Thunder-Loan Audit Report

Version 1.0

Ramprasad

April 26, 2025

Thunder-Loan Report

Ramprasad

April 26, 2025

Prepared by: Ramprasad

Table of Contents

Disclaimer

This audit was independently conducted by Ramprasad. Every effort was made to identify as many vulnerabilities as possible within the given time frame. However, no guarantees are provided regarding the discovery of all existing issues.

This report focuses solely on the security aspects of the Solidity implementation at the time of the audit. It does not constitute an endorsement of the underlying business model or project viability. Readers should perform their own independent assessments.

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 commit 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

Severity	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 the ThunderLoan system, the `ExchangeRate` system is responsible for calculating the exchange rate between the asset token and the underlying token. In effect, it keeps track of how many fees are to be given to the liquidity provider.

However, the `deposit` function updates these rates 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);

@>        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 owned tokens are 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 becomes impossible for LP to redeem.

Code

Place the following code in `ThunderLoanTest.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(liquidityProvider);
        thunderLoan.redeem(tokenA, amountToRedeem);
    }
```

Recommended Mitigation: Remove the incorrectly updated exchange rate lines 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);

-        uint256 calculatedFee = getCalculatedFee(token, amount);
-        assetToken.updateExchangeRate(calculatedFee);

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

[H-2] Mixing up variable location causes storage collision in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol

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

```
    uint256 private s_feePrecision;
    uint256 private s_flashLoanFee;
```

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 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 after the upgrade will be charged the wrong fee.

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

```
-   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;
```