SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Seismic
Prepared by:	Sherlock
Lead Security Expert:	mstpr-brainbot
Dates Audited:	March 22 - March 25, 2024
Prepared on:	May 10, 2024

Introduction

Seismic is a money market on blast that focuses on being the most competitive place for the biggest assets on chain.

Scope

Repository: seismic-finance/seismic-protocol-v2

Branch: master

Commit: 13070b9ec891d9e7c5fb7450adfb3e61bc58d199

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

Medium	High
2	3

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

mstpr-brainbot	0xdeadbeef
KupiaSec	nirohgo

Issue H-1: Index logic is flawed

Source:
https://github.com/sherlock-audit/2024-03-seismic-finance-judging/issues/3

Found by

KupiaSec, mstpr-brainbot

Summary

When indexes are updated, on top of the interest rate updates the rebasing rewards will also be added. However, the handling of it is wrong.

Vulnerability Detail

AAVE V2 liquidityIndex is calculated as: newLiquidityIndex = previousLiquidityIndex * (total interest accrued since last time + 1) in RAY units.

This is how the (total interest accrued since last time) is calculated:

```
function calculateLinearInterest(uint256 rate, uint40 lastUpdateTimestamp)
    internal
    view
    returns (uint256)
  {
    //solium-disable-next-line
    uint256 timeDifference = block.timestamp.sub(uint256(lastUpdateTimestamp));

    return (rate.mul(timeDifference) / SECONDS_PER_YEAR).add(WadRayMath.ray());
  }
```

The crucial aspect here is the addition of 1e27 to the value. The return of the `calculateLinearInterest` function is guaranteed to be a number greater than 1e27. Therefore, multiplying the `previousIndex` by the new value will also ensure that the result is in 1e27 decimals.

Regarding the addition lines that Seismic adds to include rebasing rewards in the index:

```
if (claimableAmount > 0) {
      uint256 totalPoolHoldings =
↪   IERC20(underlyingAsset).balanceOf(aTokenAddress) + // pool liquidity
        IERC20(reserve.stableDebtTokenAddress).totalSupply() + // total stable
↪   debt
```

```
        IERC20(reserve.variableDebtTokenAddress).totalSupply(); // total
↪   variable debt

      // express claimable amount as a percentage of pool assets and convert
↪   from wad to ray
      uint256 claimedInterestIndex =
↪   claimableAmount.wadDiv(totalPoolHoldings).wadToRay();

      // update pool liquidity index to reflect accrued native
      newLiquidityIndex = claimedInterestIndex.rayMul(newLiquidityIndex);
      reserve.liquidityIndex = uint128(newLiquidityIndex);
      require(newLiquidityIndex <= type(uint128).max,
↪   Errors.RL_LIQUIDITY_INDEX_OVERFLOW);

      // claim and send yield to the aToken
      IERC20Rebasing(underlyingAsset).claim(address(aTokenAddress),
↪   claimableAmount);
    }

    //solium-disable-next-line
    reserve.lastUpdateTimestamp = uint40(block.timestamp);
```

now, let's assume the claimable rewards are 1e18 and there are 100e18 tokens in total at aToken/vTokens. claimableAmount = 1e18 totalPoolHoldings = 100e18 claimedInterestIndex = (1e18 * 1e18 / 100 * 1e18) * 1e9 = 10000000000000000000000000 = 0.01 * 1e27 **the claimedInterestIndex is a value way lesser than 1e27 !** newLiquidityIndex = 0.01 * 1e27 * 1e27 / 1e27 = 0.01 * 1e27 **new index is a value way lesser than 1e27!**

The default value for both the `liquidityIndex` and `variableBorrowIndex` is 1e27 and should NEVER go below 1e27. However, as observed in the above example, it is very likely to occur. Even worse, when the claimable amount is too small, the `claimedInterestIndex` can be "0", resulting in the new liquidity index being "0", which would cause chaos for all users and prevent them from withdrawing, depositing, or borrowing.

Impact

Code Snippet

https://github.com/sherlock-audit/2024-03-seismic-finance/blob/main/seismic-protocol-v2/contracts/protocol/libraries/logic/ReserveLogic.sol#L340-L412

Tool used

Manual Review

Recommendation

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/seismic-finance/seismic-protocol-v2/pull/23

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-2: Borrow index isn't updated correctly

Source:
https://github.com/sherlock-audit/2024-03-seismic-finance-judging/issues/4

Found by

mstpr-brainbot

Summary

Rebasing tokens are not supported for both collateral and borrowable assets in AAVE V2. stETH is currently the only rebasing token supported by AAVE V2, and it has specific implementations. Additionally, stETH cannot be borrowed; it can only be used as collateral. Previously, AMPL was the only rebasing token that was both supported as collateral and borrowable, but it is no longer supported. AMPL had different implementations compared to other generic Aave contracts like AToken and DebtToken. The same applies to Seismic's USDB and WETH, which are rebasing tokens.

https://docs.aave.com/developers/v/2.0/guides/ampl-asset-listing https://etherscan.io/address/0xbd233d4ffdaa9b7d1d3e6b18cccb8d091142893a#code (steth atoken specific implementation)

Vulnerability Detail

The rebasing rewards are reflected to the `liquidityIndex` as follows:

```
// claimableAmount always has 18 decimals, since both USDB and WETH have 18
↪ decimals
    uint256 claimableAmount = (underlyingAsset == USDB || underlyingAsset ==
↪ WETH)
      ? IERC20Rebasing(underlyingAsset).getClaimableAmount(address(this))
      : 0;

    // only accrue native yield if there is something to be claimed
    if (claimableAmount > 0) {
      uint256 totalPoolHoldings =
↪ IERC20(underlyingAsset).balanceOf(aTokenAddress) + // pool liquidity
        IERC20(reserve.stableDebtTokenAddress).totalSupply() + // total stable
↪ debt
        IERC20(reserve.variableDebtTokenAddress).totalSupply(); // total
↪ variable debt

      // express claimable amount as a percentage of pool assets and convert
↪ from wad to ray
```

```
      uint256 claimedInterestIndex =
↪   claimableAmount.wadDiv(totalPoolHoldings).wadToRay();

      // update pool liquidity index to reflect accrued native
      newLiquidityIndex = claimedInterestIndex.rayMul(newLiquidityIndex);
      reserve.liquidityIndex = uint128(newLiquidityIndex);
      require(newLiquidityIndex <= type(uint128).max,
↪   Errors.RL_LIQUIDITY_INDEX_OVERFLOW);

      // claim and send yield to the aToken
      IERC20Rebasing(underlyingAsset).claim(address(aTokenAddress),
↪   claimableAmount);
    }
```

