T-Swap Audit Report

Version 1.0

Jeremy Bru

February 29, 2024

T-Swap Audit Report

Jeremy Bru

Feb 29, 2024

Minimal Audit Report - T-Swap

Prepared by: Jeremy Bru (Link) Lead Security Researcher:

- Jeremy Bru

Contact: –

Table of Contents

Protocol Summary

From my understanding, This project is meant to be a permissionless way for users to swap assets between each other at a fair price. The protocol is an AMM (Automated Market Maker) that should do the following:

- Respect the constant formula `x * y = k` for each pools.
  - The ratios of tokens should always stay the same.
- In order for the protocol to work, it needs to have liquidity providers.
  - Their shares are represented by an `LP` ERC20 tokens. ie `T-SwapWeth`.
  - They gain a 0.3% fee every time a swap is made.
  - The generated gain to the liquidity providers will be based in the form of a liquidity pool token (`LP`s) number increasing.
- Liquidity can be deposited, added and withdrawn by those 2 functions:
  - `TSwapPool::deposit`
  - `TSwapPool::withdraw`
  - Note: If there is a deposit of liquidity from a user already in place, the deposit function will be considered as `adding liquidity`.

`PoolFactory` contract is the contract used to create new "pools" of tokens via the `PoolFactory::createPool` function. It helps make sure every pool token uses the correct logic.

- Pools are made of 2 assets a Token A (x) and the WETH token (y).
- Allows swapping of assets between each other at a fair price.

There are 2 functions users can call to swap tokens in the pool, from the `TSwapPool` contract:

- `TSwapPool::swapExactInput`
- `TSwapPool::swapExactOutput`
- Users are able to swap tokens based on the amount of tokens they want to receive or the amount of tokens they want to give in exchange of the desired asset.

The chain to which it is gonna be deployed is `Ethereum` and `any ERC20` token can be used with WETH as a paired token to create pools.

Disclaimer

I, Jeremy Bru, did 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

Uses the CodeHawks (Link) severity matrix to determine severity. See the documentation for more details.

Audit Details

Commit Hash: `e643a8d4c2c802490976b538dd009b351b1c8dda`

Scope

```
./src/
#-- PoolFactory.sol
#-- TSwapPool.sol
```

Roles

- Swapper - Users that swapping between 2 assets in a pool.
- Liquidity Providers - Users that initialize and provide liquidity in pools for users to be able to swap their assets. Gaining fees on each swap based on their shares.

Executive Summary

Used `Forge Foundry`, `Aderyn`, `Slither`, Stateful Fuzzing with Handler and manual review to find the following issues and wrote test cases to show the issues.

Issues found

Severity	Number of findings
High	6
Medium	2
Low	3
Infos	10
———	———————
Total	21

Findings

High

[S-H1] Core Invariant `x * y = k` is broken by the _swap function incentives, pool ratio is broken.

Description:

The protocol is giving extra incentive to users who are swapping, `swappers`, in WETH tokens every 10 transactions. This, to keep users exchanging their assets on the protocol.

The incentive is about `1WETH` every `10 transactions`.

So, the invariant `x * y = k`, that should conserves a constant ratio between the two assets, breaks due to the above incentive.

The pool ratio is completly broken, as users are getting more from the pool as they should.

Impact:

- The pool can be drained to 0 by `swappers` without big effort.
- And this every 10th transaction per users.

Proof of Concept:

- In the test folder there is a test case using a Stateful Fuzzing with Handler method to show the issue.
- Also adding a third test not based on the fuzzing method, by just doing 10 swaps.

The first test:

- `test/Invariant.t.sol::statefulFuzz_constantProductFormulaStaysTheSameTokenA`

- It checks for equality of what was deposited and the difference in change of the amount in the pool based on the first paired token.
- Equality matches.

run the test with:

```
1 forge test --mt statefulFuzz_constantProductFormulaStaysTheSameTokenA -
    vvvv
```

The second test, aims for the same verification but on the Weth token.

- `test/Invariant.t.sol::statefulFuzz_constantProductFormulaStaysTheSameWeth`

- Equality does not match due to the incentive. run the test with:

```
forge test --mt statefulFuzz_constantProductFormulaStaysTheSameWeth -vvvv
```

The incentive `TSwapPool::_swap`: Dev Comments: `* @dev Every 10 swaps, we give the caller an extra token as an extra incentive to keep trading on T-Swap.`

```
swap_count++;
if (swap_count >= SWAP_COUNT_MAX) {
  swap_count = 0;
  outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000);
}
```

Third test `TSwapPool.t.sol`:

Proof of Code

```

    function testDepositSwapAudit() public {
        uint256 swapOutput = 1e17;

        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        vm.startPrank(user);
        poolToken.approve(address(pool), 1000e18);
        for (uint256 i = 0; i < 8; i++) {
            console.log("Swap number: ", i);
            pool.swapExactOutput(poolToken, weth, swapOutput, uint64(
                block.timestamp));
        }

        int256 startingWethBalance = int256(weth.balanceOf(address(pool
            )));
        int256 expectedWethBalanceChange = int256(-1) * int256(
            swapOutput);

        console.log("Swap number: 9");
        pool.swapExactOutput(poolToken, weth, swapOutput, uint64(block.
            timestamp));

```

```
24          // comment swap number 10 and the test will pass
25          // the purpose of this test is to show the invariant breaking
                due to the incentives coming in on the 10th swap
26          console.log("Swap number: 10"); // when incentive comes in
27          pool.swapExactOutput(poolToken, weth, swapOutput, uint64(block.
                timestamp));
28
29          vm.stopPrank();
30
31          uint256 endingWethBalance = weth.balanceOf(address(pool));
32          int256 actualWethBalanceChange = int256(endingWethBalance) -
                int256(startingWethBalance);
33
34          assertEq(actualWethBalanceChange, expectedWethBalanceChange);
35      }
```

Recommended Mitigation:

- Change the incentive logic, or remove it. It is breaking the pool ratio and the core invariant of the protocol.
- Users could be rewarded in TSwap tokens, a token made for the protocol, earned and mint initially only by swappers. Then a TSwap / Weth pool could be created to allow users to swap their TSwap tokens for Weth tokens.
- Lower the incentive amount to avoid any mathematical issue with the minimum required for a swap or / and for the minimum required to be a liquidity provider.

