// Private Smart Contract Security Assessment 07.22.2024 - 08.09.2024

Zerolend-One

ZeroLend

/ DRAFT /

HALBORN

Zerolend-One - ZeroLend

Prepared by: HALBORN

Last Updated 08/09/2024

Date of Engagement by: July 22nd, 2024 - August 9th, 2024

Summary

0% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
19	1	1	9	0	8

TABLE OF CONTENTS

1. Introduction

ZeroLend engaged Halborn to conduct a security assessment on their smart contracts beginning on 2024-07-22 and ending on 2024-08-09. The security assessment was scoped to the smart contracts provided in directly be the team.

Commit hashes and further details can be found in the Scope section of this report.

2. Assessment Summary

The team at Halborn was provided 3 weeks for the engagement and assigned one full-time security engineer to check the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified several security concerns that should be addressed by the ZeroLend team.

3. Test Approach And Methodology

Halborn performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Smart contract manual code review and walkthrough to identify any logic issue.
- Thorough assessment of safety and usage of critical Solidity variables and functions in scope that could lead to arithmetic related vulnerabilities.
- Manual testing by custom scripts.
- Graphing out functionality and contract logic/connectivity/functions (solgraph).
- Static Analysis of security for scoped contract, and imported functions. (Slither).
- Local or public testnet deployment (Foundry, Remix IDE).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: zerolend-one

(b) Assessed Commit ID: f7f26be

(c) Items in scope:

- contracts/core/pool/Pool.sol
- contracts/core/pool/PoolFactory.sol
- contracts/core/pool/PoolGetters.sol
- contracts/core/pool/PoolRentrancyGuard.sol
- contracts/core/pool/PoolSetters.sol
- contracts/core/pool/PoolStorage.sol
- contracts/core/pool/configuration/DataTypes.sol
- contracts/core/pool/configuration/PositionBalanceConfiguration.sol
- contracts/core/pool/configuration/ReserveConfiguration.sol
- contracts/core/pool/configuration/ReserveSuppliesConfiguration.sol
- contracts/core/pool/configuration/TokenConfiguration.sol
- contracts/core/pool/configuration/UserConfiguration.sol
- contracts/core/pool/logic/BorrowLogic.sol
- contracts/core/pool/logic/FlashLoanLogic.sol
- contracts/core/pool/logic/GenericLogic.sol
- contracts/core/pool/logic/LiquidationLogic.sol
- contracts/core/pool/logic/PoolLogic.sol
- contracts/core/pool/logic/ReserveLogic.sol
- contracts/core/pool/logic/SupplyLogic.sol
- contracts/core/pool/logic/ValidationLogic.sol
- contracts/core/pool/manager/PoolConfigurator.sol
- contracts/core/pool/manager/PoolManager.sol
- contracts/core/pool/utils/MathUtils.sol
- contracts/core/pool/utils/PercentageMath.sol
- contracts/core/pool/utils/WadRayMath.sol
- contracts/core/positions/NFTPositionManager.sol
- contracts/core/positions/NFTPositionManagerGetters.sol
- contracts/core/positions/NFTPositionManagerSetters.sol
- contracts/core/positions/NFTPositionManagerStorage.sol
- contracts/core/positions/NFTRewardsDistributor.sol
- contracts/core/proxy/RevokableBeaconProxy.sol
- contracts/core/vaults/CuratedVault.sol
- contracts/core/vaults/CuratedVaultBase.sol
- contracts/core/vaults/CuratedVaultFactory.sol
- contracts/core/vaults/CuratedVaultGetters.sol
- contracts/core/vaults/CuratedVaultRoles.sol

- contracts/core/vaults/CuratedVaultSetters.sol
- contracts/core/vaults/CuratedVaultStorage.sol
- contracts/core/vaults/libraries/ConstantsLib.sol
- contracts/core/vaults/libraries/MathLib.sol
- contracts/core/vaults/libraries/PendingLib.sol
- contracts/core/vaults/libraries/SharesMathLib.sol
- contracts/core/vaults/libraries/UtilsLib.sol

Out-of-Scope:

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	1	9	0	8

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-04 - LIQUIDITY MISALLOCATION DUE TO FRONT-RUNNING VECTOR IN NFTPOSITIONMANAGER	CRITICAL	-
HAL-07 - IMPOSSIBILITY OF LIQUIDATION LEADING TO PROTOCOL INSOLVENCY	HIGH	-
HAL-08 - BYPASS OF INTEREST ACCRUAL TO TREASURY	MEDIUM	-
HAL-03 - LACK OF BALANCE CHECK IN SUPPLYETH AND REPAYETH FUNCTIONS	MEDIUM	-
HAL-01 - SUPPLY CAP BYPASS VIA FLASHLOAN	MEDIUM	-

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-05 - INCORRECT ARGUMENT ORDER IN INTEREST CALCULATIONS	MEDIUM	-
HAL-02 - USE OF DEPRECATED CHAINLINK API	MEDIUM	-
HAL-11 - MISSING MINIMUM ACCEPTABLE AMOUNTS IN THE VAULT FLOWS	MEDIUM	-
HAL-13 - INABILITY TO ACCRUE LIQUIDATION PROTOCOL FEE	MEDIUM	-
HAL-15 - INCOMPLETE IMPLEMENTATION OF TREASURY ACCRUALS	MEDIUM	-
HAL-18 - POTENTIAL TO SKEW LIQUIDITY ALLOCATION	MEDIUM	-
HAL-09 - USE OF OWNABLE LIBRARY WITH SINGLE-STEP OWNERSHIP TRANSFER	INFORMATIONAL	-
HAL-06 - REDUNDANT CHECK IN REPAYETH FUNCTION	INFORMATIONAL	-
HAL-10 - USE OF CUSTOM ERRORS INSTEAD OF REVERT STRINGS	INFORMATIONAL	-
HAL-12 - UNUSED IMPORTS	INFORMATIONAL	-

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-14 - UNUSED HOOKS IN _FLASHLOAN FUNCTION	INFORMATIONAL	-
HAL-16 - MISSING CALLS TO INIT FUNCTIONS	INFORMATIONAL	-
HAL-17 - LACK OF CONSISTENCY WITH PRAGMA VERSIONS	INFORMATIONAL	-
HAL-19 - OPEN TODO AND OBSOLETE COMMENTS	INFORMATIONAL	-

7. FINDINGS & TECH DETAILS

7.1 (HAL-04) LIQUIDITY MISALLOCATION DUE TO FRONT-RUNNING VECTOR IN NFTPOSITIONMANAGER

// CRITICAL

Description

The NFTPositionManager contract implements methods that enable users to operate on lending pools indirectly, by use of pre-minted NFT tokens that represent a liquidity position in particular pool. The logic for supplying assets to these pools is implemented in the `supply` and `supplyETH` methods.

```
64    /// @inheritdoc INFTPositionManager
65    function supply(AssetOperationParams memory params) external {
66      if (params.asset == address(0)) revert NFTErrorsLib.ZeroAddressNotA
67      IERC20Upgradeable(params.asset).safeTransferFrom(msg.sender, addres
68      _supply(params);
69    }
70
71    /// @inheritdoc INFTPositionManager
72    function supplyETH(AssetOperationParams memory params) external payab
73      params.asset = address(weth);
74      weth.deposit{value: params.amount}();
75      _supply(params);
76    }
```

Both methods internally call the `_supply` to handle the actual supply logic.

```
45    function _supply(AssetOperationParams memory params) internal nonReer
46      if (params.amount == 0) revert NFTErrorsLib.ZeroValueNotAllowed();
47      if (params.tokenId == 0) params.tokenId = _nextId - 1;
48      IPool pool = IPool(_positions[params.tokenId].pool);
49
50      IERC20(params.asset).forceApprove(address(pool), params.amount);
51      pool.supply(params.asset, address(this), params.amount, params.toke
52
53      // update incentives
54      uint256 balance = pool.getBalance(params.asset, address(this), para
55      _handleSupplies(address(pool), params.asset, params.tokenId, balanc
56
57      emit NFTEventsLib.Supply(params.asset, params.tokenId, params.amour
58    }
```

This one, before calling the pool to manage the incoming liquidity, checks if the provided `tokenId` is equal to 0, indicating that the user's intention was to point to the last minted NFT. An attacker, observing that

a transaction with `tokenId` 0 is sent to the mempool, can front-run it by minting a new token. As a result, the victim's liquidity will be directed to the attacker's token, leading to financial losses of the entire deposit.
Note that this attack vector can similarly be exploited with use of repay flow.

