BLOCKCHAIN IN AUDIT

PRESENTED BY: ANSELMO SANCHEZ

Prepared by: Anselmo Ramon Sanchez Titla

Table of Contents

Protocol Summary

This project is meant to be a permissionless way for users to swap assets between each other at a fair price. You can think of T-Swap as a decentralized asset/token exchange (DEX). T-Swap is known as an Automated Market Maker (AMM) because it doesn't use a normal "order book" style exchange, instead it uses "Pools" of an asset. It is similar to Uniswap. To understand Uniswap, please watch this video: Uniswap Explained

Disclaimer

The Anselmo Ramon Sanchez Titla 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
	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 to the following commit hash:

- Commit Hash: e643a8d4c2c802490976b538dd009b351b1c8dda

Scope

```
./src/
└── PoolFactory.sol
└── TSwapPool.sol
```

Roles

- `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.
- `Users`: Users who want to swap tokens.

Executive Summary

We spent 50 hours with 3 auditors using manual review, aderyn, slither finding the below vulnerabilities classified from high to informational

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	2
Infomational	11
Gas	1
total	19

Findings

High

[H-1] Magic Number in getInputAmountBasedOnOutput Function

Description:

The vulnerability exists in the `TSwapPool::getInputAmountBasedOnOutput` function, which uses a magic number `10000` in the calculation. The use of magic numbers in code is generally problematic as they lack context and can lead to unintended behavior. In this case, the use of `10000` instead of a more appropriate value like `1000` results in a much higher value being returned than intended.

Impact:

This vulnerability has a high impact because it directly affects the amount users are charged in the protocol. By using the magic number `10000`, the function overcharges users in transactions, leading to significant financial loss. Given that this function is part of the `TSwapPool::swapExactOutput`, one of the main functions of the protocol, the vulnerability is likely to affect a large number of users.

Proof of Concept:

The issue arises in the following code:

```
function getInputAmountBasedOnOutput(
    uint256 outputAmount,
```

```
    uint256 inputReserves,
    uint256 outputReserves
)
    public
    pure
    revertIfZero(outputAmount)
    revertIfZero(outputReserves)
    returns (uint256 inputAmount)
{
    // @audit - info magic number
    // @audit - high they are using 10_000 instead of 1_000. They are
stilling a lot of money from the users
    // IMPACT: HIGH users are charge way too much
    // LIKELIHOOD: HIGH this is used in swapExactOutput which is one the main
functions of the protocol
    return
        ((inputReserves * outputAmount) * 10000) /
        ((outputReserves - outputAmount) * 997);
}
```

Section-1: The calculation uses a magic number `10000` instead of `1000`, causing the overcharge issue.

Section-2: The function should be adjusted to use a more appropriate factor (e.g., `1000`) to avoid overcharging users.

Section-3: This vulnerability is especially critical because the TSwapPool::getInputAmountBasedOnOutput function is central to the protocol's operations, and users rely on accurate calculations for fair transactions.

Recommended Mitigation

The recommended mitigation is to replace the magic number `10000` with a more appropriate value such as `1000` in the calculation. This change will ensure that users are charged accurately and fairly.

```
solidity
Copy
function getInputAmountBasedOnOutput(
    uint256 outputAmount,
    uint256 inputReserves,
    uint256 outputReserves
)
    public
    pure
    revertIfZero(outputAmount)
    revertIfZero(outputReserves)
    returns (uint256 inputAmount)
{
    // @audit - correct magic number issue
```

```
    return
-       ((inputReserves * outputAmount) * 10000) /
+       ((inputReserves * outputAmount) * 1000) /
        ((outputReserves - outputAmount) * 997);
}
```

Section-A: The change to `1000` ensures that the calculation is more accurate, preventing overcharging.

Section-B: Updating the function's calculation will improve user trust and prevent financial losses.

Section-C: After applying this fix, users will pay a fair amount according to the reserves and output amount, improving the overall reliability of the protocol.