[S-H2] Grief attack on swap incentives, users == `swappers` can drained pools and the protocol won't survive the loss.

Description:

- The minimum required for `liquidity providers` to deposit liquidity of WETH is `1_000_000_000` wei.
- Every 10 swaps, `swappers` are rewarded with `1_000_000_000_000_000_000` wei of WETH.
- `Liquidity providers` needs to put in at least 0,000000001 weth <–> `swappers` get 1WETH

Just by swapping at least 10 times, `swappers` can drain the pool to 0.

Impact:

- Swappers can drain any pools to 0.

Proof of Concept:

Incentives code from `TSwapPool::_swap`:

```
swap_count++;
if (swap_count >= SWAP_COUNT_MAX) {
  swap_count = 0;
  outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000);
}
```

Minimum deposit checker from `TSwapPool::deposit`:

```
if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
    revert TSwapPool__WethDepositAmountTooLow(
        MINIMUM_WETH_LIQUIDITY,
        wethToDeposit
    );
}
```

The constant variable `MINIMUM_WETH_LIQUIDITY` from `TSwapPool.sol`:

```
uint256 private constant MINIMUM_WETH_LIQUIDITY = 1_000_000_000;
```

Recommended Mitigation:

- Same as the first issue of this repport, see below:
- Change the incentive logic, or remove it. It is breaking the pool ratio and the core invariant of the protocol.
- Users could be rewarded in TSwap tokens, a token made for the protocol, earned and mint initially only by swappers. Then a TSwap / Weth pool could be created to allow users to swap their TSwap tokens for Weth tokens.
- Lower the incentive amount to avoid any mathematical issue with the minimum required for a swap or / and for the minimum required to be a liquidity provider.

[S-H3] Unused parameter `deadline` in the `TSwapPool::deposit` function, disrupting logic that transaction should not go through if deadline is passed.

Description:

If someone sets the deadline to a block to come, they can deposit at the current block and can still deposit until the deadline of the block they set. Whatever a user expects something, in the actual code, it is always the case.

- Modifier `revertIfDeadlinePassed` is also not used in the `TSwapPool::deposit` function.
- `uint64 deadline` parameter is not used in the `TSwapPool::deposit` function.

Impact:

- If a user expects a deposit to fails due to the deadline, it won't. The transaction will go through. Whatever a user expects something, in the actual code, it is always the case.
- Users should be able to withdraw when they want. There is no vesting or minimum time to wait for a withdrawal that explains that.

Proof of Concept:

- The below is a simple test where `user` calls the `TSwapPool::deposit` function to deposit liquidity in a pool, using an already passed `deadline` at a time of where `deadline` is already `1000` seconds in the future.
- Deposit is successfull. But fails if the modifier checking for the deadline is set in the function signature like described in below `recommended mitigation`.

Proof Of Code

```
1     function testBlockTimestamp() public {
2         uint256 minimumDeposit = 1e18;
3         uint256 currentBalanceOfWethInPool = weth.balanceOf(address(
              pool));
4 
5         vm.getBlockTimestamp();
6         uint64 currentBlock = uint64(block.timestamp);
7         console.log("Current block timestamp: ", currentBlock);
8 
9         // increase timestamp
10        vm.warp(1000);
11        vm.assume(currentBlock != block.timestamp);
12        require(currentBlock != block.timestamp);
13 
14        // as pool and liquidity already exist (see initial setup
              config)
15        // prank with user for just depositing on the initial block.
              timestamp value when deposing at a time where the
16        // blockchain is already at a different block forward.
17        vm.startPrank(user);
18        weth.approve(address(pool), minimumDeposit);
19        poolToken.approve(address(pool), minimumDeposit);
20        pool.deposit(minimumDeposit, 0, minimumDeposit, currentBlock);
21        vm.stopPrank();
22 
23        //check deposit by checking that the liquidity pool has
              increased by the amount deposited by the user
24        assertEq(weth.balanceOf(address(pool)),
              currentBalanceOfWethInPool + minimumDeposit); // ok
25    }
```

Recommended Mitigation:

- Change the logic about the deadline, or remove it, or adapt functions and parameters correctly for its use.
- Review restrictions based on the `deadline` logic.
- Can consider following change for the `TSwapPool::deposit` function, using the deadline modifier:

```
    function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
    )
        external
+       revertIfDeadlinePassed(deadline)
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
```

[S-H4] Fees math in `TSwapPool::getInputAmountBasedOnOutput` is wrong, it is set to 10.13% instead of 0.03% as it should be.

Description:

`10000` is used instead of `1000` in the calculation of the return value.

- Might be due to the fact of dealing with magic numbers instead of constant variable names.

Impact:

- Calculate 10.13% instead of 0.03%
- Users are charged 10.13% instead of 0.03% of the swap amount each time.
- Users not getting the expected amount of tokens in return, because they are charged too much when paying the swap.
- The extra paid by users will be withdrawn by liquidity providers.

Proof of Concept:

- In the test case below, available that can be added to the protocol test suite `TSwapPool.t.sol`, here what is happening:

1. `Liquidity provider` provide liquidity for a pool at a ratio 1:1.
2. `user == swapper` wishing to swap 1 Token for 1 Weth, calls the `TSwapPool::swapExactOutput` function.
3. Checking balance before / after of the `swapper` and `pool`.
4. `swapper` balance paid 10.13 tokens instead of 1 for getting 1 weth.
5. `liquidity provider` withdraw all liquidity from the pool + the extra from miscalculation.