However, the borrowIndex is not updated as accordingly. The `liquidityIndex` updates assuming the rebase rewards will be back by borrowers. However, the borrowers are not encouraged to do it since the borrowIndex is NOT updated.

```
if (scaledVariableDebt != 0) {
      uint256 cumulatedVariableBorrowInterest =
↪   MathUtils.calculateCompoundedInterest(
          reserve.currentVariableBorrowRate,
          timestamp
      );
      newVariableBorrowIndex =
↪   cumulatedVariableBorrowInterest.rayMul(variableBorrowIndex);
      require(
          newVariableBorrowIndex <= type(uint128).max,
          Errors.RL_VARIABLE_BORROW_INDEX_OVERFLOW
      );
      reserve.variableBorrowIndex = uint128(newVariableBorrowIndex);
    }
  }
```

`liquidityIndex` will be way ahead of the `borrowIndex`, which will mean that the supplier will assume that there are enough funds returned by the borrowers. However, borrowers are not entitled to the extra borrowing APY that rebasing causes.

Additional read: https://ethereum.stackexchange.com/questions/154265/why-are-the-aave-v2-supply-and-borrow-index-rate-calculations-done-differently

Impact

Insolvency.

Code Snippet

https://github.com/sherlock-audit/2024-03-seismic-finance/blob/main/seismic-protocol-v2/contracts/protocol/libraries/logic/ReserveLogic.sol#L340-L412

Tool used

Manual Review

Recommendation

Use NrETH, NrUSDB, the wrapped versions of rebasing tokens like wstETH if the rebasing points accounting works. That way the AAVE V2 code can be used without adding any extra code.

I think rebasing tokens can't work properly with AAVE V2/V3.

Discussion

nevillehuang

I believe not being entitled to additional borrowing APY from rebasing logic constrained to yield accrued of USDB/WETH is medium severity and does not cause serious enough non-material losses

mstpr

Escalate

The issue impact is more serious than just a lesser borrowing APY. Suppliers will assume that the rebasing yield of 15% is always accruing regardless of the funds are borrowed or not. If the funds are borrowed with lesser apy which will be the case always, suppliers aTokens will not be 1:1 redeemable in future because the supply interest index assumes the yield is back but the borrowers are not forced to give it back. In result, the borrowers will be able to borrow a token that rebasing 15% with a borrowing apy of lower than 15%.

For example: There are 100 token supplied and 10 token borrowed. The supply APY is rebasing yield + the supply apy in market activity. However, borrowing apy is just the market activity, so it is something around 2% since it is 10% utilized market. Borrowers who borrows USDB/WETH does not need to return the rebasing yield. This means that the protocol will accrue bad debt every second and potentially opens up other attack vectors that I dont think of as of now. Though, that means the borrowers can **steal** the rebasing yield of supplied funds.

So at future someone holding 100 aToken might not redeem it because the funds are not going to be returned back by the borrowers. The supplier will **lose** the rebasing yield of USDB/WETH and will only entitled to market supply rate.

I think this is a core logic broken and is a very serious bug. I think this deserves a high severity.

sherlock-admin2

> Escalate
>
> The issue impact is more serious than just a lesser borrowing APY. Suppliers will assume that the rebasing yield of 15% is always accruing regardless of the funds are borrowed or not. If the funds are borrowed with lesser apy which will be the case always, suppliers aTokens will not be 1:1 redeemable in future because the supply interest index assumes the yield is back but the borrowers are not forced to give it back. In result, the borrowers will be able to borrow a token that rebasing 15% with a borrowing apy of lower than 15%.
>
> For example: There are 100 token supplied and 10 token borrowed. The supply APY is rebasing yield + the supply apy in market activity. However, borrowing apy is just the market activity, so it is something around 2% since it is 10% utilized market. Borrowers who borrows USDB/WETH does not need to return the rebasing yield. This means that the protocol will accrue bad debt every second and potentially opens up other attack vectors that I dont think of as of now. Though, that means the borrowers can **steal** the rebasing yield of supplied funds.
>
> So at future someone holding 100 aToken might not redeem it because the funds are not going to be returned back by the borrowers. The supplier will **lose** the rebasing yield of USDB/WETH and will only entitled to market supply rate.
>
> I think this is a core logic broken and is a very serious bug. I think this deserves a high severity.

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.

KupiaSecAdmin

https://docs.blast.io/user-incentives

Based on the Blast docs, the APY is like ~4% for ETH/WETH and 5% for USDB. Couldn't be like 15% I think.

Czar102

@mstpr @nevillehuang Given that borrowers could extract more value, wouldn't they borrow more and that would increase the interest rates to valid levels?

nevillehuang

@Czar102 I don't think borrowing more solves the problem of the rebasing yield creating a gap between `borrowIndex` and `liquidityIndex`, given `borrowIndex` remains unchanged and not including the potential rebasing yield. @mstpr Is my understanding correct?

mstpr

> @Czar102 I don't think borrowing more solves the problem of the rebasing yield creating a gap between `borrowIndex` and `liquidityIndex`, given `borrowIndex` remains unchanged and not including the potential rebasing yield. @mstpr Is my understanding correct?

exactly. There can be many things crafted from this root cause.

First, we need a utilization level such that the borrowing apy > supply apy + yield apy. This can be achieved in very high utilized markets, like 95% utilization in default interest rate strategy the borrowing apy would be 65% and supply apy would be 50%. 50% + 15% would sum up to 65%. Though that means the market has to be extremely utilized. That means users who depositing tokens should expect to withdraw only 5% of their deposits since the market will never go below the 95% utilization. If it does, then bad debt will accrue. So both sides will be extremely problematic for users and the market in general.

So to wrap up, if market acts rational and borrows, then the market is always highly utilized and suppliers never get back their full funds. If market is not rational then the bad debt will accrue.