[H-2] Missing Slippage Protection in `TSwapPool::swapExactOutput`

Description:

The vulnerability is found in the `TSwapPool::swapExactOutput` function where there is a lack of slippage protection. Slippage occurs when the price at which the transaction is executed differs from the expected price. This can be exploited by attackers, especially in the case of MEV (Maximal Extractable Value) attacks, which can lead to significant financial losses. The function does not ensure that the input amount remains within an acceptable range, leaving it vulnerable to fluctuations in price due to network congestion or changes in reserves during the transaction.

Impact:

The impact of this vulnerability is high as it could lead to unintended losses for users. Without slippage protection, users may end up paying more than they intended or expected, as the actual input amount needed to complete the transaction could exceed the intended amount. This vulnerability also opens the door for potential MEV attacks, where attackers could manipulate the transaction by exploiting price changes and causing further losses to users. It could damage the trust in the protocol and cause a loss of funds for the users involved.

Proof of Concept:

The vulnerability exists due to the absence of slippage protection in the following code snippet:

```
function swapExactOutput(
    IERC20 inputToken,
    IERC20 outputToken,
    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
    );
    _swap(inputToken, inputAmount, outputToken, outputAmount);
}
```

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

To mitigate the risk of slippage and protect against MEV attacks, the function should be updated to include slippage protection and a maximum input amount check. By introducing a mechanism to limit the maximum input amount based on the output amount and slippage tolerance, users will have more control over their transactions, and the protocol will be more resilient to price manipulation.

```
function swapExactOutput(
    IERC20 inputToken,
    IERC20 outputToken,
    uint256 outputAmount,
    uint64 deadline,
    uint256 maxInputAmount // Added parameter for maximum input amount
)
    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
        );

+       // Check if inputAmount is within the acceptable range
+       require(inputAmount <= maxInputAmount, "Slippage too high");

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

The new `maxInputAmount` parameter should be passed into the function to set the limit on how much input a user is willing to pay. This mitigates slippage by ensuring the input amount remains within acceptable bounds.

The function now includes a check with `require(inputAmount <= maxInputAmount)` to ensure that the transaction does not proceed if the slippage exceeds the set threshold.

By introducing slippage protection, the protocol reduces the potential for price manipulation, protecting users from potential losses due to fluctuating market conditions or MEV attacks.

[H-3] Incorrect Argument Passed to swapExactOutput in sellPoolTokens

Description:

The vulnerability arises in the `TSwapPool::sellPoolTokens` function where the third argument passed to the `swapExactOutput` function is incorrect. The third argument in `swapExactOutput` should represent the amount of WETH to be sent from the protocol (the output amount), but instead, the `poolTokenAmount` is being used, which is the input amount (the amount of pool tokens being sold). This mistake could lead to unexpected behavior during the swap, as the wrong values are being passed to the swap logic, potentially resulting in incorrect token exchanges.

This is to the fact that the `TSwapPool::swapExactOutput` is called, whereas the `TSwapPool::swapExactInput` is the one that should be called, because users specify the exact amount of input tokens, not output.

Impact:

This vulnerability can result in a miscalculation of the expected amount of WETH to be returned to the caller, which could lead to users receiving an incorrect amount of WETH when selling pool tokens. This could cause financial losses to users as they may receive less WETH than expected, or the transaction may fail due to mismatched amounts being passed to the swap function. Additionally, this could cause the protocol to behave unpredictably, leading to user dissatisfaction and trust issues in the platform.

Proof of Concept:

The issue occurs because the third argument passed to `swapExactOutput` in `sellPoolTokens` is incorrect. Instead of passing the expected output amount (the WETH to be received), the input amount (the pool tokens

being sold) is passed, which is not how the function should operate.

```
function sellPoolTokens(
    uint256 poolTokenAmount
) external returns (uint256 wethAmount) {
    return
        swapExactOutput(
            i_poolToken,
            i_wethToken,
            poolTokenAmount, // This should be the amount of WETH to go out
from the protocol
            uint64(block.timestamp)
        );
}
```

Recommended Mitigation

To fix this vulnerability, the sellPoolTokens function should be updated to correctly pass the expected amount of WETH to be received as the third argument to the swapExactOutput function. The output amount should be calculated based on the pool token amount and the reserves, ensuring that the right amount of WETH is passed to the swap function.

