code4rena

Silo Finance

Smart Contract Security Assessment

Audit dates: Mar 24 — Mar 31, 2025

Overview

About C4

Code4rena (C4) is a competitive audit platform where security researchers, referred to as Wardens, review, audit, and analyze codebases for security vulnerabilities in exchange for bounties provided by sponsoring projects.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Silo Finance smart contract system. The audit took place from March 24 to March 31, 2025.

Following the C4 audit, 3 wardens (d3e4, t0x1c, and Drynooo) reviewed the mitigations for all identified issues; the mitigation review report is appended below the audit report. Additional details can be found within the C4 Silo Mitigation Review repository.

Final report assembled by Code4rena.

Summary

The C4 analysis yielded an aggregated total of 6 unique vulnerabilities. Of these vulnerabilities, 6 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 13 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding, which may include relevant context from the judge and Silo Finance team.

Scope

The code under review can be found within the C4 Silo Finance repository, and is composed of 20 smart contracts written in the Solidity programming language and includes 1697 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

Medium Risk Findings (6)

[M-01] Supply function doesn't account for market maxDeposit when providing assets to it

Submitted by SpicyMeatball, also found by 056Security, d3e4, DanielArmstrong, Dulgiq, and Rampage

https://github.com/code-423n4/2025-03-silo-finance/blob/main/silo-vaults/contracts/SiloVault.sol#L879

Finding description and impact

Some vaults that the Silo vault deposits into have their own supply caps (do not confuse with `config[market].cap`), which may prevent `_supplyERC4626` from fully depositing user-provided assets. If these caps are not accounted for, the deposit function may revert instead of distributing assets across multiple vaults.

Proof of Concept

Consider a scenario where there are two vaults available for deposits:

- Vault 1 has a supply cap of 10,000 assets and currently holds 5,000, meaning `vault1.maxDeposit` is 10,000 - 5,000 = 5,000.
- Vault 2 is in the same condition.

In total, 10,000 assets of deposit space are available. However, if a user tries to deposit 10,000 assets through the Silo vault, `_supplyERC4626` is called:

```
    function _supplyERC4626(uint256 _assets) internal virtual {
        for (uint256 i; i < supplyQueue.length; ++i) {
            IERC4626 market = supplyQueue[i];

            uint256 supplyCap = config[market].cap;
            if (supplyCap == 0) continue;
```

```
            // Update internal balance for market to include interest if
any.
            // `supplyAssets` needs to be rounded up for `toSupply` to be
rounded down.
            uint256 supplyAssets =
_updateInternalBalanceForMarket(market);

            uint256 toSupply =
UtilsLib.min(UtilsLib.zeroFloorSub(supplyCap, supplyAssets), _assets);

            if (toSupply != 0) {
                uint256 newBalance = balanceTracker[market] + toSupply;
                // As `_supplyBalance` reads the balance directly from
the market,
                // we have additional check to ensure that the market did
not report wrong supply.
                if (newBalance <= supplyCap) {
                    // Using try/catch to skip markets that revert.
>>                  try market.deposit(toSupply, address(this)) {
                        _assets -= toSupply;
                        balanceTracker[market] = newBalance;
                    } catch {}
                }
            }

            if (_assets == 0) return;
        }
```

The function will first attempt to deposit 10,000 assets into Vault 1, but since `vault1.maxDeposit` < 10,000, the transaction will revert. The same issue occurs with Vault 2, causing the entire deposit operation to fail—even though sufficient space exists across both vaults.

Recommended mitigation steps

To avoid this issue, the function should check `market.maxDeposit` before attempting a deposit, similar to how it is handled in the `_maxDeposit` function:

```
    function _maxDeposit() internal view virtual returns (uint256
totalSuppliable) {
        for (uint256 i; i < supplyQueue.length; ++i) {
            IERC4626 market = supplyQueue[i];

            uint256 supplyCap = config[market].cap;
            if (supplyCap == 0) continue;
```

```
            (uint256 assets, ) = _supplyBalance(market);
>>          uint256 depositMax = market.maxDeposit(address(this));
>>          uint256 suppliable = Math.min(depositMax,
UtilsLib.zeroFloorSub(supplyCap, assets));
```

In the `_supplyERC4626` function:

```
    function _supplyERC4626(uint256 _assets) internal virtual {
        for (uint256 i; i < supplyQueue.length; ++i) {
            IERC4626 market = supplyQueue[i];

            uint256 supplyCap = config[market].cap;
            if (supplyCap == 0) continue;

            // Update internal balance for market to include interest if
any.
            // `supplyAssets` needs to be rounded up for `toSupply` to be
rounded down.
            uint256 supplyAssets =
_updateInternalBalanceForMarket(market);

            uint256 toSupply =
UtilsLib.min(UtilsLib.zeroFloorSub(supplyCap, supplyAssets), _assets);
+           toSupply = Math.min(market.maxDeposit(address(this),
toSupply));
```

With this fix, the Silo vault will distribute deposits correctly:

- 5,000 assets will be deposited into Vault 1.
- 5,000 assets will be deposited into Vault 2.