Proof of Code

```
    // test based on getInputAmountBasedOnOutput function and fee
       miscalculation
    function testCalculationFee() public {
        uint256 liquidityFromProvider = 100e18;

        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), liquidityFromProvider);
        poolToken.approve(address(pool), liquidityFromProvider);

        // deposit liquidity into a pool
        //    function deposit(uint256 wethToDeposit, uint256
           minimumLiquidityTokensToMint, uint256
        // maximumPoolTokensToDeposit, uint64 deadline)
        pool.deposit(liquidityFromProvider, 0, liquidityFromProvider,
           uint64(block.timestamp));

        vm.stopPrank(); // end prank liquidity provider

        // liquidity deposited checkers
        assertEq(pool.balanceOf(liquidityProvider),
           liquidityFromProvider);
        console.log("Pool balance of Token A from liquidity provider: "
           , poolToken.balanceOf(address(pool)));
        console.log("Pool balance of Weth from liquidity provider: ",
           weth.balanceOf(address(pool)));

        // start prank user
        vm.startPrank(user); // mint 100e18 in setup == 100 Token A -
           LP / 100 Weth
        uint256 userStartingWethBalance = weth.balanceOf(user);
        uint256 userStartingTokenABalance = poolToken.balanceOf(user);
        console.log("User starting Weth balance: ",
           userStartingWethBalance);
        console.log("User starting Token A balance: ",
           userStartingTokenABalance);
```

```
27
28          // approve token A to be used for swap
29          poolToken.approve(address(pool), type(uint256).max);
30
31          // get 1 Weth in exchange
32          pool.swapExactOutput(poolToken, weth, 1e18, uint64(block.
                timestamp));
33
34          uint256 userEndingWethBalanceAfterSwap = weth.balanceOf(user);
35          uint256 userEndingTokenABalanceAfterSwap = poolToken.balanceOf(
                user);
36
37          vm.stopPrank(); // end prank user
38
39          // Weth balance of Weth should have increase by one
40          assertEq(userEndingWethBalanceAfterSwap,
                userStartingWethBalance + 1e18);
41          console.log("User balance of Weth after swap: ",
                userEndingWethBalanceAfterSwap);
42
43          // As pool initial ratio is 1:1, when buying 1 Weth, 1 Token A
                should be removed from the pool
44          // so user balance of Token A should decrease by 1
45          assertEq(userEndingTokenABalanceAfterSwap,
                userStartingTokenABalance - 1e18);
46          console.log("User balance of Token A after swap: ",
                userEndingTokenABalanceAfterSwap);
47          // for 1 weth
48          // Token A before                100,000000000000000000
49          // Token A Expected after swap    99,000000000000000000
50          // Actual Token A balance after   89,868595686048043120
51          //WHAAAAAT nearly - 10.14 tokens instead of -1 ??
52
53          //so the pool is ...
54          console.log("Pool balance after user swap: ", poolToken.
                balanceOf(address(pool)));
55          // Pool Token A after swap       110,131404313951956880
56          // + 10.13....
57          console.log("Pool Weth after swap ", weth.balanceOf(address(
                pool)));
58          // Pool Weth after swap           99,000000000000000000
59
60          // Give place to the liquidity provider to rug the pool
                including the extra of Token A
61          vm.startPrank(liquidityProvider);
62          pool.withdraw(
63              pool.balanceOf(liquidityProvider),
64              1, // minWethToWithdraw
65              1, // minPoolTokensToWithdraw
66              uint64(block.timestamp)
67          );
```

```
68
69          console.log(
70              "Pool balance of Token A after liquidity provider withdraw
                    all: ", poolToken.balanceOf(address(pool))
71          );
72          console.log("Pool balance of Weth after liquidity provider
                withdraw all: ", weth.balanceOf(address(pool)));
73
74          // new balance of liquidity provider
75          console.log("Liquidity provider balance of Weth after withdraw
                all: ", weth.balanceOf(liquidityProvider));
76          console.log(
77              "Liquidity provider balance of Token A after withdraw all:
                    ", poolToken.balanceOf(liquidityProvider)
78          );
79          // loosing 1 weth but gained 10 token A instead of 1
80
81          assertEq(weth.balanceOf(address(pool)), 0);
82          assertEq(poolToken.balanceOf(address(pool)), 0);
83      }
```

Recommended Mitigation:

- Change `10000` for `1000` in the calculation of the return value.

```
1       {
2   -         return ((inputReserves * outputAmount) * 10000) / ((
        outputReserves - outputAmount) * 997);
3   +         return ((inputReserves * outputAmount) * 1_000) / ((
        outputReserves - outputAmount) * 997);
4       }
```

[S-H5] No slippage protection in `TSwapPool::swapExactOutput`, in the case of a price spike or huge movement of money in the liquidity pool users are not protected.

Description:

Price spike or massive transaction to the pool can occur any time and affects users on slippage as there is no slippage protection in `TSwapPool::swapExactOutput`, like it is existing in `TSwapPool::swapExactInput`.

Can lead to pay for a fraction of what the users should get in returns in an instant.

Impact:

- Price spike or massive transaction to the pool can occur any time and affects users on slippage.
- Also an MEV (Miner Extractable Value) attack can occur.

Recommended Mitigation:

- Needs an extra parameter in the function declaration checking for a maximum output and a new error statement for it.

```
+ error TSwapPool__OutputTooBig(uint256 actual, uint256 max);

    function swapExactOutput(
        IERC20 inputToken,
        IERC20 outputToken,
+       uint256 maxOutputAmount,
        uint256 outputAmount,
        uint64 deadline
    )
        public
        revertIfZero(outputAmount)
        revertIfDeadlinePassed(deadline)
        returns (uint256 inputAmount)
    {
        uint256 inputReserves = inputToken.balanceOf(address(this));
        uint256 outputReserves = outputToken.balanceOf(address(this));

        inputAmount = getInputAmountBasedOnOutput(outputAmount,
           inputReserves, outputReserves);

+       if (outputAmount > maxOutputAmount) {
+           revert TSwapPool__OutputTooBig(outputAmount,
    maxOutputAmount);
+       }

        _swap(inputToken, inputAmount, outputToken, outputAmount);
    }
```

[S-H6] In `TSwapPool::sellPoolTokens` function, users receives incorrect amount of tokens.

Description:

- Business logic, wrong function used in the return statement and wrong parameters.
- The function `swapExactOutput` is used instead of `swapExactInput` in the return statement.
- Wrong maths and lacks of slippage protection as stated in another issues of this report, leading to users suffering big losses when swapping tokens for `Weth`.
- The ratio 1:1 is not respected.

Impact:

- Due to the wrong logic as described above users are not getting the expected amount of tokens in return, because they are charged too much when paying the swap, suffering an enormous loss.