Proof of Concept

```
function test_supplyFrontRunning() external {
    tokenA.mint(alice, 10 ether);
    //Alice mints a new position
    vm.prank(alice);
    nftPositionManager.mint(address(pool));

    //Attacker mints a new token to front-run the observed transaction
    vm.prank(attacker);
    nftPositionManager.mint(address(pool));

    //Alice supplies liquidity
    DataTypes.ExtraData memory extraData = DataTypes.ExtraData({
        hookData: "",
        interestRateData: ""
    });
    INFTPositionManager.AssetOperationParams memory params = INFTPositionMa
        asset: address(tokenA),
        target: alice,
        amount: 10 ether,
        tokenId: 0,
        data: extraData
    });
    vm.startPrank(alice);
        tokenA.approve(address(nftPositionManager), 10 ether);
        nftPositionManager.supply(params);
    vm.stopPrank();

    //Liquidity is misdirected to attacker's position
    bytes32 position = keccak256(abi.encodePacked(address(nftPositionManage
    assertEq(pool.supplyShares(address(tokenA), position), 10 ether);
    assertEq(nftPositionManager.ownerOf(2), attacker);
}
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:H/Y:N/R:N/S:U (10.0)

Recommendation

Ensure that the `tokenId` is explicitly specified by the user and not defaulted to the last minted NFT, or allow this only when the address set in the provided `AssetOperationParams` struct equals the token owner.

References

zerolend/zerolend-one/contracts/core/positions/NFTPositionManager.sol#L65-L76
zerolend/zerolend-one/contracts/core/positions/NFTPositionManager.sol#L112-L124
zerolend/zerolend-one/contracts/core/positions/NFTPositionManagerSetters.sol#L45-L58

7.2 (HAL-07) IMPOSSIBILITY OF LIQUIDATION LEADING TO PROTOCOL INSOLVENCY

// HIGH

Description

During the liquidation process, the `executeLiquidationCall` function is responsible for performing necessary calculations of the user's debt and collateral balances, validating the conditions for liquidation, updating interest rates, and then executing the transfer of assets to cover the debt. To reflect the actual collateral liquidation, the `_burnCollateralTokens` method is called.

```
208   function _burnCollateralTokens(
209     DataTypes.ReserveData storage collateralReserve,
210     DataTypes.ExecuteLiquidationCallParams memory params,
211     LiquidationCallLocalVars memory vars,
212     DataTypes.PositionBalance storage balances,
213     DataTypes.ReserveSupplies storage totalSupplies
214   ) internal {
215     DataTypes.ReserveCache memory collateralReserveCache = collateralRe
216     collateralReserve.updateState(params.reserveFactor, collateralReser
217     collateralReserve.updateInterestRates(
218       totalSupplies,
219       collateralReserveCache,
220       params.collateralAsset,
221       0,
222       IPool(params.pool).getReserveFactor(),
223       vars.actualCollateralToLiquidate,
224       params.position,
225       params.data.interestRateData
226     );
227
228     // Burn the equivalent amount of aToken, sending the underlying to
229     balances.withdrawCollateral(totalSupplies, vars.actualCollateralTol
230     IERC20(params.collateralAsset).transfer(msg.sender, vars.actualColl
231   }
```

The primary purpose of this function is to handle the internal accounting of collateral tokens and the subsequent transfer of the underlying collateral asset to the liquidator.
However, the address of the collateral token is wrapped in the IERC20 interface. Some tokens do not return a bool (e.g., USDT, BNB, OMG) on ERC20 methods, and wrapping them in this way will lead to a revert, as it expects a boolean return. This vulnerability allows an adversary to:

- create a pool consisting of such token,
- take a risk-free loan by providing it as collateral,
- avoid liquidation even if appropriate circumstances occur.

If the value of the collateral becomes less than the loan, this will deepen the protocol's insolvency.

Proof of Concept

```
  function test_USDTCannotBeLiquidable() external {
    //Attacker deployed a pool with USDT and has 100e18 USDT
    usdt.mint(attacker, 1000 ether);
    usdt.mint(bob, 100 ether);

    _mintAndApprove(bob, tokenB, 2000 ether, address(poolUSDT));
    vm.prank(bob);
      poolUSDT.supplySimple(address(tokenB), bob, 750 ether, 0);

    //Attacker provides USDT as liquidity and borrow another token
    vm.startPrank(attacker);
      usdt.approve(address(poolUSDT), 550 ether);
      poolUSDT.supplySimple(address(usdt), attacker, 550 ether, 0);
      poolUSDT.borrowSimple(address(tokenB), attacker, 100 ether, 0);
    vm.stopPrank();

    //After some time attacker's position becomes liquidable
    oracleA.setAnswer(5e3);
    bytes32 position = keccak256(abi.encodePacked(address(attacker), 'index

    //Liquidator attempts to liquidate position
    vm.prank(bob);
    vm.expectRevert();
    poolUSDT.liquidateSimple(address(usdt), address(tokenB), position, 10 e
  }
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:H/Y:M/R:N/S:U (8.8)

Recommendation

Use the SafeERC20 library implementation from OpenZeppelin and call `safeTransfer` or `safeTransferFrom` when handling ERC20 tokens. Replace every occurrence of `transfer`/`transferFrom` with the safe wrapper.

References

zerolend/zerolend-one/contracts/core/pool/logic/LiquidationLogic.sol#L230
zerolend/zerolend-one/contracts/core/pool/logic/LiquidationLogic.sol#L189

7.3 (HAL-08) BYPASS OF INTEREST ACCRUAL TO TREASURY

// MEDIUM

Description

The `forceUpdateReserve` function is used during vault-related operations to update the cumulative and borrow indexes. It calls the `updateState` method on the `DataTypes.ReserveData` structure, which handles the updating of liquidity cumulative index and the borrow index, and includes logic to accrue interest to the treasury. However, instead of the reserve factor associated with the proper pool, the function pass 0.

```
161   function forceUpdateReserve(address asset) public {
162     DataTypes.ReserveData storage reserve = _reserves[asset];
163     DataTypes.ReserveCache memory cache = reserve.cache(_totalSupplies[
164     reserve.updateState(0, cache);
165   }
```

The `updateState` function, defined in the ReserveLogic contract, performs the following operations:

- checks if the last update timestamp is equal to the current block timestamp. If they are the same, the function returns early, skipping the state update,
- updates the liquidity cumulative index and borrow index by calling `_updateIndexes`,
- accrues interest to the treasury by calling `_accrueToTreasury`.

```
88    function updateState(DataTypes.ReserveData storage self, uint256 _res
89      // If time didn't pass since last stored timestamp, skip state upda
90      if (self.lastUpdateTimestamp == uint40(block.timestamp)) return;
91
92      _updateIndexes(self, _cache);
93      _accrueToTreasury(_reserveFactor, self, _cache);
94
95      self.lastUpdateTimestamp = uint40(block.timestamp);
96    }
```

The `_accrueToTreasury` method is responsible for minting part of the repaid interest to the reserve treasury based on the reserve factor.

```
198   function _accrueToTreasury(uint256 reserveFactor, DataTypes.ReserveDa
199     if (reserveFactor == 0) return;
200     AccrueToTreasuryLocalVars memory vars;
201
202     // calculate the total variable debt at moment of the last interact
203     vars.prevtotalDebt = _cache.currDebtShares.rayMul(_cache.currBorrow
204
205     // calculate the new total variable debt after accumulation of the
206
```

```
207       vars.currtotalDebt = _cache.currDebtShares.rayMul(_cache.nextBorrow
208
209       // debt accrued is the sum of the current debt minus the sum of the
210       vars.totalDebtAccrued = vars.currtotalDebt - vars.prevtotalDebt;
211
212       vars.amountToMint = vars.totalDebtAccrued.percentMul(reserveFactor)
213
214       if (vars.amountToMint != 0) _reserve.accruedToTreasuryShares += var
      }
```

As the `forceUpdateReserve` function is publicly accessible, users can call this function immediately before any direct pool operation. By doing so, they can set the `lastUpdateTimestamp` to the current block timestamp. Consequently, when the pool operation attempts to call `updateState`, the timestamp check causes the function to return early, skipping the state update and, more importantly, the accrual of interest to the treasury. This can result in a loss of revenue for the treasury, as the interest that should be accrued to it is bypassed.