Overall, I think this is a very critical issue in a lending borrowing protocol. Also fits the description of "Inflicts serious non-material losses" perfectly imo.

Czar102

@mstpr are you sure you mean bad debt here? I think it's just a net APY loss for lenders, they are effectively paying for others to borrow. But there is no bad debt.

This situation won't occur if the markets work efficiently in the first place.

@zugdev @0xmegaman @0x-xamurai @cachegrab do you agree with the above?

mstpr

> @mstpr are you sure you mean bad debt here? I think it's just a net APY loss for lenders, they are effectively paying for others to borrow. But there is no bad debt.
>
> This situation won't occur if the markets work efficiently in the first place.
>
> @zugdev @0xmegaman @0x-xamurai @cachegrab do you agree with the above?

You are right, no "bad debt" just loss of yield for lenders. Though, up to some utilization the borrowing is free. So this market would be always extremely utilized.

Czar102

> Though, up to some utilization the borrowing is free. So this market would be always extremely utilized.

That can be fixed with an interest model parameters change if the rebasing APY is predictable, right?

Evert0x

My current understanding is that the issue needs to be high severity as it's a definite loss of funds (loss of yield) for lenders in the protocol. Although the risk can be mitigated by setting specific parameters in the interest model. I believe it results in loss in normal situations.

Also, these rebasing tokens are the only intended tokens for his protocol.

Planning to accept validation and make severity high

nevillehuang

Hi @Evert0x I believe this should be medium severity, because

- In the **original** submission, the impact wasn't quantified. As per comment here, the APY is 4% and 5% respectively for ETH and USDB respectively. Note that this is APY spread out over a year, so the actual impact is even more constraint.
- Risk can be reduced by specific interest rate model parameters

Evert0x

But the lender loses out on the full 4%/5%, correct? That's a really serious issue in the protocol.

@mstpr or someone else, do you have more information about the interest rate model parameters required?

mstpr

> But the lender loses out on the full 4%/5%, correct? That's a really serious issue in the protocol.
>
> @mstpr or someone else, do you have more information about the interest rate model parameters required?

If the market is rational, then the utilization will always be high. Currently, USD has a 15% APY since it deposits to sDAI and to achieve an APY such that borrowing is feasible considering the supply+yield APY with the current interest model utilization has to be very high, like 95%.

So, if the market is rational, then the market will always be 95% utilized and no one will be able to withdraw their full aToken amount. If the market is irrational, then borrowing up to 95% will be free since the borrowing APY will be lower than the supply+yield APY up to some very high utilization. In such a scenario, lenders are losing the potential yield from rebasing.

Overall, if the market is rational, then funds are locked. If the market is irrational, then yield is lost.

KupiaSecAdmin

I think the issue still should be Low/Invalid. Here's my thoughts:

[Information]

1. Blast provides ~4% APY on ETH/WETH and ~5% on USDB.
2. Interest parameters can be controlled for borrowing rate.

[Assumption] The total collateral: T Utilization rate(borrow / collateral): U Supply APY: SPY Borrow APY: BPY The amount a lender deposited: L The amount a borrower borrowed: D Blast yield rate: R(~4-5%)

[How much do lenders earn from Seismic with ETH/USDB] By lending on Seismic, lenders earn from interest rate from borrowers + Blast yield from tokens sitting in the pool. If they didn't lend on Seismic, they will earn only from Blast yield.

Thus the actual amount a lender earns is: `InterestRate + MarketBlastYield - PersonalBlastYield = L * SPY + T * (1 - U) * R * L / T - L * R = L * SPY - L * U * R = L * (SPY - R* U)`

[How much do borrowers pay to Seismic with ETH/USDB] By borrowing from Seismic, borrowers pay the interest rate, and with borrowed tokens, they earn APY from Blast. Thus the actual amount a borrower needs to pay is: `D * BPY - D * R = D * (BPY - R)`

1. When utilization is low, the reporter claims that the borrowers don't need to return back their assets because BPY < R. This assumption is wrong because, the borrower need to deposit more amount as collateral to borrow something, and if they don't borrow but own ETH/USDB with collateral directly, they earn more than they deposit into Seismic, thus it becomes less incentive.
2. When utilization is high, the yield that lenders earn becomes smaller because it depends on utilization U as shown above. However, if utilization goes high, BPY and SPY becomes bigger as well, thus it can totally cover R * U < 4%.

Thus, without even controlling market's interest rate params, the amount that lenders earn always exceeds the yield from the Blast, thus it can be considered as no-problem.

mstpr

> I think the issue still should be Low/Invalid. Here's my thoughts:
>
> [Information]
>
> 1. Blast provides ~4% APY on ETH/WETH and ~5% on USDB.
> 2. Interest parameters can be controlled for borrowing rate.
>
> [Assumption] The total collateral: T Utilization rate(borrow / collateral): U Supply APY: SPY Borrow APY: BPY The amount a lender deposited: L The amount a borrower borrowed: D Blast yield rate: R(~4-5%)
>
> [How much do lenders earn from Seismic with ETH/USDB] By lending on Seismic, lenders earn from interest rate from borrowers + Blast yield from tokens sitting in the pool. If they didn't lend on Seismic, they will earn only from Blast yield.
>
> Thus the actual amount a lender earns is: `InterestRate + MarketBlastYield - PersonalBlastYield = L * SPY + T * (1 - U) * R * L / T - L * R = L * SPY - L * U * R = L * (SPY - R* U)`
>
> [How much do borrowers pay to Seismic with ETH/USDB] By borrowing from Seismic, borrowers pay the interest rate, and with borrowed tokens, they earn APY from Blast. Thus the actual amount a borrower needs to pay is: `D * BPY - D * R = D * (BPY - R)`
>
> 1. When utilization is low, the reporter claims that the borrowers don't need to return back their assets because BPY < R. This assumption is wrong because, the borrower need to deposit more amount as collateral to borrow something, and if they don't borrow but own ETH/USDB with collateral directly, they earn more than they deposit into Seismic, thus it becomes less incentive.
> 2. When utilization is high, the yield that lenders earn becomes smaller because it depends on utilization U as shown above. However, if utilization goes high, BPY and SPY becomes bigger as well, thus it can totally cover R * U < 4%.
>
> Thus, without even controlling market's interest rate params, the amount that lenders earn always exceeds the yield from the Blast, thus it can be considered as no-problem.