```
function sellPoolTokens(
    uint256 poolTokenAmount
) external returns (uint256 wethAmount) {
+    // Calculate the expected WETH amount based on pool token reserves
+    uint256 wethAmount = calculateWethAmount(poolTokenAmount);

    return
        swapExactOutput(
            i_poolToken,
            i_wethToken,
-           poolTokenAmount,
+            wethAmount, // Corrected to the amount of WETH to be sent from
the protocol
            uint64(block.timestamp)
        );
}
```

The function should calculate the amount of WETH the user should receive for the poolTokenAmount. This can be done by using the appropriate formula based on the pool reserves.

By passing the correct output amount (wethAmount) to swapExactOutput, the transaction will execute as intended, and users will receive the correct amount of WETH.

This change ensures that the protocol behaves as expected and prevents any financial loss for the user due to incorrect token swaps.

Consider changing the implementation to use `swapExactInput` instead of `swapExactOutput`. Note that this 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) {
+    // Calculate the expected WETH amount based on pool token reserves
+    uint256 wethAmount = calculateWethAmount(poolTokenAmount);

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

additionally it might be wise to add a deadline to the function, as there is currently no deadline

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

Description:

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

- `x`: The balance of the pool token
- `y`: The balance of the WETH token
- `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 every 10 swaps due to the extra incentive in the `_swap` function. Meaning that over time the protocol funds will be drained

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.

The following 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)
}
```

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 until all the protocol funds are drained

▶ Proof of Code

```
    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();

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

        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));
```

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

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

        assert(actualDeltaY == expectedDeltaY*10);
    }
```

Recommended Mitigations

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, or we should set aside tokens in the same way we do with fees.

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

Medium

[M-1] Unused Deadline Parameter in Deposit Function can cause user misunderstanding

Description:

The `TSwapPool::deposit` function in the `TSwapPool` contract accepts a `deadline` parameter, but this parameter is not used in the function logic. The purpose of a deadline is typically to prevent transactions from being executed after a specific time, ensuring that deposits happen within an acceptable timeframe. However, in this case, the `deadline` parameter is provided, but not referenced anywhere in the function, leading to confusion about its purpose and potentially misleading users into thinking it impacts the deposit process.

Impact:

A user may assume that providing a `deadline` will cause their deposit to fail if the deadline is exceeded. However, since the parameter is not used in the code, a user could set a deadline expecting the deposit to fail after that time, but the deposit would still go through. This discrepancy could lead to severe disruptions in functionality, where users may expect transactions to fail but they are processed anyway, causing a misalignment between user expectations and actual behavior. The potential for such confusion is high, as the `deadline` parameter is often associated with critical transaction timing.

Proof of Concept:

The `deposit` function is shown below, where the `deadline` parameter is declared but never used:

```
function deposit(
    uint256 wethToDeposit,
    uint256 minimumLiquidityTokensToMint,
    uint256 maximumPoolTokensToDeposit,
    uint64 deadline
)
```

Since the deadline parameter isn't utilized, users may believe their deposits are subject to a time limit that isn't enforced.

Recommended Mitigation:

To mitigate this issue, one of the following actions should be taken:

Use the `deadline` parameter: Implement logic that enforces the `deadline`, ensuring that deposits are only allowed if the current time is before the specified `deadline`. This can be done by adding a check for the deadline in the function.

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

This change ensures that the deposit function enforces the specified time limit.

Remove the `deadline` parameter: If the `deadline` feature is not required, the best course of action is to remove the parameter entirely to avoid confusion. This prevents any misunderstandings regarding its intended functionality.