Proof of Concept

```
function test_bypassInterestAccrual() external {
  _mintAndApprove(alice, tokenA, 4000 ether, address(pool));

  // Set the reserve factor to 1000 bp (10%)
  poolFactory.setReserveFactor(10000);

  // Alice supplies and borrows tokenA from the pool
  vm.startPrank(alice);
    pool.supplySimple(address(tokenA), alice, 2000 ether, 0);
    pool.borrowSimple(address(tokenA), alice, 800 ether, 0);

    vm.warp(block.timestamp + 10 days);

    assertGt(pool.getDebt(address(tokenA), alice, 0), 0);
  vm.stopPrank();

  //Anyone can front-run repayment skipping the interest accrual to the t
  pool.forceUpdateReserve(address(tokenA));

  vm.startPrank(alice);
    tokenA.approve(address(pool), UINT256_MAX);
    pool.repaySimple(address(tokenA), UINT256_MAX, 0);
  vm.stopPrank();

  //No interest was accrued to the treasury
  DataTypes.ReserveData memory data = pool.getReserveData(address(tokenA))
```

```
    assertEq(data.accruedToTreasuryShares, 0);
}
```

```
Ran 1 test for test/forge/core/pool/PoolBorrowTests.t.sol:PoolBorrowTests
[PASS] test_bypassInterestAccrual() (gas: 453750)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.47ms (637.42µs CPU time)

Ran 1 test suite in 138.46ms (3.47ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:M/R:N/S:U (6.3)

Recommendation

Add checks to ensure that `forceUpdateReserve` is called only by the proper vault. Additionally, as a similar griefing attack is possible with the use of the vault, consider reimplementing the reserve update mechanism to always use the proper reserve factor.

References

zerolend/zerolend-one/contracts/core/pool/Pool.sol#L164

7.4 (HAL-03) LACK OF BALANCE CHECK IN SUPPLYETH AND REPAYETH FUNCTIONS

// MEDIUM

Description

The `sweep` function, implemented in the NFTPositionManager contract, allows an actor with a DEFAULT_ADMIN_ROLE role to transfer all mistakenly sent or outstanding tokens to their address.

```
127   function sweep(address token) external onlyRole(DEFAULT_ADMIN_ROLE) {
128     if (token == address(0)) {
129       uint256 bal = address(this).balance;
130       payable(msg.sender).transfer(bal);
131     } else {
132       IERC20Upgradeable erc20 = IERC20Upgradeable(token);
133       erc20.transfer(msg.sender, erc20.balanceOf(address(this)));
134     }
135   }
```

The `supplyETH` and `repayETH` methods in the NFTPositionManager contract allow users to interact with lending pools using ETH. However, these functions do not check if the transaction sender has provided the required amount of native coins. This allows anyone to use the outstanding ETH balance in the contract to supply or repay positions without sending any additional ETH themselves.

```
72   function supplyETH(AssetOperationParams memory params) external payab
73     params.asset = address(weth);
74     weth.deposit{value: params.amount}();
75     _supply(params);
76   }
```

```
119   function repayETH(AssetOperationParams memory params) external payabl
120     params.asset = address(weth);
121     weth.deposit{value: params.amount}();
122     require(params.asset == address(weth), 'not weth');
123     _repay(params);
124   }
```

Proof of Concept

```
function test_lackOfBalanceCheck() public {
    //10 ethers are initially in the NFTPositionManager contract
    vm.deal(address(nftPositionManager), 10 ether);
```

```
    DataTypes.ExtraData memory extraData = DataTypes.ExtraData({
        hookData: "",
        interestRateData: ""
    });

    INFTPositionManager.AssetOperationParams memory params = INFTPositionMa
        asset: address(0),
        target: alice,
        amount: 10 ether,
        tokenId: 1,
        data: extraData
    });

    //Alice mints a new position and and supply 10 eth
    vm.startPrank(alice);
        nftPositionManager.mint(address(poolWETH));
        nftPositionManager.supplyETH{value: 0}(params);
    vm.stopPrank();

    //The position now has 10 ether supplied, taken from the contract's bal
    bytes32 position = keccak256(abi.encodePacked(address(nftPositionManage

    assertEq(poolWETH.supplyShares(address(weth), position), 10 ether);
    assertEq(nftPositionManager.ownerOf(1), alice);
}
```

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:C (6.2)

Recommendation

Implement a check within the `supplyETH` and `repayETH` functions to ensure that the transaction sender has provided the required amount of ETH before proceeding with the operation.

References

zerolend/zerolend-one/contracts/core/positions/NFTPositionManager.sol#L72-L76
zerolend/zerolend-one/contracts/core/positions/NFTPositionManager.sol#L119-L124

7.5 (HAL-01) SUPPLY CAP BYPASS VIA FLASHLOAN

// MEDIUM

Description

The `executeSupply` function in the SupplyLogic contract is responsible for managing the supply of tokens within a reserve. It uses the `validateSupply` function from the ValidationLogic library to ensure that the supply cap for a reserve is not exceeded.

```
58  function executeSupply(
59    DataTypes.ReserveData storage reserve,
60    DataTypes.UserConfigurationMap storage userConfig,
61    DataTypes.PositionBalance storage balance,
62    DataTypes.ReserveSupplies storage totalSupplies,
63    DataTypes.ExecuteSupplyParams memory params
64  ) external returns (DataTypes.SharesType memory minted) {
65    DataTypes.ReserveCache memory cache = reserve.cache(totalSupplies);
66    reserve.updateState(params.reserveFactor, cache);
67    ValidationLogic.validateSupply(cache, reserve, params, params.pool)
```

The `validateSupply` function checks if the supply cap of the reserve is set to 0 (indicating no supply limit) or sums the balance of the reserve token with the accrued treasury reward to ensure it is below the reserve's supply cap. However, it relies on the current balance of the token, which can be manipulated using a flashloan:

- A flashloan is taken for the pre-deployed contract,
- The contract supplies the borrowed assets to the lending pool,
- As the underlying balance decreases by the loaned amount, this will bypass the supply cap check.

Bypassing the supply cap limits on a reserve leads to over-supply and excessive protocol exposure to a single asset, which is not in accordance with the protocol's intended risk management.

