TSwap Audit Report

Version 0.1

Riiz0

Date: May 20, 2024

TSwap Audit Report

Prepared by:

- Shawn Rizo

Lead Auditor(s):

- Shawn Rizo

Assisting Auditors:

- None

Table of Contents

About Shawn Rizo

I am a seasoned Smart Contract Engineer, adept at utilizing agile methodologies to deliver comprehensive insights and high-level overviews of blockchain projects. Specialized in developing and deploying decentralized applications (DApps) on Ethereum and EVM compatible chains. Expertise in Solidity, and security auditing, leading to a significant reduction in vulnerabilities through the strategic use of Foundry and Security Tools like Slither and Aderyn.

Disclaimer

The Riiz0 team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described in this document correspond the following commit hash:

```
e643a8d4c2c802490976b538dd009b351b1c8dda
```

Scope

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

Protocol Summary

The TSwap project aims to offer a decentralized platform for users to trade assets among themselves at equitable prices, functioning as a Decentralized Exchange (DEX). Unlike traditional exchanges that rely on order books, TSwap operates as an Automated Market Maker (AMM), utilizing pools of assets. This model is akin to platforms like Uniswap, emphasizing a permissionless trading environment where participants can swap tokens directly through liquidity pools.

Roles

- Liquidity Providers: Users who have liquidity deposited into the pools.
- Users: Users who want to swap tokens.

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	2
Info	10
Gas	1
Total	18

Findings

High

[H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` casuses protocol to take too many tokens from users, resulting in lost fees

Description: The `getInputAmountBasedOnOutput` function is intended to calculate the amoun of tokens a user should deposit given an amount of tokens of output tokens. However, the function currently miscalculates the resulting amount. When calculating the fee, it scales the amount by 10_000 instead of 1_000.

Impact: Protocol takes more fees than expected from users.

Proof of Concept: The provided code snippet highlights the incorrect calculation in the getInputAmountBasedOnOutput function:

```
function testFlawedSwapExactOutput() public {
    uint256 initialLiquidity = 100e18;
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), initialLiquidity);
    poolToken.approve(address(pool), initialLiquidity);

    pool.deposit({
        wethToDeposit: initialLiquidity,
        minimumLiquidityTokensToMint: 0,
        maximumPoolTokensToDeposit: initialLiquidity,
        deadline: uint64(block.timestamp)
    });
    vm.stopPrank();

    // User has 11 pool tokens
    address someUser = makeAddr("someUser");
    uint256 userInitialPoolTokenBalance = 11e18;
    poolToken.mint(someUser, userInitialPoolTokenBalance);
    vm.startPrank(someUser);

    // Users buys 1 WETH from the pool, paying with pool tokens
    poolToken.approve(address(pool), type(uint256).max);
    pool.swapExactOutput(
        poolToken,
        weth,
        1 ether,
        uint64(block.timestamp)
    );

    // Initial liquidity was 1:1, so user should have paid ~1 pool token
    // However, it spent much more than that. The user started with 11
tokens, and now only has less than 1.
    assertLt(poolToken.balanceOf(someUser), 1 ether);
    vm.stopPrank();

    // The liquidity provider can rug all funds from the pool now,
    // including those deposited by user.
    vm.startPrank(liquidityProvider);
    pool.withdraw(
        pool.balanceOf(liquidityProvider),
        1, // minWethToWithdraw
        1, // minPoolTokensToWithdraw
        uint64(block.timestamp)
    );

    assertEq(weth.balanceOf(address(pool)), 0);
    assertEq(poolToken.balanceOf(address(pool)), 0);
}
```

Recommended Mitigation: Avoid using magic numbers like "1000" and "10000", instead make constant variables to avoid inputting incorrect values or do the following code snippet:

```
    function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
    {
-         return ((inputReserves * outputAmount) * 10_000) /
((outputReserves - outputAmount) * 997);
+         return ((inputReserves * outputAmount) * 1_000) /
((outputReserves - outputAmount) * 997);
    }
```

[H-2] Lack of slippage protection in swapExactOutput function

Description: The swapExactOutput function does not include any sort of slippage protection to protect user funds that swap tokens in the pool. Similar to what is done in the swapExactInput function, it should include a parameter (e.g., maxInputAmount) that allows callers to specify the maximum amount of tokens they're willing to pay in their trades.

