TSwap Audit Report

Version 1.0

NGU Associates

November 5, 2024

TSwap Pool Audit Report

NGU Associates

November 5, 2024

Prepared by: SADFrancis Lead Auditor: - Sean Francis

Table of Contents

- [M-1] `TSwapPool::deposit()` neglecting to use the `deadline` parameter misleads a user to believe they can cause a revert!
- [M-2] The values stored in private `TSwapPool::swap_count` and constant `TSwapPool::SWAP_COUNT_MAX` used for tracking the number of `TSwapPool::_swap()` calls before gifting a Pool Token ARE NOT PRIVATE
- [M-3] Rebasing and fee on transfer Weird ERC-20 tokens will break the invariant!
- [L-1] `TSwapPool::_addLiquidityMintAndTransfer()`'s event `LiquidityAdded` has incorrect parameters
- [L-2] `TSwapPool::swapExactInput()` returns `uint256 output` but is never defined in the function thus giving an incorrect return value
- [I-1] The error `PoolFactory::PoolFactory__PoolDoesNotExist(address tokenAddress)` is not used
- [I-2] Neither the `PoolFactory::constructor` nor the `TSwapPool::constructor` have a zero check for the address parameters
- [I-3] The string `liquidityTokenSymbol` assigned within `PoolFactory::createPool` should use the `ERC20` function `symbol()` instead of `name()`
- [I-4] Event `TSwapPool::Swap` should have 3 indexed events when containing 3 or more parameters
- [I-5] In `TSwapPool::deposit()`, `MINIMUM_WETH_LIQUIDITY` is an constant that does not need to be emitted in the error `TSwapPool__WethDepositAmountTooLow`
- [I-6] In `TSwapPool::deposit()`, the uint256 variable `poolTokenReserves` is unused and should be taken out to save on gas
- [I-7] In `TSwapPool::deposit()`, the uint256 variable `liquidityTokensToMint` is not a state variable and doesn't follow CEI
- [I-8] `TSwapPool::getOutputAmountBasedOnInput` among other functions contain literal numbers and should be replaced with literal CONSTANTS
- [I-9] `TSwapPool::swapExactInput()` does not have natspec written out
- [I-10] `TSwapPool::swapExactInput()` should be changed to `external` to save on gas since no contract functions calls it
- [I-11] `TSwapPool::totalLiquidityTokenSupply()` should be `external view` to save on gas since no contract function calls it

Protocol Summary

TSwap is a protocol that allows users to create their own pools of two tokens maintaining a specific ratio as traders trade between the two assets of the pool. Pool contract, with each trade takes a 0.3% fee of the value of the trade and generously gifts a lucky user once certain conditions are met.

Disclaimer

The NGU Associates 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

Commit Hash: e643a8d4c2c802490976b538dd009b351b1c8dda

Scope

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

Solc Version: 0.8.20 Chain(s) to deploy contract to: Ethereum Tokens: Any ERC20 token

Roles

1. Liquidity Providers: Users who have liquidity deposited into the pools. Their shares are represented by the LP ERC20 tokens. They gain a 0.3% fee every time a swap is made.

2. Users: Users who want to swap tokens.

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	2
Info/Gas	11
Total	20

Findings

[H-1] `TSwapPool::_swap()`'s generous Gifting Tokens for reaching 10 swaps breaks Protocol Invariant of maintaining Pool Ratios

Description: The core invariant of the protocol is that for any pool, the ratio of the pool token and WETH should be:

`x * y = k` - x = Token Balance X - y = Token Balance Y - k = The constant ratio between X & Y

However, within `TSwapPool::_swap()` function, the global private uint256 variable `TSwapPool::swap_count` increments by 1 (starting from 0) each time the function is called. Once the value equals the private constant `TSwapPool::SWAP_COUNT_MAX` (which is currently set to 10), the lucky `_swap()` function caller receives an extra token (1e18 wei) of whatever is being swapped to.

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

Impact: This generous gifting mechanism breaks the protocol invariant of maintaining the pool ratio. A malicious user could also drain the protocol by initiating a ton of swaps to collect the gifted tokens.

