Thunder-Loan Audit Report

Version 1.0

Ramprasad

June 27, 2024

Thunder-Loan Report

Ramprasad

june 27, 2024

Prepared by: Ramprasad

Table of Contents

Disclaimer

I 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

The findings described in this document corresponded the following commmit hash:**

```
8803f851f6b37e99eab2e94b4690c8b70e26b3f6
```

Scope

```
#-- interfaces
|   #-- IFlashLoanReceiver.sol
|   #-- IPoolFactory.sol
|   #-- ITSwapPool.sol
|   #-- IThunderLoan.sol
#-- protocol
```

```
|    #-- AssetToken.sol
|    #-- OracleUpgradeable.sol
|    #-- ThunderLoan.sol
#-- upgradedProtocol
     #-- ThunderLoanUpgraded.sol
```

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	Number of issues found
High	2
Medium	0
Low	0
Information	0
Total	2

Findings

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

Description: In ThunderLoan system the `ExchangeRate` syatem is responsible for the calculating of exchange rate between asset token and underlaying token. In a way, it is responsible for the keeping track of how many fee is to give to the liquidity provider.

However the `deposit` function, updates this rates, without collecting any fees.

"'java script 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 several impacts to this bug.
1. The `redeem` function is blocked, beacause the protocol thinks the
   owned tokens is more than it has.
2. Rewards are incorrectly calculated, leading to liquidity providers
   potentially getting way more or lessthan deserved.

Proof of Concept:
1. LP deposits.
2. users takes out a flash loan
3. It is now impossible for LP to redeem.

<details>
<summary>Code</summary>

place the following code in `ThunderLoanTest.t.sol`

```java script
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 ammountToRedeem = type(uint256).max;
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, ammountToRedeem);
```

```
    }
```

Recommended Mitigation: Removed the incorrectly updated exchange rates lines from the `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);

        //@audit-high
-        uint256 calculatedFee = getCalculatedFee(token, amount);
-        assetToken.updateExchangeRate(calculatedFee);

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

[H-2] Mixingup variable location causes storage collision n `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol

Description: `ThunderLoan.sol` has 2 variables in the following order:

"'java script uint256 private s_feePrecision; uint256 private s_flashLoanFee;

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

```java script
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 posotion of storage variables, and removing storage variables for constanr varoables, bearks the storage locations as well.

Impact: After the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. this means that user who takes out the flash loans after the upgrade will be charged the wrong fee.

Recommended Mitigation: If you must remove the storage variable, leave it as blank as o not mess up with the storage slot.

“‘diff - 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;

```
```
```
```