```
function deposit(
     uint256 wethToDeposit,
     uint256 minimumLiquidityTokensToMint,
+    uint256 maximumPoolTokensToDeposit
-    uint256 maximumPoolTokensToDeposit,
-    uint64 deadline
)
```

In either case, clarifying the functionality (or lack thereof) of the `deadline` parameter will help align user expectations with the actual behavior of the contract.

Low

[L-1] Unused Function Parameter in swapExactInput (Root Cause + Impact)

Description:

The vulnerability exists in the `TSwapPool::swapExactInput` function, where the return type `uint256 output` is specified, but the `output` variable is never actually defined or used within the function. This results in an unused function parameter, which is confusing and could lead to errors in the future if the function signature is not updated. Additionally, it can lead to unnecessary gas costs if the function signature remains as is.

Impact:

This issue has a low impact in terms of security but can cause confusion and inefficiency in the codebase. The unused function parameter increases the complexity of the contract and could lead to further mistakes or misunderstandings in the future. Furthermore, the incorrect return type may cause issues if someone attempts to interact with the function, expecting the return value to be properly set.

Proof of Concept:

The issue arises in the following code, where the `output` variable is specified in the return type but is never defined or used:

```
function swapExactInput(
    IERC20 inputToken,
    uint256 inputAmount,
    IERC20 outputToken,
    uint256 minOutputAmount,
    uint64 deadline
)
    public
    revertIfZero(inputAmount)
    revertIfDeadlinePassed(deadline)
    returns (uint256 output)
{
    uint256 inputReserves = inputToken.balanceOf(address(this));
    uint256 outputReserves = outputToken.balanceOf(address(this));

    uint256 outputAmount = getOutputAmountBasedOnInput(
        inputAmount,
        inputReserves,
        outputReserves
    );

    if (outputAmount < minOutputAmount) {
```

```
            revert TSwapPool__OutputTooLow(outputAmount, minOutputAmount);
        }

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

Recommended Mitigation

The recommended mitigation is to remove the unused return type uint256 output from the function signature, as the function does not return any value. This will clarify the function's behavior and reduce unnecessary complexity.

```
    function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    )
        public
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
    -   returns (uint256 output)
    {
        uint256 inputReserves = inputToken.balanceOf(address(this));
        uint256 outputReserves = outputToken.balanceOf(address(this));

        uint256 outputAmount = getOutputAmountBasedOnInput(
            inputAmount,
            inputReserves,
            outputReserves
        );

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

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

Removing the unused return type uint256 output clarifies the function’s purpose and ensures the function signature accurately reflects its behavior.

This change reduces unnecessary complexity and potential sources of confusion for developers interacting with the contract.

After this fix, the function becomes simpler and more understandable, ensuring that it no longer misleads developers regarding its return type and expected behavior.

[L-2] `TSwapPool::LiquidityAdded` event has parameters out of order causing event to emit incorrect information

Description:

When `TSwapPool::LiquidityAdded` event is emmited in `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

Recomended Mitigations

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

Informational

[I-1] Unused Error Declaration in PoolFactory Contract

Description:

The `PoolFactory::PoolFactory__PoolDoesNotExist` error is declared in the `PoolFactory` contract but is never used anywhere in the contract. This could lead to unnecessary code bloat, as the error does not provide any functionality or contribute to the contract's behavior. It's important to remove unused variables or errors to keep the codebase clean and efficient.

Impact:

The unused error declaration may confuse developers or auditors reviewing the contract. It increases the size of the bytecode without offering any value and could potentially be a source of misunderstanding regarding the contract's error-handling logic. Additionally, it may lead to security risks if developers mistakenly rely on or assume the presence of such errors in their implementation, without any actual usage or handling.

Proof of Concept:

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

The error PoolFactory::PoolFactory__PoolDoesNotExist is declared, but it is not invoked in any function within the contract, making it redundant.

Recommended Mitigation

To mitigate the issue, the unused error declaration should be removed from the PoolFactory contract. This will improve code clarity, reduce bytecode size, and eliminate unnecessary confusion. If the error was intended for future use, ensure that it is properly integrated into the contract's logic.

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

By removing this unused error, the contract will be more streamlined, and there will be no ambiguity regarding unused elements.

[I-2] Missing Zero Address Check in Constructor

Description:

The `PoolFactory::constructor` lacks a check to ensure that the provided `wethToken` address is not the zero address. This oversight could lead to unintended behavior if a contract is deployed with a zero address for `wethToken`.

Impact:

If the `PoolFactory::constructor` is deployed with a zero address for the `wethToken` parameter, any operations that rely on this token address will fail or behave unpredictably. It could lead to lost funds or contract malfunction, particularly in functions that interact with the `wethToken` address.

Proof of Concept:

In the current implementation of the `PoolFactory::constructor`, there is no validation to ensure that `wethToken` is not the zero address:

```
constructor(address wethToken) {
    i_wethToken = wethToken;
}
```

To exploit this, a malicious actor could deploy the contract with the wethToken parameter set to `0x0000000000000000000000000000000000000000`, which could break subsequent contract logic that expects a valid token address.

Recommended Mitigation

To mitigate this issue, a validation should be added in the PoolFactory::constructor to ensure that the provided `wethToken` address is not the zero address. This will prevent the contract from being deployed with an invalid

token address.

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

With this modification, the contract will throw an error during deployment if the zero address is provided for wethToken, ensuring that the contract cannot be deployed in an invalid state.

Additionally, consider adding additional checks for the validity of the token address, such as ensuring that the address points to a contract with the expected token interface.

[I-3] Incorrect Function Call in Token Symbol Retrieval

Description:

The code attempts to concatenate the string `"ts"` with the result of `IERC20(tokenAddress).name()` to form a liquidity token symbol. Do you mean to call `IERC20(tokenAddress).symbol()`.

Impact:

Using .name() could cause inefficiencies if the token's name is excessively long. It is likely more efficient to use .symbol() instead.

Proof of Concept:

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

Recommended Mitigation:

To mitigate this issue, the recommended action is to use the symbol() function, which typically returns a shorter, more appropriate string for token identification.

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

Additionally, it is suggested to consider verifying that the token address implements the ERC-20 interface properly before making calls to the symbol() function, ensuring robust behavior and avoiding potential runtime errors.

[I-4] Insufficient Indexed Parameters in Event Declaration.

Description:

In the `TSwap::Swap` event declaration, there are more than three parameters, but only one parameter (`swapper`) is indexed. According to the Solidity best practices, it is recommended to index all the parameters that will be frequently used for event filtering, especially when the parameters are critical to the event's logic, such as `tokenIn`, `amountTokenIn`, `tokenOut`, and `amountTokenOut`. By indexing these additional parameters, it becomes easier and more efficient to filter events in external tools, such as when using `web3.js` or `ethers.js`.

Impact:

The current event declaration will make it inefficient to filter or search for logs based on the non-indexed parameters. Since only the `swapper` address is indexed, querying or tracking swaps based on tokens or amounts will require additional processing and might result in higher gas costs for retrieving events. This also impacts the ability to easily track or monitor specific swaps by token types or amounts. Additionally, if external applications or users rely on querying these events, they may experience slower performance and increased complexity.

Proof of Concept:

The following code shows the `TSwap::Swap` event, where only one parameter (`swapper`) is indexed, despite the presence of additional relevant parameters:

```
event Swap(
    address indexed swapper,
    IERC20 tokenIn,
    uint256 amountTokenIn,
    IERC20 tokenOut,
    uint256 amountTokenOut
);
```

In this code, only swapper is indexed, while tokenIn, amountTokenIn, tokenOut, and amountTokenOut could be indexed to improve event querying efficiency.

Recommended Mitigation:

To resolve this, it is recommended to index additional parameters in the Swap event declaration. Specifically, the tokenIn, amountTokenIn, tokenOut, and amountTokenOut should be indexed for better filtering and performance in event logs.