Proof of Concept:

- The below is a simple test where `user` calls the `TSwapPool::sellPoolTokens` function to sell pool tokens for Weth.
- Amount of tokens to sell is 3.
- 32 tokens disappeared in the process, instead of 3 tokens.

```
1     function testSellPoolTokens() public {
2         //result
3         //   amount sold: 3 ether
              300000000000000000
4         //   User balance of pool token before sell:
              10000000000000000000
5         //   User balance of weth before sell:
              10000000000000000000
6         //   User balance of pool token after sell:
              6897910225521926604 6  // 32 tokens disappeared instead of 3
7         //   User balance of weth after sell:
              10300000000000000000
8         //   Error: a == b not satisfied [uint]
9         //         Left: 6897910225521926604 6
10        //        Right: 9700000000000000000
11        uint256 liquidityFromProvider = 100e18;
12        uint256 amountOfTokenToSell = 3 ether;
13
14        vm.startPrank(liquidityProvider);
15        weth.approve(address(pool), liquidityFromProvider);
16        poolToken.approve(address(pool), liquidityFromProvider);
17
```

```
18          pool.deposit(liquidityFromProvider, 0, liquidityFromProvider,
                uint64(block.timestamp));
19
20          vm.stopPrank();
21
22          // Use another user trying to sell pool tokens for weth
23          vm.startPrank(user);
24          uint256 poolTokenBalanceBeforeSell = poolToken.balanceOf(user);
25          uint256 wethBalanceBeforeSell = weth.balanceOf(user);
26          console.log("User balance of pool token before sell: ",
                poolTokenBalanceBeforeSell);
27          console.log("User balance of weth before sell: ",
                wethBalanceBeforeSell);
28
29          poolToken.approve(address(pool), liquidityFromProvider);
30          pool.sellPoolTokens(amountOfTokenToSell);
31
32          uint256 poolTokenBalanceAfterSell = poolToken.balanceOf(user);
33          uint256 wethBalanceAfterSell = weth.balanceOf(user);
34          console.log("User balance of pool token after sell: ",
                poolTokenBalanceAfterSell);
35          console.log("User balance of weth after sell: ",
                wethBalanceAfterSell);
36
37          vm.stopPrank();
38
39          assertLt(poolTokenBalanceAfterSell, poolTokenBalanceBeforeSell)
                ; // ok
40          assertGt(wethBalanceAfterSell, wethBalanceBeforeSell); // ok
41          assertEq(wethBalanceAfterSell, wethBalanceBeforeSell +
                amountOfTokenToSell); // ok
42
43          //The below fail, 32 tokens disappeared... for 1 weth:
44          // |- emit log_named_uint(key: "      Left", val:
                6897910225521926 6046 [6.897e19])
45          // |- emit log_named_uint(key: "     Right", val:
                9700000000000000000 [9.7e19])
46          assertEq(poolTokenBalanceAfterSell, poolTokenBalanceBeforeSell
                - amountOfTokenToSell); // not ok
47      }
```

Recommended Mitigation:

- Please review other function issue of this report and adapt with the below recommandation:
  - Change the function used in the return statement:

```
1     function sellPoolTokens(uint256 poolTokenAmount) external  returns
          (uint256 wethAmount) {
2 -       return swapExactOutput(i_poolToken, i_wethToken,
      poolTokenAmount, uint64(block.timestamp));
3 +       return swapExactInput(i_poolToken, poolTokenAmount, i_wethToken
      , poolWethAmount, uint64(block.timestamp));
4     }
```

Medium

[S-M1] Using `ERC721::_mint()` can be dangerous, replace with `_safeMint()` instead and a `nonreentrant` guard from openzeppelin.

Description:

- In `TSwapPool::_addLiquidityMintAndTransfer`, using `ERC721::_mint()` can mint ERC721 tokens to addresses which don't support ERC721 tokens. If it happens, the tokens will be stuck forever.
- **Impact:**
- using `ERC721::_mint()` can mint ERC721 tokens to addresses which don't support ERC721 tokens (not able to use ERC721 even if owning some).
- `_safeMint()` can also be used maliciously to allows reentrancy attacks. To prevent it, a `nonreentrant` guard from openzeppelin can be used.

Recommended Mitigation:

- Use `_safeMint()` instead of `_mint()` for ERC721. To ensure that the receiver is a contract that can handle ERC721 tokens and able to use them.
- Use the modifier `nonReentrant` from openzeppelin to prevent reentrancy attacks as a re-entrancy guard, if a malicious contract tries to use `_safeMint()` to attack the protocol. https://docs.openzeppelin.com/contracts/5.x/api/utils#ReentrancyGuard

The below is recommended:

`TSwapPool::_addLiquidityMintAndTransfer`:

```
+ import {ReentrancyGuard} from "@openzeppelin/contracts/utils/
    ReentrancyGuard.sol";

- contract TSwapPool is ERC20
+ contract TSwapPool is ERC20, ReentrancyGuard
.
.
.
    function _addLiquidityMintAndTransfer(
        uint256 wethToDeposit,
        uint256 poolTokensToDeposit,
        uint256 liquidityTokensToMint
    )
        private
+       nonReentrant
    {
```

```
16 -         _mint(msg.sender, liquidityTokensToMint);
17 +         _safeMint(msg.sender, liquidityTokensToMint);
18
19 .
20 ,
21 ,
22      }
```

[S-M2] Fee-on-transfert, rebase logics from weird ERC20s and ERC777 tokens breaks the invariant, due to extra tokens minted or reentrancy attack.

Description:

Weird tokens, that can be rebase, or have a fee-on-transfert logic, can break the invariant of the protocol. There is also tokens with reentrancy problems.

Tokens having a compounding interest, or getting their number increased over time will break the pool ratio as the number of WETH doesn't increase over time. So the ratio of 1:1 will be broken.

A list of weird ERC20 https://github.com/d-xo/weird-erc20

Impact:

- Breaking pool ratio and possible reentrancy attack on various tokens.
- The `SafeErc20` library could not protect against all of those tokens.

Proof of Concept:

Example Using a basic weird token that use the `fee-on-transfer` logic:

- The fee is set to the number `1000`. So please, be careful to also look at last digits of numbers in the result below. Don't look only at first digits.

