



Protocol Audit Report

Version 1.0

Cyfrin.io

July 27, 2024

Protocol Audit Report

Cyfrin.io

March 7, 2023

Prepared by: Cyfrin Lead Auditors: - xxxxxxxx

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
 - Scope
 - Roles
- Executive Summary
 - Issues found
- Findings
- High
- Medium
- Low
- Informational
- Gas

Protocol Summary

Protocol does X, Y, Z

Disclaimer

The YOUR_NAME_HERE 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
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

everything. ## Roles owner, user, flashloanreceiver (contract) # Executive Summary here it is ## Issues found some

Findings

[H-1] Erroneous ThunderLoan::updateExchangeRate in the deposit function causes protocol to have 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. In a way, it's responsible for keeping track of how many fees to give to liquidity providers. However, the `deposit` function updates this rate, without collecting any fees!

```
1      function deposit(IERC20 token, uint256 amount) external
2          revertIfZero(amount) revertIfNotAllowedToken(token) {
3          AssetToken assetToken = s_tokenToAssetToken[token];
4          uint256 exchangeRate = assetToken.getExchangeRate();
5          uint256 mintAmount = (amount * assetToken.
6              EXCHANGE_RATE_PRECISION()) / exchangeRate;
7          emit Deposit(msg.sender, token, amount);
8          assetToken.mint(msg.sender, mintAmount);
9          uint256 calculatedFee = getCalculatedFee(token, amount);
10         // audit high this should not update the exchange rate here
11         @> assetToken.updateExchangeRate(calculatedFee);
12         @> token.safeTransferFrom(msg.sender, address(assetToken), amount)
13     ;
14 }
15 }
```

Impact: There are several impacts to this bug.

1. The `redeem` function is blocked, because the protocol thinks the owed tokens is more than it has
2. Rewards are incorrectly calculated, leading to providers 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.

Proof of Code

Place the following into `ThunderLoanTest.t.sol`

```
1      function testRedeemAfterLoan() public setAllowedToken hasDeposits {
2          uint256 amountToBorrow = AMOUNT * 10;
```

```
3      uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
4          amountToBorrow);
5      vm.startPrank(user);
6      tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);
7      thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
8          amountToBorrow, "");
9      vm.stopPrank();
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
2          revertIfZero(amount) revertIfNotAllowedToken(token) {
3          AssetToken assetToken = s_tokenToAssetToken[token];
4          uint256 exchangeRate = assetToken.getExchangeRate();
5          uint256 mintAmount = (amount * assetToken.
6              EXCHANGE_RATE_PRECISION()) / exchangeRate;
7          emit Deposit(msg.sender, token, amount);
8          assetToken.mint(msg.sender, mintAmount);
9          uint256 calculatedFee = getCalculatedFee(token, amount);
10         // audit high this should not update the exchange rate here
11         - assetToken.updateExchangeRate(calculatedFee);
12         - token.safeTransferFrom(msg.sender, address(assetToken), amount)
13         ;
14     }
```

[H-2] Mixing up variable location causes storage collisions in

ThunderLoan::s_flashLoanFee and ThunderLoan::s_currentlyFlashLoaning

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

```
1      uint256 private s_feePrecision;
2      uint256 private s_flashLoanFee; // 0.3% ETH fee
```

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

```
1      uint256 private s_flashLoanFee; // 0.3% ETH fee
2      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 positions of storage variables when working with upgradeable contracts.

Impact: After upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. This means that users who take out flash loans right after an upgrade will be charged the wrong fee. Additionally the `s_currentlyFlashLoaning` mapping will start on the wrong storage slot.

Proof of Code:

Code

Add the following code to the `ThunderLoanTest.t.sol` file.

```
1 // You'll need to import `ThunderLoanUpgraded` as well
2 import { ThunderLoanUpgraded } from "../src/upgradedProtocol/
  ThunderLoanUpgraded.sol";
3
4 function testUpgradeBreaks() public {
5     uint256 feeBeforeUpgrade = thunderLoan.getFee();
6     vm.startPrank(thunderLoan.owner());
7     ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
8     thunderLoan.upgradeTo(address(upgraded));
9     uint256 feeAfterUpgrade = thunderLoan.getFee();
10
11     assert(feeBeforeUpgrade != feeAfterUpgrade);
12 }
```

You can also see the storage layout difference by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage`

Recommended Mitigation: Do not switch the positions of the storage variables on upgrade, and leave a blank if you're going to replace a storage variable with a constant. In `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;
```

[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 a token is determined by how many reserves are on either side of the pool. Because of this, it is easy for malicious users 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 drastically reduced fees for providing liquidity.

Proof of Concept:

The following all happens in 1 transaction.

1. User takes a flash loan from [ThunderLoan](#) for 1000 [tokenA](#). They are charged the original fee [fee1](#). During the flash loan, they do the following:
 1. User sells 1000 [tokenA](#), 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 loan, 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 TWAP fallback oracle.