```
70  function validateSupply(
71    DataTypes.ReserveCache memory cache,
72    DataTypes.ReserveData storage reserve,
73    DataTypes.ExecuteSupplyParams memory params,
74    address pool
75  ) internal view {
76    require(params.amount != 0, PoolErrorsLib.INVALID_AMOUNT);
77
78    (bool isFrozen,) = cache.reserveConfiguration.getFlags();
79    require(!isFrozen, PoolErrorsLib.RESERVE_FROZEN);
80
81    uint256 supplyCap = cache.reserveConfiguration.getSupplyCap();
82    // todo
83    require(
```

```
        supplyCap == 0
          || (
            (IERC20(params.asset).balanceOf(pool) + uint256(reserve.accru
          ) <= supplyCap * (10 ** cache.reserveConfiguration.getDecimals(
        PoolErrorsLib.SUPPLY_CAP_EXCEEDED
      );
```

Proof of Concept

```
  function test_supplyCapBypass() external {
    //Mint and approve tokens
    _mintAndApprove(bob, tokenA, 200 ether, address(pool));
    _mintAndApprove(bob, tokenB, 200 ether, address(pool));
    _mintAndApprove(attacker, tokenA, 100 ether, address(pool));

    //Bob supplies tokens to the pool to generate proper flashloan conditio
    vm.startPrank(bob);
      pool.supplySimple(address(tokenA), bob, 200 ether, 0);
      pool.supplySimple(address(tokenB), bob, 200 ether, 0);
    vm.stopPrank();

    vm.prank(attacker);
    flashLoanReceiver = new FlashLoanReceiver(pool, attacker);
    tokenA.mint(address(flashLoanReceiver), 205 ether);

    //Set a supply cap for tokenA
    configurator.setSupplyCap(IPool(address(pool)), address(tokenA), 200);

    //Attacker tries to supply more than the cap, expecting a revert
    vm.startPrank(attacker);
      vm.expectRevert(bytes('SUPPLY_CAP_EXCEEDED'));
      pool.supplySimple(address(tokenA), attacker, 1, 0);

    //Attacker uses a flashloan to manipulate the supply cap check
      bytes memory emptyParams;
      pool.flashLoanSimple(address(address(flashLoanReceiver)), address(tok
    vm.stopPrank();
  }



contract FlashLoanReceiver {
  IPool public pool;
  address public owner;
  constructor(IPool _pool, address _owner) {
```

```
    pool = _pool;
    owner = _owner;
  }
  function executeOperation(
    address asset,
    uint256 amount,
    uint256 premium,
    address,
    bytes memory
  ) public returns (bool) {
    //Approve the pool to pull the flashloaned amount
    IERC20(asset).approve(address(pool), type(uint256).max);

    //Supply the flashloaned amount to the pool
    pool.supplySimple(address(asset), owner, amount, 0);

    return true;
  }
}
```

```
Ran 1 test for test/forge/core/pool/PoolSupplyTests.t.sol:PoolSupplyTests
[PASS] test_supplyCapBypass() (gas: 1090235)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 10.16ms (2.30ms CPU time)

Ran 1 test suite in 140.83ms (10.16ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:L/R:N/S:U (5.6)

Recommendation

Disallow supplying liquidity in the same block as a flashloan is taken. Alternatively, use internal accounting mechanisms to determine how many tokens can be borrowed, ensuring that the temporary increase in token balance due to flashloans does not affect supply cap validations.

References

zerolend/zerolend-one/contracts/core/pool/logic/ValidationLogic.sol#L86-L88

7.6 (HAL-05) INCORRECT ARGUMENT ORDER IN INTEREST CALCULATIONS

// MEDIUM

Description

During the liquidation workflow, the amount of collateral available for liquidation is calculated, and the internal accounting is adjusted accordingly. For this purpose, the `_burnCollateralTokens` method is called.

```
208   function _burnCollateralTokens(
209     DataTypes.ReserveData storage collateralReserve,
210     DataTypes.ExecuteLiquidationCallParams memory params,
211     LiquidationCallLocalVars memory vars,
212     DataTypes.PositionBalance storage balances,
213     DataTypes.ReserveSupplies storage totalSupplies
214   ) internal {
215     DataTypes.ReserveCache memory collateralReserveCache = collateralRe
216     collateralReserve.updateState(params.reserveFactor, collateralReser
217     collateralReserve.updateInterestRates(
218       totalSupplies,
219       collateralReserveCache,
220       params.collateralAsset,
221       0,
222       IPool(params.pool).getReserveFactor(),
223       vars.actualCollateralToLiquidate,
224       params.position,
225       params.data.interestRateData
226     );
```

The call to `updateInterestRates` in the `_burnCollateralTokens` method passes the arguments in an incorrect order. Specifically, the method incorrectly passes the `reserveFactor` (5th argument) and `liquidityAdded` (6th argument) in reverse order.

```
146   function updateInterestRates(
147     DataTypes.ReserveData storage _reserve,
148     DataTypes.ReserveSupplies storage totalSupplies,
149     DataTypes.ReserveCache memory _cache,
150     address _reserveAddress,
151     uint256 _reserveFactor,
152     uint256 _liquidityAdded,
153     uint256 _liquidityTaken,
154     bytes32 _position,
155
156
```

```
        bytes memory _data
    ) internal {
```

This will lead to incorrect interest rate calculations due to the misallocation of the reserve factor, which is used to allocate a portion of the borrow interest to the protocol. Additionally, financial discrepancies in the internal accounting, worsening over time, will affect the revenue calculations.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:M/R:N/S:U (5.6)

Recommendation

Update the `_burnCollateralTokens` method to correctly pass the `reserveFactor` and `liquidityAdded` arguments to the `updateInterestRates` function to ensure accurate interest rate calculations.

References

zerolend/zerolend-one/contracts/core/pool/logic/LiquidationLogic.sol#L208-L226
zerolend/zerolend-one/contracts/core/pool/logic/ReserveLogic.sol#L146-L156

7.7 (HAL-02) USE OF DEPRECATED CHAINLINK API

// MEDIUM

Description

The `getAssetPrice` function is used in the protocol to discover the price of the intended asset. It internally calls Chainlink's deprecated `latestAnswer` function. This function also does not guarantee that the price returned by the Chainlink price feed is not stale, and there are no additional checks to ensure that the return values are valid.

```
58    function getAssetPrice(address reserve) public view override returns
59      return uint256(IAggregatorInterface(_reserves[reserve].oracle).late
60    }
```

The lack of proper checks on the staleness of Chainlink's return values can lead to incorrect calculation of the collateral and borrow prices of the underlying assets, and allows for excessive sandwich attacks.

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:U (5.0)

Recommendation

Modify the function to utilize Chainlink's `latestRoundData`. This approach provides detailed information about the most recent price round, ensuring that retrieved the price remains up-to-date.
Example implementation:

```
uint256 private constant GRACE_PERIOD_TIME = 3600; // duration until the pr

function getChainlinkPrice(AggregatorV2V3Interface feed) internal {
    (uint80 roundId, int256 price, uint startedAt, uint updatedAt, uint80 a
    require(price > 0, "Invalid price");
    require(block.timestamp <= updatedAt + GRACE_PERIOD_TIME, "Stale price"
    return price;
}
```

References

zerolend/zerolend-one/contracts/core/pool/PoolGetters.sol#L159

7.8 (HAL-11) MISSING MINIMUM ACCEPTABLE AMOUNTS IN THE VAULT FLOWS

// MEDIUM

Description

The ERC4626 standard was implemented by a vault system presented in the CuratedVault. As stated in EIP-4626:

`If implementors intend to support EOA account access directly, they should consider adding an additional function call for deposit/mint/withdraw/redeem with the means to accommodate slippage loss or unexpected deposit/withdrawal limits, since they have no other means to revert the transaction if the exact output amount is not achieved.`

As the vault is intended to be used directly by EOA accounts, the lack of minimum acceptable amount controls can result in significant slippage. This slippage can cause users to receive fewer tokens or shares than anticipated due to variations in the exchange rate that may occur before the actual transaction execution.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:M/Y:N/R:N/S:U (5.0)

Recommendation

Consider adding parameters to both the `deposit`, `withdraw`, `mint` and `redeem` functions allowing users to specify minimum acceptable amounts for their transactions and ensure these amounts are checked and enforced.