This prevents unnecessary reverts and ensures that all available deposit space is properly utilized.

IhorSF (Silo Finance) confirmed

Silo Finance mitigated:

This PR here accounts for `MaxDeposit` when completing a deposit

Status: Mitigation confirmed. Full details in reports from d3e4, t0x1c, and Drynooo.

[M-02] SiloVault will incorrectly accrue rewards during user transfer/transferFrom actions due to unsynced totalSupply()

Submitted by oakcobalt, also found by aldarion and seeques

https://github.com/code-423n4/2025-03-silo-finance/blob/0409be5b85d7aabfbbe10de1de1890d4b862d2d5/silo-vaults/contracts/SiloVault.sol#L985

Finding description and impact

SiloVault's totalSupply() accrueFees from interests (delta totalAssets). Such fee accrual is updated through _accrueFee() which mints an additional share to the fee reciever.

We see that when claiming rewards directly through `claimRewards()` the accrued extra share is updated first before _claimRewards().

The vulnerability is when `_claimRewards()` is invoked atomically through hooks (_update), the `_accrueFee()` will be missed in the user's transfer/transferFrom call. In this case, an incorrect totalSupply() will be used for fee accrual, leading to incorrect fee accrual.

Impacts: incorrect and inconsistent reward accrual due to unsynced `totalSupply()`.

Recommended mitigation steps

In _update() hook, consider adding `_accrueFee()` or `_updateLastTotalAssets(_accrueFee())` before `_claimRewards()`.

Proof of Concept

We see in direct `claimRewards` flow. `totalSupply()` will be updated in `_accrueFee()`.

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

|>      _updateLastTotalAssets(_accrueFee());
        _claimRewards();

        _nonReentrantOff();
    }
```

```
    function _accrueFee() internal virtual returns (uint256
newTotalAssets) {
        uint256 feeShares;
        (feeShares, newTotalAssets) = _accruedFeeShares();
        //@audit this will increase totalSupply()
|>      if (feeShares != 0) _mint(feeRecipient, feeShares);
```

```
        emit EventsLib.AccrueInterest(newTotalAssets, feeShares);
    }
```

However, the vulnerable flow is transfer/transferFrom -> _update(), where `_accrueFee()` is missed.

```
    function _update(address _from, address _to, uint256 _value) internal
virtual override {
        // on deposit, claim must be first action, new user should not
get reward

        // on withdraw, claim must be first action, user that is leaving
should get rewards
        // immediate deposit-withdraw operation will not abused it,
because before deposit all rewards will be
        // claimed, so on withdraw on the same block no additional
rewards will be generated.

        // transfer shares is basically withdraw->deposit, so claiming
rewards should be done before any state changes

|>      _claimRewards(); //@audit rewards is claimed without first
updating totalSupply(). transfer/transferFrom flow is vulnerable.

        super._update(_from, _to, _value);

        if (_value == 0) return;

        _afterTokenTransfer(_from, _to, _value);
    }
```

We know transfer/transferFrom flow is vulnerable because unlike deposit/withdraw which calls `_accrueFee()` first, transfer/transferFrom will directly call `_update()` without updating totalSupply. This causes an incorrect/inconsistent reward accrual.

IhorSF (Silo Finance) disputed

Silo Finance mitigated:

This PR here fixes the accrue on transfer for SiloVault.

Status: Mitigation confirmed. Full details in reports from d3e4, t0x1c, and Drynooo.

[M-03] SiloVault.sol :: Markets with assets that revert on zero approvals cannot be removed.

Submitted by Fitro, also found by grearlake, nuthan2x, and Samueltroydomi

https://github.com/code-423n4/2025-03-silo-finance/blob/0409be5b85d7aabfbbe10de1de1890d4b862d2d5/silo-vaults/contracts/SiloVault.sol#L252-L267

https://github.com/code-423n4/2025-03-silo-finance/blob/0409be5b85d7aabfbbe10de1de1890d4b862d2d5/silo-vaults/contracts/SiloVault.sol#L272

https://github.com/code-423n4/2025-03-silo-finance/blob/0409be5b85d7aabfbbe10de1de1890d4b862d2d5/silo-vaults/contracts/libraries/SiloVaultActionsLib.sol#L29-L65

https://github.com/code-423n4/2025-03-silo-finance/blob/0409be5b85d7aabfbbe10de1de1890d4b862d2d5/silo-vaults/contracts/SiloVault.sol#L329

Finding description and impact

To remove a market from the vault, the supply `cap` must be set to 0. However, when this happens, the market's allowance to use tokens from the vault is also reset to 0. The issue arises because some tokens revert when attempting to approve a 0 value, preventing these markets from being removed from the vault.

Proof of Concept

`submitMarketRemoval()` is implemented as follows.

