Security Assessment & Formal Verification Draft Report

Silo Vault

February 2025

Prepared for Silo

Table of content

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
silo-vaults	https://github.com/silo-finance/silo-contracts-v2/tree/develop/silo-vaults/contracts	8ec7238	EVM

Project Overview

This document describes the specification and verification of silo-vaults using the Certora Prover and manual code review findings. The work was undertaken from 27.1.25 to 10.2.25.

The following contract list is included in our scope:

```
contracts/SiloVault.sol
contracts/SiloVaultFactory.sol
contracts/PublicAllocator.sol
contracts/IdleVault.sol
contracts/libraries/ConstantsLib.sol
contracts/libraries/ErrorsLib.sol
contracts/libraries/EventsLib.sol
contracts/libraries/PendingLib.sol
contracts/incentives/VaultIncentiveModule.sol
contracts/incentives/claiming-logics/SiloIncentivesControllerCL.sol
contracts/incentives/claiming-logics/SiloIncentivesControllerCLFactory.sol
```

The Certora Prover demonstrated that the implementation of the **Solidity** contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all the Solidity contracts. During the verification process and the manual audit, the Certora team discovered bugs in the Solidity contracts code, as listed on the following page.

Protocol Overview

Silo Vault is an ERC4626 Vault which allows users to deposit an underlying ERC20 asset. The Vault would then invest those underlying asset tokens into other yield-generating and reward-earning ERC4626 vaults called Markets. The Vault allows for privileged roles to add and remove Markets, and for unprivileged users to move funds in between different markets, for a fee.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	1		
High	1		
Medium	3		
Low	8		
Informational	2		
Total	15		

Severity Matrix

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

Detailed Findings

ID	Title	Severity	Status
C-01	Funds could be permanently lost due to a share price inflation attack in ERC4626 markets.	Critical	Not yet fixed
H-01	Permissionless skim() function allows draining market tokens	High	Not yet fixed
M-01	Missing VaultIncentivesModule initialization by SiloVaultsFactory	Medium	Not yet fixed
M-02	The Incentive Module's owner can execute arbitrary code on behalf of the Vault	Medium	Not yet fixed
M-03	Public Allocator could be DoS	Medium	Not yet fixed
L-01	Factories using CREATE opcode create contracts vulnerable to reorgs	Low	Not yet fixed
L-02	Vault does not revoke its infinite approval from removed markets	Low	Not yet fixed
L-03	Vault's transfer and transferFrom are not protected for reentrancy	Low	Not yet fixed

L-04	Insufficient gas for PublicAllocator's native fee collection	Low	Not yet fixed
L-05	Fee Recipient could lose rewards for newly generated fees	Low	Not yet fixed
L-06	Vault could be vulnerable to an inflation attack	Low	Not yet fixed
L-07	Faulty or malicious markets could drain the Vault	Low	Not yet fixed
L-08	Removing an active Notification Receiver could drain the incentive program	Low	Not yet fixed

Critical Severity Issues

C-01 Funds could be permanently lost due to a share price inflation attack in ERC4626 markets

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: SiloVault.sol	Status: Not yet fixed	Violated Property:

Description: As the markets themselves are ERC4626, a share inflation attack (first depositor) in any of them may result in the vault being drained, as users can call the `reallocateTo()` or the `deposit()` functions to constantly deposit into the vulnerable ERC4626 and lose funds.

The way the standard implementation of ERC4626 deals with a first depositor attack is by making such an attack unprofitable for an attacker, which will discourage anyone from actually inflating the share price. An attacker would need to invest a certain amount of funds in order to inflate the price, and that amount must be greater than any loss caused due to rounding to any future depositor.

The implied assumption is that the victim must be front-runned and will not repeat this deposit more than once. This assumption does not actually hold true in our case because the attacker has some control over the Silo Vault. He can control how many times the vault deposits into the ERC4626 market, repeating this action as many times as he wants via the `reallocateTo()` function in the PublicAllocator.sol contract which would cost the attacker some fees, or via the `deposit()` function if the vulnerable market is the next market in the `supplyQueue` (This could be forced by taking a large flashloan and filling up the caps of the markets ahead of it in the queue), and controlling the amount that is being deposited (making it such that the rounding errors would be most impactful).

