Prepared by: mhmojtaba Lead security researcher: mhmojtaba

- Audit Date: April 17, 2024

# Table of contents

▶ Details
See table

# Protocol Summary

`ThunderLoan` is a flash loan protocol that allows users to borrow assets without collateral, provided the loan is repaid within the same transaction. Liquidity providers can deposit assets into the protocol and receive `AssetTokens`, which accrue interest based on flash loan usage. Fees are calculated using the on-chain TSwap price oracle. An upgrade from the current `ThunderLoan` contract to `ThunderLoanUpgraded` is planned and included in the audit scope.

# Disclaimer

The `mhmojtaba` 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

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

```
026da6e73fde0dd0a650d623d0411547e3188909
```

## Scope

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

## Roles

- Liquidity Providers: Users who have liquidity deposited into the pools. Their shares are represented by the LP ERC20 tokens. They gain a 0.3% fee every time a swap is made.
- Users: Users who want to swap tokens.

# Executive Summary

## Issues found

| Severity | Number of issues found |
|---|---|
| High | 4 |
| Medium | 3 |
| Low | 3 |

| Severity | Number of issues found |
|---|---|
| Info | 4 |
| Gas | 1 |
| Total | 15 |

# Findings

## High Risk Findings

### [H-1] Reward Manipulation: In the `ThunderLoan::updateExchangeRate` in the `deposit` function, the fee is calculated in the wrong place and makes the protocol vulnerable make the fee higher than it should be. It cause the protocol block redeem the user's assets.

**Description:** In the Thunderloan protocol, the `exchangeRate` is updated in the `updateExchangeRate` function. However, the fee is calculated in the `deposit` function. This makes the protocol vulnerable to reward manipulation.

```
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
>>>     uint256 calculatedFee = getCalculatedFee(token, amount);
>>>     assetToken.updateExchangeRate(calculatedFee);

        token.safeTransferFrom(msg.sender, address(assetToken), amount);
    }
```

**Impact:** A liquidity provider will deposit assets into the protocol to earn interest. The protocol will calculate the fee and update the exchange rate. The fee is calculated in the wrong place and makes the protocol vulnerable make the fee higher than it should be. It cause the protocol block redeem the provider's assets.

**Proof of Concept:** The following unit test can reproduce the issue:

1. Providers deposit assets into the protocol.
2. users take out flash loans.
3. The protocol will calculate the fee and update the exchange rate.
4. providers try to redeem their assets but the protocol will revert.

▶ POC

```
function testRedeem() 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();

        vm.expectRevert();
        uint256 amountToRedeem = type(uint256).max;
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, amountToRedeem);
    }
```

**Recommended Mitigation:** The calculation of the fee should be removed from `deposite` 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);
    }
```

/////// oracle manipulation depist instead of repay storage collision

### [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 expected 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 positions of storage variables when working with upgradeable contracts.

**Impact:** After 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. Additionally the `s_currentlyFlashLoaning` mapping will start on the wrong storage slot.

**Proof of Concept:**

▶ Code

Add the following code to the `ThunderLoanTest.t.sol` file.

```
// You'll need to import `ThunderLoanUpgraded` as well
import { ThunderLoanUpgraded } from
"../../src/upgradedProtocol/ThunderLoanUpgraded.sol";

function testStorageCollision() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());

        ThunderLoanUpgraded thunderLoanUpgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(thunderLoanUpgraded), "");

        uint256 feeAfterUpgrade = thunderLoan.getFee();
        vm.stopPrank();

        console.log("fee before", feeBeforeUpgrade);
        console.log("fee after", 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:**

Do not switch the positions of the storage variables on upgrade, and leave a blank if you're going to replace a storage variable with a constant. In `ThunderLoanUpgraded.sol`:

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

### [H-4] getPriceOfOnePoolTokenInWeth uses the TSwap price which doesn't account for decimals, also fee precision is 18 decimals

## Medium Risk Findings

### [M-1] Centralization risk for trusted owners

**Description:**

**Impact:**

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

**Instances (2):**

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

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

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

Contralized owners can brick redemptions by disapproving of a specific token

### [M-2] Using TSwap as price oracle leads to price and oracle manipulation attacks

**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 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: i. User sells 1000 tokenA, tanking the price. ii. Instead of repaying right away, the user takes out another flash loan for another 1000 tokenA. a. Due to the fact

that the way ThunderLoan calculates price based on the 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. The user then repays the first flash loan, and then repays the 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.

### [M-3] Fee on transfer, rebase, etc

## Low Risk Findings

### [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 (1):**

```
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] 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);
```

**Impact:**

**Proof of Concept:**

**Recommended Mitigation:**

## 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