References

zerolend/zerolend-one/contracts/core/vaults/CuratedVault.sol#L322
zerolend/zerolend-one/contracts/core/vaults/CuratedVault.sol#L333
zerolend/zerolend-one/contracts/core/vaults/CuratedVault.sol#L344
zerolend/zerolend-one/contracts/core/vaults/CuratedVault.sol#L356

7.9 (HAL-13) INABILITY TO ACCRUE LIQUIDATION PROTOCOL FEE

// MEDIUM

Description

During the liquidation, the `_calculateAvailableCollateralToLiquidate` function, as the name suggest, calculates how much of a specific collateral can be liquidated for provided amount of debt asset. On the last part of this method, it calculates the final amounts of collateral to be returned to the user and the protocol fee during a liquidation.

```
367     if (liquidationProtocolFeePercentage != 0) {
368       vars.bonusCollateral = vars.collateralAmount - vars.collateralAmo
369       vars.liquidationProtocolFee = vars.bonusCollateral.percentMul(liq
370       return (vars.collateralAmount - vars.liquidationProtocolFee, vars
371     } else {
372       return (vars.collateralAmount, vars.debtAmountNeeded, 0);
373     }
```

Depending on whether the calculated liquidation protocol fee amount is a positive number, it is transferred to the treasury.

```
179     if (vars.liquidationProtocolFeeAmount != 0) {
180       uint256 liquidityIndex = collateralReserve.getNormalizedIncome();
181       uint256 scaledDownLiquidationProtocolFee = vars.liquidationProtoc
182       // todo
183       uint256 scaledDownUserBalance = balances[params.collateralAsset][
184
185       if (scaledDownLiquidationProtocolFee > scaledDownUserBalance) {
186         vars.liquidationProtocolFeeAmount = scaledDownUserBalance.rayMu
187       }
188
189       IERC20(params.collateralAsset).transfer(IPool(params.pool).factor
190     }
```

The `liquidationProtocolFeePercentage` variable, which is crucial for these calculations, is declared in the PoolFactory contract. However, it does not have a setter function, resulting in its value remaining permanently at 0. This prevent the application of the protocol liquidation fee, leading to loss of revenue for the protocol.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:M/R:N/S:U (5.0)

Recommendation

Implement a setter function that will allow setting the liquidation protocol fee within reasonable boundaries.

References

zerolend/zerolend-one/contracts/core/pool/PoolFactory.sol#L51

7.10 (HAL-15) INCOMPLETE IMPLEMENTATION OF TREASURY ACCRUALS

// MEDIUM

Description

The `executeMintToTreasury` function implemented in PoolLogic library, is intended to allow the treasury to pull the rewards accrued for the particular asset.

```
84    function executeMintToTreasury(mapping(address => DataTypes.ReserveDa
85      DataTypes.ReserveData storage reserve = reservesData[asset];
86
87      uint256 accruedToTreasuryShares = reserve.accruedToTreasuryShares;
88
89      if (accruedToTreasuryShares != 0) {
90        reserve.accruedToTreasuryShares = 0;
91        uint256 normalizedIncome = reserve.getNormalizedIncome();
92        uint256 amountToMint = accruedToTreasuryShares.rayMul(normalizedI
93
94        // todo mint and unwrap to treasury
95        // IAToken(reserve.aTokenAddress).mintToTreasury(amountToMint, no
96
97        emit PoolEventsLib.MintedToTreasury(asset, amountToMint);
98      }
99    }
```

However, the actual taking possession process (in the case of a forked protocol, minting native ERC20 tokens) is commented out.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:M/R:N/S:U (5.0)

Recommendation

Consider implementing the intended logic or completely removing the treasury rewards functionality.

References

zerolend/zerolend-one/contracts/core/pool/logic/PoolLogic.sol#L94-L95

7.11 (HAL-18) POTENTIAL TO SKEW LIQUIDITY ALLOCATION

// MEDIUM

Description

The `_supplyPool` function, defined in the CuratedVaultSetters contract, is designed to allocate the deposited liquidity across pools listed in the `supplyQueue`. The function attempts to deposit funds into, respecting the predefined caps for each one. If a pool has reached its cap, the function moves on to the next one, continuing this process until either all the liquidity is deposited or all pools are capped. If there is still liquidity left that cannot be allocated due to all caps being reached, the function reverts with an `AllCapsReached` error.

```
115   function _supplyPool(uint256 assets) internal {
116     for (uint256 i; i < supplyQueue.length; ++i) {
117       IPool pool = supplyQueue[i];
118
119       uint256 supplyCap = config[pool].cap;
120       if (supplyCap == 0) continue;
121
122       pool.forceUpdateReserve(asset());
123
124       uint256 supplyShares = pool.supplyShares(asset(), positionId);
125
126       // `supplyAssets` needs to be rounded up for `toSupply` to be rou
127       (uint256 totalSupplyAssets, uint256 totalSupplyShares,,) = pool.m
128       uint256 supplyAssets = supplyShares.toAssetsUp(totalSupplyAssets,
129
130       uint256 toSupply = UtilsLib.min(supplyCap.zeroFloorSub(supplyAsse
131
132       if (toSupply > 0) {
133         // Using try/catch to skip markets that revert.
134         try pool.supplySimple(asset(), address(this), toSupply, 0) {
135           assets -= toSupply;
136         } catch {}
137       }
138
139       if (assets == 0) return;
140     }
141
142     if (assets != 0) revert CuratedErrorsLib.AllCapsReached();
143   }
```

Attackers can manipulate the liquidity to force the system into reallocating liquidity in a way that skews distribution towards the next pools.

Consider a scenario where there are two pools, `Pool A` and `Pool B`, in the `supplyQueue`. Both `Pool A` and `Pool B` have a cap of 10M. Currently, `Pool A` holds 5M in liquidity, and `Pool B` holds 1M.

- The attacker deposits 10M into `Pool A`. This deposit brings `Pool A`'s total liquidity to 15M, fully reaching its cap. Since `Pool A` can no longer accept additional liquidity, the system attempts to allocate the remaining 5M to the `Pool B`, so now it hold 6M.
- The attacker withdraws the 10M, reducing the `Pool A` capacity back to 0, effectively emptying the pool. The `Pool B` is holding 6M of tokens.

This scenario might create significant imbalances in liquidity distribution, potentially leading to underfunded pools and distorted market dynamics. Over time, such skewed allocations could destabilize the system, compromising the efficiency of the underlying pools by distorting the interest rates.

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:U (5.0)

Recommendation

Consider recommending in the documentation or comments that vault creators should be aware of the potential risks associated with using pools that have a finite cap as the first market in the supplied pools list.

References

zerolend/zerolend-one/contracts/core/vaults/CuratedVaultSetters.sol

7.12 (HAL-09) USE OF OWNABLE LIBRARY WITH SINGLE-STEP OWNERSHIP TRANSFER

// INFORMATIONAL

Description

The current ownership transfer process for PoolFactory and CuratedVaultFactory contracts, inheriting from Ownable, involves the current owner calling the `transferOwnership` function

```
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero add
        _transferOwnership(newOwner);
    }
```

If the nominated EOA account is not a valid account, it is entirely possible that the owner may accidentally transfer ownership to an uncontrolled account, losing the access to all functions with the `onlyOwner` modifier.

BVSS

AO:S/AC:L/AX:L/C:N/I:M/A:L/D:N/Y:N/R:N/S:C (1.4)

Recommendation

It is recommended to implement a two-step process where the owner nominates an account and the nominated account needs to call an `acceptOwnership` function for the transfer of the ownership to fully succeed. This ensures the nominated EOA account is a valid and active account. This can be achieved by using OpenZeppelin's Ownable2Step contract instead of the Ownable.