While it would be best for an attacker if he would be able to inflate the share price in any regular market (as he would be able to be a shareholder in that market and gain the funds that the Silo Vault will lose), there's no guarantee that it would be possible. However, the Idle Vault should always be vulnerable to a share price inflation attack. It inherits from the standard ERC4626

implementation and it restricts anyone who isn't the Silo Vault from being a shareholder. In that case, the attacker can forcibly make the Silo Vault withdraw the funds from there (if caps allow it) and inflate the share price through a donation. After the inflation, the attacker can force the Silo Vault to deposit funds into the Idle Vault that will be lost due to rounding, causing a permanent loss of funds, as they will be owned by the “virtual user” in the Idle Vault.

Recommendations: Firstly, we would recommend adding a sanity check that whenever the Silo Vault deposits funds into an ERC4626 market, the difference in Silo Vault-owned assets reported by the market is not too different from the amount that was actually deposited. Secondly, we would recommend setting the _decimalsOffset() in the Idle Vault to be very large (say, 18). This would make the amount that the user would need to "gift" the market in order to significantly inflate the share price very large and impractical.

Lastly, we would also recommend making a design change and cap the amounts that could be deposited (decrease back when funds are withdrawn) into each market (and not just the amount that it currently **holds** on the Silo Vault's behalf). This could limit any damage to the Silo Vault that could occur as a result of a faulty market.

Customer’s response:

Fix Review:

High Severity Issues

H-01 Permissionless skim() function allows draining market tokens

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: SiloVault.sol	Status: Not yet fixed	Violated Property: `SiloVault`'s market balance decreases only through `withdraw` or `reallocate` calls

Description: The `skim()` function in `SiloVault` can be used to transfer ERC-20 assets held by the SiloVault contract to a predefined `skimRecipient`. While it's true that `SiloVault` doesn't directly hold any assets because all deposits are immediately routed to the markets, the market shares minted in exchange for these deposits can be seen as ERC-20 assets, because ERC-4626 markets have an ERC-20 sub-interface.

JavaScript

```
File: SiloVault.sol
492:     /// @inheritdoc ISiloVaultBase
493:     function skim(address _token) external virtual {
494:         if (skimRecipient == address(0)) revert ErrorsLib.ZeroAddress();
495:
496:         uint256 amount = _ERC20BalanceOf(_token, address(this));
497:
498:         IERC20(_token).safeTransfer(skimRecipient, amount);
499:
500:         emit EventsLib.Skim(_msgSender(), _token, amount);
501:     }
```

Exploit Scenario: It is possible for anyone to use the `skim()` function to move market share balance away from the Vault with the effect of deflating the Vault's asset supply, which would deflate the Vault's share price and allow anyone to mint a very large amount of shares at a discount, practically nullifying the value of the pre-existing shares. If the `skimRecipient` sends

the markets' shares back to Vault, the attacker would then be able to withdraw his over-minted shares and drain the returned assets from the Vault. Even if the `skimRecipient` doesn't send the funds back, the attacker could permanently DoS the Vault by deflating the share price. When all the assets have been removed from the Vault, new deposits would mint more shares than the previous `totalSupply()` of shares. By repeating this process, the attacker could further deflate the share price, making it such that `type(uint256).max` shares would be worth `1` asset, at which point the Vault will be permanently bricked.

Recommendations: Make the `skim()` function permissioned and/or prevent its call with any `_token` that is present in the Vault's `supplyQueue` or `withdrawQueue`.

Customer's response:

Fix Review:

Medium Severity Issues

M-01 Missing VaultIncentivesModule initialization by SiloVaultsFactory

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: SiloVaultsFactory.sol	Status: Not yet fixed	Violated Property: initialOwner should be set for created VaultIncentivesModule and SiloVault instances

Description: When creating a `SiloVault`, `SiloVaultsFactory` also creates a `VaultIncentivesModule` by cloning a pre-existing instance used as implementation. The newly created `VaultIncentivesModule` is a proxy that was not initialized, in particular on its `owner` storage slot.

