ThunderLoan Protocol Audit Report

Version 1.0

Aditya Mohan

January 11, 2025

ThunderLoan Protocol Audit Report

Aditya Mohan Jha

11 January, 2025

Prepared by: Aditya Lead Security Researcher: - Aditya

Table of Contents

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

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!

What is a flash loan?

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.

Users additionally have to pay a small fee to the protocol depending on how much money they borrow. To calculate the fee, we're using the famous on-chain TSwap price oracle.

We are planning to upgrade from the current `ThunderLoan` contract to the `ThunderLoanUpgraded` contract. Please include this upgrade in scope of a security review.

Disclaimer

The 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

		Impact		
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

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
- In Scope:

```
#-- 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
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 the calculating the exchange rate between assetToken and it's underlying token.In a way it's responsible for keeping track of how many fees to give to liquidity provider.

However,the `deposit` function,update the 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 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 to liquidity providers potentially getting more or less than deserved.

Proof of Concept:

1. LP deposits
2. Users take out a flash loan
3. It is now impossible to redeem

Proof of Code

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

[H-2] User able to repay the flashLoan amount by calling `ThunderLoan::deposit` instead of `ThunderLoan::redeem`.This will allow user to steal all the fund.

Description: The `flashloan()` perform the balance check to ensure the ending balance, after the flash loan should exceed the initial balance, accounting the borrower fee.The verification achieved by comparing `endingBalance` with `startingBalance+fee`. However, a vulnerability emerges when calculating endingBalance using `token.balanceOf(address(assetToken))`.

Exploiting this vulnerability, an attacker can return the flash loan using the `deposit()` instead of `repay()`. This action allows the attacker to mint `AssetToken` and subsequently redeem it using `redeem()`. What makes this possible is the apparent increase in the Asset contract's balance, even though it resulted from the use of the incorrect function. Consequently, the flash loan doesn't trigger a revert.

Impact: A attacker can acquire a fund using flash loan and deposit funds directly using `deposit()`, enabling stealing all funds.

Proof of Concept: 1. User will create a flashloan 2. And repay the borrow amount using `deposit`.

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

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

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

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

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

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

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

Due to how solidity storage work, after the upgrade `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot adjust the value of storage variables, and removing storage variables for constant variables, breaks the storage location as well.

Impact: After the upgrade `s_flashLoanFee` will have the value of `s_feePrecision`. This means the user who take the flash loan after the upgrade will charged wrong fee.

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

Proof of Concept:

Proof of Code:

Put below code in `ThunderLoanTest.t.sol` file.

```
    import {ThunderLoanUpgraded} from "../../src/upgradedProtocol/
        ThunderLoanUpgraded.sol";
    .
    .
    .
    .
     function testUpgradeBreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgrade = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgrade),"");
        uint256 feeAfterUpgrade = thunderLoan.getFee();
        vm.stopPrank();

        console.log("Fee Before:",feeBeforeUpgrade);
        console.log("Fee After:",feeAfterUpgrade);
        vm.expectRevert();
        assertEq(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 slot.

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

Medium

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

Description: The TSwap protocol user constant product formula based AMM (automated market maker). The price of a token is determine by how many token in the reserve pool of each token.Because of this it is easy for malicious user to manipulate the price of by buying and selling the large amount of token in same transaction, essentially ignoring the protocol fee.

Impact: Liquidity provider will drastically reduced fee for providing liquidity.

Proof of Concept:

The following all happen in 1 transaction.

1. User take the flash loan from `ThunderLoan` for 1000 `tokenA`. They are charged the original fee `fee1`. During the flash loan they do following:
   1. User sells 1000 `tokenA`, tanking the price.
   2. Instead of repaying right away,user take flash loan for another 1000 `tokenA`.
      1. Due to the fact that the way `ThunderLoan` calculate price based on `TSwapPool` this second flash loan is substantially cheaper.

```
    function getPriceInWeth(address token) public view returns (
        uint256) {
        address swapPoolOfToken = IPoolFactory(s_poolFactory).
            getPool(token);
@>      return ITSwapPool(swapPoolOfToken).
            getPriceOfOnePoolTokenInWeth();
    }
```

   3. Than user repay the first flash loan and then 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.