"By borrowing from Seismic, borrowers pay the interest rate, and with borrowed tokens, they earn APY from Blast." correct, the borrowing apy is lesser than the rebasing apy up to very high utilization which what this issue tries to point.

If market is rational utilization will always be high and you will never be able to withdraw all your aToken balances. If not, then the borrowers borrows free money and lenders lose out on the rebasing yield.

AAVE strictly forbids rebasing tokens on borrows. This is why stETH is only used as collateral. This is why AMPL token had a different AToken implementation.

KupiaSecAdmin

@mstpr - Here's my opinion:

1. When utilization rate is high, the borrow APY will be like 15% or 20% based on parameters, in this case borrowers must return their tokens back to avoid liquidations.
2. When utilization rate is low < 5%, you said borrowers earn free money. This is not true because to borrow assets, borrowers need collateral. If they buy ETH or USDB directly with the collateral, they earn more than from borrowing. Thus it's not free money, it's losing money for borrowers.

Evert0x

Thanks for the discussion. Although the risk can be mitigated by certain parameters, I believe High severity is appropriate as it breaks a core mechanism in the protocol which leads to inferior yields (as described in the original report). Having competitive yields is of upmost importance to a lending protocol.

Planning to accept escalation and assign High severity

KupiaSecAdmin

@Evert0x - Based on Sherlock's issue validity criteria, I think it deserves Medium. Because:

In V.1 it is mentioned that the issue with certain external conditions or specific states is considered Medium. Regarding this issue, the loss can only happens with inappropriate market params and utilization rate is high.

Evert0x

Result: High Unique

Comment above doesn't provide new information to change the escalation outcome

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- mstpr: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/seismic-finance/seismic-protocol-v2/pull/23

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-3: Claiming rebasing token rewards in ReserveLogic is done from the wrong account (LendingPool instead of AToken) and will fail on mainnet

Source:
https://github.com/sherlock-audit/2024-03-seismic-finance-judging/issues/23

Found by

0xdeadbeef, KupiaSec, mstpr-brainbot, nirohgo

Summary

The code added to ReserveLogic.sol::_updateIndexes in order to claim rebasing interest, uses address(this) instead of the Atoken address for getClaimableAmount and calls claim directly instead of through the Atoken, both of which are wrong and will cause the function to fail.

Vulnerability Detail

The following code snippet was added to ReserveLogic.sol::_updateIndexes with the intention of claiming any rebasing interest accrued in rebasing underlying tokens, and add it to the relevant AToken as additional LP profit:

```
 // claimableAmount always has 18 decimals, since both USDB and WETH have 18
↪   decimals
uint256 claimableAmount = (underlyingAsset == USDB || underlyingAsset == WETH)
    ? IERC20Rebasing(underlyingAsset).getClaimableAmount(address(this))
    : 0;

// only accrue native yield if there is something to be claimed
if (claimableAmount > 0) {
    uint256 totalPoolHoldings = IERC20(underlyingAsset).balanceOf(aTokenAddress)
↪   + // pool liquidity
    IERC20(reserve.stableDebtTokenAddress).totalSupply() + // total stable debt
    IERC20(reserve.variableDebtTokenAddress).totalSupply(); // total variable
↪   debt

    // express claimable amount as a percentage of pool assets and convert from
↪   wad to ray
    uint256 claimedInterestIndex =
↪   claimableAmount.wadDiv(totalPoolHoldings).wadToRay();

    // update pool liquidity index to reflect accrued native
```

SHERLOCK

```
    newLiquidityIndex = claimedInterestIndex.rayMul(newLiquidityIndex);
    reserve.liquidityIndex = uint128(newLiquidityIndex);
    require(newLiquidityIndex <= type(uint128).max,
↪   Errors.RL_LIQUIDITY_INDEX_OVERFLOW);

    // claim and send yield to the aToken
    IERC20Rebasing(underlyingAsset).claim(address(aTokenAddress),
↪   claimableAmount);
    }
```

The root cause of the issue is that the code uses address(this) (the LendingPool in this context) instead of the AToken address which is the real owner of the rebalancing token liquidity. This happens in two places:

1. The getClaimableAmount function is called with address(this) (the LendingPool address in this context) instead of the AToken which is the real owner of the underlying asset (and that is configured as CLAIMABLE). On mainnet the function would fail because getClaimableAmount fails when called on an account that is not configured as CLAIMABLE (which is the case for the LendingPool address) as seen here (from the WETH ERC20Rebasing source code on the Blast mainnet explorer):

```
function getClaimableAmount(address account) public view returns (uint256) {
    if (getConfiguration(account) != YieldMode.CLAIMABLE) {
        revert NotClaimableAccount();
    }

    uint256 shareValue = _computeShareValue(sharePrice(), _shares[account],
↪   _remainders[account]);
    return shareValue - _fixed[account];
}
```

2. The function calls the rebasing token's claim function directly (with LendingPool as the msg.sender), with the AToken address as the recipient, again wrongfully because the LendingPool contract doesn't hold the underlying balance (the AToken does).

Note that another problem currently shadows this issue on Sepolia: Currently the WETH and USDB addresses are wrongfully configured in ReserveLogic (supposedly to sepolia addresses but there is an error in the addresses):

```
address constant USDB = 0x4300000000000000000000000000000000000022;
address constant WETH = 0x4300000000000000000000000000000000000023;
```

Compare with the correct sepolia addresses as configured in the AToken.sol contract:

```
IERC20Rebasing public constant USDB =
↪   IERC20Rebasing(0x4200000000000000000000000000000000000022);
IERC20Rebasing public constant WETH =
↪   IERC20Rebasing(0x4200000000000000000000000000000000000023);
```

The POC below shows how, if the addresses are corrected in ReserveLogic, the function fails with a call that requires a status update on a rebalancing reserve token.