JavaScript

```
File: SiloVaultsFactory.sol
34:     /// @inheritdoc ISiloVaultsFactory
35:     function createSiloVault(
36:         address initialOwner,
37:         uint256 initialTimelock,
38:         address asset,
39:         string memory name,
40:         string memory symbol
41:     ) external virtual returns (ISiloVault siloVault) {
42:         VaultIncentivesModule vaultIncentivesModule = VaultIncentivesModule(
43:             Clones.clone(VAULT_INCENTIVES_MODULE_IMPLEMENTATION)
44:         );
45:
46:         siloVault = ISiloVault(address(
47:             new SiloVault(initialOwner, initialTimelock, vaultIncentivesModule, asset,
name, symbol))
48:         );
49:
50:         isSiloVault[address(siloVault)] = true;
51:
```

```
52:         emit EventsLib.CreateSiloVault(
53:             address(siloVault), msg.sender, initialOwner, initialTimelock, asset, name,
symbol
54:         );
55:     }
```

Exploit Scenario: `VaultIncentivesModule` instances created through the `SiloVaults` factory are unusable because they come with no `owner`.

Recommendations: Add an initializer function to `VaultIncentivesModule` that could be called after cloning.

Customer's response:

Fix Review:

M-02 The Incentive Module's owner can execute arbitrary code on behalf of the Vault		
Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: SiloVault.sol VaultIncentivesModule.sol	Status: Not yet fixed	Violated Property:

Description: When the function `_claimRewards()` is being called, the Vault delegatecalls the addresses in the `logics[]` array:

```
JavaScript
    function _claimRewards() internal virtual {
        address[] memory logics = INCENTIVES_MODULE.getAllIncentivesClaimingLogics();
        bytes memory data =
abi.encodeWithSelector(IIncentivesClaimingLogic.claimRewardsAndDistribute.selector);

        for (uint256 i; i < logics.length; i++) {
            (bool success,) = logics[i].delegatecall(data);
            if (!success) revert ErrorsLib.ClaimRewardsFailed();
        }
    }
```

However, those addresses come from the Incentive Module and are controlled by the Incentive Module's owner (which is presumably the same one as the Vault's owner) via the `addIncentivesClaimingLogic()` function in `VaultIncentivesModule.sol`.

According to the design of the protocol, the Owner should not have unlimited power, and he should be restricted both by the code itself and by the Vault's guardian, which is supposed to have the power to restrict the Owner from performing certain actions.

Exploit Scenario: A malicious Owner of the Incentive Module can deploy a contract that features the `claimRewardsAndDistribute()` function with arbitrary logic, add that contract as one of the addresses in the `logics[]` array and execute whatever he wants unopposed.

Recommendations: Change the Incentive Module to be more consistent with the design of the Vault and add a Guardian and a timelock. That way, there would be at least some restriction on the power of the owner.

Customer's response:

Fix Review:

M-03 Public Allocator could be DoS

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: SiloVault.sol PublicAllocator.sol	Status: Not yet fixed	Violated Property:

Description: Users can move funds between markets through two different mechanisms. One is by depositing and withdrawing from the Vault, and the other is through the public allocator (an action which costs fees). The existence of both of these mechanisms simultaneously enables all sorts of DoS and griefing attacks. For example, a user could pay the required fee and call the public allocator in order to move funds from one market to another. A different user could then immediately ruin this allocation by either calling the Public Allocator again (and also paying fees), or by depositing and withdrawing a large amount (for example by taking a flashloan), which would change the allocation according to the Supply and the Withdraw queues.

Similarly, a user could DoS the allocation to certain markets using the Public Allocator by filling the flowCaps. For example, a user could target a certain market and transfer the maximal possible amount to it. After the flowCap has been reached, the user can move those funds out (again, by depositing and withdrawing a large amount, which would move the funds back to the "natural" allocation), which would DoS any future attempt to move those funds back again into that market using the Public allocator.