7.13 (HAL-06) REDUNDANT CHECK IN REPAYETH FUNCTION

// INFORMATIONAL

Description

In the repayETH function implemented in the NFTPositionManager contract, the check `require(params.asset == address(weth), 'not weth');` is redundant because params.asset is already set to address(weth) at the beginning of the function.

```
function repayETH(AssetOperationParams memory params) external payable {
  params.asset = address(weth);
  weth.deposit{value: params.amount}();
  require(params.asset == address(weth), 'not weth');
  _repay(params);
}
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Remove the redundant check.

References

zerolend/zerolend-one/contracts/core/positions/NFTPositionManager.sol#L119-L124

7.14 (HAL-10) USE OF CUSTOM ERRORS INSTEAD OF REVERT STRINGS

// INFORMATIONAL

Description

In Solidity development, replacing hard-coded revert message strings with the Error() syntax is an optimization strategy that can significantly reduce gas costs. Hard-coded strings, stored on the blockchain, increase the size and cost of deploying and executing contracts. The Error() syntax allows for the definition of reusable, parameterized custom errors, leading to a more efficient use of storage and reduced gas consumption. This approach not only optimizes gas usage during deployment and interaction with the contract but also enhances code maintainability and readability by providing clearer, context-specific error information.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Consider replacing all revert strings with custom errors.

7.15 (HAL-12) UNUSED IMPORTS

// INFORMATIONAL

Description

Throughout the code in scope, there are several instances where the imports are declared but never used.
In `NFTPositionManagerSetters`:

- `import {ERC721Upgradeable} from '@openzeppelin/contracts-upgradeable/token/ERC721/ERC721Upgradeable.sol';`

In `PoolLogic`:

- `import {ValidationLogic} from './ValidationLogic.sol';`

In `ReserveLogic`:

- `import {PoolErrorsLib} from '../../../interfaces/errors/PoolErrorsLib.sol';`

In `PoolConfiguration`:

- `import {PoolErrorsLib} from '../../../interfaces/errors/PoolErrorsLib.sol';`

In `PoolManager`:

- `import {PoolErrorsLib} from '../../../interfaces/errors/PoolErrorsLib.sol';`

In `CuratedVaultSetters`:

- `import {ICuratedVaultBase, MarketConfig} from '../../interfaces/vaults/ICuratedVaultBase.sol';`

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Consider removing all unused imports.

7.16 (HAL-14) UNUSED HOOKS IN _FLASHLOAN FUNCTION

// INFORMATIONAL

Description

Both `flashLoan` and `flashLoanSimple` functions, defined in the Pool contract, internally call `_liquidate`, providing `DataTypes.ExtraData`, to maintain consistency with the other workflows, which use hooks before and after the proper operation execution.

```
139   /// @inheritdoc IPoolSetters
140   function flashLoan(address receiverAddress, address asset, uint256 am
141     _flashLoan(receiverAddress, asset, amount, params, data);
142   }
143
144   /// @inheritdoc IPoolSetters
145   function flashLoanSimple(address receiverAddress, address asset, uint
146     _flashLoan(receiverAddress, asset, amount, params, DataTypes.ExtraD
147   }
```

However, the `_flashLoan` function and subsequent execution never execute the hooks' logic.

```
178   function _flashLoan(
179     address receiverAddress,
180     address asset,
181     uint256 amount,
182     bytes calldata params,
183     DataTypes.ExtraData memory data
184   ) public virtual nonReentrant(RentrancyKind.FLASHLOAN) {
185     require(receiverAddress != address(0), PoolErrorsLib.ZERO_ADDRESS_N
186     DataTypes.FlashloanSimpleParams memory flashParams = DataTypes.Flas
187       receiverAddress: receiverAddress,
188       asset: asset,
189       amount: amount,
190       data: data,
191       reserveFactor: _factory.reserveFactor(),
192       params: params,
193       flashLoanPremiumTotal: _factory.flashLoanPremiumToProtocol()
194     });
195     FlashLoanLogic.executeFlashLoanSimple(address(this), _reserves[asse
196   }
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

If the `_flashLoan` function is not intended to use hooks, references to `DataTypes.ExtraData` should be removed from both `flashLoan` and `flashLoanSimple` to simplify the code and avoid confusion.

References

zerolend/zerolend-one/contracts/core/pool/Pool.sol#L140-L142
zerolend/zerolend-one/contracts/core/pool/Pool.sol#L145-L147

7.17 (HAL-16) MISSING CALLS TO INIT FUNCTIONS

// INFORMATIONAL

Description

The NFTPositionManager contracts are meant to be used via proxy pattern, and hence their implementations require the use of `initialize` functions instead of constructors. This requires derived contracts to call the corresponding init functions of their inherited base contracts. The mentioned contract is inheriting indirectly from ReentrancyGuardUpgradeable and MulticallUpgradeable, but their its function is missing the call to `__ReentrancyGuard_init` and `__Multicall_init`.

```
38    function initialize(address _factory, address _staking, address _owne
39      __ERC721Enumerable_init();
40      __ERC721_init('ZeroLend One Position', 'ZL-POS-ONE');
41      __AccessControlEnumerable_init();
42      __NFTRewardsDistributor_init(50_000_000, _staking, 14 days, _zero);
43
44      _grantRole(DEFAULT_ADMIN_ROLE, _owner);
45
46      factory = IPoolFactory(_factory);
47      weth = IWETH(_weth);
48      _nextId = 1;
49    }
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Consider implementing the intended logic or completely removing the treasury rewards functionality.

References

zerolend/zerolend-one/contracts/core/positions/NFTPositionManager.sol#L38-L49

7.18 (HAL-17) LACK OF CONSISTENCY WITH PRAGMA VERSIONS

// INFORMATIONAL

Description

The FlashLoanLogic, BorrowLogic, and NFTRewardDistributor (which also use an inconsistent version) contracts use floating pragma, in contrast with the rest of the codebase, which uses pragma pinned to version `0.8.19`. This bad practice can lead to unpredictable behavior due to differences in features and compatibility between minor versions.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Consider pinning the Solidity pragma to a specific version or using the floating pragma only for library contracts.

References

zerolend/zerolend-one/contracts/core/pool/logic/BorrowLogic.sol#L2
zerolend/zerolend-one/contracts/core/pool/logic/FlashLoanLogic.sol#L2
zerolend/zerolend-one/contracts/core/positions/NFTRewardsDistributor.sol#L2

7.19 (HAL-19) OPEN TODO AND OBSOLETE COMMENTS

// INFORMATIONAL

Description

The codebase contains TODOs and obsolete comments that originated from the forked codebase and may no longer be relevant to the current implementation. These comments can create confusion for readers, leading to potential misinterpretations of the code's intent. The TODO comments may unnecessarily suggest that the contract is not fully implemented.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Review and remove or update any TODOs and obsolete comments from the codebase to prevent developer confusion and ensure clarity regarding the contract's implementation status.

8. AUTOMATED TESTING

Static Analysis Report

Description

Halborn used automated testing techniques to enhance the coverage of certain areas of the scoped contracts. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified all the contracts in the repository and was able to compile them correctly into their abi and binary formats, Slither was run on the all-scoped contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

Slither results

Pool.sol

```
INFO:Detectors:
FlashLoanLogic._handleFlashLoanRepayment(DataTypes.ReserveData,DataTypes.ReserveSupplies,DataTypes.FlashLoanRepaymentParams) (contracts/core/pool/logic/FlashLoanLogic.sol#94-118) uses arbitrary from in transferFrom
: IERC20(_params.asset).safeTransferFrom(_params.receiverAddress,address(_params.pool),amountPlusPremium) (contracts/core/pool/logic/FlashLoanLogic.sol#108)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#arbitrary-from-in-transferfrom
INFO:Detectors:
FlashLoanLogic.executeFlashLoanSimple(address,DataTypes.ReserveData,DataTypes.ReserveSupplies,DataTypes.FlashloanSimpleParams) (contracts/core/pool/logic/FlashLoanLogic.sol#53-86) ignores return value by IERC20(_pa
rams.asset).transfer(_params.receiverAddress,_params.amount) (contracts/core/pool/logic/FlashLoanLogic.sol#67)
LiquidationLogic.executeLiquidationCall(mapping(address => DataTypes.ReserveData),mapping(uint256 => address),mapping(address => mapping(bytes32 => DataTypes.PositionBalance)),mapping(address => DataTypes.ReserveSu
pplies),mapping(bytes32 => DataTypes.UserConfigurationMap),DataTypes.ExecuteLiquidationCallParams) (contracts/core/pool/logic/LiquidationLogic.sol#96-200) ignores return value by IERC20(params.collateralAsset).tran
sfer(IPool(params.pool).factory().treasury(),vars.liquidationProtocolFeeAmount) (contracts/core/pool/logic/LiquidationLogic.sol#191)
LiquidationLogic._burnCollateralTokens(DataTypes.ReserveData,DataTypes.ExecuteLiquidationCallParams,LiquidationLogic.LiquidationCallLocalVars,DataTypes.PositionBalance,DataTypes.ReserveSupplies) (contracts/core/poo
l/logic/LiquidationLogic.sol#210-233) ignores return value by IERC20(params.collateralAsset).transfer(msg.sender,vars.actualCollateralToLiquidate) (contracts/core/pool/logic/LiquidationLogic.sol#232)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
```

PoolGetters.sol