```
function submitMarketRemoval(IERC4626 _market) external virtual
onlyCuratorRole {
        if (config[_market].removableAt != 0) revert
ErrorsLib.AlreadyPending();
@>      if (config[_market].cap != 0) revert ErrorsLib.NonZeroCap();
        if (!config[_market].enabled) revert
ErrorsLib.MarketNotEnabled(_market);
        if (pendingCap[_market].validAt != 0) revert
ErrorsLib.PendingCap(_market);

        // Safe "unchecked" cast because timelock <= MAX_TIMELOCK.
        config[_market].removableAt = uint64(block.timestamp + timelock);

        emit EventsLib.SubmitMarketRemoval(_msgSender(), _market);
```

```
    }
```

As you can see, removing a market requires setting the `cap` to 0 (the same applies to `updateWithdrawQueue()`.). This is done by calling `submitCap()` with `_newSupplyCap` set to 0.

```
function submitCap(IERC4626 _market, uint256 _newSupplyCap) external
virtual onlyCuratorRole {
        if (_market.asset() != asset()) revert
ErrorsLib.InconsistentAsset(_market);
        if (pendingCap[_market].validAt != 0) revert
ErrorsLib.AlreadyPending();
        if (config[_market].removableAt != 0) revert
ErrorsLib.PendingRemoval();
        uint256 supplyCap = config[_market].cap;
        if (_newSupplyCap == supplyCap) revert ErrorsLib.AlreadySet();

        if (_newSupplyCap < supplyCap) {
            _setCap(_market, SafeCast.toUint184(_newSupplyCap));
        } else {
            pendingCap[_market].update(SafeCast.toUint184(_newSupplyCap),
timelock);

            emit EventsLib.SubmitCap(_msgSender(), _market,
_newSupplyCap);
        }
    }
```

In this case, the execution will enter the `if` section, which calls `_setCap()`, which invokes `setCap()` from the `SiloVaultActionsLib`.

```
function setCap(
        IERC4626 _market,
        uint184 _supplyCap,
        address _asset,
        mapping(IERC4626 => MarketConfig) storage _config,
        mapping(IERC4626 => PendingUint192) storage _pendingCap,
        IERC4626[] storage _withdrawQueue
    ) external returns (bool updateTotalAssets) {
        MarketConfig storage marketConfig = _config[_market];
        uint256 approveValue;

        if (_supplyCap > 0) {
            if (!marketConfig.enabled) {
```

```
                _withdrawQueue.push(_market);

                if (_withdrawQueue.length >
ConstantsLib.MAX_QUEUE_LENGTH) revert ErrorsLib.MaxQueueLengthExceeded();

                marketConfig.enabled = true;

                // Take into account assets of the new market without
applying a fee.
                updateTotalAssets = true;

                emit EventsLib.SetWithdrawQueue(msg.sender,
_withdrawQueue);
            }

            marketConfig.removableAt = 0;
            // one time approval, so market can pull any amount of tokens
from SiloVault in a future
            approveValue = type(uint256).max;
        }

        marketConfig.cap = _supplyCap;

@>      IERC20(_asset).forceApprove(address(_market), approveValue);

        emit EventsLib.SetCap(msg.sender, _market, _supplyCap);

        delete _pendingCap[_market];
    }
```

As you can see, we do not enter the `if` block because `_supplyCap = 0`, which results in `approveValue = 0` (default value). This is expected since we want to clear the market's allowance. However, some assets (such as BNB) revert when the approval value is set to `0`, causing `forceApprove()` to fail.

As a result, the cap cannot be set to `0`, preventing the market from being removed. Furthermore, if the vault relies on markets with such assets and they cannot be removed, new ones cannot be added due to the `MAX_QUEUE_LENGTH` restriction.

In this case, forceApprove() will not resolve the issue because it is only useful for tokens that revert when the previous allowance is not set to zero. It does not address tokens that revert due to 0 approval amounts.

```
 function forceApprove(IERC20 token, address spender, uint256 value)
internal {
```

```
        bytes memory approvalCall = abi.encodeCall(token.approve,
(spender, value));

        if (!_callOptionalReturnBool(token, approvalCall)) {
            _callOptionalReturn(token, abi.encodeCall(token.approve,
(spender, 0)));
            _callOptionalReturn(token, approvalCall);
        }
    }
```

As you can see, if the initial approval call fails, the function first resets the allowance to zero and then attempts to approve the provided value again. However, since the value is 0, it will fail another time reverting the transaction.

According to the contest specifications, tokens that **Revert on zero value approvals** are explicitly within scope.

Recommended mitigation steps

To resolve this issue, `approveValue` can be set to `1` instead of leaving it at the default `uint256` value.