Recommendations: The existence of two separate mechanisms to move funds in between markets could result in them interfering with each other. Assess whether or not this is an acceptable risk and consider making a design change.

In the context of the Public Allocator, we would also recommend allowing users to specify `withdrawal.amount = type(uint256).max` as a convention to move the maximal amount,

which might prevent some unintended DoS occurring as a result of several users interacting with the protocol simultaneously.

Customer's response:

Fix Review:

Low Severity Issues

L-01 Factories using CREATE opcode create contracts vulnerable to reorgs

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: SiloVaultFactory.sol SiloIncentivesControllerCLFactory.sol	Status: Not yet fixed	Violated Property: Created contract addresses should not depend on creation sequence

Description: Both factories in scope `SiloIncentivesControllerCLFactory` and `SiloVaultFactory`, create contracts using the CREATE opcode. This is an opcode that is especially insecure for factories that permissionlessly create contracts that hold assets, because frontrunning attacks and/or reorgs can divert funds to contracts other than the intended ones.

```
JavaScript
File: SiloVaultsFactory.sol
42:         VaultIncentivesModule vaultIncentivesModule = VaultIncentivesModule(
43:             Clones.clone(VAULT_INCENTIVES_MODULE_IMPLEMENTATION)
44:         );
45:
46:         siloVault = ISiloVault(address(
47:             new SiloVault(initialOwner, initialTimelock, vaultIncentivesModule, asset,
name, symbol))
48:         );

File: SiloIncentivesControllerCLFactory.sol
16:         logic = new SiloIncentivesControllerCL(_vaultIncentivesController,
_siloIncentivesController);
```

Exploit Scenario: Alice sends two transactions to the mempool, one to create a vault, and another one to fund it with its own assets at its expected address A. Bob observes these two transactions and frontruns Alice's creation transaction. Bob's SiloVault will be created at address

A, Alice's will be created at address B, but Alice's second transaction will fund Bob's vault instead of hers.

Recommendations: Consider using CREATE2 with a deterministic salt ideally dependent on the Vault's owner.

Customer's response:

Fix Review:

L-02 Vault does not revoke its infinite approval from removed markets

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: SiloVault.sol	Status: Not yet fixed	Violated Property: `notInWithdrawQThenNoApproval`

Description: When ERC4626 markets are registered on the Vault, an approval of an infinite amount of the asset token is granted to the added markets. This approval is however not revoked when markets are removed.

```
JavaScript
File: SiloVault.sol
338:             if (!seen[i]) {
339:                 IERC4626 market = withdrawQueue[i];
340:
341:                 if (config[market].cap != 0) revert
ErrorsLib.InvalidMarketRemovalNonZeroCap(market);
342:                 if (pendingCap[market].validAt != 0) revert
ErrorsLib.PendingCap(market);
343:
344:                 if (_ERC20BalanceOf(address(market), address(this)) != 0) {
345:                     if (config[market].removableAt == 0) revert
ErrorsLib.InvalidMarketRemovalNonZeroSupply(market);
346:
347:                     if (block.timestamp < config[market].removableAt) {
348:                         revert ErrorsLib.InvalidMarketRemovalTimelockNotElapsed(market);
349:                     }
350:                 }
351:
352:                 delete config[market];
---
File: SiloVault.sol
808:         // one time approval, so market can pull any amount of tokens from SiloVault in
a future
809:         IERC20(asset()).forceApprove(address(_market), type(uint256).max);
```

Exploit Scenario: We don't foresee any likely exploit scenario for this finding.

Recommendations: Revoke asset approvals to markets when they are removed.