```
INFO:Detectors:
PoolStorage._usersConfig (contracts/core/pool/PoolStorage.sol#31) is never initialized. It is used in:
        - PoolGetters.getUserAccountData(address,uint256) (contracts/core/pool/PoolGetters.sol#100-111)
        - PoolGetters.getUserConfiguration(address,uint256) (contracts/core/pool/PoolGetters.sol#119-121)
PoolStorage._reservesCount (contracts/core/pool/PoolStorage.sol#44) is never initialized. It is used in:
        - PoolGetters.getReservesList() (contracts/core/pool/PoolGetters.sol#134-140)
        - PoolGetters.getReservesCount() (contracts/core/pool/PoolGetters.sol#143-145)
PoolStorage._factory (contracts/core/pool/PoolStorage.sol#47) is never initialized. It is used in:
        - PoolGetters.getConfigurator() (contracts/core/pool/PoolGetters.sol#148-150)
        - PoolGetters.factory() (contracts/core/pool/PoolGetters.sol#163-165)
        - PoolGetters.getReserveFactor() (contracts/core/pool/PoolGetters.sol#168-170)
PoolStorage._hook (contracts/core/pool/PoolStorage.sol#50) is never initialized. It is used in:
        - PoolGetters.getHook() (contracts/core/pool/PoolGetters.sol#52-54)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables
```

PoolSetters.sol

```
INFO:Detectors:
FlashLoanLogic._handleFlashLoanRepayment(DataTypes.ReserveData,DataTypes.ReserveSupplies,DataTypes.FlashLoanRepaymentParams) (contracts/core/pool/logic/FlashLoanLogic.sol#94-118) uses arbitrary from in transferFrom
: IERC20(_params.asset).safeTransferFrom(_params.receiverAddress,address(_params.pool),amountPlusPremium) (contracts/core/pool/logic/FlashLoanLogic.sol#108)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#arbitrary-from-in-transferfrom
INFO:Detectors:
FlashLoanLogic.executeFlashLoanSimple(address,DataTypes.ReserveData,DataTypes.ReserveSupplies,DataTypes.FlashloanSimpleParams) (contracts/core/pool/logic/FlashLoanLogic.sol#53-86) ignores return value by IERC20(_pa
rams.asset).transfer(_params.receiverAddress,_params.amount) (contracts/core/pool/logic/FlashLoanLogic.sol#67)
LiquidationLogic.executeLiquidationCall(mapping(address => DataTypes.ReserveData),mapping(uint256 => address),mapping(address => mapping(bytes32 => DataTypes.PositionBalance)),mapping(address => DataTypes.ReserveSu
pplies),mapping(bytes32 => DataTypes.UserConfigurationMap),DataTypes.ExecuteLiquidationCallParams) (contracts/core/pool/logic/LiquidationLogic.sol#96-200) ignores return value by IERC20(params.collateralAsset).tran
sfer(IPool(params.pool).factory().treasury(),vars.liquidationProtocolFeeAmount) (contracts/core/pool/logic/LiquidationLogic.sol#191)
LiquidationLogic._burnCollateralTokens(DataTypes.ReserveData,DataTypes.ExecuteLiquidationCallParams,LiquidationLogic.LiquidationCallLocalVars,DataTypes.PositionBalance,DataTypes.ReserveSupplies) (contracts/core/poo
l/logic/LiquidationLogic.sol#210-233) ignores return value by IERC20(params.collateralAsset).transfer(msg.sender,vars.actualCollateralToLiquidate) (contracts/core/pool/logic/LiquidationLogic.sol#232)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
PoolStorage._reservesCount (contracts/core/pool/PoolStorage.sol#44) is never initialized. It is used in:
        - PoolGetters.getReservesList() (contracts/core/pool/PoolGetters.sol#134-140)
        - PoolGetters.getReservesCount() (contracts/core/pool/PoolGetters.sol#143-145)
        - PoolSetters._borrow(address,address,uint256,bytes32,DataTypes.ExtraData) (contracts/core/pool/PoolSetters.sol#89-119)
        - PoolSetters._liquidate(address,address,bytes32,uint256,DataTypes.ExtraData) (contracts/core/pool/PoolSetters.sol#148-176)
PoolStorage._factory (contracts/core/pool/PoolStorage.sol#47) is never initialized. It is used in:
        - PoolGetters.getConfigurator() (contracts/core/pool/PoolGetters.sol#148-150)
        - PoolGetters.factory() (contracts/core/pool/PoolGetters.sol#163-165)
        - PoolGetters.getReserveFactor() (contracts/core/pool/PoolGetters.sol#168-170)
        - PoolSetters._supply(address,uint256,bytes32,DataTypes.ExtraData) (contracts/core/pool/PoolSetters.sol#36-60)
        - PoolSetters._withdraw(address,address,uint256,bytes32,DataTypes.ExtraData) (contracts/core/pool/PoolSetters.sol#62-87)
        - PoolSetters._borrow(address,address,uint256,bytes32,DataTypes.ExtraData) (contracts/core/pool/PoolSetters.sol#89-119)
        - PoolSetters._repay(address,uint256,bytes32,DataTypes.ExtraData) (contracts/core/pool/PoolSetters.sol#121-146)
        - PoolSetters._liquidate(address,address,bytes32,uint256,DataTypes.ExtraData) (contracts/core/pool/PoolSetters.sol#148-176)
        - PoolSetters._flashLoan(address,address,uint256,bytes,DataTypes.ExtraData) (contracts/core/pool/PoolSetters.sol#178-196)
        - PoolSetters._setUserUseReserveAsCollateral(address,uint256,bool) (contracts/core/pool/PoolSetters.sol#198-217)
PoolStorage._hook (contracts/core/pool/PoolStorage.sol#50) is never initialized. It is used in:
        - PoolGetters.getHook() (contracts/core/pool/PoolGetters.sol#52-54)
        - PoolSetters._supply(address,uint256,bytes32,DataTypes.ExtraData) (contracts/core/pool/PoolSetters.sol#36-60)
        - PoolSetters._withdraw(address,address,uint256,bytes32,DataTypes.ExtraData) (contracts/core/pool/PoolSetters.sol#62-87)
        - PoolSetters._borrow(address,address,uint256,bytes32,DataTypes.ExtraData) (contracts/core/pool/PoolSetters.sol#89-119)
        - PoolSetters._repay(address,uint256,bytes32,DataTypes.ExtraData) (contracts/core/pool/PoolSetters.sol#121-146)
        - PoolSetters._liquidate(address,address,bytes32,uint256,DataTypes.ExtraData) (contracts/core/pool/PoolSetters.sol#148-176)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables
```

FlashLoanLogic.sol

```
INFO:Detectors:
FlashLoanLogic._handleFlashLoanRepayment(DataTypes.ReserveData,DataTypes.ReserveSupplies,DataTypes.FlashLoanRepaymentParams) (contracts/core/pool/logic/FlashLoanLogic.sol#94-118) uses arbitrary from in transferFrom
: IERC20(_params.asset).safeTransferFrom(_params.receiverAddress,address(_params.pool),amountPlusPremium) (contracts/core/pool/logic/FlashLoanLogic.sol#108)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#arbitrary-from-in-transferfrom
INFO:Detectors:
FlashLoanLogic.executeFlashLoanSimple(address,DataTypes.ReserveData,DataTypes.ReserveSupplies,DataTypes.FlashloanSimpleParams) (contracts/core/pool/logic/FlashLoanLogic.sol#53-86) ignores return value by IERC20(_pa
rams.asset).transfer(_params.receiverAddress,_params.amount) (contracts/core/pool/logic/FlashLoanLogic.sol#67)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
```

LiquidationLogic.sol