First here is the result:

```
1 //   Liquidity provider balance of shitcoin:
    10000000000000000000
2 //   User balance of shitcoin:
    99999999999999000
3 //   Liquidity provider balance of shitcoin after transfer to User:
    99000000000000000000
4 //   Balance of Weth in the pool:
    10000000000000000000
5 //   Balance of WeirdERC20 with fee on transfer:
    99999999999999000
6 //   Balance of Liquidity provider after deposit:
    98000000000000000000
7 //   Balance of Liquidity provider after withdraw:
    98999999999999998000
8 //   Balance of shitcoin in the pool after withdraw:
                                        0
```

Steps taken for the above result:

1. Create the token, mint it to the `liquidity provider`. -> `10000000000000000000` tokens.
2. Transfer the equivalent of `1e18` of the token to `user`. -> To show in a more easy way, that `user` is not getting the expected amount of tokens. So the same will also happen if the `user` is the liquidity pool. -> `99999999999999000` tokens received.
3. Check the balance of the `liquidity provider` after the transfer. -> `99000000000000000000` tokens.
4. Create a liquidity pool and deposit tokens at a ratio of 1:1.
5. Check the balance of the pool after deposit -> `99999999999999000` tokens.
6. Balance of the `liquidity provider` after deposit -> `98000000000000000000` tokens.
7. Withdraw all liquidity from the pool.
8. Check the balance of the `liquidity provider` after withdraw -> `98999999999999998000` tokens.

In the process, due to fees, when a transfer occur, tokens amount is decreasing by `1000`, `WETH` doesn't decrease by default, so the ratio is broken.

- requirements for the above test: requires an additional file to place in `./test/mocks/` called `ERC20MockFeeOnTransfer.sol` containing the below code:

ERC20MockFeeOnTransfer code

```
// Copyright (C) 2017, 2018, 2019, 2020 dbrock, rain, mrchico, d-xo
// SPDX-License-Identifier: AGPL-3.0-only

// adapted from https://github.com/d-xo/weird-erc20/blob/main/src/TransferFee.sol

pragma solidity >=0.6.12;

contract Math {
// --- Math ---
function add(uint256 x, uint256 y) internal pure returns (uint256 z) {
require((z = x + y) >= x);
}

    function sub(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require((z = x - y) <= x);
    }

}

contract WeirdERC20 is Math {
// --- ERC20 Data ---
string public name;
string public symbol;
uint8 public decimals;
uint256 public totalSupply;
bool internal allowMint = true;

    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    event Approval(address indexed src, address indexed guy, uint256 wad);
    event Transfer(address indexed src, address indexed dst, uint256 wad);

    // --- Init ---
    constructor(string memory _name, string memory _symbol, uint8 _decimalPlaces) public {
        name = _name;
        symbol = _symbol;
        decimals = _decimalPlaces;
```

```
39      }
40
41      // --- Token ---
42      function transfer(address dst, uint256 wad) public virtual returns
            (bool) {
43          return transferFrom(msg.sender, dst, wad);
44      }
45
46      function transferFrom(address src, address dst, uint256 wad) public
             virtual returns (bool) {
47          require(balanceOf[src] >= wad, "WeirdERC20: insufficient-
                balance");
48          if (src != msg.sender && allowance[src][msg.sender] != type(
                uint256).max) {
49              require(allowance[src][msg.sender] >= wad, "WeirdERC20:
                    insufficient-allowance");
50              allowance[src][msg.sender] = sub(allowance[src][msg.sender
                    ], wad);
51          }
52          balanceOf[src] = sub(balanceOf[src], wad);
53          balanceOf[dst] = add(balanceOf[dst], wad);
54          emit Transfer(src, dst, wad);
55          return true;
56      }
57
58      function approve(address usr, uint256 wad) public virtual returns (
            bool) {
59          allowance[msg.sender][usr] = wad;
60          emit Approval(msg.sender, usr, wad);
61          return true;
62      }
63
64      function mint(address to, uint256 _amount) public {
65          require(allowMint, "WeirdERC20: minting is off");
66
67          _mint(to, _amount);
68      }
69
70      function _mint(address account, uint256 amount) internal virtual {
71          require(account != address(0), "WeirdERC20: mint to the zero
                address");
72
73          totalSupply += amount;
74          unchecked {
75              // Overflow not possible: balance + amount is at most
                    totalSupply + amount, which is checked above.
76              balanceOf[account] += amount;
77          }
78          emit Transfer(address(0), account, amount);
79      }
80
```

```
81      function burn(address from, uint256 _amount) public {
82          _burn(from, _amount);
83      }
84
85      function _burn(address account, uint256 amount) internal virtual {
86          require(account != address(0), "WeirdERC20: burn from the zero
                address");
87
88          uint256 accountBalance = balanceOf[account];
89          require(accountBalance >= amount, "WeirdERC20: burn amount
                exceeds balance");
90          unchecked {
91              balanceOf[account] = accountBalance - amount;
92              // Overflow not possible: amount <= accountBalance <=
                    totalSupply.
93              totalSupply -= amount;
94          }
95
96          emit Transfer(account, address(0), amount);
97      }
98
99      function toggleMint() public {
100         allowMint = !allowMint;
101     }
102
103 }
104
105 contract ERC20MockFeeOnTransfer is WeirdERC20 {
106 uint256 private fee;
107
108     // --- Init ---
109     constructor(
110         string memory _name,
111         string memory _symbol,
112         uint8 _decimalPlaces,
113         uint256 _fee
114     )
115         WeirdERC20(_name, _symbol, _decimalPlaces)
116     {
117         fee = _fee;
118     }
119
120     // --- Token ---
121     function transferFrom(address src, address dst, uint256 wad) public
            override returns (bool) {
122         require(balanceOf[src] >= wad, "ERC20MockFeeOnTransfer:
                insufficient-balance");
123         // don't worry about allowances for this mock
124         //if (src != msg.sender && allowance[src][msg.sender] != type(
                uint).max) {
125         //    require(allowance[src][msg.sender] >= wad, "
```

```
                ERC20MockFeeOnTransfer insufficient-allowance");
126         //    allowance[src][msg.sender] = sub(allowance[src][msg.
                sender], wad);
127         //}
128
129         balanceOf[src] = sub(balanceOf[src], wad);
130         balanceOf[dst] = add(balanceOf[dst], sub(wad, fee));
131         balanceOf[address(0)] = add(balanceOf[address(0)], fee);
132
133         emit Transfer(src, dst, sub(wad, fee));
134         emit Transfer(src, address(0), fee);
135
136         return true;
137     }
138
139 }
```