Customer's response:

Fix Review:

L-03 Vault's transfer and transferFrom are not protected for reentrancy		
Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: SiloVault.sol	Status: Not yet fixed	Violated Property:

Description: The Vault's `_update()` function overridden implementation performs several external calls within the `_claimRewards()` and `_afterTokenTransfer()` internal calls.

```
JavaScript
File: SiloVault.sol
922:     function _update(address _from, address _to, uint256 _value) internal virtual
override {
---
931:         _claimRewards();
933:         super._update(_from, _to, _value);
935:         if (_value == 0) return;
937:         _afterTokenTransfer(_from, _to, _value);
938:     }
```

Among the external entry points that trigger an `_update()` internal call, we have `transfer()` and `transferFrom()` that aren't overridden from the contract's ERC4626/ERC20 parents, and therefore aren't protected from reentrancy like `mint()`, `deposit`, `redeem`, `withdraw` are.

Exploit Scenario: While an exploit scenario is somewhat unlikely due to the controlled nature of the called contracts, we believe that there is a potential for using reentrancy to change the order in which external calls to the downstream incentive claiming logic and `INotificationReceiver` contracts are made.

Recommendations: Protect the `transfer()` and `transferFrom()` functions with reentrancy guards.

Customer's response:

Fix Review:

L-04 Insufficient gas for PublicAllocator's native fee collection

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: PublicAllocator.sol	Status: Not yet fixed	Violated Property:

Description: The `PublicAllocator` contract allows withdrawing fees collected in native tokens via the `transferFee()` function. This function sends the tokens to a provided `feeRecipient` however using an unnecessarily strict `transfer()` call which limits the transfer gas to `2300`. This gas amount can be insufficient if `feeRecipient` is a contract.

```
JavaScript
File: PublicAllocator.sol
88:     /// @inheritdoc IPublicAllocatorBase
89:     function transferFee(ISiloVault vault, address payable feeRecipient) external virtual
onlyAdminOrVaultOwner(vault) {
90:         uint256 claimed = accruedFee[vault];
91:         accruedFee[vault] = 0;
92:         feeRecipient.transfer(claimed);
93:         emit EventsLib.TransferFee(msg.sender, vault, claimed, feeRecipient);
94:     }
```

Exploit Scenario: The Gnosis multisig is a popular example for which the provided gas would not suffice to complete the reception of native tokens.

Recommendations: Forward all available gas, for example via the `call` keyword:

```
JavaScript
       feeRecipient.call{value: claimed}("")
```

Customer's response:

Fix Review:

L-05 Fee Recipient could lose rewards for newly generated fees

Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: SiloVault.sol	Status: Not yet fixed	Violated Property:

Description: There is an inconsistency in the way the Vault treats the rewards that were generated since the last time the Vault was updated.

If a user mints, redeems or transfers shares, the `_accrueFee()` function is being called before `_claimRewards()`. As `_accrueFee()` mints some amount of shares for the Fee Recipient, this means that Fee Recipient will receive a larger share of the newly generated rewards than he would if fees were not accrued first.

However, rewards generated since the last update could also be claimed by calling the `claimRewards()` function, but this function does not accrue fees and therefore does not mint new shares to the Fee Recipient before the generated rewards are distributed.

JavaScript

```
        function claimRewards() public virtual {
        _nonReentrantOn();

        _claimRewards();

        _nonReentrantOff();
    }
```

The implication is that whenever `claimRewards()` would be called, the Fee Recipient would earn slightly less rewards.

Recommendations: The `claimRewards()` function could be modified to accrue fees before the rewards are being distributed.

```JavaScript
            function claimRewards() public virtual {
            _nonReentrantOn();

            _updateLastTotalAssets(_accrueFee());

            _claimRewards();

            _nonReentrantOff();
        }
```

Customer's response:

Fix Review:

L-06 Vault could be vulnerable to an inflation attack

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: SiloVault.sol	Status: Not yet fixed	Violated Property:

Description: Silo Vault inherits from the standard Open-Zeppelin implementation of ERC4626, which uses the `_decimalsOffset()` to determine the initial shares to assets ratio of the vault.

As stated before, the standard implementation deals with an inflation attack (first depositor attack) by disincentivizing users from inflating the share price, as this would cost the attacker more than what any one victim would lose in a single deposit due to rounding.

