ThunderLoan Audit Report

Prepared by: Tadeo

Table of Contents

Protocol Summary

ThunderLoan allows users to take flashloans for their different DeFi strategies, and Liquidity Providers earn fees for allowing them to lend their money.

Disclaimer

The Tadeo 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

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:

```
463672a5d8cfce3c3ae63549eb096ea1165eedbd
```

Scope

```
./src/
└──interfaces
    └── IFlashLoanReceiver.sol
    └── IPoolFactory.sol
    └── IThunderLoan.sol
    └── ITSwapPool.sol
└──protocol
    └── AssetToken.sol
    └── IPoolFactory.sol
    └── ThunderLoan.sol
└──upgradedProtocol
    └── ThunderLoanUpgraded.sol
```

Roles

- Owner: The user which can modify some configuration parameters of the ThunderLoan contract.
- Liquidity Providers: They deposit their money to be loaned in exchange for the profit of a Fee.
- User: You can apply for flash loans on the platform.

Executive Summary

It took 8 hours to audit the code, and 6 vulnerabilities were found in the code. The entire audit was done under manual review.

Issues found

Severity	Numbers of issues found
High	3
Mdium	1
Low	1
Gas	1
Total	6

Findings

High

[H-1] Erroneus `ThunderLoan::updateExchangeRate` in `deposit` function causes protocol to think it has more fees than it really does, which blocks redemptions and incorrectly sets the exchange rate.

Description: In the ThunderLoan system, the `exchangeRate` is responsible for calculating the exchange rate between assets and underlying tokens. In a way, it's responsible for keeping track of how many fees to give to liquidity providers.

However, the `deposit` functions, updates this rate, whithout 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 way more or less than deserved.

Proof of Concept:

1. LP deposits
2. User take out a flash loan
3. It's 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), AMOUNT);
        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 update exchange rate lines from `deposit` funciton.

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

Description: The `ThunderLoan::flashloan()` performs a crucial balance check to ensure that the ending balance, after the flash loan, exceeds the initial balance, accounting for any borrower fees. This verification is 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 `ThunderLoan::deposit()` instead of `ThunderLoan::repay()`. This action allows the attacker to mint AssetToken and subsequently redeem it using `ThunderLoan::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.

```
    function flashloan(
        address receiverAddress,
```

```
        IERC20 token,
        uint256 amount,
        bytes calldata params
    )
        external
        revertIfZero(amount)
        revertIfNotAllowedToken(token)
    {
        AssetToken assetToken = s_tokenToAssetToken[token];
@>      uint256 startingBalance =
IERC20(token).balanceOf(address(assetToken));

        ....

@>      uint256 endingBalance = token.balanceOf(address(assetToken));
        if (endingBalance < startingBalance + fee) {
            revert ThunderLoan__NotPaidBack(startingBalance + fee,
endingBalance);
        }
        s_currentlyFlashLoaning[token] = false;
    }
```

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

Proof of Concept: This proof happens in 1 transaction:

1. Make a flashloan with `ThunderLoan::flashloan()`
2. During the flash loan they do the following:
    1. Deposit to the `ThunderLoan`contract using `deposit()` function the borrowed tokens.
    2. Complete the transaction and the `ThunderLoan`contract thinks you pay the loan.
3. Call `ThunderLoan::redeem()` and steal the tokens borrow plus fees.

I make a copy of `ThunderLoanTest::testUseDepositToPayFlashLoanAndStealFunds` and add a proof of code you can see in my `audit-data` folder, because is too large to include in the report.

Recommended Mitigation: Add a check in `ThunderLoan::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 `ThunderLoan::flashloan()` and checking it in `ThunderLoan::deposit()`.

[H-3] Mixing up variable locations causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentFlashLoaning`, 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 different order:

```
    uint256 private s_flashLoanFee; // 0.3% ETH fee
    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 can't adjust the position of storage variables, and removing storage variables for constant variables, breaks the storage location 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 rigth after an upgrade will be charged the wrong fee.

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

Proof of Concept:

▶ PoC

Paste the following code in `ThunderLoanTest.t.sol`:

```
    import { ThunderLoanUpgraded } from
"../../src/upgradedProtocol/ThunderLoanUpgraded.sol";
    .
    .
    .
    .

    function testupgradeBreaks() public {
        uint256 beforeUpgradeFee = thunderLoan.getFee();

        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded newThunderLoan = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(newThunderLoan),"");
        uint256 afterUpgradeFee = thunderLoan.getFee();
        vm.stopPrank();


        console.log("beforeUpgradeFee: ",beforeUpgradeFee);
        console.log("afterUpgradeFee: ",afterUpgradeFee);

        assert(beforeUpgradeFee != afterUpgradeFee);
    }
```

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;
```

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, the price of a token is determined by hoy many reserves are on either side of the pool. Because of this, it's easy for malicious users to manipulate the price of the token by buying or selling a large amount of the token in the same transaction, ignoring the protocol fees.

Impact: Liquidity providers will drastically reduce fees for providing liquidity.

Proof of Concept: This proof happens in 1 transaction

1. User takes a flash loan from `ThunderLoan::flashloan` for 1000 `tokenA`. They are charged the original fee (`fee1`).
2. During the flash loan they do the following:
    1. User sells 1000 `tokenA`, taking the price.
    2. The user take another flash loan for another 1000 `tokenA`.
    3. Due to the fact the `ThunderLoan` calculates price based on the `TSwapPool` this secong flash loan is more cheaper than the original.

In `ThunderLoan`:

```
    function getCalculatedFee(IERC20 token, uint256 amount) public view
returns (uint256 fee) {
@>      uint256 valueOfBorrowedToken = (amount *
getPriceInWeth(address(token))) / s_feePrecision;
        fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
    }
```

In `OracleUpgradeable`:

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

3. The user repay the flash loans.

I make a copy of `ThunderLoanTest::testOracleManipulationAttack` and add a proof of code you can see in my `audit-data` folder, because is too large to include in the report.

Recommended Mitigation: Consider using different price oracle mecanism, like chainlink price feed or Uniswa TSWAP oracle.

Low

[L-1] `ThunderLoan::getCalculatedFee` can be 0.

Description: Any value up to 333 for "amount" can result in 0 fee based on calculation

Impact: Low as this amount is really small

Proof of Concept: Add this code in `ThunderLoanTest.t.sol`:

▶ PoC

```
    function testGetCalculatedFeeIsZero() public {
        uint256 calculatedFee = thunderLoan.getCalculatedFee(
            tokenA,
            333
        );

        assertEq(calculatedFee ,0);

        console.log(calculatedFee);
    }
```

Recommended Mitigation: A minimum fee can be used to offset the calculation, though it is not that important.

Gas

[G-1] Using private rather than public for constants, saves gas

Description: If needed, the values can be read from the verified contract source code. Saves 3406-3606 gas in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata.

In `AssetToken.sol`:

```
-    uint256 public constant EXCHANGE_RATE_PRECISION = 1e18;
+    uint256 private constant EXCHANGE_RATE_PRECISION = 1e18;
```

In `ThunderLoan.sol`:

```
-   uint256 public constant FLASH_LOAN_FEE = 3e15; // 0.3% ETH fee 4
-   uint256 public constant FEE_PRECISION = 1e18;

+   uint256 private constant FLASH_LOAN_FEE = 3e15; // 0.3% ETH fee 4
+   uint256 private constant FEE_PRECISION = 1e18;
```

In `ThunderLoanUpgraded.sol`:

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