Proof of Concept

1. A user swaps 10 times, and is gifted an extra pool token
2. A user continues till the pool is drained

Proof of Code

We will implement the following test in `test/unit/TSwapPool.t.sol`:

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


        uint256 outputWeth = 1e17;
        int256 startingY = int256(weth.balanceOf(address(pool)));
        int256 expectedDeltaY= int256(-1) *int256(outputWeth);

        poolToken.mint(user,100e18);
        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);

        for(uint i =0;i < 10; ++i){
            pool.swapExactOutput(poolToken,weth,outputWeth,uint64(block
                .timestamp));
        }
        vm.stopPrank();

        uint256 endingY = weth.balanceOf(address(pool));
        int256 actualDeltaY = int256(endingY) - startingY;

        assertEq(actualDeltaY,expectedDeltaY, "we assert that the
            change in Y is less than expected due to Invariant break" );


    }
```

And within the foundry environment, run:

```
forge test --mt testInvariantBroken -vvvv
```

The test will fail. This test has the user swap poolTokens for Weth 10 times to reach the critical swap count to be gifted a token. Once that happens, the invariant breaks!

Recommended Mitigation: Take it out!

```
-swap_count++;
-        if (swap_count >= SWAP_COUNT_MAX) {
```

```
3 -             swap_count = 0;
4 -             // @report-written replace literal numbers with magic
      number constants to be more descriptive
5 -             outputToken.safeTransfer(msg.sender, 1
      _000_000_000_000_000_000);
6 -         }
```

You'll need to look for another accounting mechanism to maintain the ratio formula while rewarding your users. Consider setting aside a portion of the fees to gift to users.

[H-2] `TwSwapPool::getInputAmountBasedOnOutput()` has a mathematical error that drives sub 1 percent protocol fee to over 90 percent

Description: The pure view function contains this block:

```
1             ((inputReserves * outputAmount) * 10000) /
2             ((outputReserves - outputAmount) * 997);
```

Where the variables are defined according to the formula as follows:

```
1 formula: xdy/(y - dy)
2 inputReserves: x
3 outputAmount: dy
4 outputReserves: y
5 outputAmount: dy
```

The magic numbers multiplied are meant to adjust the formula to account for a 0.3% fee when the Swap takes place (997/1000). However, there is `10_000` instead of `1000` which ends up driving the fee to 90.03%!

Impact: This function is called in the `TSwapPool::SwapExactInput"()` and thus will take a 90.03% fee from the user!

Recommended Mitigation: Save `997` and `1000` to global CONSTANTS and use them in your functions. Magic numbers are bad!

[H-3] `TSwapPool::swapExactOutput()` has no slippage protection when calling the internal `_swap` function causing users to receive unintended amounts of tokens swapped

Description: The function calculates the exact amount of output token to swap to the input token and calls the `_swap` function internally but if the price changes due to some other transaction that is processed before this transaction is processed, the price could change how many output tokens are needed.

Impact: For example, if the input token is `WETH` and the output token is `DAI` and the ETH price skyrockets, you'll end up spending a lot more DAI to complete the transaction.

Proof of Concept: 1. The price of WETH == 1000 USDC 2. User inputs the following parameters:

```
swapExactOutput(
        /*IERC20 inputToken*/ USDC,
        /*IERC20 outputToken*/ WETH,
        /*uint256 outputAmount*/ 1,
        /*uint64 deadline but this value isn't used*/ 10,
    )
```

With the intention of buying 1 WETH for current market price of 1000 USDC at the time submitting the transaction. 3. The transaction is sent and is left pending in the mempool. In that time, a massive market price correction takes place, driving the price of WETH from 1000 USDC to 10_000 USDC. 4. The user now accidentally spends 9000 more than tokens intended.

Proof of Code:

```
function testShowswapExactOutputHasNoSlippageProtection() public {

        // Prep attacker
        address attacker = makeAddr("attacker");
        poolToken.mint(attacker, 100e18);
        vm.startPrank(attacker);
        poolToken.approve(address(pool), 100e18);
        weth.approve(address(pool), 24962443665498247371);

        // Provide Liquidity
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(50e18, 50e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        // simulate user that wants to swap
        vm.prank(user);
        poolToken.approve(address(pool), 10e18);
        uint256 expected = 2_046_957_198_124_987_206; // this is what
           the user expects to receive swapping 1e17 Wei pool tokens