However, if `_decimalsOffset()` is set to `0`, this would be a tight bound, meaning that this attack could be profitable for an attacker even if there would be only two later deposits that will lose funds due to rounding.

In the case of the Silo Vault, the `_decimalsOffset()` would be `0` for any asset that has at least `18` decimals.

```
JavaScript
DECIMALS_OFFSET = uint8(UtilsLib.zeroFloorSub(18, IERC20Metadata(_asset).decimals()));
```

Recommendations: Increase `_decimalsOffset()` for all assets, which would exponentially increase the ratio between the amount that an attacker would need to invest to inflate the share price and the maximal amount that any victim would lose in a single deposit.

Customer's response:

Fix Review:

L-07 Faulty or malicious markets could drain the Vault

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: SiloVault.sol	Status: Not yet fixed	Violated Property:

Description: Markets report the amount of assets they currently hold on behalf of the Vault. As the Vault imposes the market's cap on the amount that the market currently holds (and not on the amount that was actually deposited), it means that any faulty market could lead to the vault being drained, as there would not be any limitations on moving more funds into the market.
A malicious market could also falsely report that it holds a large amount of assets, which would lead to an inflation of the vault's share price and to the possible draining of vault's funds invested in other markets.

Recommendations: Be aware of the dangers of adding a faulty or a malicious market. Consider capping the amounts that could be deposited into each market to deal with a faulty market, and perhaps even capping the markets' maximal reported revenue for a period of time to deal with a malicious market that attempts to inflate the vault's share price, if that's a concern.

Customer's response:

Fix Review:

L-08 Removing an active Notification Receiver could drain the incentive program

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: VaultIncentivesModule.sol SiloVault.sol	Status: Not yet fixed	Violated Property:

Description: Removing a Notification Receiver from the Incentive Module would allow users to transfer their `shareToken` without updating the state of the incentive program. As the accrued rewards are proportional to the amount of `shareToken` held by the users, it would be possible to transfer a large amount of `shareToken` from one address to the other without updating the state, claiming a large amount of rewards on behalf of a different address each time.

Recommendations: Be careful when you remove a Notification Receiver from the Incentive Module.

Customer's response:

Fix Review:

Informational Severity Issues

I-01. Rewards distribution consumes a lot of gas

Description: The `_claimRewards()` function is being called frequently (every time the `_update()` function is being called) and it may consume a lot of gas. It iterates over all the claiming logics of every market, claiming rewards from those markets and distributing them back to the users through the Vault's own incentive program.

Recommendation: Consider adding a `_lastUpdated` variable in the Vault to keep track of the last time rewards were distributed. If rewards have already been distributed in the current block, no new rewards should be distributed and therefore the rewards distribution code could be skipped.
Another improvement that could be considered is to optimize the amount of times that `immediateDistribution()` is being called. Currently, the same `rewardToken` could be distributed many times in the same transaction, once for each claiming logic. If instead the distribution process would only happen after all the rewards from all the markets have been claimed, it would be possible to only distribute the rewards once for each `rewardToken`.

Customer's response:

Fix Review:

I-02. Redundant setting of withdrawn variable to 0

Description: The `withdrawn` variable in `SiloVault.sol` line `383` is being set to `0`, but this appears to be redundant. The new value of `withdrawn` is only being used if the `supplyShares` variable is `0`, but then the `supplyAssets` variable should also be `0` and therefore the `withdrawn` variable should be set to `0` in line `374`.

Recommendation: Consider removing this line.

Customer's response:

Fix Review:

Formal Verification

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Formally Verified After Fix	The rule was violated due to an issue in the code and was successfully verified after fixing the issue
Violated	A counter-example exists that violates one of the assertions of the rule.

General Assumptions and Simplifications

1. We work with objects inherited from the original contracts that we call harnesses. In the inherited objects we add more view methods, flags, etc. In cases where it was not possible to collect the required information via the inherited object.
2. We replaced some functions with equivalent CVL implementations. Notably *mulDiv, safeTransfer and safeTransferFrom*. This speeds up the verification process and doesn't affect the results.
3. We assume that loops are not iterated through more than two times.