```
event Swap(
    address indexed swapper,
-    IERC20 tokenIn,
+    IERC20 indexed tokenIn,
     uint256 amountTokenIn,
-    IERC20 tokenOut,
+    IERC20 indexed tokenOut,
     uint256 amountTokenOut
);
```

This ensures that the tokens involved in the swap and their respective amounts are indexed, allowing for more efficient queries based on these parameters.

Additionally, be mindful of the gas costs when adding indexed parameters. Although indexing provides better filtering, it increases the cost of emitting the event, so only index parameters that are likely to be queried frequently.

[I-5] Usage of Magic Number in getOutputAmountBasedOnInput Function

Description:

The vulnerability in the `TSwapPool::getOutputAmountBasedOnInput` function lies in the unclear use of a "magic number" (`997` in the calculation for `inputAmountMinusFee`). A magic number is a constant used directly in code without explanation, making it difficult for developers to understand its purpose and increasing the risk of future errors when maintaining or updating the code. The code snippet `inputAmount * 997` is part of the formula for determining the output amount, but the meaning of the number `997` is not explained, leading to potential confusion.

Impact:

Using a magic number can introduce bugs or unexpected behavior in the smart contract. If the value `997` is incorrect, misinterpreted, or needs to be changed, it will be difficult for future developers to identify and modify it properly. Without proper documentation or clarity about the reason for using this constant, the logic of the contract may break, potentially leading to incorrect calculations or unintended consequences in liquidity pools. This can directly affect users' transactions, leading to loss of funds or inaccurate calculations in liquidity provisioning.

Proof of Concept:

The following section highlights where the magic numbers `997` and `1000` are used without explanation in the code.

```
        uint256 inputAmountMinusFee = inputAmount * 997;
        uint256 numerator = inputAmountMinusFee * outputReserves;
        uint256 denominator = (inputReserves * 1000) + inputAmountMinusFee;
```

Recommended Mitigation:

Clarify the Use of the Magic Number:

The first step in addressing this vulnerability is to clarify the purpose of the magic numbers `997` and `1000`. If it represents a fee, it should be documented, or better yet, placed in a constant variable with a meaningful name. For example:

```
uint256 constant FEE_MULTIPLIER = `997`;
```

Then, use this constant in the calculation to make the code more understandable:

```
uint256 inputAmountMinusFee = inputAmount * FEE_MULTIPLIER;
```

After renaming the constant, ensure that the smart contract is tested thoroughly to ensure the behavior remains as intended. In particular, validate the calculations for inputAmountMinusFee to confirm the correct fee multiplier is used.

It's recommended to add comments or documentation explaining why the constant `997` is used in the contract, especially if it is related to business logic or specific optimizations.

[I-6] Missing Natspec Documentation in swapExactInput Function (Root Cause + Impact)

Description:

The vulnerability exists in the `TSwapPool::swapExactInput` function, where the function lacks proper Natspec documentation. Natspec comments are essential for improving code readability, understandability, and ensuring that other developers or auditors can easily interpret the purpose and behavior of the function. In this case, the absence of Natspec documentation on a critical function makes it harder for auditors or developers to fully understand its behavior, which can lead to errors or missed vulnerabilities.

Impact:

This vulnerability has a moderate impact in terms of code readability and maintainability. While the functionality may not directly cause an immediate bug, the lack of Natspec documentation increases the risk of misinterpretation, potential misuse, and complicates future auditing and review processes. This could lead to difficulties in detecting vulnerabilities or issues when updates or changes are made to the smart contract.

Proof of Concept:

The issue arises in the following code, where there is no Natspec documentation:

```
function swapExactInput(
    IERC20 inputToken,
    uint256 inputAmount,
    IERC20 outputToken,
    uint256 minOutputAmount,
    uint64 deadline
)
    public
    revertIfZero(inputAmount)
    revertIfDeadlinePassed(deadline)
    returns (uint256 output)
{
    uint256 inputReserves = inputToken.balanceOf(address(this));
    uint256 outputReserves = outputToken.balanceOf(address(this));

    uint256 outputAmount = getOutputAmountBasedOnInput(
        inputAmount,
        inputReserves,
        outputReserves
    );