```
function setCap(
        IERC4626 _market,
        uint184 _supplyCap,
        address _asset,
        mapping(IERC4626 => MarketConfig) storage _config,
        mapping(IERC4626 => PendingUint192) storage _pendingCap,
        IERC4626[] storage _withdrawQueue
    ) external returns (bool updateTotalAssets) {
        MarketConfig storage marketConfig = _config[_market];
-       uint256 approveValue;
+       uint256 approveValue = 1;

        if (_supplyCap > 0) {
            if (!marketConfig.enabled) {
                _withdrawQueue.push(_market);

                if (_withdrawQueue.length >
ConstantsLib.MAX_QUEUE_LENGTH) revert ErrorsLib.MaxQueueLengthExceeded();

                marketConfig.enabled = true;

                // Take into account assets of the new market without
applying a fee.
```

```
                updateTotalAssets = true;

                emit EventsLib.SetWithdrawQueue(msg.sender,
_withdrawQueue);
            }

            marketConfig.removableAt = 0;
            // one time approval, so market can pull any amount of tokens
from SiloVault in a future
            approveValue = type(uint256).max;
        }

        marketConfig.cap = _supplyCap;

        IERC20(_asset).forceApprove(address(_market), approveValue);

        emit EventsLib.SetCap(msg.sender, _market, _supplyCap);

        delete _pendingCap[_market];
    }
```

IhorSF (Silo Finance) confirmed

Silo Finance mitigated:

This PR here resets approval to 1 wei.

Status: Mitigation confirmed. Full details in reports from d3e4, t0x1c, and Drynooo.

[M-04] Lack of slippage and deadline protection in deposit(), withdraw() and redeem()

Submitted by t0x1c, also found by anchabadze, Aristos, falconhoof, harsh123, hezze, NexusAudits, and RaOne

https://github.com/code-423n4/2025-03-silo-finance/blob/main/silo-vaults/contracts/SiloVault.sol#L569

https://github.com/code-423n4/2025-03-silo-finance/blob/main/silo-vaults/contracts/SiloVault.sol#L942

Description

1. When users deposit funds, those assets are allocated to one or more underlying ERC4626 markets according to the supply queue.

2. When withdrawing or redeeming, assets are pulled from the underlying markets according to the withdraw queue and shares burned.

None of these actions allow the user to specify any acceptable slippage or deadline.

- The vault interacts with multiple ERC4626 vaults. Share price in these underlying vaults can change between transaction submission and execution.
- During deposit, the protocol might need to distribute assets across multiple markets based on their caps. This multi-step process could expose users to price changes.
- During withdrawals or redemptions, the protocol attempts to pull assets from markets in a specific order. If a market has insufficient liquidity, the next market is tried, which might have different share pricing.
- The _accrueFee() function is called during both deposit and withdrawal, which can change the conversion rate between shares and assets.

Additionally, there's a risk of transactions getting stuck in the mempool during periods of network congestion and hence deadline protection is needed.

Impact

User may recieve less than expected shares or assets due to unfavourable price movement or execution delays.

Recommendation

Allow the user to specify paramaters like `minShares`, `minAssets` while calling these functions.

IhorSF (Silo Finance) disputed

[M-05] Incorrect reward distribution due to feeShares minting order

Submitted by t0x1c, also found by Drynooo

https://github.com/code-423n4/2025-03-silo-finance/blob/main/silo-vaults/contracts/SiloVault.sol#L976-L992

Summary

The current implementation distributes rewards *before* minting fee shares, resulting in the fee recipient receiving shares but no rewards for the corresponding interest accrual period. This creates an inconsistency where the exisiting share owners receive higher than deserved portion of rewards.

Description

Let's assume Bob is the sole shareholder. What's happening right now is:

- Bob deposits at `t` and receives `100%` shares (for simplicity let's ignore, for now, the `DECIMALS_OFFSET` strategy deployed by the protocol to thwart the first-depositor attack).
- At some time, `t2`, we see that yield & reward has accrued.
- Someone calls claimRewards() which first internally calls `_updateLastTotalAssets(_accrueFee())` and then `_claimRewards()`:

```
File: silo-vaults/contracts/SiloVault.sol

 495:              function claimRewards() public virtual {
 496:                  _nonReentrantOn();
 497:
 498:@--->             _updateLastTotalAssets(_accrueFee());
 499:@--->             _claimRewards();
 500:
 501:                  _nonReentrantOff();
 502:              }
```

- _accrueFee() internally calls _mint() if `feeShares != 0`.

