ThunderLoan Audit Report

Version 1.0

QV.io

May 8, 2024

ThunderLoan Audit Report

QV

May 8, 2024

Prepared by: [QV]

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
- High
  - [H-1] The `ThunderLoan::updateExchangeRate` in the `deposit` function causes the protocal to updating more fee then it really have, which blocks the `redeem` fuction and sets incorrect exchange rate.
  - [H-2] Mixing up state variable order after upgrading causing users pay wrong fee.
- Medium
- Low
- Informational
- Gas

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

QV makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

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

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

Audit Details

Scope

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6

- In Scope:

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

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- ERC20s:
  - USDC
  - DAI
  - LINK
  - WETH ## Roles
- Owner: The owner of the protocol who has the power to upgrade the implementation.
- Liquidity Provider: A user who deposits assets into the protocol to earn interest.
- User: A user who takes out flash loans from the protocol.

Executive Summary

Issues found

Severtity	# of issues found
High	2
Medium	0
Low	0
Info	0
Total	2

Findings

High

[H-1] The `ThunderLoan::updateExchangeRate` in the `deposit` function causes the protocal to updating more fee then it really have, which blocks the `redeem` fuction and sets incorrect exchange rate.

Description: In the `ThunderLoan` contract system, the `exchangeRate` is used to calculating the exchange rate between assetToken and underlying tokens. Also, it's responsible for keeping track of how many fees to give to liquidity providers. However, the `deposit` function does not collect any fee but keep updating this rate.

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

Impact There are several impacts to this bug: 1. The `redeem` function is blocked because the protocal think it has more fee than it has 2. Rewards are incorrectly calculated, leading to liquidity provider receive wrong redemtion amout.

Proof of Concept 1. LP deposits 2. User taks out a flask loan 3. Fee calculated incorrectly 4. It is now impossible for LP to redeem

Place the following into `ThunderLoan.t.sol`

```
1   function testRedeemAfterLoan() public setAllowedToken hasDeposits {
2          uint256 amountToBorrow = AMOUNT * 10;
3          uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
               amountToBorrow);
4
5          vm.startPrank(user);
6          tokenA.mint(address(mockFlashLoanReceiver), calculatedFee); //
               borrow with some fee
7          thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
               amountToBorrow, "");
```

```
8           vm.stopPrank();
9
10          uint256 amountRedeem = type(uint256).max;
11          vm.startPrank(liquidityProvider);
12          thunderLoan.redeem(tokenA, amountRedeem);
```

Recommended Mitigation Remove the incorrectly updated exchange rate lines from `deposit`

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

[H-2] Mixing up state variable order after upgrading causing users pay wrong fee.

Description: The `ThunderLoan::s_feePrecision` and `ThunderLoan::s_flashLoanFee` swap their slot in the `ThunderLoanUpgraded` contract. In addition, after the upgrade `ThunderLoan::s_feePrecision` becomes constant variable, which is not stored on storage. The value of `ThunderLoanUpgraded::s_flashLoanFee` now gets the value of `ThunderLoan::s_feePrecision`

Most importantly, the `s_currentFlashingLoaning` mapping will be store in the wrong storage slot after the upgrade.

In`ThunderLoan` contract:

```
1 mapping(IERC20 => AssetToken) public s_tokenToAssetToken;
2 uint256 public s_feePrcision = 1e18;
3 uint256 private s_flashLoanFee;
4 mapping(IERC20 token => bool currentlyFlashLoaning) private
    s_currentlyFlashLoaning;
```

In `ThunderLoanUpgraded` contract:

```
1 mapping(IERC20 => AssetToken) public s_tokenToAssetToken;
2 uint256 private s_flashLoanFee;
3 uint256 public constant FEE_PRECISION = 1e18;
```

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

Impact Flashloan fees after the upgrade will be higher (3e17 –>1e18)

Proof of Concept

Place the following into the `ThunderLoan.t.sol`

```
1  import {ThunderLoanUpgraded} from "../../src/upgradedProtocol/
       ThunderLoanUpgraded.sol";
2
3  function testUpgradeBreaks() public {
4          uint256 feeBeforeUpdrade = thunderLoan.getFee();
5          vm.startPrank(thunderLoan.owner());
6          ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
7          thunderLoan.upgradeToAndCall(address(upgraded),"");
8          uint256 feeAfterUpgrade = thunderLoan.getFee();
9          vm.stopPrank();
10
11         console2.log("Fee before:",feeBeforeUpdrade);
12         console2.log("Fee after:", feeAfterUpgrade);
13         assert(feeBeforeUpdrade != feeAfterUpgrade);
14 }
```

You can run `forge inspect [contracname] storage` in Foundry to get more deails about the storage slot of each contract.

Recommended Mitigation Do not switch the position of variable in the upgrade. In the `ThunderLoanUpgraded.sol`:

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

Medium

Low

Informational

Gas