        // SUDDENLY THE ATTACKER ATTACKS! OPERATION SANDWICH ATTACK:
           COMMENCE
        // BUY WETH TO DRIVE UP THE PRICE!
        vm.prank(attacker);
        pool.swapExactInput(
            poolToken,
            100e18,
            weth,
```

```
            0,
            uint64(block.timestamp)
        );

        // The user's transaction in purchasing Weth with the Pool Token finally goes through
        // BUT THE PRICE OF WETH JUST PUMPED!
        vm.prank(user);
        uint256 actual = pool.swapExactOutput(
            poolToken,
            weth,
            1e17,
            uint64(block.timestamp)
        );

        // SANDWICH ATTACK THE USER!
        // SELL WETH FOR POOL TOKEN AT HIGHER PRICES
        vm.startPrank(attacker);
        pool.swapExactInput(
            weth,
            weth.balanceOf(attacker),
            poolToken,
            0,
            uint64(block.timestamp)
        );

        console.log("Expected ETH amount: ", expected);
        console.log("Expected ETH amount x 3: ", expected * 3); // x4 = 8187828792499948824, so we'll drop down to x3 to make a statement
        console.log("Actual ETH amount:", actual); // 8_044_164_501_343_340_225
        console.log("Pool Token Address:", address(poolToken));
        console.log("Weth Address:", address(weth));

        assert(actual > expected * 3);
        assert(weth.balanceOf(attacker) >= 0);
        assert(poolToken.balanceOf(attacker) > 100e18); // attacker had 0 weth and 100e18 PT initially check if he has more now
    }
```

Where with foundry, you can run the command

```
forge test --mt testShowswapExactOutputHasNoSlippageProtection -vvvv
```

To show the user unwittingly paying nearly 4x for WETH than expected.

Recommended Mitigation: There should be a maxOutput parameter to use as a conditional to wrap around the `_swap` function to prevent the any unintentional overexpenditures.

```
    function swapExactOutput(
        IERC20 inputToken,
        IERC20 outputToken,
        uint256 outputAmount,
+       uint256 maxInputAmount,
        uint64 deadline
    )
.
.
.
        inputAmount = getInputAmountBasedOnOutput(outputAmount,
           inputReserves,outputReserves);
+        if (inputAmount > maxInputAmount){
+            revert;
+        }
        _swap(inputToken, inputAmount, outputToken, outputAmount);
```

[H-4] `TSwapPool::sellPoolTokens()` has incorrect input parameters leading to incorrect amount of PoolTokens to be sold

Description: The function calls `swapExactOutput()` but switches the input and output token parameters which leads to incorrect calculations on how many poolTokens to be sold despite the user specifying the amount of poolTokens they want to sell.

Impact: This leads to failed transactions for users who do not approve more poolTokens than they intend to spend and this will cause them to sell more tokens than intended if they approve far more tokens than needed.

Proof of Code:

Within the `TSwapPool.t.sol` test file in the `test/unit` folder, we include the following test functions:

```

    function
       testexpectRevertFromsellPoolTokensDueToInsufficientAllowance()
       public {

        // Display user balances
        uint256 initialPoolTokenBalance = poolToken.balanceOf(user);
        uint256 initialWethBalance= weth.balanceOf(user);
        console.log("Initial Pool balance: ", initialPoolTokenBalance);
        console.log("Initial Weth balance: ", initialWethBalance);
        uint256 POOL_TOKENS_TO_SELL = 1e17;

        // Provide Liquidity
        vm.startPrank(liquidityProvider);
```

```
13          weth.approve(address(pool), 100e18);
14          poolToken.approve(address(pool), 100e18);
15          pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
16          vm.stopPrank();
17
18          // run swapExactOutput
19          initialPoolTokenBalance = poolToken.balanceOf(user);
20          initialWethBalance= weth.balanceOf(user);
21          vm.startPrank(user);
22          poolToken.approve(address(pool), POOL_TOKENS_TO_SELL);
23          vm.expectRevert(); // the sellPoolTokens function will attempt
                to sell more than user approved amount
24          uint256 wethAmount= pool.sellPoolTokens(POOL_TOKENS_TO_SELL);
25      }
26
27
28      function testsellPoolTokensActsAsIntended() public {
29
30          // Display user balances
31          uint256 initialPoolTokenBalance = poolToken.balanceOf(user);
32          uint256 initialWethBalance= weth.balanceOf(user);
33          console.log("Initial Pool balance: ", initialPoolTokenBalance);
34          console.log("Initial Weth balance: ", initialWethBalance);
35          uint256 POOL_TOKENS_TO_SELL = 1e17;
36
37          // Provide Liquidity
38          vm.startPrank(liquidityProvider);
39          weth.approve(address(pool), 100e18);
40          poolToken.approve(address(pool), 100e18);
41          pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
42          vm.stopPrank();
43
44          // run swapExactOutput
45          initialPoolTokenBalance = poolToken.balanceOf(user);
46          initialWethBalance= weth.balanceOf(user);
47          vm.startPrank(user);
48          poolToken.approve(address(pool), type(uint256).max);
49          uint256 wethAmount= pool.sellPoolTokens(POOL_TOKENS_TO_SELL);
50          vm.stopPrank();
51
52          console.log(poolToken.balanceOf(user));
53          console.log(weth.balanceOf(user));
54
55          // We assert that the
56          assertLt(poolToken.balanceOf(user), initialPoolTokenBalance -
                POOL_TOKENS_TO_SELL, "We assert that the parameter error
                causes more pool tokens to be sold");
57      }
```

And running the following foundry commands:

```
forge test --mt testexpectRevertFromsellPoolTokensDueToInsufficientAllowance
 -vvvv
```

```
forge test --mt testsellPoolTokensActsAsIntended -vvvv
```

The first test will fail due to insufficient allowance. The second test set the approval to the max to avoid the first error to demonstrate the function will sell more tokens than the user specified to sell.

Recommended Mitigation:

Consider changing the implementation to use `swapExectInput` instead of `swapExactOutput`, Note that this would also require changing the `sellPoolTokens` function to accept a new parameter.

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

[M-1] `TSwapPool::deposit()` neglecting to use the `deadline` parameter misleads a user to believe they can cause a revert!

Description: The natspec specifies the uint64 parameter `deadline` to specify when the transaction should be settled by. But it's not used in the function!

Impact: If a user relied on the deadline to prevent a transaction to go through, it wouldn't protect them from its misuse because it's not used at all!

Recommended Mitigation: Make a modifier to use it!

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

[M-2] The values stored in private `TSwapPool::swap_count` and constant `TSwapPool::SWAP_COUNT_MAX` used for tracking the number of `TSwapPool::_swap()` calls before gifting a Pool Token ARE NOT PRIVATE

Description: Private variables are still accessible to be read from storage!

Impact: This means a user could time their function call to collect an extra token!

Proof of Concept

First we will deploy the `TSwapPool` contract, which first involves deploying the `PoolFactory` contract and a Mock Weth `ERC20` contract (a second `MockERC20` contract is optional for this proof of concept). Once the TSwapPool contract is deployed, we'll use foundry tools to inspect the storage layout and retrieve the value of `swap_count`.

Proof of Code

In a separate terminal, we run the command

```
anvil
```

Then, we'll simplify the future commands by exporting the following anvil generated Private key and rpc-url,

```
export ANVIL_KEY=0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2f
export ANVIL_RPC_URL=http://127.0.0.1:8545
```

to spin up a local chain to deploy the contract to. For deploying the contract, run the terminal command,

```
forge script script/DeployTSwap.t.sol --private-key $ANVIL_KEY --rpc-url $ANVIL_RPC_URL --broadcast
```

Where this deploys first a WETH MockERC20 contract and secondly a PoolFactory contract to anvil. Take note of the Pool Factory address and export it with this command

```
export WETH_CONTRACT=0x5FbDB2315678afecb367f032d93F642f64180aa3 export POOLFACTORY_CONTRACT=0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512
```

To generate a Pool contract from the Pool Factory contract, we need to submit two ERC20 contracts to the constructor, WETH and a second token. Currently the Pool Factory contract does not check if the second token is Weth so you can submit the same Weth contract address in both constructor parameters. However, I'll include an optional ROUTE B below for deploying another ERC20 contract to properly deploy the a Pool contract.

— ROUTE OPTIONAL B —-

Next, we need to deploy a MockERC20 contract to anvil. We first create a deploy script in the `script` folder from the root directory under the file name `DeployMockERC20.s.sol`.

```
1
2  // SPDX-License-Identifier: MIT
3  pragma solidity 0.8.20;
4
5  import { Script } from "forge-std/Script.sol";
6  import {ERC20Mock} from "../test/unit/mocks/ERC20Mock.sol";
7
8  contract DeployERC20Mock is Script {
9      function run() public {
10         vm.startBroadcast();
11         new ERC20Mock();
12         vm.stopBroadcast();
13     }
14 }
```

Then we deploy the contract using the following terminal command:

‘forge script script/DeployMockERC20.s.sol --private-key $ANVIL_KEY --rpc-url $ANVIL_RPC_URL --broadcast

— ROUTE OPTIONAL B END —-

Then, within the PoolFactory contract, we can finally deploy the `TSwapPool` contract by running the command where the address parameter will be the same WETH contract for a WETH/WETH Pool (if you went with route B, swap the parameter for your deployed ERC20 contract address):

```
cast send $POOLFACTORY_CONTRACT "createPool(address)"$WETH_CONTRACT --private-key $ANVIL_KEY --rpc-url $ANVIL_RPC_URL
```

We can get the address of the new WETH/WETH TSwapPool generated from the factory contract using the command:

```
cast call $POOLFACTORY_CONTRACT "getPool(address)"$WETH_CONTRACT --rpc-url $ANVIL_RPC_URL
```

You can export the output to `CALL_DATA`

Then we can run the command to extract the last 40 characters for the address and prefix 0x to the start of the address ot get a proper Ethereum address:

```
echo "0x$(echo $CALL_DATA | tail -c 41)"
```

And export that address to `POOL` or a variable of your choice. Next, we’ll run the command

```
cast storage $POOL 5 --rpc-url $ANVIL_RPC_URL
```

And within the output you can find the slot number for swap_count

```
{
        "astId": 41638,
        "contract": "src/TSwapPool.sol:TSwapPool",
        "label": "swap_count",
        "offset": 0,
        "slot": "5",
        "type": "t_uint256"
}
```

And you'll be able to retrieve the value in hex using the command:

```
cast storage $POOL 5 --rpc-url $ANVIL_RPC_URL
```

It should currently return a hex value of zero as there have been no changes to the function. To convert to decimal, run the command

```
cast to-dec 0x0000000000000000000000000000000000000000000000000000000000000000
```

Recommended Mitigation: For gifting of a token, consider using an RNG like Chainlink VRF once the `swap_count` reaches the minimum swap threshold `SWAP_COUNT_MAX` to randomize the chance of the next user winning a token, whether it may be a 50% or 33% chance. This mitigates the chance of a user tracking `swap_count` to game the system.

[M-3] Rebasing and fee on transfer Weird ERC-20 tokens will break the invariant!

Description: There are custom ERC-20 tokens, Weird ERC-20 tokens that will take or give value from or to the user trading the token. If they generate a pool with said ERC-20 token, it will inevitably break the invariant.

Impact: This means that ERC-20 tokens can steal value from unwitting users!

Proof of Concept: 1. Someone create a new pool with weird-ERC20 and Weth and provides liquidity. 2. User trades with the token, and not notice that the weird token is taking fees from user with every transaction 3. Invariant breaks in the process

Proof of Code:

We first create an example ERC20 token with the file `YeildERC20.sol` in the `test/unit/mocks` folder:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

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

```

```
contract YeildERC20 is ERC20 {
    uint256 public constant INITIAL_SUPPLY = 1_000_000e18;
    address public immutable owner;
    // We take a fee once every 10 transactions
    uint256 public count = 0;
    uint256 public constant FEE = 10;
    uint256 public constant USER_AMOUNT = 90;
    uint256 public constant PRECISION = 100;

    constructor() ERC20("MockYeildERC20", "MYEILD") {
        owner = msg.sender;
        _mint(msg.sender, INITIAL_SUPPLY);
    }

    /**
     * @dev Transfers a `value` amount of tokens from `from` to `to`, or alternatively mints (or burns) if `from`
     * (or `to`) is the zero address. All customizations to transfers, mints, and burns should be done by overriding
     * this function.
     *
     * Every 5 transactions, we take a fee of 5% and send it to the owner.
     */
    function _update(address from, address to, uint256 value) internal virtual override {
        if (to == owner) {
            super._update(from, to, value);
        } else if (count >= 1) {
            uint256 userAmount = value * USER_AMOUNT / PRECISION;
            uint256 ownerAmount = value * FEE / PRECISION;
            count = 0;
            super._update(from, to, userAmount);
            super._update(from, owner, ownerAmount);
        } else {
            count++;
            super._update(from, to, value);
        }
    }
}
```

We then implement this test in `test/unit/TSwapPool.t.sol`:

```

function testInvariantBrokenWithWeirdERC20() public {

        YeildERC20 yield = new YeildERC20();

        TSwapPool ypool = new TSwapPool(address(yield), address(weth), "LTokenY", "LY");

```

```
8           yield.transfer(liquidityProvider,100e18);
9           vm.startPrank(liquidityProvider);
10          weth.approve(address(ypool), 100e18);
11          yield.approve(address(ypool), 100e18);
12          ypool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
13          vm.stopPrank();
14
15
16          uint256 outputWeth = 1e17;
17          int256 startingX = int256(yield.balanceOf(address(ypool)));
18          int256 expectedDeltaX = 90000000000000000*5;
19
20          yield.transfer(user,100e18);
21          vm.startPrank(user);
22          yield.approve(address(ypool), type(uint256).max);
23
24          console.log("pool yield Before Swap balance: ", yield.balanceOf
                (address(this)));
25
26          for(uint i =0;i < 5; ++i){
27              ypool.swapExactInput(yield,1e17,weth,outputWeth,uint64(
                    block.timestamp));
28          }
29          vm.stopPrank();
30
31          int256 endingX = int256(yield.balanceOf(address(ypool)));
32          int256 actualDeltaX = endingX - startingX;
33
34          console.log("this contract yield balance: ", yield.balanceOf(
                address(this)));
35          console.log("startingX: ", startingX);
36          console.log("endingX: ", endingX);
37          console.log("Delta: ", endingX - startingX);
38          assertEq(actualDeltaX,expectedDeltaX, "we assert that the swaps
                should remain the same 5 times over" );
39      }
```

And run this in a foundry environment with the command:

```
forge test --mt testInvariantBrokenWithWeirdERC20 -vvvv
```

The test assertion of the change in X, yield token, remains the same across 5 swaps. It fails due to the example weird ERC20 token YeildERC20 sending 5% of the transfer value to the YeildERC20 token contract deployer, thereby breaking the invariant.

Recommended Mitigation: It is recommended to create a whitelist of ERC20 tokens to vet the ERC20 tokens that don't introduce hidden functionality to skim users.

[L-1] `TSwapPool::_addLiquidityMintAndTransfer()`'s event `LiquidityAdded` has incorrect parameters

Description: The event is defined as follows

```
event LiquidityAdded(
        address indexed liquidityProvider,
        uint256 wethDeposited,
        uint256 poolTokensDeposited
    );
```

While the emitted event emits in the order:

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

Impact: This leads to an incorrect event log

Recommended Mitigation: Swap 2nd and 3rd parameters

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

[L-2] `TSwapPool::swapExactInput()` returns `uint256 output` but is never defined in the function thus giving an incorrect return value

Description:

```
function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    )
        public
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
        returns (uint256 output){}
```

The return value is declared in the function declaration but never used in function.

Impact: Thus it will return incorrect values to the user.

Proof of Code:

within `TSwapPool.t.sol` in the `test` folder, we include the test:

```
function testgetInputAmountBasedOnOutputReturnsProperOutput() public {

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

        // run swapExactOutput
        uint256 initialWethBalance = weth.balanceOf(user);
        vm.startPrank(user);
        uint256 expected = 9e18;
        poolToken.approve(address(pool), 10e18);
        uint256 output= pool.swapExactInput(poolToken, 10e18, weth,
           expected, uint64(block.timestamp));
        vm.stopPrank();

        assertEq(output, 0, "we assert that the output was never
           updated from zero");
        assert(weth.balanceOf(user) > initialWethBalance);
    }
```

To which we assert that the output is 0 despite the wethbalance increasing before the `swapExactInput` call. Running the foundry command

`forge test --mt testgetInputAmountBasedOnOutputReturnsProperOutput`

passes the assertions.