```
 942:              function _accrueFee() internal virtual returns
(uint256 newTotalAssets) {
 943:                  uint256 feeShares;
 944:                  (feeShares, newTotalAssets) =
_accruedFeeShares();
 945:
 946:@--->             if (feeShares != 0) _mint(feeRecipient,
feeShares);
 947:
 948:                  emit EventsLib.AccrueInterest(newTotalAssets,
feeShares);
 949:              }
```

- Note that `_accruedFeeShares()` on L944 is a ***retrospective way*** to calculate the `feeShares` which should correspond to the `feeAssets` amount applied on the accumulated interest. This is done because the total assets have already grown between `t` and `t2`. This is evident from the `newTotalAssets - feeAssets` term inside _accruedFeeShares() and also the comments on L960-961:

```
 951:              /// @dev Computes and returns the fee shares
(`feeShares`) to mint and the new vault's total assets
```

```
   952:              /// (`newTotalAssets`).
   953:              function _accruedFeeShares() internal view virtual
returns (uint256 feeShares, uint256 newTotalAssets) {
   954:                  newTotalAssets = totalAssets();
   955:
   956:                  uint256 totalInterest =
UtilsLib.zeroFloorSub(newTotalAssets, lastTotalAssets);
   957:                  if (totalInterest != 0 && fee != 0) {
   958:                      // It is acknowledged that `feeAssets` may
be rounded down to 0 if `totalInterest * fee < WAD`.
   959:                      uint256 feeAssets =
totalInterest.mulDiv(fee, WAD);
   960:@--->              // The fee assets is subtracted from the
total assets in this calculation to compensate for the fact
   961:@--->              // that total assets is already increased by
the total interest (including the fee assets).
   962:                      feeShares = _convertToSharesWithTotals(
   963:                          feeAssets,
   964:                          totalSupply(),
   965:@--->                  newTotalAssets - feeAssets,
   966:                          Math.Rounding.Floor
   967:                      );
   968:                  }
   969:              }
```

What this means is that the fee recipient is going to be minted the `feeShares` currently because they have a rightful claim to the `feeAssets` which started to accrue right from timestamp `t`.

With that in mind, let's see the remaining steps -

- On L946 `_accrueFee() --> _mint()` internally calls the overridden `_update()` function, which in turn calls `_claimRewards()` before `super._update()` actually mints these `feeShares` and increases `totalSupply()`. As the inline code comments explain, this is meant to be a safeguard and it is required so that rewards can be claimed before a new deposit/withdraw/transfer:

```
   976:              function _update(address _from, address _to, uint256
_value) internal virtual override {
   977:                  // on deposit, claim must be first action, new
user should not get reward
   978:
   979:                  // on withdraw, claim must be first action, user
that is leaving should get rewards
```