- Add import and contract declaration / creation in the `TSwapPool.t.sol`, see below:

```
1  import { Test, console } from "forge-std/Test.sol";
2  import { TSwapPool } from "../../src/PoolFactory.sol";
3  import { ERC20Mock } from "@openzeppelin/contracts/mocks/token/
       ERC20Mock.sol";
4  import { IERC20 } from "@openzeppelin/contracts/interfaces/IERC20.sol";
5  + import { ERC20MockFeeOnTransfer } from "../mocks/
       ERC20MockFeeOnTransfer.sol"; // ERC20MockFeeOnTransfer.sol for weird
       // token test
6
7  contract TSwapPoolTest is Test {
8      TSwapPool pool;
9      ERC20Mock poolToken;
10     ERC20Mock weth;
11 +    ERC20MockFeeOnTransfer shitcoin; // ERC20MockFeeOnTransfer.sol for
       weird token test
12
13     address liquidityProvider = makeAddr("liquidityProvider");
14     address user = makeAddr("user");
15
16     function setUp() public {
17         poolToken = new ERC20Mock();
18         weth = new ERC20Mock();
19
20 +        shitcoin = new ERC20MockFeeOnTransfer("FeeOnTransferCoin", "
       SHIT", 18, 1000); // ERC20MockFeeOnTransfer.sol
21             // for
22             // weird token test || parameters : name, symbol, decimals
                   number, fee when transfer occur
23
24         pool = new TSwapPool(address(poolToken), address(weth), "
               LTokenA", "LA");
25
26
```

```
        weth.mint(liquidityProvider, 200e18);
        poolToken.mint(liquidityProvider, 200e18);

        weth.mint(user, 100e18);
        poolToken.mint(user, 100e18);
    }
.
.
.
}
```

- Then add the following test case to `TSwapPool.t.sol`:

Test code

```
    function testWeirdErc20WithFee() public {
        // fee set to 1000 at contract creation in setup

        //   Liquidity provider balance of shitcoin:
                                    10000000000000000000
        //   User balance of shitcoin:
                                                  999999999999999000
        //   Liquidity provider balance of shitcoin after transfer to
           User:  990000000000000000
        //   Balance of Weth in the pool:
                                               1000000000000000000
        //   Balance of WeirdERC20 with fee on transfer:
                                    999999999999999000
        //   Balance of Liquidity provider after deposit:
                                  980000000000000000
        //   Balance of Liquidity provider after withdraw:
                                 989999999999998000
        //   Balance of shitcoin in the pool after withdraw:
                                               0
        uint256 amountUsedForInteraction = 1 ether;

        vm.startPrank(liquidityProvider);
        shitcoin.mint(address(liquidityProvider), 100e18);
        // balance of shitcoin in liquidity provider
        uint256 startingShitcoinBalance = shitcoin.balanceOf(
           liquidityProvider);
        console.log("Liquidity provider balance of shitcoin: ",
           startingShitcoinBalance);

        // transfer shitcoin token to user and check user balance,
           transfer 2e18 (2 ether value)
        shitcoin.transferFrom(address(liquidityProvider), address(user)
           , amountUsedForInteraction);
        console.log("User balance of shitcoin: ", shitcoin.balanceOf(
           user));
        // 1 ether instead of 2 has been transferred, due to 1 ether
```

```
            fee.

        // check liquidity provider after transfer
        console.log(
            "Liquidity provider balance of shitcoin after transfer to
                User: ", shitcoin.balanceOf(liquidityProvider)
        );

        // now let deposit into a pool and withdraw from the pool
        // create the pool
        pool = new TSwapPool(address(shitcoin), address(weth), "SHIT/
            WETH", "SW");
        // approve token
        weth.approve(address(pool), amountUsedForInteraction);
        shitcoin.approve(address(pool), amountUsedForInteraction);

        //capture balance before deposit, as in this test a transfer to
             external user is also made, for easy
        // understanding.
        uint256 balanceBeforeDeposit = shitcoin.balanceOf(address(
            liquidityProvider));

        //deposit to the pool
        pool.deposit(amountUsedForInteraction, 0,
            amountUsedForInteraction, uint64(block.timestamp));

        // check deposited tokens
        console.log("Balance of Weth in the pool: ", weth.balanceOf(
            address(pool)));
        console.log("Balance of WeirdERC20 with fee on transfer: ",
            shitcoin.balanceOf(address(pool)));

        // capture the actual balance of shitcoin after deposit
        uint256 shitcoinBalanceAfterDeposit = shitcoin.balanceOf(
            address(liquidityProvider));
        console.log("Balance of Liquidity provider after deposit: ",
            shitcoinBalanceAfterDeposit);

        // withdraw tokens from the pool
        pool.withdraw(
            pool.balanceOf(liquidityProvider),
            1, // minWethToWithdraw
            1, // minPoolTokensToWithdraw
            uint64(block.timestamp)
        );

        // check if initial shitcoin balance and ending balance is
            equal
        uint256 shitcoinBalanceAfterWithdraw = shitcoin.balanceOf(
            liquidityProvider);
        console.log("Balance of Liquidity provider after withdraw: ",
```

```
            shitcoinBalanceAfterWithdraw);

        // pool balance
        console.log("Balance of shitcoin in the pool after withdraw: ",
            shitcoin.balanceOf(address(pool)));
        vm.stopPrank();

        assertLt(shitcoinBalanceAfterWithdraw, balanceBeforeDeposit);
    }
```

Recommended Mitigation:

- Create a whitelist / blacklist array of tokens that can be used or not used in the protocol.
- restrict the use of such token if it is not well known token.
- Need manual review of such tokens.
- If a token is reviewed, whitelist it for other pool creation
- or stick to well known tokens only, like the AAVE protocol

Low

[S-L1] `PoolCreated` event should be `indexed` for better searchability and filtering.

Description:

- Since it is an AMM, better to index event for third party app to track the state of pools.
- `PoolFactory` contract is not indexed, which makes it difficult to search and filter for specific transactions and state changes in case you need to.
- Major events in `TSwapPool` using 3 parameters, are `indexed`.
- Note: Indexed event are stored more efficiently.

Impact:

- Hard to retrieve and filter events. It is a low severity issue, but it is a good practice to index events for better searchability and filtering.

Recommended Mitigation:

Add the `indexed` keyword:

```
1 -    event PoolCreated(address tokenAddress, address poolAddress);
2
3 +    event PoolCreated(address tokenAddress, address poolAddress)
    indexed;
```

[S-L2] The returned value `uint256 output` declared in the function signature doesn't exist in the `TSwapPool::swapExactInput` function,

Description:

- The returned value `uint256 output` doesn't exist in the `TSwapPool::swapExactInput` function, but it is declared in the function signature.

Impact:

- Returned value will always be 0.
- The impact is low, but if logic is to have it output a value, it should be declared in the function.

Recommended Mitigation:

- I think the value that should be returned is `outputAmount` instead of `output`. If it is the case it also needs to be set inside the function itself to be returned.

[S-L3] `TSwapPool::LiquidityAdded` event parameters in wrong order, returned information is erroneous.

Description:

`uint256 poolTokensDeposited` and `uint256 wethDeposited` are inverted.

- Returned information is erroneous.

Emitted event:

```
1  emit LiquidityAdded(msg.sender, poolTokensToDeposit, wethToDeposit);
```

Declared Event:

```
1  -    event LiquidityAdded(address indexed liquidityProvider, uint256
       wethDeposited, uint256 poolTokensDeposited);
2  +    event LiquidityAdded(address indexed liquidityProvider, uint256
       poolTokensDeposited, uint256 wethDeposited);
```

Informational

[S-Info1] Magic Numbers

Description:

All number literals should be replaced with constants. This makes the code more readable and easier to maintain. Numbers without context are called “magic numbers”.

Recommended Mitigation:

- Replace all magic numbers with constants.

`TSwapPool.sol` contract:

```
+       uint256 public constant WHATEVER_IS_997 = 997;
+       uint256 public constant WHATEVER_IS_1000 = 1000;
+       uint256 public constant WHATEVER_IS_10000 = 10000;
+       uint256 public constant WHATEVER_IS_1_000_000_000_000_000_000 = 1_000_000_000_000_000_000;
+       uint256 public constant WHATEVER_IS_1e18 = 1e18;
.
.
.
-       uint256 inputAmountMinusFee = inputAmount * 997;
-       uint256 denominator = (inputReserves * 1000) + inputAmountMinusFee;
+       uint256 inputAmountMinusFee = inputAmount * WHATEVER_IS_997;
+       uint256 denominator = (inputReserves * WHATEVER_IS_1000) + inputAmountMinusFee;
.
.
.
-        return ((inputReserves * outputAmount) * 10000) / ((outputReserves - outputAmount) * 997);
+        return ((inputReserves * outputAmount) * WHATEVER_IS_10000) / ((outputReserves - outputAmount) * WHATEVER_IS_997);
.
.
.
-       outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000);
+       outputToken.safeTransfer(msg.sender, WHATEVER_IS_1_000_000_000_000_000_000);
.
.
.
-       getOutputAmountBasedOnInput(1e18, i_wethToken.balanceOf(address(this)), i_poolToken.balanceOf(address(this)));
```

```
27 +         getOutputAmountBasedOnInput(WHATEVER_IS_1e18, i_wethToken.
       balanceOf(address(this)), i_poolToken.balanceOf(address(this)));
28 .
29 .
30 .
31 -         getOutputAmountBasedOnInput(1e18, i_poolToken.balanceOf(
       address(this)), i_wethToken.balanceOf(address(this)));
32 +         getOutputAmountBasedOnInput(WHATEVER_IS_1e18, i_poolToken.
       balanceOf(address(this)), i_wethToken.balanceOf(address(this)));
```

[S-Info2] The function `swapExactInput()` is never used in `TSwapPool` contract, remove it or change its use case.

Description:

The function `swapExactInput()` is never used and should be removed or changed to `external` to be used. Or change the logic of its use case depending on the intended use of it.

```
1     function swapExactInput(
2         IERC20 inputToken,
3         uint256 inputAmount,
4         IERC20 outputToken,
5         uint256 minOutputAmount,
6         uint64 deadline
7     )
8 -       public
9 +       external
10        revertIfZero(inputAmount)
11        revertIfDeadlinePassed(deadline)
12        returns (uint256 output)
13    {
14        uint256 inputReserves = inputToken.balanceOf(address(this));
15        uint256 outputReserves = outputToken.balanceOf(address(this));
16
17        uint256 outputAmount = getOutputAmountBasedOnInput(
18            inputAmount,
19            inputReserves,
20            outputReserves
21        );
22
23        if (outputAmount < minOutputAmount) {
24            revert TSwapPool__OutputTooLow(outputAmount,
                  minOutputAmount);
25        }
26
```

```
27          _swap(inputToken, inputAmount, outputToken, outputAmount);
28      }
```

[S-Info3] Solidity 0.8.20 by default target EVM version to `Shanghai` by default, `PUSH0` is not supported by all chains

Description:

Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes. Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.

- Since the protocol is meant to be deployed on Ethereum, it is not an issue, more of an information just in case.

[S-Info4] The variable `poolTokenReserves` is never used in `TSwapPool::deposit` function, remove it or change its use case.

Description:

The variable `poolTokenReserves` is never used in `TSwapPool::deposit` function and should be removed or changed integrated following the logic explained in comments belows it. Or change the logic of it depending on the intended use of it.

```
1           if (totalLiquidityTokenSupply() > 0) {
2               uint256 wethReserves = i_wethToken.balanceOf(address(this))
                    ;
3   -           uint256 poolTokenReserves = i_poolToken.balanceOf(address(
        this)); // @audit unused local variable
4               // Our invariant says weth, poolTokens, and liquidity
                    tokens must always have the same ratio after the
5               // initial deposit
6               // poolTokens / constant(k) = weth
7               // weth / constant(k) = liquidityTokens
8               // aka...
9               // weth / poolTokens = constant(k)
```

```
10              // To make sure this holds, we can make sure the new
                    balance will match the old balance
11              // (wethReserves + wethToDeposit) / (poolTokenReserves +
                    poolTokensToDeposit) = constant(k)
12              // (wethReserves + wethToDeposit) / (poolTokenReserves +
                    poolTokensToDeposit) =
13              // (wethReserves / poolTokenReserves)
14              //
15              // So we can do some elementary math now to figure out
                    poolTokensToDeposit...
16              // (wethReserves + wethToDeposit) / poolTokensToDeposit =
                    wethReserves
17              // (wethReserves + wethToDeposit)  = wethReserves *
                    poolTokensToDeposit
18              // (wethReserves + wethToDeposit) / wethReserves  =
                    poolTokensToDeposit
19              uint256 poolTokensToDeposit =
                    getPoolTokensToDepositBasedOnWeth(
20                  wethToDeposit
21              );
22              if (maximumPoolTokensToDeposit < poolTokensToDeposit) {
23                  revert TSwapPool__MaxPoolTokenDepositTooHigh(
24                      maximumPoolTokensToDeposit,
25                      poolTokensToDeposit
26                  );
27              }
28
29              // We do the same thing for liquidity tokens. Similar math.
30              liquidityTokensToMint =
31                  (wethToDeposit * totalLiquidityTokenSupply()) /
32                  wethReserves;
33              if (liquidityTokensToMint < minimumLiquidityTokensToMint) {
34                  revert TSwapPool__MinLiquidityTokensToMintTooLow(
35                      minimumLiquidityTokensToMint,
36                      liquidityTokensToMint
37                  );
38              }
39              _addLiquidityMintAndTransfer(
40                  wethToDeposit,
41                  poolTokensToDeposit,
42                  liquidityTokensToMint
43              );
44          }
```

