tests-local/Silo/Silo.sigp.t.sol:SiloTestSigp
[PASS] test_sigp_borrow_same_silo() (gas: 1031649)
[PASS] test_sigp_cannot_borrow_more_than_maxBorrow() (gas: 849766)
[PASS] test_sigp_collateral_transition_effect_on_apr() (gas: 1544729)
[PASS] test_sigp_deposit_and_withdraw() (gas: 339927)
[PASS] test_sigp_deposit_borrow() (gas: 1025277)
[PASS] test_sigp_deposit_single(uint256) (runs: 257, µ: 442658, ~: 441986)
[PASS] test_sigp_deposit_transition_collateral_and_withdraw_correct_collateral_type() (gas: 459640)
[PASS] test_sigp_deposit_transition_collateral_and_withdraw_wrong_collateral_type() (gas: 424345)
[PASS] test_sigp_deposit_twice() (gas: 295036)
[PASS] test_sigp_deposit_withdraw(uint256,uint256,uint256) (runs: 257, µ: 453000, ~: 440565)
[PASS] test_sigp_maxWithdraw_protected(uint256,uint256) (runs: 257, µ: 357496, ~: 357527)
[PASS] test_sigp_transitionCollateral() (gas: 380980)
[PASS] test_sigp_transition_collateral_after_borrow() (gas: 989668)
[PASS] test_sigp_withdraw_effect_on_apr() (gas: 1332275)
Suite result: ok. 14 passed; 0 failed; 0 skipped; finished in 248.34ms (424.82ms CPU time)

Ran 5 test suites in 249.84ms (391.64ms CPU time): 31 tests passed, 0 failed, 0 skipped (31 total tests)
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact \ Likelihood	Low	Medium	High
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Low	Low	Medium

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: `https://blog.sigmaprime.io/solidity-security.html`. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: `http://www.dasp.co/`. [Accessed 2018].

σ'