Recommended Mitigation:

```
    {
        uint256 inputReserves = inputToken.balanceOf(address(this));
        uint256 outputReserves = outputToken.balanceOf(address(this));

-       uint256 outputAmount = getOutputAmountBasedOnInput(inputAmount,
    inputReserves,outputReserves);
+       uint256 output = getOutputAmountBasedOnInput(inputAmount,
    inputReserves,outputReserves);


-       if (outputAmount < minOutputAmount) {
-           revert TSwapPool__OutputTooLow(outputAmount,
    minOutputAmount);
-       }

+       if (output < minOutputAmount) {
+           revert TSwapPool__OutputTooLow(output, minOutputAmount);
```

```
+       }
-       _swap(inputToken, inputAmount, outputToken, outputAmount);
+       _swap(inputToken, inputAmount, outputToken, output);
    }
```

[I-1] The error `PoolFactory::PoolFactory__PoolDoesNotExist(address tokenAddress)` is not used

It is worth omitting to reduce gas consumption in deployment.

[I-2] Neither the `PoolFactory::constructor` nor the `TSwapPool::constructor` have a zero check for the address parameters

`PoolFactory.sol`

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

`TSwapPool::constructor`

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

[I-3] The string `liquidityTokenSymbol` assigned within `PoolFactory::createPool` should use the ERC20 function `symbol()` instead of `name()`

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

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

[I-4] Event `TSwapPool::Swap` should have 3 indexed events when containing 3 or more parameters

```
1 event Swap(
2         address indexed swapper,
3 -         IERC20 tokenIn,
4 +         IERC20 indexed tokenIn,
5         uint256 amountTokenIn,
6 -         IERC20 tokenOut,
7 +         IERC20 indexed tokenOut,
8         uint256 amountTokenOut
9     );
```

[I-5] In `TSwapPool::deposit()`, `MINIMUM_WETH_LIQUIDITY` is an constant that does not need to be emitted in the error `TSwapPool__WethDepositAmountTooLow`

```
1         if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
2             revert TSwapPool__WethDepositAmountTooLow(
3 -                 MINIMUM_WETH_LIQUIDITY,
4                 wethToDeposit
5             );
6         }
```

[I-6] In `TSwapPool::deposit()`, the uint256 variable `poolTokenReserves` is unused and should be taken out to save on gas

```
1 -   uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
```

[I-7] In `TSwapPool::deposit()`, the uint256 variable `liquidityTokensToMint` is not a state variable and doesn't follow CEI

```
1 +           liquidityTokensToMint = wethToDeposit;
2     _addLiquidityMintAndTransfer(
3                 wethToDeposit,
4                 maximumPoolTokensToDeposit,
5                 wethToDeposit
6             );
7 -           liquidityTokensToMint = wethToDeposit;
```

```
8         }
```

[I-8] `TSwapPool::getOutputAmountBasedOnInput` among other functions contain literal numbers and should be replaced with literal CONSTANTS

Recommended Mitigation:: For example, with `TSwapPool::getOutputAmountBasedOnInput()`:

Define global variables:

```
1             uint256 constant FEE_MULTIPLIER = 997;
2             uint256 constant PRECISION = 1000;
```

Changes within getter function `getOutputAmountBasedOnInput()`

```
1 -       uint256 inputAmountMinusFee = inputAmount * 997;
2 +       uint256 inputAmountMinusFee = inputAmount * FEE_MULTIPLIER;
3         uint256 numerator = inputAmountMinusFee * outputReserves;
4 -       uint256 denominator = (inputReserves * 1000) +
      inputAmountMinusFee;
5 +       uint256 denominator = (inputReserves * PRECISION) +
      inputAmountMinusFee;
```

The following functions also have magic numbers that should be replaced:

`TSwapPool::_swap()` `TSwapPool::getPriceOfOnePoolTokenInWeth()`

[I-9] `TSwapPool::swapExactInput()` does not have natspec written out

[I-10] `TSwapPool::swapExactInput()` should be changed to `external` to save on gas since no contract functions calls it

[I-11] `TSwapPool::totalLiquidityTokenSupply()` should be `external view` to save on gas since no contract function calls it