Impact: If market conditions change before the transaciton processes, the user could get a much worse swap.

Proof of Concept:

1. The price of 1 WETH right now is 1,000 USDC
2. User inputs a `swapExactOutput` looking for 1 WETH
    1. inputToken = USDC
    2. outputToken = WETH
    3. outputAmount = 1
    4. deadline = whatever
3. The function does not offer a maxInput amount
4. As the transaction is pending in the mempool, the market changes! And the price moves HUGE -> 1 WETH is now 10,000 USDC. 10x more than the user expected
5. The transaction completes, but the user sent the protocol 10,000 USDC instead of the expected 1,000 USDC

Recommended Mitigation: We should include a `maxInputAmount` so the user only has to spend up to a specific amount, and can predict how much they will spend on the protocol.

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

[H-3] `TSwapPool::sellPoolTokens` mismatches input and output tokens causing users to receive the incorrect amount of tokens

Description: The `sellPoolTokens` function is intended to allow users to easily sell pool tokens and receive WETH in exchange. Users indicate how many pool tokens they're willing to sell in the `poolTokenAmount` parameter. However, the function currently miscalculaes the swapped amount.

Impact: Users will swap the wrong amount of tokens, which is a severe disruption of protcol functionality.

Proof of Concept:

1. Scenario Setup: Assume a user wants to sell 100 pool tokens for WETH. The current conversion rate between pool tokens and WETH is 1:10 (meaning 1 pool token equals 10 WETH).
2. Incorrect Swap Logic: When the user calls sellPoolTokens with poolTokenAmount set to 100, the function internally uses swapExactOutput to attempt to convert 100 pool tokens into WETH. Given the conversion rate, the expected output should indeed be 1000 WETH.
3. Actual Outcome: Due to the misuse of swapExactOutput, the function incorrectly calculates the output amount. Instead of calculating the output based on the conversion rate, it might mistakenly use the input amount itself as the output, resulting in an incorrect calculation like returning 100 WETH instead of the expected 1000 WETH.
4. Impact: This error means the user receives significantly fewer WETH than they expected, leading to dissatisfaction and potentially disrupting trust in the protocol.

Recommended Mitigation: Consider changing the implementation to use `swapExactInput` instead of `swapExactOutput`. Note that this would also require changing the `sellPoolTokens` function to accept a new parameter (ie `minWethToReceive` to be passed to `swapExactInput`)

```
    function sellPoolTokens(
        uint256 poolTokenAmount,
+       uint256 minWethToReceive,
        ) external returns (uint256 wethAmount) {
```

```
-         return swapExactOutput(i_poolToken, i_wethToken, poolTokenAmount,
uint64(block.timestamp));
+         return swapExactInput(i_poolToken, poolTokenAmount, i_wethToken,
minWethToReceive, uint64(block.timestamp));
      }
```

[H-4] In `TSwapPool::_swap` the extra tokens given to users after every `swapCount` breaks the protocol invariant of `x * y = k`

Description: The protocol follows a strict invariant of `x * y = k`. Where:

- `x`: The balance of the pool token
- `y`: The balance of WETH
- `k`: The constant product of the two balances

This means, that whenever the balances change in the protocol, the ratio between the two amounts should remain constant, hence the `k`. However, this is broken due to the extra incentive in the `_swap` function. Meaning that over time the protocol funds will be drained.

The follow block of code is responsible for the issue.

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

Impact: A user could maliciously drain the protocol of funds by doing a lot of swaps and collecting the extra incentive given out by the protocol. Most simply put, the protocol's core invariant is broken.