POC

how to run (The first three steps can be done once and apply to all my POCs in this contest)

1. Run `forge init` in a new folder
2. Run `forge install openzeppelin/openzeppelin-contracts@v3.1.0 --no-commit`
3. Copy everything under seismic-protocol-v2\contracts to the src folder
4. Create a test.sol file under the test folder and copy the code below to it.
5. Run `forge test --match-test testClaimProblemPOC --fork-url https://sepolia.blast.io -vvv`
6. change token addresses in ReserveLogic.sol line 30 to:

```
//address constant USDB = 0x4200000000000000000000000000000000000022;
//address constant WETH = 0x4200000000000000000000000000000000000023;
```

5. Run `forge test --match-test testClaimProblemPOC --fork-url https://sepolia.blast.io -vvv` again to see the revert.

Test.sol Code:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.6.12;
pragma experimental ABIEncoderV2;

import {Test, console2} from "forge-std/Test.sol";
import {IERC20} from '../src/dependencies/openzeppelin/contracts/IERC20.sol';
import {IERC20Detailed} from
↪   '../src/dependencies/openzeppelin/contracts/IERC20Detailed.sol';

import {LendingPool} from "../src/protocol/lendingpool/LendingPool.sol";
import {ILendingPoolAddressesProvider} from
↪   "../src/interfaces/ILendingPoolAddressesProvider.sol";
```

```
import {ILendingRateOracle} from "../src/interfaces/ILendingRateOracle.sol";
import {IPriceOracleGetter} from "../src/interfaces/IPriceOracleGetter.sol";

import {LendingPoolConfigurator} from
↪  "../src/protocol/lendingpool/LendingPoolConfigurator.sol";
import "../src/protocol/libraries/types/DataTypes.sol";
import {LendingPoolCollateralManager} from
↪  "../src/protocol/lendingpool/LendingPoolCollateralManager.sol";
import {LendingPoolAddressesProvider} from
↪  "../src/protocol/configuration/LendingPoolAddressesProvider.sol";
import {AaveProtocolDataProvider} from
↪  "../src/misc/AaveProtocolDataProvider.sol";
import {AaveOracle} from "../src/misc/AaveOracle.sol";

import {IERC20Rebasing, YieldMode} from
↪  '../src/misc/interfaces/IERC20Rebasing.sol';
import {IBlastPoints} from '../src/misc/interfaces/IBlastPoints.sol';
import {IWETH} from '../src/misc/interfaces/IWETH.sol';
import {ILendingPoolConfigurator} from
↪  "../src/interfaces/ILendingPoolConfigurator.sol";
import {AToken} from "../src/protocol/tokenization/AToken.sol";
import {StableDebtToken} from "../src/protocol/tokenization/StableDebtToken.sol";
import {VariableDebtToken} from
↪  "../src/protocol/tokenization/VariableDebtToken.sol";
import {DefaultReserveInterestRateStrategy} from
↪  "../src/protocol/lendingpool/DefaultReserveInterestRateStrategy.sol";
import {LendingRateOracle} from "../src/mocks/oracle/LendingRateOracle.sol";




contract AaveTest is Test {
     address public user = makeAddr("user");
     address public user1 = makeAddr("user1");
     address public pool_admin = makeAddr("poolAdmin");
     address public pool_emerg_admin = makeAddr("pool_emerg_admin");
     address public treasury_usdb =  makeAddr("treasury_usdb");
     address public treasury_weth =  makeAddr("treasury_weth");

     address public PYTH_CONTRACT = 0xA2aa501b19aff244D90cc15a4Cf739D2725B5729;
     bytes32 PYTH_USDB_FEED_ID =
↪  0x433faaa801ecdb6618e3897177a118b273a8e18cc3ff545aadfc207d58d028f7;

     address POINTS_OPERATOR = 0xc783df8a850f42e7F7e57013759C285caa701eB6;
     address public usdb_holder = 0x3df9C3E7105B2bEdd0b7b75c9ECF5C9041313186;
```

```
    IERC20Rebasing public USDB =
↪  IERC20Rebasing(0x4200000000000000000000000000000000000022);
    IERC20Rebasing public WETH =
↪  IERC20Rebasing(0x4200000000000000000000000000000000000023);
    string public marketId = "testMarket";

    LendingPool public lending_pool;
    LendingPoolAddressesProvider public address_provider;
    LendingPoolConfigurator public pool_configurator;
    LendingPoolCollateralManager public pool_collateral_manager;
    AaveOracle public price_oracle;
    ILendingRateOracle public lendingRateOracle;
    AaveProtocolDataProvider public aave_protocol_data_provider;

    DefaultReserveInterestRateStrategy public defualt_strategy;

    function setUp() public virtual {

        address_provider = new LendingPoolAddressesProvider(marketId);

        //init lending pool
        LendingPool lending_pool_impl = new LendingPool();
        address_provider.setLendingPoolImpl(address(lending_pool_impl));
        lending_pool = LendingPool(address_provider.getLendingPool());


        //init lending pool configurator
        LendingPoolConfigurator pool_configurator_impl = new
↪  LendingPoolConfigurator();
        address_provider.setLendingPoolConfiguratorImpl(address(pool_configura⌋
↪  tor_impl));
        pool_configurator =
↪  LendingPoolConfigurator(address_provider.getLendingPoolConfigurator());

        //init LendingPoolCollateralManager
        LendingPoolCollateralManager pool_collateral_mamager_impl = new
↪  LendingPoolCollateralManager();
        address_provider.setLendingPoolCollateralManager(address(pool_collater⌋
↪  al_mamager_impl));
        pool_collateral_manager = LendingPoolCollateralManager(address_provide⌋
↪  r.getLendingPoolCollateralManager());

        //init Pool Oracle
        address[] memory assets = new address[](1);
        assets[0] = address(USDB);
        bytes32[] memory feedids = new bytes32[](1);
        feedids[0] = PYTH_USDB_FEED_ID;
```

SHERLOCK

```
        price_oracle = new AaveOracle(PYTH_CONTRACT, assets,feedids);
        address_provider.setPriceOracle(address(price_oracle));


        //init lendingRateOracle
        lendingRateOracle = new LendingRateOracle();
        address_provider.setLendingRateOracle(address(lendingRateOracle));
       lendingRateOracle.setMarketBorrowRate(address(WETH), 0.02*1e27);
       lendingRateOracle.setMarketBorrowRate(address(USDB), 0.02*1e27);

        address_provider.setPoolAdmin(pool_admin);
        address_provider.setEmergencyAdmin(pool_emerg_admin);
        address_provider.setMarketId(marketId);