Formal Verification Properties

SiloVault

Module General Assumptions

We introduced two more mappings to the SiloVault contract:

- mapping(address => uint256) public withdrawRank
- mapping(address => uint256) public deletedAt

We also modify some of its methods to correctly maintain these. These changes don't affect the original functionality of the contract and help us to verify rules about the withdrawQueue.

Module Properties

P-01. Reachable states are consistent	
Status: Verified	

Rule Name	Status	Description	Link to rule report
noFeeToUnsetFeeRecipient	Verified	*If feeRecepient is not set then fee() returns zero.*	
supplyCapIsEnabled	Verified	*If the market has a cap greater than 0 then it is enabled.*	
pendingSupplyCapHasConsistentAsset	Verified	*If the market has a pending cap then its token is the same as the asset of the vault.*	
enabledHasConsistentAsset	Verified	*If the market is enabled then its token is the same as the asset of the vault.*	
supplyCapIsNotMarkedForRemoval	Verified	*If the market has a non-zero supply cap then it's not marked for removal. (I.e., removableAt == 0)*	

notEnabledIsNotMarkedForRemoval	Verified	*If the market is enabled then it's not marked for removal. (I.e., removableAt == 0}*	
pendingCapIsNotMarkedForRemoval	Verified	*If the market has a pending cap then it's not marked for removal. (I.e., removableAt == 0}*	
newSupplyQueueEnsuresPositiveCap	Verified	*Method SetSupplyQueue can only add markets with non-zero caps.*	

P-02. Contract variables stay within allowed ranges

Status: Verified

Rule Name	Status	Description	Link to rule report
pendingTimelockInRange	Verified	*pendingTimelock_ is within minTimelock and maxTimelock at all times.*	
timelockInRange	Verified	*timelock is within minTimelock and maxTimelock at all times.*	
feeInRange	Verified	*fee is less than maxFee at all times.*	
supplyQueueLengthInRange	Verified	*The length of SupplyQueue is less than maxQueueLength at all times.*	
withdrawQueueLengthInRange	Verified	*The length of WithdrawQueue is less than maxQueueLength at all times.*	

P-03. Pending values are consistent

Status: Verified

Rule Name	Status	Description	Link to rule report
noBadPendingTimelock	Verified	*pendingTimelock.validAt is zero if and only if the pendingTimelock value is zero.*	
smallerPendingTimelock	Verified	*The pending timelock value is always strictly smaller than the current timelock value.*	
noBadPendingCap	Verified	*pendingCap.validAt is zero if and only if the pendingCap value is zero.*	
greaterPendingCap	Verified	*The pending cap value is either 0 or strictly greater than the current cap value.*	
noBadPendingGuardian	Verified	*If pendingGuardian.validAt is zero then pendingGuardian value is the zero address.*	
differentPendingGuardian	Verified	*The pending guardian is either the zero address or it is different from the current guardian.*	

P-04. Roles hierarchy

Status: Verified

Rule Name	Status	Description	Link to rule report

ownerIsGuardian	Verified	*If the Guardian can perform an action then the Owner can also perform it.*	
ownerIsCurator	Verified	*If the Curator can perform an action then the Owner can also perform it.*	
curatorIsAllocator	Verified	*If the Allocator can perform an action then the Curator can also perform it.*	

P-05. Methods update balances correctly

Status: Verified

Rule Name	Status	Description	Link to rule report
depositTokenChange	Verified	*Depositing correctly updates balances of all involved parties.*	
withdrawTokenChange	Verified	*Withdrawing correctly updates balances of all involved parties.*	
reallocateTokenChange	Verified	*Calling reallocate doesn't affect balances of SiloVault, msg.sender or any market.*	