Proof of Concept:

1. A user swaps 10 times, and collects the extra incentive of `1_000_000_000_000_000_000` tokens
2. That user continues to swap untill all the protocol funds are drained

```
    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;
```

```
        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        poolToken.mint(user, 100e18);
        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));

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

        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
        vm.stopPrank();

        uint256 endingY = weth.balanceOf(address(pool));
        int256 actualDeltaY = int256(endingY) - int256(startingY);
        assertEq(actualDeltaY, expectedDeltaY);
    }
```

Recommended Mitigation: Remove the extra incentive mechanism. If you want to keep this in, we should account for the change in the x * y = k protocol invariant. Or, we should set aside tokens in the same way we do with fees.

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

Medium

[M-1] `TSwapPool::deposit` is missing deadline check causing transactions to complete even after the deadline

Description: The `deposit` function accepts a deadline parameter, which according to the documentation is "The deadline for the transaction to be completed by". However, this parameter is never used, as a consequence, operations that add liquidity to the pool might be executed at unexpected times, in market conditions where the deposit rate is unfavorable.

Impact: Transactions could be sent when market conditions are unfavorable, even when adding a deadline parameter.

Proof of Concept: The `deadline` parameter is unused.

```
Warning (5667): Unused function parameter. Remove or comment out the
variable name to silence this warning.
   --> src/TSwapPool.sol:100:9:
    |
100 |         uint64 deadline
    |         ^^^^^^^^^^^^^^^
```

Recommended Mitigation: Consider making the following change to the function.

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

Low

[L-1] `TSwapPool::LiquidityAdded` event has parameters out of order

Description: When the `LiquidityAdded` event is emitted in the `TSwapPool::_addLiquidityMintAndTransfer` function, it logs values in an incorrect order. The `poolTokensToDeposit` value should go in the third parameter position, whereas the `wethToDeposit` value should go second.

Impact: Event emission is incorrect, leading to off-chain functions potentially malfunctioning.

Proof of Concept:

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

Recommended Mitigation: Test all events and double check and see if the correct log values are being displayed for all events.

[L-2] Default value returned by `TSwapPool::swapExactInput` results incorrect return value given

Description: The `swapExactInput` function is expected to return the actual amount of tokens bought by the caller. However, while it declares the named return value `output` it is never assigned a value, nor uses an explicit return statement.

Impact: The return value will always be 0, giving incorrect information to the caller

Proof of Concept: The swapExactInput function calculates the outputAmount based on the input amount, input reserves, and output reserves, and then compares this calculated outputAmount against the minOutputAmount provided by the caller. If the calculated outputAmount is less than the minOutputAmount, the function reverts with the TSwapPool__OutputTooLow error. The protocol is giving the wrong return, which implies that the outputAmount returned by the function might not be what the caller expects. This could lead to confusion about whether the error is due to the calculation logic itself or the comparison against minOutputAmount.

Recommended Mitigation:

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

-        uint256 outputAmount = getOutputAmountBasedOnInput(inputAmount,
inputReserves, outputReserves);
+        output = getOutputAmountBasedOnInput(inputAmount, inputReserves,
outputReserves);
-        if (outputAmount < minOutputAmount) {
-            revert TSwapPool__OutputTooLow(outputAmount,
minOutputAmount);
         }
+         if (output < minOutputAmount) {
+           revert TSwapPool__OutputTooLow(outputAmount,
minOutputAmount);
         }
-        _swap(inputToken, inputAmount, outputToken, outputAmount);
+        _swap(inputToken, inputAmount, outputToken, output);
    }
```

Informational

[I-1] `PoolFactory::PoolFactory__PoolAlreadyExists` is not used and should be removed

Description: In the `PoolFactory` there is a custom error that checks for if a pool does not exist. Unfortunately the custom error isn't being used and therefore is not need.

Impact: Having an unused custom error does not directly cause gas wastage, but it's good practice to keep the contract clean and focused on its essential functionalities.

Proof of Concept:

```
- error PoolFactory__PoolDoesNotExist(address tokenAddress);
```

Recommended Mitigation: Proof read code and double check variables, functions, custom errors for any unused objects.

[I-2] Event PoolCreated in `PoolFactory` & `TSwapPool` should have indexed events

Description: Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

Impact: Not indexing enough parameters in events can lead to inefficient querying of event logs, potentially impacting the performance and responsiveness of applications that depend on these events for real-time data analysis or notifications. Additionally, not indexing parameters that are frequently used for filtering or searching can limit the effectiveness of these applications.

Proof of Concept:

- Found in src/TSwapPool.sol: Line: 44
- Found in src/PoolFactory.sol: Line: 37
- Found in src/TSwapPool.sol: Line: 46
- Found in src/TSwapPool.sol: Line: 43

Recommended Mitigation: To mitigate this issue, the events in the contracts should be reviewed to determine which parameters are most likely to be used for filtering or searching. Parameters identified as candidates for frequent filtering or searching should be indexed. At least three of them should be indexed to support efficient querying. The implementation would look something like this:

[I-3] `.name()` should be `.symbol()` in `PoolFactory::createPool` function

Description: In the `createPool` function of the `PoolFactory` smart contract, the token's symbol is incorrectly retrieved using `.name()` instead of `.symbol()`. This mistake leads to incorrect data being assigned to the liquidity token's symbol, which could mislead users and affect the functionality of the contract.

Impact: Using `.name()` to fetch the token's symbol results in assigning the token's full name as its symbol, which goes against the standard convention of ERC20 tokens where the symbol is a shorter identifier. This discrepancy can cause confusion for users and may lead to errors in interactions with the contract, as the symbol is a critical piece of information for identifying and distinguishing between different tokens.

Proof of Concept:

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

Recommended Mitigation: To rectify this issue, the `createPool` function should be updated to correctly use `.symbol()` to retrieve the token's symbol. This change ensures that the correct symbol is assigned to the liquidity token, aligning with the standard expectations for ERC20 tokens and improving the clarity and reliability of the contract's operations. The corrected line of code would be:

[I-4] In the `TSwapPool` Contructor check for zero address

Description: The constructor of the `TSwapPool` smart contract should include checks to ensure that the addresses passed as arguments are not zero addresses. This is a critical security measure to prevent vulnerabilities such as loss of funds, unauthorized access, and unexpected behavior due to interactions with zero addresses.

Impact: If a zero address is used as a recipient for token transfers, the tokens sent will be irretrievable. Additionally, granting permissions or privileges to a zero address can lead to unauthorized actions or control over the contract.

Proof of Concept:

```
    constructor(
        address poolToken,
        address wethToken,
        string memory liquidityTokenName,
        string memory liquidityTokenSymbol
    )
        ERC20(liquidityTokenName, liquidityTokenSymbol)
    {
+       if(wethToken == address(0)) {
+            revert();
+        }
+       if(poolToken == address(0)) {
+            revert();
+        }

