Thunder Loan Initial Audit Report

Version 0.1

BowTiedBrothers

September 26, 2024

Thunder Loan Audit Report

BowTiedBrothers

September 26, 2023

Thunder Loan Audit Report

Prepared by: BowTiedBrothers Lead Auditors:

- BowTiedBrothers

Assisting Auditors:

- None

Table of contents

See table

  - Issues found
- Findings
  - High
    * [H-1] Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes protocol to think it has more fees than it really does, which blocks redemption and incorrectly sets the exchange rate
    * [H-2] Changing variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, causing protocol to freeze.
  - Medium
    * [M-1] Using TSwap as a price oracle leads to price and oracle manipulation attacks.

About BowTiedBrothers

I am a freelance junior security researcher specializing in auditing Solidity and Vyper code.

Disclaimer

BowTiedBrothers puts in as much effort as possible 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

Audit Details

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

```
3bb18274ae8d6f72b49f79a6621223a17a9a2038
```

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:

Give users a way to create flash loans 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!

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	1
Low	0
Info	0
Gas	0
Total	3

Findings

High

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

Description In the ThunderLoan system, the `exchangeRate` is responsible for calculating the exchange rate between assetTokens and underlying tokens. It is directly responsible for keeping track of how many fees to go to liquidity providers (LP's).

The `deposit` function ends up updating the rate but 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);
        // @audit-high: should not be updating exchange rate as it
           breaks the redeem function
@>      uint256 calculatedFee = getCalculatedFee(token, amount);
@>      assetToken.updateExchangeRate(calculatedFee);

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

Impact Several impacts from this bug.

1. The `redeem` function is blocked because the protocol thinks the amount of tokens to be redeemed is more than the users balance.
2. Rewards are incorrectly calculated, leading to LP's potentially getting a different amount of tokens than deserved.

Proof of Concept

1. LP deposits
2. User takes out a flash loan
3. LP is now unable to redeem

Proof of Code

Place the following into `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 calculation lines from `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: should not be updating exchange rate as it
            breaks the redeem function
-       uint256 calculatedFee = getCalculatedFee(token, amount);
-       assetToken.updateExchangeRate(calculatedFee);

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

[H-2] Changing variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, causing protocol to freeze.

Description `ThunderLoan.sol` has 2 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 functions, after the upgrade the `s_flashLoanFee` will have the value of `s_feePrecision`. The position of storage variables cannot be adjusted and removing storage variables for constant varaibles breaks 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 afer an upgrade will be charged the wrong fee.

More importantly, the `s_currentlyFlashLoaning` mapping will start in the wrong storage slot, leading to additional storage collisions.

Proof of Concept

Proof of Code

Place the following into `ThunderLoanTest.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 upgrade: ", feeBeforeUpgrade);
        console.log("Fee after upgrade: ", feeAfterUpgrade);
        assert(feeBeforeUpgrade != feeAfterUpgrade);
    }
```

You can also see the storage layout difference 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 not mess up storage slot order.

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

Medium

[M-1] Using TSwap as a price oracle leads to price and oracle manipulation attacks.

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. This in turn makes it easy for malicious users to manipulate prices of tokens by either buying or selling large quantities of a specific token in the same transaction, leading to protocol fees essentially being ignored.

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

Proof of Concept:

The following happens in 1 transaction:

1. User takes a flash loan from `ThunderLoan` for 1000 `tokenA`. They are charged the original fee `feeOne`. During the flash loan, they do the following:
    1. User sells 1000 `tokenA`, tanking the price.
    2. Instead of repaying immediately, the user takes out another flash loan for another 1000 `tokenA`.
        1. Due to the `ThunderLoan` using `TSwapPool` as a price oracle, the second flash loan is substantially cheaper.

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

```
}
```

3. The user then repays the first flash loan, and immediately repays the second flash loan.

I have created a proof of code located in my `audit-data` folder since it is too large to include here.

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