[S-Info5] Zero-check missing in constructors.

Description:

Missing zero-check in `PoolFactory::constructor` and `TSwapPool::constructor` on `wethToken` parameter.

`TSwapPool::constructor`:

```
    constructor(
        address poolToken,
        address wethToken,
        string memory liquidityTokenName,
        string memory liquidityTokenSymbol
    ) ERC20(liquidityTokenName, liquidityTokenSymbol) {
+       if (poolToken == address(0) || wethToken == address(0)) {
+           revert TSwapPool__InvalidToken();
*       }
        i_wethToken = IERC20(wethToken);
        i_poolToken = IERC20(poolToken);
    }
```

`PoolFactory::constructor`:

```
+   error PoolFactory__ZeroAddress();

    constructor(address wethToken) {
+       if (wethToken == address(0)) {
+           revert PoolFactory__ZeroAddress();
+       }
        i_wethToken = wethToken;
    }
```

[S-Info6] IERC20 interface duplicate, same function available in `ERC20.sol` of OpenZeppelin using the latest version of solidity.

Description:

Importing 2 `IERC20` interfaces, one from `OpenZeppelin` in `TSwapPool` contract and one from `Forge-Std` library in `PoolFactory` contract. The one from `OpenZeppelin` is enough and should be used.

`PoolFactory::IERC20`, the below function are available in the `ERC20.sol` library from `OpenZeppelin`:

```
    /// @notice Returns the name of the token.
    function name() external view returns (string memory);

    /// @notice Returns the symbol of the token.
    function symbol() external view returns (string memory);

    /// @notice Returns the decimals places of the token.
    function decimals() external view returns (uint8);
```

Change could be made where `name()`, `symbol()` and `decimals()` are used in `PoolFactory` contract instead.

- Also the one from `Forge-Std` use a different solidity versions than others libraries used in the project.

```
    - [0.8.20](src/PoolFactory.sol#L15)
    - [0.8.20](src/TSwapPool.sol#L15)
-   - [>=0.6.2](lib/forge-std/src/interfaces/IERC20.sol#L2)
    - [^0.8.20](lib/openzeppelin-contracts/contracts/interfaces/draft-
      IERC6093.sol#L3)
    - [^0.8.20](lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.
      sol#L4)
    - [^0.8.20](lib/openzeppelin-contracts/contracts/token/ERC20/IERC20
      .sol#L4)
    - [^0.8.20](lib/openzeppelin-contracts/contracts/token/ERC20/
      extensions/IERC20Metadata.sol#L4)
    - [^0.8.20](lib/openzeppelin-contracts/contracts/token/ERC20/
      extensions/IERC20Permit.sol#L4)
    - [^0.8.20](lib/openzeppelin-contracts/contracts/token/ERC20/utils/
      SafeERC20.sol#L4)
    - [^0.8.20](lib/openzeppelin-contracts/contracts/utils/Address.sol#
      L4)
    - [^0.8.20](lib/openzeppelin-contracts/contracts/utils/Context.sol#
      L4)
```

[S-Info7] Wrong function used for grabbing the token symbol.

Description:

Use of `.name()` instead of `.symbol()` in `PoolFactory::createPool` function, when concataining token symbols strings.

```
1 -         string memory liquidityTokenSymbol = string.concat("ts", IERC20
      (tokenAddress).name());
2 -         string memory liquidityTokenSymbol = string.concat("ts", IERC20
      (tokenAddress).symbol());
```

Change could be made where `name()`, `symbol()` and `decimals()` are used in `PoolFactory` contract instead.

[S-Info8] `PoolFactory::PoolFactory__PoolDoesNotExist` unused error.

Description:

Unused error declared `PoolFactory::PoolFactory__PoolDoesNotExist`, in `PoolFactory` contract.

```
1     error PoolFactory__PoolDoesNotExist();
```

[S-Info9] Changement of "state", even if not state variables should be set before an external call to follow CEI.

Description:

In the else statement of the `TSwapPool::deposit` function, the `liquidityTokensToMint` variable should be set before the `_addLiquidityMintAndTransfer` function call to follow the Checks-Effects-Interactions pattern. Even if there is no re-entrancy danger in this case, it is good habits.

```
1  else {
2              // This will be the "initial" funding of the protocol. We
                  are starting from blank here!
3              // We just have them send the tokens in, and we mint
                  liquidity tokens based on the weth
4  +           liquidityTokensToMint = wethToDeposit;
5              _addLiquidityMintAndTransfer(wethToDeposit,
                  maximumPoolTokensToDeposit, wethToDeposit);
6  -           liquidityTokensToMint = wethToDeposit;
7          }
```

[S-Info10] `TSwapPool::totalLiquidityTokenSupply` function should be `external` instead of `public`.

Description:

`TSwapPool::totalLiquidityTokenSupply` function should be `external` instead of **`public`**.

```
1  -    function totalLiquidityTokenSupply() public view returns (uint256)
       {
2  +    function totalLiquidityTokenSupply() external view returns (uint256
       ) {
3              return totalSupply();
4        }
```

Gas

- Included in above findings.