        //init data provider
        aave_protocol_data_provider = new
↪   AaveProtocolDataProvider(ILendingPoolAddressesProvider(address_provider));

        defualt_strategy = new
↪   DefaultReserveInterestRateStrategy(address_provider,
             0.725*1e27,
             0.02*1e27,
             0.05*1e27,
             0.9*1e27,
             0.07*1e27,
             0.9*1e27);
        //add tokens as reserves
        InitToken(address(USDB),treasury_usdb,
             address(defualt_strategy),
             "aave USDB",
             "aUSDB",
             "aave USDB Var",
             "aUSDBVar",
             "aave USDB Stable",
             "aUSDBSta");
         InitToken(address(WETH),treasury_weth,
             address(defualt_strategy),
             "aave WETH",
             "aWETH",
             "aave WETH Var",
             "aWETHVar",
             "aave WETH Stable",
             "aWETHSta");
    }
    function testClaimProblemPOC() public {

        //deposit some WETH to the pool
        vm.startPrank(user);
        deal(user,100e18);
        IWETH(address(WETH)).deposit{value: 100e18}();
        IERC20(address(WETH)).approve(address(lending_pool),100e18);
        lending_pool.deposit(address(WETH),5e18,user,0);
        (address aWeth,,) =
↪   aave_protocol_data_provider.getReserveTokensAddresses(address(WETH));
        printRebalancingBals("AToken Weth balance and claimable at
↪   start:",WETH,aWeth);
        //OUTPUT
        //Balance 5000000000000000000
        //Claimable: 0

        //workaround to simulate 10% accrued interest to WETH holders (since
↪   warping time won't do it)
        deal(address(WETH),address(WETH).balance * 110 / 100);
        printRebalancingBals("AToken Weth balance and claimable after interest
↪   accrual",WETH,aWeth);
        //OUTPUT
        //Balance 5000000000000000000
        //Claimable: 613881253978048657

        //another small deposit to trigger a call to _updateIndexes where the
↪   claiming code is
        //Note: after changing the WETH/USDB addresses in ReserveLogic to:
        //address constant USDB = 0x4200000000000000000000000000000000000022;
        //address constant WETH = 0x4200000000000000000000000000000000000023;
        //this call reverts because getClaimableAmount is called with
↪   LendingPool which is not configured as CLAIMABLE
        lending_pool.deposit(address(WETH),0.00000001e18,user,0);
        printRebalancingBals("AToken Weth balance and claimable after pool
↪   claiming:",WETH,aWeth);
        //OUTPUT
        //Balance 5000000010000000000
        //Claimable: 613880470540939226

        vm.stopPrank();
    }

    //prints the current balance and claimable amounts of the given address
    function printRebalancingBals(string memory message, IERC20Rebasing token,
↪   address user) public {
        console2.log(message);
```

SHERLOCK

```
        console2.log("Balance %s",IERC20(address(token)).balanceOf(user));
        console2.log("Claimable: %s\n",token.getClaimableAmount(user));
    }

    function InitToken(address underlying,
            address treasury,
            address strategy,
            string memory atokenName,
             string memory atokenSymbol,
             string memory vatTname,
             string memory varTSymbol,
             string memory stableTname,
             string memory stableTsymbol) public {
        vm.startPrank(pool_admin);
        ILendingPoolConfigurator.InitReserveInput[] memory input = new
↪   ILendingPoolConfigurator.InitReserveInput[](1);

        input[0].aTokenImpl = address(new AToken());
        input[0].stableDebtTokenImpl = address(new StableDebtToken());
        input[0].variableDebtTokenImpl = address(new VariableDebtToken());
        input[0].underlyingAssetDecimals = 18;
        input[0].interestRateStrategyAddress = strategy;
        input[0].underlyingAsset = underlying;
        input[0].treasury = treasury;
        input[0].incentivesController = address(0);
        input[0].underlyingAssetName = "Rebasing USD";
        input[0].aTokenName = atokenName;
        input[0].aTokenSymbol = atokenSymbol;
        input[0].variableDebtTokenName = vatTname;
        input[0].variableDebtTokenSymbol = varTSymbol;
        input[0].stableDebtTokenName =stableTname;
        input[0].stableDebtTokenSymbol = stableTsymbol;
        input[0].pointsOperator =POINTS_OPERATOR;
        input[0].params = "";
        pool_configurator.batchInitReserve(input);
        pool_configurator.activateReserve(underlying);
        pool_configurator.enableBorrowingOnReserve(underlying,true);
        pool_configurator.configureReserveAsCollateral(underlying, 8000, 8250,
↪   10500);
        pool_configurator.setReserveFactor(underlying,1000);
        vm.stopPrank();
    }
}
```

Impact

This error would cause the function to fail whenever is it called for a reserve with a rebasing underlying token, because of the call to getClaimableAmount with an acount that is not configured as YieldMode.CLAIMABLE. If the LendingPool token is configured to YieldMode.CLAIMABLE the function doesn't fail but no claimable amounts are attributed to LPs because the wrong account is being checked.

Code Snippet

https://github.com/sherlock-audit/2024-03-seismic-finance/blob/main/seismic-protocol-v2/contracts/protocol/libraries/logic/ReserveLogic.sol#L391

Tool used

Manual Review Foundry

Recommendation

1. The first issue is easily fixable by calling getClaimableAmount with the relevant AToken address instead of address(this).
2. The second issue can be fixed by adding a function to AToken, callable by LendingPool, that claims the entire claimable amount to the AToken. (since only the AToken can call claim on its own balance).

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/seismic-finance/seismic-protocol-v2/pull/23

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-1: `balanceOf` function is not updated with the re-basing yield

Source:
https://github.com/sherlock-audit/2024-03-seismic-finance-judging/issues/13

Found by

mstpr-brainbot

Summary

The `balanceOf` function calculates the interest that has been paid to the aToken holders using indexes. However, it does not simulate the rebasing token yield.

Vulnerability Detail

As we can see in the `balanceOf` function, the reserved normal income is used to derive the actual value of the tokens:

```
function balanceOf(
    address user
  ) public view override(IncentivizedERC20, IERC20) returns (uint256) {
    return super.balanceOf(user).rayMul(_pool.getReserveNormalizedIncome(_underl⌋
↪   yingAsset));
  }
```

Typically, the index adds the rebasing token yield, as observed here:

```
function _updateIndexes(
    DataTypes.ReserveData storage reserve,
    uint256 scaledVariableDebt,
    uint256 liquidityIndex,
    uint256 variableBorrowIndex,
    uint40 timestamp
  ) internal returns (uint256, uint256) {
    uint256 currentLiquidityRate = reserve.currentLiquidityRate;

    uint256 newLiquidityIndex = liquidityIndex;
    uint256 newVariableBorrowIndex = variableBorrowIndex;

    //only cumulating if there is any income being produced
    if (currentLiquidityRate > 0) {
      uint256 cumulatedLiquidityInterest = MathUtils.calculateLinearInterest(
        currentLiquidityRate,
        timestamp
      );
      newLiquidityIndex = cumulatedLiquidityInterest.rayMul(liquidityIndex);
      require(newLiquidityIndex <= type(uint128).max,
↪   Errors.RL_LIQUIDITY_INDEX_OVERFLOW);

      // moved to end of the function
      reserve.liquidityIndex = uint128(newLiquidityIndex);

      //as the liquidity rate might come only from stable rate loans, we need to
↪   ensure
      //that there is actual variable debt before accumulating
      if (scaledVariableDebt != 0) {
        uint256 cumulatedVariableBorrowInterest =
↪   MathUtils.calculateCompoundedInterest(
          reserve.currentVariableBorrowRate,
          timestamp
        );
        newVariableBorrowIndex =
↪   cumulatedVariableBorrowInterest.rayMul(variableBorrowIndex);
        require(
          newVariableBorrowIndex <= type(uint128).max,
          Errors.RL_VARIABLE_BORROW_INDEX_OVERFLOW
        );
        reserve.variableBorrowIndex = uint128(newVariableBorrowIndex);
      }
    }

    // check for blast native yield if underlying asset is USDB or WETHRebasing
    // if pending, claim yield and accrue it as interest to aToken holders
    address aTokenAddress = reserve.aTokenAddress;
    address underlyingAsset = IAToken(aTokenAddress).UNDERLYING_ASSET_ADDRESS();

    // claimableAmount always has 18 decimals, since both USDB and WETH have 18
↪   decimals
    uint256 claimableAmount = (underlyingAsset == USDB || underlyingAsset ==
↪   WETH)
      ? IERC20Rebasing(underlyingAsset).getClaimableAmount(address(this))
      : 0;

    // only accrue native yield if there is something to be claimed
    if (claimableAmount > 0) {
      uint256 totalPoolHoldings =
↪   IERC20(underlyingAsset).balanceOf(aTokenAddress) + // pool liquidity
        IERC20(reserve.stableDebtTokenAddress).totalSupply() + // total stable
↪   debt
```

```
        IERC20(reserve.variableDebtTokenAddress).totalSupply(); // total
↪   variable debt

      // express claimable amount as a percentage of pool assets and convert
↪   from wad to ray
      uint256 claimedInterestIndex =
↪   claimableAmount.wadDiv(totalPoolHoldings).wadToRay();

      // update pool liquidity index to reflect accrued native
      newLiquidityIndex = claimedInterestIndex.rayMul(newLiquidityIndex);
      reserve.liquidityIndex = uint128(newLiquidityIndex);
      require(newLiquidityIndex <= type(uint128).max,
↪   Errors.RL_LIQUIDITY_INDEX_OVERFLOW);

      // claim and send yield to the aToken
      IERC20Rebasing(underlyingAsset).claim(address(aTokenAddress),
↪   claimableAmount);
    }
```

However, when calculating the view function for reserve normalized income, it does not add the rebasing yield:

```
function getNormalizedIncome(
    DataTypes.ReserveData storage reserve
  ) internal view returns (uint256) {
    uint40 timestamp = reserve.lastUpdateTimestamp;

    //solium-disable-next-line
    if (timestamp == uint40(block.timestamp)) {
      //if the index was updated in the same block, no need to perform any
↪   calculation
      return reserve.liquidityIndex;
    }

    uint256 cumulated = MathUtils
      .calculateLinearInterest(reserve.currentLiquidityRate, timestamp)
      .rayMul(reserve.liquidityIndex);