```
INFO:Detectors:
LiquidationLogic.executeLiquidationCall(mapping(address => DataTypes.ReserveData),mapping(uint256 => address),mapping(address => mapping(bytes32 => DataTypes.PositionBalance)),mapping(address => DataTypes.ReserveSupplies),mapping(bytes32 => DataTypes.UserConfigurationMap),DataTypes.ExecuteLiquidationCallParams) (contracts/core/pool/logic/LiquidationLogic.sol#96-200) ignores return value by IERC20(params.collateralAsset).transfer(IPool(params.pool).factory().treasury(),vars.liquidationProtocolFeeAmount) (contracts/core/pool/logic/LiquidationLogic.sol#191)
LiquidationLogic._burnCollateralTokens(DataTypes.ReserveData,DataTypes.ExecuteLiquidationCallParams,LiquidationLogic.LiquidationCallLocalVars,DataTypes.PositionBalance,DataTypes.ReserveSupplies) (contracts/core/pool/logic/LiquidationLogic.sol#210-233) ignores return value by IERC20(params.collateralAsset).transfer(msg.sender,vars.actualCollateralToLiquidate) (contracts/core/pool/logic/LiquidationLogic.sol#232)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
```

NFTPositionManager.sol

```
INFO:Detectors:
NFTPositionManager.supplyETH(INFTPositionManager.AssetOperationParams) (contracts/core/positions/NFTPositionManager.sol#72-76) sends eth to arbitrary user
        Dangerous calls:
        - weth.deposit{value: params.amount}() (contracts/core/positions/NFTPositionManager.sol#74)
NFTPositionManager.repayETH(INFTPositionManager.AssetOperationParams) (contracts/core/positions/NFTPositionManager.sol#119-124) sends eth to arbitrary user
        Dangerous calls:
        - weth.deposit{value: params.amount}() (contracts/core/positions/NFTPositionManager.sol#121)
NFTPositionManager.sweep(address) (contracts/core/positions/NFTPositionManager.sol#127-135) sends eth to arbitrary user
        Dangerous calls:
        - address(msg.sender).transfer(bal) (contracts/core/positions/NFTPositionManager.sol#130)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
INFO:Detectors:
MathUpgradeable.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#55-134) has bitwise-xor operator ^ instead of the exponentiation operator **:
        - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#116)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
NFTRewardsDistributor.getReward(uint256,bytes32) (contracts/core/positions/NFTRewardsDistributor.sol#76-84) ignores return value by rewardsToken.transfer(ownerOf(tokenId),reward) (contracts/core/positions/NFTRewardsDistributor.sol#81)
NFTRewardsDistributor.notifyRewardAmount(uint256,address,address,bool) (contracts/core/positions/NFTRewardsDistributor.sol#113-137) ignores return value by rewardsToken.transferFrom(msg.sender,address(this),reward) (contracts/core/positions/NFTRewardsDistributor.sol#114)
NFTPositionManager.sweep(address) (contracts/core/positions/NFTPositionManager.sol#127-135) ignores return value by erc20.transfer(msg.sender,erc20.balanceOf(address(this))) (contracts/core/positions/NFTPositionManager.sol#133)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
```

NFTPositionManagerGetters.sol

```
NFTPositionManagerStorage._positions (contracts/core/positions/NFTPositionManagerStorage.sol#33) is never initialized. It is used in:
        - NFTPositionManagerGetters.positions(uint256) (contracts/core/positions/NFTPositionManagerGetters.sol#22-24)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables
```

CuratedVault.sol

```
INFO:Detectors:
MathUpgradeable.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#55-134) has bitwise-xor operator ^ instead of the exponentiation operator **:
        - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#116)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
```

CuratedVaultGetters.sol

```
INFO:Detectors:
MathUpgradeable.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#55-134) has bitwise-xor operator ^ instead of the exponentiation operator **:
        - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#116)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
CuratedVaultStorage.DECIMALS_OFFSET (contracts/core/vaults/CuratedVaultStorage.sol#33) is never initialized. It is used in:
        - CuratedVaultBase._decimalsOffset() (contracts/core/vaults/CuratedVaultBase.sol#28-30)
CuratedVaultStorage.config (contracts/core/vaults/CuratedVaultStorage.sol#35) is never initialized. It is used in:
        - CuratedVaultGetters._maxDeposit() (contracts/core/vaults/CuratedVaultGetters.sol#131-145)
CuratedVaultStorage.fee (contracts/core/vaults/CuratedVaultStorage.sol#45) is never initialized. It is used in:
        - CuratedVaultGetters._accruedFeeShares() (contracts/core/vaults/CuratedVaultGetters.sol#185-196)
CuratedVaultStorage.supplyQueue (contracts/core/vaults/CuratedVaultStorage.sol#54) is never initialized. It is used in:
        - CuratedVaultGetters.supplyQueueLength() (contracts/core/vaults/CuratedVaultGetters.sol#38-40)
        - CuratedVaultGetters._maxDeposit() (contracts/core/vaults/CuratedVaultGetters.sol#131-145)
CuratedVaultStorage.withdrawQueue (contracts/core/vaults/CuratedVaultStorage.sol#57) is never initialized. It is used in:
        - CuratedVaultGetters.withdrawQueueLength() (contracts/core/vaults/CuratedVaultGetters.sol#43-45)
        - CuratedVaultGetters._simulateWithdraw(uint256) (contracts/core/vaults/CuratedVaultGetters.sol#92-115)
CuratedVaultStorage.lastTotalAssets (contracts/core/vaults/CuratedVaultStorage.sol#60) is never initialized. It is used in:
        - CuratedVaultGetters._accruedFeeShares() (contracts/core/vaults/CuratedVaultGetters.sol#185-196)
CuratedVaultStorage.positionId (contracts/core/vaults/CuratedVaultStorage.sol#63) is never initialized. It is used in:
        - CuratedVaultGetters._simulateWithdraw(uint256) (contracts/core/vaults/CuratedVaultGetters.sol#92-115)
        - CuratedVaultGetters._maxDeposit() (contracts/core/vaults/CuratedVaultGetters.sol#131-145)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables
```

CuratedVaultSetters.sol

```
INFO:Detectors:
MathUpgradeable.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#55-134) has bitwise-xor operator ^ instead of the exponentiation operator **:
        - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#116)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
CuratedVaultStorage.DECIMALS_OFFSET (contracts/core/vaults/CuratedVaultStorage.sol#33) is never initialized. It is used in:
        - CuratedVaultBase._decimalsOffset() (contracts/core/vaults/CuratedVaultBase.sol#28-30)
CuratedVaultStorage.fee (contracts/core/vaults/CuratedVaultStorage.sol#45) is never initialized. It is used in:
        - CuratedVaultGetters._accruedFeeShares() (contracts/core/vaults/CuratedVaultGetters.sol#185-196)
CuratedVaultStorage.feeRecipient (contracts/core/vaults/CuratedVaultStorage.sol#48) is never initialized. It is used in:
        - CuratedVaultSetters._accrueFee() (contracts/core/vaults/CuratedVaultSetters.sol#183-188)
CuratedVaultStorage.supplyQueue (contracts/core/vaults/CuratedVaultStorage.sol#54) is never initialized. It is used in:
        - CuratedVaultGetters.supplyQueueLength() (contracts/core/vaults/CuratedVaultGetters.sol#38-40)
        - CuratedVaultGetters._maxDeposit() (contracts/core/vaults/CuratedVaultGetters.sol#131-145)
        - CuratedVaultSetters._supplyPool(uint256) (contracts/core/vaults/CuratedVaultSetters.sol#117-150)
CuratedVaultStorage.positionId (contracts/core/vaults/CuratedVaultStorage.sol#63) is never initialized. It is used in:
        - CuratedVaultGetters._simulateWithdraw(uint256) (contracts/core/vaults/CuratedVaultGetters.sol#92-115)
        - CuratedVaultGetters._maxDeposit() (contracts/core/vaults/CuratedVaultGetters.sol#131-145)
        - CuratedVaultSetters._accruedSupplyBalance(IPool) (contracts/core/vaults/CuratedVaultSetters.sol#62-71)
        - CuratedVaultSetters._setCap(IPool,uint184) (contracts/core/vaults/CuratedVaultSetters.sol#87-112)
        - CuratedVaultSetters._supplyPool(uint256) (contracts/core/vaults/CuratedVaultSetters.sol#117-150)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables
```

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.