P-06. Timelocks work correctly

Status: Verified

Rule Name	Status	Description	Link to rule report

guardianUpdateTime	Verified	*No change of guardian can happen before the timelock.*	
capIncreaseTime	Verified	*No increase of cap can happen before the timelock.*	
timelockDecreaseTime	Verified	*No decrease of timelock can happen before the timelock.*	
removableTime	Verified	*Market cannot be removed before removableAt.*	

P-07. Consistency of Supply and Withdraw queues

Status: Verified

Rule Name	Status	Description	Link to rule report
enabledIsInWithdrawalQueue	Verified	*If the market is enabled then it's in the WithdrawQueue.*	
inWithdrawQueueIsEnabled	Verified	*If the market is in the WithdrawQueue then it is enabled.*	
nonZeroCapHasPositiveRank	Verified	*If the market has a non-zero cap then it's in the WithdrawQueue.*	
distinctIdentifiers	Verified	*There are no duplicate markets in the withdraw queue.*	
isInDepositQThenIsInWithdrawQ	Verified	*If a market is in the supplyQueue then is it also in the withdrawQueue.*	

P-08. Risk assessment

Status: Violated

Rule Name	Status	Description	Link to rule report
canPauseSupply	Verified	*The allocator is able to pause supply by setting an empty supplyQueue. After the pause, all deposits and mints revert.*	
canForceRemoveMarket	Verified	*The curator is able to remove a market from the WithdrawQueue and set the market as not enabled. We verified this for the case where the WithdrawQueue contains exactly two elements.*	
noDelegateCalls	Verified	*No delegateCall happens, i.e. the contract is truly immutable.*	
reentrancySafe	Verified	*There are no untrusted external calls, ensuring notably reentrancy safety.*	
notInWithdrawQThenNoApproval	Violated	*If a market is not in the withdraw queue, then SiloVault should not have approval of the asset token for it.*	

P-09. Methods revert on incorrect inputs and don't revert otherwise

Status: Verified

Rule Name	Status	Description	Link to rule report

setCuratorRevertCondition	Verified	*setCurator reverts if and only if the specified conditions occur.*	
setIsAllocatorRevertCondition	Verified	*setAllocator reverts if and only if the specified conditions occur.*	
setSkimRecipientRevertCondition	Verified	*setSkimRecipient reverts if and only if the specified conditions occur.*	
setFeeInputValidation	Verified	*setFee reverts if the specified conditions occur.*	
setFeeRecipientInputValidation	Verified	*setFeeRecipient reverts if the specified conditions occur.*	
submitGuardianRevertCondition	Verified	*submitGuardian reverts if and only if the specified conditions occur.*	
submitCapRevertCondition	Verified	*submitCap reverts if and only if the specified conditions occur.*	
submitMarketRemovalRevertCondition	Verified	*submitMarketRemoval reverts if and only if the specified conditions occur.*	
setSupplyQueueInputValidation	Verified	*setSupplyQueue reverts if the specified conditions occur.*	
updateWithdrawQueueInputValidation	Verified	*updateWithdrawQueue reverts if the specified conditions occur.*	
reallocateInputValidation	Verified	*reallocate reverts if the specified conditions occur.*	
revokePendingTimelockRevertCondition	Verified	*revokePendingTimelock reverts if and only if the specified conditions occur.*	
revokePendingGuardianRevertCondition	Verified	*revokePendingGuardian reverts if and only if the specified conditions occur.*	

revokePendingCapRevertCondition	Verified	*revokePendingCap reverts if and only if the specified conditions occur.*	
revokePendingCapRevertCondition	Verified	*revokePendingCap reverts if and only if the specified conditions occur.*	
revokePendingMarketRemovalRevertCondition	Verified	*revokePendingMarketRemoval reverts if and only if the specified conditions occur.*	
acceptTimelockRevertCondition	Verified	*acceptTimelock reverts if and only if the specified conditions occur.*	
acceptGuardianRevertCondition	Verified	*acceptGuardian reverts if and only if the specified conditions occur.*	
acceptCapInputValidation	Verified	*acceptCap reverts if the specified conditions occur.*	
skimRevertCondition	Verified	*skim reverts if and only if the specified conditions occur.*	

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.