```
   980:                    // immediate deposit-withdraw operation will not
abused it, because before deposit all rewards will be
   981:                    // claimed, so on withdraw on the same block no
additional rewards will be generated.
   982:
   983:                    // transfer shares is basically withdraw-
>deposit, so claiming rewards should be done before any state changes
   984:
   985:@--->                _claimRewards();
   986:
   987:                    super._update(_from, _to, _value);
   988:
   989:                    if (_value == 0) return;
   990:
   991:                    _afterTokenTransfer(_from, _to, _value);
   992:                }
```

In this case however, what it means is that the **entire** reward is doled out to Bob since he possesses `100%` of shares because the `feeShares` are yet to be minted. By the time the control reaches the second call to `_claimRewards()` on L499 after minting of these shares, there are no more rewards left to be distributed to the fee recipient.

Impact

- The shares of the fee recipient will now only receive any future rewards and miss out on the current one even though the shares have been rightly minted to them retrospectively. Conversely put, Bob receives more than his fair share of rewards.
- Note that this is not just a one-time loss of rewards for the fee recipient. Each time `claimRewards()` is called and there is a pending yield to be collected, `feeShares` minted in that cycle lose out on the rewards being distributed. They only get to see a portion of the rewards from the next cycle onwards.

Recommendation

The current logic of calling `_claimRewards()` from inside `_update()` is correct and works well for all the other cases, so no issues there. For cases where `feeShares` are being minted, however, we may need to introduce additional logic inside `_claimRewards()` which checks for this via a new flag and calculates the reward portion after accounting for these retrospectively minted `feeShares`.

edd (Silo Finance) acknowledged

[M-06] Deflation attack

Submitted by d3e4

https://github.com/code-423n4/2025-03-silo-finance/blob/0409be5b85d7aabfbbe10de1de1890d4b862d2d5/silo-vaults/contracts/SiloVault.sol#L1

Finding description and impact

The flow of assets between SiloVault and its markets favour the markets (per standard ERC4626 specifications). This means that SiloVault can lose assets compared to its total supply of shares. This deflates the share price. At low total asset levels, this can be exploited to deflate the share price until the shares per asset is close to overflowing.

This can be exploited to either simply brick the vault, or such that the attacker is the guaranteed sole holder and recipient of the incentive rewards.

Proof of Concept

The shares minted is calculated as `_assets.mulDiv(_newTotalSupply + 10 ** _decimalsOffset(), _newTotalAssets + 1, _rounding)`, where the total assets is based on the redeemable value of all market shares held by SiloVault.

For example, the first depositor deposits 1 wei into SiloVault, which deposits this into a market, and mints `10 ** _decimalsOffset()` shares. The market, however, is likely to round this away and not return any share to SiloVault. On the next deposit into SiloVault, `totalAssets` is zero, but `10 ** _decimalsOffset()` shares were minted. Again depositing 1 wei mints `2 * 10 ** _decimalsOffset()`, without increasing `totalAssets`. Each repetition doubles the total supply.

The same would, of course, happen if `totalAssets` on deposit did increase to `1` but due to the market updating its price, this later became `0`.

Suppose SiloVault were to instead mint shares according to the increase in `totalAssets`. Then no shares would be minted in the above example. Suppose then instead 10 wei are deposited into SiloVault, which deposits this into a market, in return, for 8 market shares (let's say the market's price is 1.13 assets/share). This is redeemable for 9 asset tokens, so SiloVault mints `9 * 10 ** _decimalsOffset()` shares. Now, redeeming `8 * 10 ** _decimalsOffset()` shares from SiloVault makes SiloVault withdraw 8 asset tokens from the market. Withdrawing 8 asset tokens burns not 7.08 but all 8 market shares, and SiloVault is now, just as above, left with zero totalAssets but a total supply of `1 * 10 ** _decimalsOffset()`.

Recommended mitigation steps

Set the virtual assets to `10**DECIMALS_OFFSET` (the same as the virtual shares, instead of the hardcoded `1`). This acts as a virtual deposit which makes both the inflation attack and the attack described here infeasible. The only downside is that interest earned in the markets is effectively accrued also to the virtual deposit, for which reason, the virtual deposit should be chosen such that it is small in monetary value but still a large integer. Then the lost interest will be negligible.

IhorSF (Silo Finance) confirmed

edd (Silo Finance) commented:

The issue is confirmed however I'd argue it should not be a high severity. There is very low likelihood of this issue happening. Taking over empty vault has no point. If there is one legit deposit, even small, issue is gone. This is similar to inflation attack or first deposit attack that are not considered critical.

Silo Finance mitigated:

The PRs here and here only ensure that deposit does not generate zero shares.

Status: Unmitigated. Full details in reports from d3e4.

Code4rena judging staff adjusted the severity of Finding [M-06], after reviewing additional context provided by the sponsor.

Low Risk and Non-Critical Issues

For this audit, 13 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **Drynooo** received the top score from the judge.

The following wardens also submitted reports: 0xterrah, c0pp3rscr3w3r, codexNature, dystopia, holtzzx, MatricksDeCoder, newspacexyz, PolarizedLight, rayss, Sparrow, TheCarrot, and Yaneca_b.

_incentivesProgramIds too long will cause the loop to be too large, and the transaction will be reverted because it exceeds the gas limit.

https://github.com/code-423n4/2025-03-silo-finance/blob/0409be5b85d7aabfbbe10de1de1890d4b862d2d5/silo-core/contracts/incentives/SiloIncentivesController.sol#L79

Finding description and impact

In the SiloIncentivesController contract, the owner can only add _incentivesProgramIds by calling the createIncentivesProgram function, with no way to reduce _incentivesProgramIds. However, the length of _incentivesProgramIds is used in multiple places for loops. For

example, in the afterTokenTransfer function, this could cause any transfer call from siloVault to fail. As a result, the funds in siloVault would be locked and unable to be withdrawn.

Proof of Concept

1. The owner can add _incentivesProgramIds through the createIncentivesProgram function but cannot reduce it.

```
require(_incentivesProgramIds.add(programId),
IncentivesProgramAlreadyExists());
```

2. In the afterTokenTransfer function, the loop is based on the length of _incentivesProgramIds. When _incentivesProgramIds reaches a certain length, the loop may cause the transaction to exceed the gas limit and revert.

```
        uint256 numberOfPrograms = _incentivesProgramIds.length();