        i_wethToken = IERC20(wethToken);
        i_poolToken = IERC20(poolToken);
    }
```

Recommended Mitigation: To mitigate this risk, the constructor should include `require` statements to check that neither `poolToken` nor `wethToken` is a zero address before proceeding with the initialization. Here's an example of how to implement this check:

[I-5] `MINIMUM_WETH_LIQUIDITY` is a constant and therefore not required to be emitted in revert `TSwapPool::deposit`

Description: In the `TSwapPool::deposit` function, the `MINIMUM_WETH_LIQUIDITY` constant is incorrectly included in the revert message. Since constants are immutable and their values are known at compile time, they do not change and thus do not need to be part of the revert message.

Impact: Including constants in revert messages can lead to confusion and inefficiency. It unnecessarily inflates the size of the revert message, which can impact gas costs and make debugging more difficult due to the increased complexity of the error message.

Proof of Concept:

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

Recommended Mitigation: To mitigate this issue, the revert message should be simplified to exclude the `MINIMUM_WETH_LIQUIDITY` constant. The revised revert message should focus on the key issue—whether the deposit amount is too low—and remove the inclusion of the constant value. The corrected code would look like this:

[I-6] Define and use `constant` variables instead of using literals or magic numbers

Description: The smart contract uses literal values (magic numbers) directly in the code, specifically in mathematical expressions and calculations. These literals represent fixed values that are used repeatedly throughout the contract. Defining these values as `constant` variables improves readability, maintainability, and reduces the risk of errors. Constants are named entities that clearly express the purpose of the value they hold, making the code more self-documenting and easier to understand.

Impact: Using literal values directly in the code can lead to several issues:

- Readability and Maintainability: Magic numbers can make the code harder to read and maintain, as their purpose is not immediately clear without looking up their definitions elsewhere in the code.
- Error Prone: Hardcoding values can introduce errors, especially if the same value is used in multiple places and needs to be changed. Changing a magic number requires updating every occurrence, increasing the risk of oversight.
- Gas Efficiency: Although not directly related to the use of constants versus literals, defining constants can sometimes offer minor optimizations in terms of gas efficiency, as the compiler might optimize the use of constants better than literals.

Proof of Concept:

- Found in src/TSwapPool.sol Line: 239

```
        uint256 inputAmountMinusFee = inputAmount * 997;
```

- Found in src/TSwapPool.sol Line: 241

```
        uint256 denominator = (inputReserves * 1000) +
inputAmountMinusFee;
```

- Found in src/TSwapPool.sol Line: 259

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

- Found in src/TSwapPool.sol Line: 396

```
            1e18, i_wethToken.balanceOf(address(this)),
i_poolToken.balanceOf(address(this))
```

- Found in src/TSwapPool.sol Line: 402

```
            1e18, i_poolToken.balanceOf(address(this)),
i_wethToken.balanceOf(address(this))
```

Recommended Mitigation: To address this issue, the contract should define constants for all literal values used in calculations. This involves identifying all unique literal values and replacing them with named constants. For example, the literal `997` could be replaced with a constant named `FEE_REDUCTION_FACTOR`, and `1e18` could be replaced with a constant named `ONE_ETHER_IN_WEI`. This approach enhances the clarity and robustness of the contract's code.

[I-7] Restructure order following CEI

Description: The current implementation of the contract performs an external call followed by a state-changing operation, which is flagged as a potential vulnerability according to the Common Exploit Identifier (CEI) principles. Specifically, the assignment of `liquidityTokensToMint` occurs after the `_addLiquidityMintAndTransfer` external function call, creating a window where the outcome of the external call could be influenced by malicious actors.

Impact: Performing state-changing operations after external calls can lead to vulnerabilities that allow attackers to manipulate the contract's state or exploit the contract's logic. This can result in financial losses, loss of control over the contract, or other unintended consequences.

Proof of Concept:

```
        } else {
-           _addLiquidityMintAndTransfer(wethToDeposit,
maximumPoolTokensToDeposit, wethToDeposit);
-           liquidityTokensToMint = wethToDeposit;
        }

        } else {
+           liquidityTokensToMint = wethToDeposit;
+           _addLiquidityMintAndTransfer(wethToDeposit,
maximumPoolTokensToDeposit, wethToDeposit);
        }
```

Recommended Mitigation: To mitigate this issue and adhere to CEI principles, the assignment of `liquidityTokensToMint` should be moved before the `_addLiquidityMintAndTransfer` external call. This change minimizes the window of opportunity for potential exploits by reducing the time between the external call and the subsequent state change. The revised code would look like this:

[I-8] Missing natspec for `TSwapPool::swapExactInput` function

Description: The `swapExactInput` function lacks NatSpec comments, which are essential for documenting the purpose, parameters, and return values of the function. NatSpec comments are a standard way of adding documentation to Solidity functions, enabling tools and developers to easily understand the function's intent and usage.

Impact: Without NatSpec comments, understanding the functionality and expected behavior of the `swapExactInput` function becomes more challenging. This lack of documentation can hinder the development process, impacting the readability and maintainability of the codebase.

Proof of Concept:

```
/*
// Missing natspec
*/
    function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,  receive
        uint64 deadline
    )
```

Recommended Mitigation: To address this issue, add a NatSpec comment block above the `swapExactInput` function declaration. This comment should describe the function's purpose, its parameters, and what it returns. Here's an example of how to add NatSpec comments to the function:

[I-9] `public` functions not used internally could be marked `external`

Description: Instead of marking a function as `public`, consider marking it as `external` if it is not used internally.

Impact: Marking functions as `external` when they are not meant to be called from outside the contract can lead to several benefits:

- Cost Savings: Calls to `external` functions are cheaper in terms of gas fees, as they avoid the overhead of copying function arguments.
- Security: Reducing the surface area of functions callable from outside the contract can reduce the attack vector for potential exploits.
- Clarity: Marking functions appropriately as `internal`, `private`, or `external` improves code readability and understanding of the contract's intended usage patterns.

Proof of Concept:

- Found in src/TSwapPool.sol Line: 263

```
    function swapExactInput(
```

- Found in src/TSwap.sol Line:380

```
function totalLiquidityTokenSupply(
```

Recommended Mitigation: To address this issue, review the functions in your contract to identify those that are not intended to be called from outside the contract. Functions that meet this criterion should be marked as `external` instead of `public`. This change will optimize the contract's performance and security posture. For example, if the `swapExactInput` function in `TSwapPool.sol` is not meant to be called from outside the contract, it should be marked as `external`.

[I-10] 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.

Impact: The inability to deploy contracts on certain chains due to unsupported opcodes can significantly limit the accessibility and usability of your decentralized application (dApp). It restricts the ability to scale applications across different layers of the Ethereum ecosystem, potentially hindering adoption and growth.

Proof of Concept:

- Found in src/PoolFactory.sol Line: 15

```
pragma solidity 0.8.20;
```

- Found in src/TSwapPool.sol Line: 15

```
pragma solidity 0.8.20;
```

Recommended Mitigation: To mitigate this issue, you should specify the EVM version explicitly in your contract's pragma directive if you plan to deploy on chains that do not support the Shanghai EVM version. This ensures compatibility with the target EVM version and avoids the use of unsupported opcodes. For example, if deploying on an L2 chain that supports the London EVM version, you should modify the pragma directive accordingly:

Gas

[G-1] Variable `uint256 poolTokenReserves` in `TSwapPool::deposit` funciton is not used/not needed

Description: The variable `poolTokenReserves` is assigned the result of `i_poolToken.balanceOf(address(this))`, which retrieves the balance of pool tokens held by the contract. However, this variable is declared but never used anywhere in the code. Unused variables consume storage space and can lead to unnecessary gas costs during contract execution.

Impact: Unused variables increase the storage footprint of the contract, which can lead to higher gas costs for transactions and deployments. Additionally, unused variables can obscure the code's logic, making it harder to understand and maintain.

Proof of Concept:

```
        if (totalLiquidityTokenSupply() > 0) {
            uint256 wethReserves = i_wethToken.balanceOf(address(this));
-            uint256 poolTokenReserves =
i_poolToken.balanceOf(address(this));
```

Recommended Mitigation: To address this issue, you should remove the unused variable `poolTokenReserves` from your code. If the intention behind declaring this variable was to calculate or compare the pool token reserves, ensure that the necessary logic utilizing this variable is implemented correctly. If the variable is indeed not needed, simply removing it will clean up the code and save on gas costs. Here's the revised code without the unused variable: