Thunder Loan Audit Report

Prepared by Pluton

Version 1.0

Lead Auditor

Herman Effendi

August 16, 2024

Table of Contents

Introduction

The Pluton team strives to identify as many vulnerabilities as possible within the allotted time but accepts no responsibility for the results detailed in this document. Their security audit should not be considered an endorsement of the business or product. The audit was limited in scope and focused exclusively on the security aspects of the Solidity code for 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

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

```
8803f851f6b37e99eab2e94b4690c8b70e26b3f6
```

Scope

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

```
10 #-- upgradedProtocol
11     #-- 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

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	2
Info	2
Gas	2
Total	11

Findings

High

[H-1] Erronous `ThunderLoan::updateExchangeRate` in the deposit function causes the protocol to think it has more fees that it really does, which block redemption and incorrectly sets the exchange rate

Description

In the ThunderLoan system, the `exchangeRate`is responsible for calculating the exchane rate between assetTokens and underlying tokens. In a way, it's responsible for keeping track how may fees

to give to liquidity provider. 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          // @audit high
8  @>      uint256 calculatedFee = getCalculatedFee(token, amount);
9  @>      assetToken.updateExchangeRate(calculatedFee);
10         token.safeTransferFrom(msg.sender, address(assetToken), amount)
               ;
11     }
```

Impact

There are several impact to this bug :

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

Proof of Concepts

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

Proof of codes

```
1  function testCantRedeemAfterFlashLoan() public setAllowedToken
       hasDeposits {
2          uint256 amountToBorrow = AMOUNT * 10;
3          vm.startPrank(user);
4          tokenA.mint(address(mockFlashLoanReceiver), AMOUNT);
5          thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
               amountToBorrow, "");
6          vm.stopPrank();
7
8          vm.startPrank(liquidityProvider);
9          thunderLoan.redeem(tokenA, type(uint256).max);
10         vm.stopPrank();
11     }
```

Recommended mitigation

Remove the incorrectly update 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] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`

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 has them in different order :

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

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

Impact

After the upgrade, `s_flashLoanFee` will have value of `s_feePrecision`, make a user have wrong fee. More important thins that the variable of `s_currentlyFlashLoaing` mapping with storage in the wrong torage slot.

Proof of Concepts

The test, following `ThunderLoanTest.t.sol::testUpgradeBreaks`

PoC

```
function testUpgradeBreaks() public {
        uint256 feeBefore = thunderLoan.getFee();

        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgrade = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgrade), "");
        vm.stopPrank();

        uint256 feeAfter = thunderLoan.getFee();

        console.log("fee before :", feeBefore);
        console.log("fee after :", feeAfter);

        assert(feeBefore != feeAfter);
    }
```

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

Recommended mitigation

If you want to avoid the 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 allows a user to borrow some funds and then repay them with fees. A way to repay not only using `ThunderLoan::repay` function, but also using `ThunderLoan::deposit` function. a checking for flashloaning is done through the total asset tokens. A normal case is when a user does flashloan and then repays the overall amount with a fee. However, an abnormal case is when a user does flashloan, and instead of repaying, a user deposits token to reduce the count of token for passing the check on the total asset tokens. Otherwise, a user can steal some funds through the remaining tokens from the deposit.

Impact

The impact is a user can charge a fee lower than expected.

Proof of Concepts

1. A user takes a flashloan
2. The user takes some funds (from the flashloan) and then makes deposit into the protocol
3. Because of the deposit, the check on the total asset passes
4. The user receive some remaining funds from deposits

PoC

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

        vm.startPrank(user);
        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)) > amountToBorrow + fee);
    }
```

Recommended mitigation

The `ThunderLoan::deposit` function is recommended to using `s_flashloaning` check to avoid this problem. It has behavior with `ThunderLoan::repay` as well.

```
    function deposit(IERC20 token, uint256 amount) external
        revertIfZero(amount) revertIfNotAllowedToken(token) {
+       if (!s_currentlyFlashLoaning[token]) {
+           revert ThunderLoan__NotCurrentlyFlashLoaning();
+       }
        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)
            ;
```

```
15         }
```

Medium

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

Description

The TSwap protocol is a constant product formula based AMM (Automated Market Model). The price of a token is determined by how many reserves are on either side of the pool. Because of this, it's easy to malicious users to manipulate the price of the token by buying or selling large amount of token in the same transaction. esentially ignoring protocol fees.

Impact

Liquidity providers drastically reduced fees for providing liquidity.

Proof of Concepts

The following all happens in one transaction.

1. User takes a flashloan from `ThunderLoan` for 1000 `tokenA`. They are charged the original fee `fee1` . During flash loan, they do following :
    a. User sells 1000 `tokenA`, tanking the price down.
    b. Instead of repaying right away, the user takes another flash loan for another 1000 `tokenA`.
        - Due to the fact the way `ThunderLoan` calculates price based on the `TSwapPool`, this second flash loan supposedly more cheaper than expected.
    c. The user then repays the first flash loan ,and then repays the second flash loan.

Proof of code, following `ThuderLoanTest.t.sol::testOracleManipulation`.

Recommended mitigation

Consider using a different price oracle mechanism, like a chainlink price feed with a Uniswap TWAP fallback oracle.

[M-2] Centralization risk causes the owner using unfair token address

Description

The `ThunderLoan.sol::setAllowedToken` function is intended to set allowed token for conducting a thunderloan. However, only the owner has the ability to set a token as allowed.

```
1   function setAllowedToken(IERC20 token, bool allowed) external
       onlyOwner returns (AssetToken) {
2     ...
3   }
```

Impact

If the owner has malicious intent, they could set an unfair token allowed.

Low

[L-1] Empty Function Body - Consider commenting why

```
1 // src/ThunerLoan.sol
2 function _authorizeUpgrade(address newImplementation) internal override
      onlyOwner { }
```

[L-2] Missing important event emissions

Description

When the `ThunderLoan::s_flashLoanFee` is updated, there is no event emitted.

Recommended mitigation

Create a emit event when the `ThunderLoan::s_flashLoanFee` is updated.

```
1  +  event FlashLoanFeeUpdated(uint256 newFee);
2     .
3     .
4     function updateFlashLoanFee(uint256 newFee) external onlyOwner {
5           if (newFee > s_feePrecision) {
6               revert ThunderLoan__BadNewFee();
7           }
8           s_flashLoanFee = newFee;
9  +        emit FlashLoanFeeUpdated(newFee);
10    }
```

Informational

[I-1] Poor test coverage

[I-2] Not using __gap[50] for future storage collision mitigation

Gas

[G-1] Unnecessary SLOAD when using emit

In `AssetToken.sol::updateExchangeRate`, we create new variable `newExchangeRate` as memory variable. However, the `emit` statement uses storage variable `s_exchangeRate` as a parameter instead of `newExchangeRate`. This can result in an unnecessary SLOAD. To avoid this, use the memory variable `newExchangeRate` rather than the storage variable `s_exchangeRate`.

```
1       s_exchangeRate = newExchangeRate;
2   -   emit ExchangeRateUpdated(s_exchangeRate);
3   +   emit ExchangeRateUpdated(newExchangeRate);
```

[G-2] Using bools for storage incurs overhead

Use uint256(1) and uint256(2) for true/false to avoid a Gwarmaccess (100 gas), and to avoid Gsset (20000 gas) when changing from ‘false’ to ‘true’, after having been ‘true’ in the past.

```
1       mapping(IERC20 token => bool currentlyFlashLoaning) private
            s_currentlyFlashLoaning;
```