...
        for (uint256 i = 0; i < numberOfPrograms; i++) {
```

3. Since the siloVault also calls afterTokenTransfer during withdrawal, this could result in users' funds being stuck in the contract and unable to be withdrawn.

Recommended mitigation steps

It is recommended to add a function that allows the administrator to remove unnecessary elements from _incentivesProgramIds.

The SiloIncentivesControllerFactory contract may cause the protocol to lose funds due to chain reorganization.

https://github.com/code-423n4/2025-03-silo-finance/blob/0409be5b85d7aabfbbe10de1de1890d4b862d2d5/silo-core/contracts/incentives/SiloIncentivesControllerFactory. sol#L14

Finding description and impact

When the protocol uses the SiloIncentivesControllerFactory contract to create a SiloIncentivesController contract and injects incentive funds into it, the system becomes highly vulnerable to chain reorganization (reorg) attacks, potentially leading to fund theft.

Proof of Concept

Chain reorganizations (reorgs) can occur across all EVM-compatible chains, including major L2 solutions such as Arbitrum and Polygon. For reference, here's a link documenting Polygon's forked blocks - representing blocks that were excluded due to "Block Reorganizations": https://polygonscan.com/blocks_forked?p=1.

The attack may occur as follows:

1. The project first deploys SiloIncentivesControllerA through SiloIncentivesControllerFactory and transfers reward funds into it.
2. The attacker detects a chain reorganization.
3. The attacker front-runs by calling create() to deploy SiloIncentivesController, which deploys to the same address as SiloIncentivesControllerA, but with the attacker as owner. Funds are then transferred to this address.
4. The attacker withdraws the funds via rescueRewards to profit.

This causes direct financial loss to the protocol. However, due to its low probability, it is assessed as medium severity.

Recommended mitigation steps

It is recommended to:

Add an owner role to the SiloIncentivesControllerFactory contract, where only the owner can perform create operations. Alternatively, use CREATE2 and include msg.sender in the salt parameter.

The SiloVaultsFactory contract may cause the protocol to lose funds due to chain reorganization.

https://github.com/code-423n4/2025-03-silo-finance/blob/0409be5b85d7aabfbbe10de1de1890d4b862d2d5/silo-vaults/contracts/SiloVaultsFactory.sol#L53

Finding description and impact

When the protocol uses the SiloVaultsFactory contract to create a SiloVault contract and injects incentive funds into it, the system becomes highly vulnerable to chain reorganization (reorg) attacks, potentially leading to fund theft.

Proof of Concept

Chain reorganizations (reorgs) can occur across all EVM-compatible chains, including major L2 solutions such as Arbitrum and Polygon. For reference, here's a link documenting Polygon's forked blocks - representing blocks that were excluded due to "Block Reorganizations": https://polygonscan.com/blocks_forked?p=1.

The attack may occur as follows:

1. The project team first deploys SiloVaultA through SiloVaultsFactory and transfers initial staking funds into it.
2. The attacker detects a chain reorganization.
3. The attacker front-runs the transaction by calling create to deploy SiloVault, which results in deploying to the same address as SiloVaultA, but with the attacker as the owner. The funds are also transferred to this address. Additionally, the owner of the vaultIncentivesModule contract created here is also the attacker.
4. Subsequently, the attacker can modify configurations so that when the `_claimRewards` function is called, a delegatecall executes logic specified by the attacker, thereby draining all funds from the vault.

This poses a direct financial loss to the protocol, but due to its low probability, it is assessed as a medium-severity issue.

Recommended mitigation steps

It is recommended to: Add an owner role to the SiloIncentivesControllerFactory contract, where only the owner can perform create operations. Alternatively, use CREATE2 and include msg.sender in the salt parameter.

SiloVault does not comply with ERC4626.

https://github.com/code-423n4/2025-03-silo-finance/blob/0409be5b85d7aabfbbe10de1de1890d4b862d2d5/silo-vaults/contracts/SiloVault.sol#L647

https://github.com/code-423n4/2025-03-silo-finance/blob/0409be5b85d7aabfbbe10de1de1890d4b862d2d5/silo-vaults/contracts/SiloVault.sol#L518

Finding description and impact

The ERC4626 standard explicitly mandates that the totalAssets and maxDeposit functions must not revert, yet they remain susceptible to reverting.

Proof of Concept

The ERC4626 standard specifies the following requirements:

totalAssets : MUST NOT revert. maxDeposit : MUST NOT revert. previewRedeem : MUST NOT revert due to vault specific user/global limits. MAY revert due to other conditions that would also cause redeem to revert.

However, both the totalAssets and maxDeposit functions call the previewRedeem function of another ERC4626 contract. This means that both totalAssets and maxDeposit could potentially revert, which violates the standards requirement that they must not revert.

Since the protocol explicitly requires compliance with ERC4626 in its documentation, this should be classified as a Medium severity issue.

Recommended mitigation steps

It is recommended to use a try-catch block when calling the previewRedeem function to prevent potential reverts.

The vault's decimal precision is hardcoded to return 18, which doesn't match its actual precision. This discrepancy may lead to incorrect value assessments when inherited by external contracts.

https://github.com/code-423n4/2025-03-silo-finance/blob/0409be5b85d7aabfbbe10de1de1890d4b862d2d5/silo-vaults/contracts/SiloVault.sol#L122

https://github.com/code-423n4/2025-03-silo-finance/blob/0409be5b85d7aabfbbe10de1de1890d4b862d2d5/silo-vaults/contracts/SiloVault.sol#L513

Finding description and impact

The vault's decimal precision is hardcoded to return 18, which doesn't match its actual precision. This discrepancy may lead to incorrect value assessments when inherited by external contracts.

Proof of Concept

When inheriting from OpenZeppelin's ERC4626, the share decimals should be (asset decimals + DECIMALS_OFFSET), which should be 24 in this contract.