    return cumulated;
  }
```

Impact

If balanceOf returns different numbers, lot's of functionality will not work such as repaying the entire balance of a user, depositing and withdrawing since the

balanceOf used inside these functions and expected to be same as the actual balance that the user holds. For example:
https://github.com/sherlock-audit/2024-03-seismic-finance/blob/main/seismic-protocol-v2/contracts/protocol/lendingpool/LendingPool.sol#L151-L157
https://github.com/sherlock-audit/2024-03-seismic-finance/blob/main/seismic-protocol-v2/contracts/protocol/lendingpool/LendingPool.sol#L285

Code Snippet

https://github.com/sherlock-audit/2024-03-seismic-finance/blob/main/seismic-protocol-v2/contracts/protocol/tokenization/AToken.sol#L228-L232

https://github.com/sherlock-audit/2024-03-seismic-finance/blob/main/seismic-protocol-v2/contracts/protocol/libraries/logic/ReserveLogic.sol#L63-L79

https://github.com/sherlock-audit/2024-03-seismic-finance/blob/main/seismic-protocol-v2/contracts/protocol/libraries/logic/ReserveLogic.sol#L382-L411

Tool used

Manual Review

Recommendation

Discussion

KupiaSecAdmin

I think the yield should not be included in real-time `balanceOf` function. Because the yield is in claimable, not automatically increasing. Including real-time yield in `balanceOf` function breaks the overall functionality of the protocol.

nevillehuang

@KupiaSecAdmin Your comment seems to be correct. Wouldn't the user be able to separate calls to claiming yield and withdrawing/depositing?

cc: @mstpr

mstpr

Yield should be displayed in `balanceOf`. The way AAVE interest works is hypothetical: if someone has 100 aUSDC at t=0, then they will have, say, 101 aUSDC at t=10. However, that does not necessarily mean that the 1 USDC is returned back from the borrowers. Withdrawing the exact amount might not be possible if the funds are illiquid. It is just that the borrower will repay that amount at some stage later, and the 101 aUSDC will be redeemed to 101 USDC. With the current code not reflecting the latest yield to aToken, the rebasing behavior of aToken 1:1 is completely off. aTokens are not 1:1 if you don't account for the exact logic as the `updateIndexes` has. Lots of functionality relies on aTokens being 1:1 in Aave v2, and there will be many unwanted outcomes of this. Some of the major issues are:

1- User collateral value will be underestimated because the last yield is not reflected in the `liquidityIndex`, so balanceOf returns a lesser value, resulting in a lower total collateral value. This can lead to premature, unfair liquidations of users. https://github.com/aave/protocol-v2/blob/ce53c4a8c8620125063168620eba0a8a92854eb8/contracts/protocol/libraries/logic/GenericLogic.sol#L203-L205

2- Withdrawing the entire aToken balance will always redeem lesser amounts due to `balanceOf` not adding the claimable yield. https://github.com/aave/protocol-v2/blob/ce53c4a8c8620125063168620eba0a8a92854eb8/contracts/protocol/lendingpool/LendingPool.sol#L151-L157

mstpr

> I think the yield should not be included in real-time `balanceOf` function. Because the yield is in claimable, not automatically increasing. Including real-time yield in `balanceOf` function breaks the overall functionality of the protocol.

balanceOf is view function. When the state is updated via deposit/withdraw/repay the yield will be claimed and balanceOf will be 100% accurate inside the execution.

nevillehuang

I believe based on this comment this is medium severity, given the constraint of yield accrued on USDB/WETH

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/seismic-finance/seismic-protocol-v2/pull/23

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-2: Incorrect calculation for total pool holding

Source:
https://github.com/sherlock-audit/2024-03-seismic-finance-judging/issues/19

Found by

KupiaSec

Summary

The amount of pool's total holding is calculated with wrong formula, thus result in wrong liquidity index.

Vulnerability Detail

In `_updateIndexes` function of `ReserveLogic.sol`:

```
uint256 totalPoolHoldings = IERC20(underlyingAsset).balanceOf(aTokenAddress) +
↪   // pool liquidity
    IERC20(reserve.stableDebtTokenAddress).totalSupply() + // total stable debt
    IERC20(reserve.variableDebtTokenAddress).totalSupply(); // total variable
↪   debt
```

As shown in the code snippet, totalPoolHoldings is calculated by summing up current balance of underlying asset and total debts. The calculated amount does not reflect correct amount because:

1. balance of underlying asset does not reflect correct value because it might be affected by flashloan.
2. debt amount does not reflect latest value because it does not count debt to accrue.

As a result, totalPoolHoldings does not represent correct number and it causes wrong liquidity index.

Impact

Liquidity index is calculated incorrectly which affects balance of depositors.

Code Snippet

https://github.com/sherlock-audit/2024-03-seismic-finance/blob/main/seismic-protocol-v2/contracts/protocol/libraries/logic/ReserveLogic.sol#L392-L394

Tool used

Manual Review

Recommendation

Instead of summing up balance and debt, `aToken.totalSupply()` function has to used. It is the same way as how the AAVE V2 handles fees accrued from flashloan.

Discussion

nevillehuang

request PoC

Could the watson please highlight a better impact to justify high severity? It is missing in the original submission

sherlock-admin3

PoC requested from @KupiaSecAdmin

Requests remaining: **1**

KupiaSecAdmin

@nevillehuang - Thanks for informing about the clarification, I also think what I described above doesn't justify high severity. So here it is:

`currentTotalPoolHoldings = underlyingToken.balanceOf(aToken) + stableDebt + variableDebt` As time of calculation, `stableDebt` and `variableDebt` do not imply interests earned. Logically thinking, any claimed amounts have to be distributed through every unit of asset that the depositors have control of, which `aToken.totalSupply()` represents. Since debt is not accrued as time of calculation, `currentTotalPoolHoldings < aToken.totalSupply()`, which means `claimedAmount/currentTotalPoolHoldings > claimedAmount/aToken.totalSupply()`. As a result, the index value of collateral asset becomes bigger than it deserves, which can be thought as loss of protocol. One more point to consider is that, if above calculation is processed during `flashloan`, `underlyingToken.balanceOf(aToken) becomes as small as zero, thus` currentTotalPoolHoldings' gets significantly small.

nevillehuang

@KupiaSecAdmin I think the impact of this could be significant but without numbers I cannot verify. Do you have a more elaborate example on the impact based on the incorrect computation of `newLiquidityIndex`?

KupiaSecAdmin

Sure, happy to take an example here: Lets assume that users deposited 10K WETH to the lending pool(10K aToken amount exists), current index value is 1.1(increased from previous borrowings). And now lets say 2K WETH is borrowed and 8K WETH sitting on the lending pool earning yields. And at some point, we can say we have like 0.1 WETH yield generated to claim.

An attacker comes in using floashloan, borrowed 8K WETH leaving empty on the lending pool, and calculates new index.

When poolHoldings uses current formula which is `underlyingAsset.balanceOf(aToken) + stableDebt + variableDebt` The value will be 2K, and index update will be 0.1 / 2000 = 5e-5, newIndex will be 1.10005. This means total liquidity becomes 1.10005 * 10K = 11000.5 WETH. This means 0.4WETH generated out of nowhere, causing loss in protocol

The amount of loss is defined variously based on utilization rate and claimed amount, but above example just shows how it generates loss in protocol by increasing more in index value.

nevillehuang

@KupiaSecAdmin Can this loss accumulate?

KupiaSecAdmin

> @KupiaSecAdmin Can this loss accumulate?

Yeah, tiny loss is accumulated for every yield claims.

nevillehuang

@KupiaSecAdmin Given this issue is highly dependent on yield accrued I think medium severity is more appropriate based on this constraint

KupiaSecAdmin

> @KupiaSecAdmin Given this issue is highly dependent on yield accrued I think medium severity is more appropriate based on this constraint

Seems reasonable to me. Thanks for the clarification.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/seismic-finance/seismic-protocol-v2/pull/23

sherlock-admin2

The Lead Senior Watson signed off on the fix.

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.