```
        DECIMALS_OFFSET = uint8(UtilsLib.zeroFloorSub(18 + 6, decimals));
```

However, the contract always returns 18 as its decimals, which will cause: Frontend display confusion Off-chain value calculation inaccuracies

For example: When depositing 1 USDC (6 decimals), the protocol calculates shares as:

```
1e6 × 1e18 / 1 = 1e24 shares
```

```
_assets.mulDiv(_newTotalSupply + 10 ** _decimalsOffset(), _newTotalAssets
+ 1, _rounding);
```

However, since the contract incorrectly returns 18 decimals:

Misinterpretation: Users/contracts will assume 1e6 shares exist
Reality: Only 1 share (1e24 raw units) was actually minted

Recommended mitigation steps

Do not override the decimals function.

Mitigation Review

Introduction

Following the C4 audit, 3 wardens (d3e4, t0x1c, and Drynooo) reviewed the mitigations for all identified issues. Additional details can be found within the Silo Finance Mitigation Review repository.

Mitigation Review Scope & Summary

The wardens confirmed the mitigations for all in-scope findings except for M-06, where the finding was not mitigated. The table below provides details regarding the status of each in-scope vulnerability from the original audit and the in-scope vulnerability that was not fully mitigated.

ORIGINAL ISSUE	STATUS	MITIGATION URL
M-01	🟢 Mitigated	PR 1166
M-02	🟢 Mitigated	PR 1168
M-03	🟢 Mitigated	PR 1165
M-06	🔴 Unmitigated	PR 1162 (solution) and PR 1173 (optimization)

M-06 Unmitigated

Submitted by d3e4.

Original issue: https://code4rena.com/audits/2025-03-silo-finance/submissions/F-11

F-11 summary

The issue was that the SiloVault share supply could be inflated by deposits which suffer (rounding) losses when deposited into the markets.

Review - Unmitigated, with error

A check that the deposit return non-zero market shares has been added in `_marketSupply`.

```
if (!_revertOnFail && _market.previewDeposit(_assets) == 0) {
    return (false, 0);
}
...
try _market.deposit(_assets, address(this)) returns (uint256 gotShares) {
    require(gotShares != 0, ErrorsLib.ZeroShares());

    shares = gotShares;
    success = true;

    _priceManipulationCheck(_market, shares, _assets);
} catch (bytes memory data) {
    if (_revertOnFail) ErrorsLib.revertBytes(data);
}
```

`_supplyERC4626` calls `_marketSupply` with `_revertOnFail = false`. Then, when the market deposit would return 0 market shares `_marketSupply` will return `(false, 0)` (since `deposit` returns the same or more shares as `previewDeposit`). `reallocate` calls `_marketSupply` with `_revertOnFail = true`. Then, when the market deposit returns 0 market shares the transaction reverts. This revert is problematic because this may allow an attacker to DoS a `reallocate` by depositing such that the reallocation only attempts to deposit such a reverting amount in the market. This error is also reported separately.

This mitigation is ineffective since the attacker can just deposit e.g. 2 assets for 1 market share (redeemable for 1 asset). The typical rounding loss is 1 wei, regardless of the magnitude of the amounts. Suppose the decimals offset is 6. This will then return 2e6 shares, but the vault's totalAssets is now only 1. The price is then (2e6 + 1e6) / (1 + 1) = 1.5e6 shares per asset. The attacker can then withdraw the 1 asset for 1.5e6 shares, leaving the vault with 0.5e6 shares minted and no assets. Depositing 2 again returns 3e6 shares. The price is then (0.5e6 + 3e6 + 1e6) / (1 + 1) = 2.25e6. He can again withdraw 2.25e6 shares and leave the vault with 1.25e6 shares and no assets. We see that each deposit/withdraw iteration inflates the shares by 50%, rather than 100% if we could deposit 1 asset for 0 market shares. This only means that we need to perform 2 * 1 / log2(1.5) ≈ 3.42 times more function calls than before to inflate it as much. The calls needed were at most 256, so still at most a very feasible 876 function calls.

Recommendation

Rounding losses are a part of ERC4626. Therefore this deflation attack cannot be fully prevented in theory, but must be made unfeasible by making the effect negligible. I believe the only effective solution is to set the virtual assets using the same offset as for the virtual shares. This is equivalent to an initial deposit that cannot be withdrawn. Then a deposit increasing the totalAssets by 1 less than it should is negligible compared to a number such as 1e18, and it is unfeasible to repeat this anywhere near 1e18 times.

Links to affected code

SiloVault.sol#L1

Disclosures

C4 is an open organization governed by participants in the community.

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.