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

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

Recommended Mitigation

The recommended mitigation is to add proper Natspec documentation to the TSwapPool::swapExactInput function. This will provide clear explanations of the function's purpose, parameters, return values, and any edge cases or requirements.

```
/// @notice Swaps a specified amount of input tokens for output tokens,
ensuring that the output is at least the minimum required.
/// @dev This function calculates the output amount based on the input
reserves and output reserves, ensuring that the user receives a fair exchange
rate.
/// @param inputToken The token being exchanged (input token).
/// @param inputAmount The amount of the input token being swapped.
/// @param outputToken The token being received (output token).
/// @param minOutputAmount The minimum acceptable amount of the output token
that the user is willing to receive due to slipage
/// @param deadline The timestamp after which the transaction will no longer
be valid.
```

```
function swapExactInput(
    IERC20 inputToken,
    uint256 inputAmount,
    IERC20 outputToken,
    uint256 minOutputAmount,
    uint64 deadline
)
    public
    revertIfZero(inputAmount)
    revertIfDeadlinePassed(deadline)
    returns (uint256 output)
{
    uint256 inputReserves = inputToken.balanceOf(address(this));
    uint256 outputReserves = outputToken.balanceOf(address(this));

    uint256 outputAmount = getOutputAmountBasedOnInput(
        inputAmount,
        inputReserves,
        outputReserves
    );

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

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

Adding Natspec documentation will enhance code clarity and ensure that all developers and auditors understand the function's intended behavior.

This documentation should include descriptions for each parameter, the expected behavior, and potential revert scenarios, particularly for minOutputAmount and deadline.

By including clear and concise Natspec comments, the maintainability of the code will improve, reducing the risk of errors during future code reviews or updates.

[I-7] Function Visibility in swapExactInput should be external instead of public.

Description:

The vulnerability exists in the `TSwapPool::swapExactInput` function, where the function is marked as `public` but it is not intended to be used other than outside of the contract. The function contains logic that doesn't require internal calls, and thus, it should have a more restrictive visibility modifier, such as `external`.

Impact:

This vulnerability has a moderate impact on the security and maintainability of the smart contract. By marking the function as `public` when it should be `external`.

Proof of Concept:

The issue arises in the following code, where the function is marked as `public` but should be `external` instead:

```
function swapExactInput(
    IERC20 inputToken,
    uint256 inputAmount,
    IERC20 outputToken,
    uint256 minOutputAmount,
    uint64 deadline
)
    // @audit - info this should be external, is not used in other parts of
the same smart contract
    public
    revertIfZero(inputAmount)
    revertIfDeadlinePassed(deadline)
    returns (uint256 output)
{
    uint256 inputReserves = inputToken.balanceOf(address(this));
    uint256 outputReserves = outputToken.balanceOf(address(this));

    uint256 outputAmount = getOutputAmountBasedOnInput(
        inputAmount,
        inputReserves,
        outputReserves
    );

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

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

Recommended Mitigation

The recommended mitigation is to change the visibility of the TSwapPool::swapExactInput function from `public` to `external`. This will limit the function's accessibility to only external callers, reducing the contract's exposure to unnecessary risks.

```
function swapExactInput(
```

```
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    )
        // @audit - corrected visibility to external
    +   external
    -   public
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
        returns (uint256 output)
    {
        uint256 inputReserves = inputToken.balanceOf(address(this));
        uint256 outputReserves = outputToken.balanceOf(address(this));

        uint256 outputAmount = getOutputAmountBasedOnInput(
            inputAmount,
            inputReserves,
            outputReserves
        );

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

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

[I-8] Missing Deadline Parameter in Natspec for swapExactOutput.

Description:

The vulnerability exists in the `TSwapPool::swapExactOutput` function's natspec comment, where the `deadline` parameter is not included in the function's documentation. Natspec comments provide crucial information for developers and auditors to understand the function's parameters and behavior, and the omission of `deadline` in the natspec can lead to confusion or misinterpretation of the function’s purpose, especially in time-sensitive operations like swaps where a deadline is involved.

Impact:

The impact of this vulnerability is low in terms of security but can lead to operational errors or misunderstandings. Developers interacting with the contract may overlook the importance of the `deadline` parameter, which is critical to prevent transactions from being executed beyond a certain time. This oversight could result in failed transactions or an incorrect user experience.

Proof of Concept:

The issue arises in the following code, where the `deadline` parameter is not documented in the natspec comment:

```
/*
 * @notice figures out how much you need to input based on how much
 * output you want to receive.
 *
 * Example: You say "I want 10 output WETH, and my input is DAI"
 * The function will figure out how much DAI you need to input to get 10 WETH
 * And then execute the swap
 * @param inputToken ERC20 token to pull from caller
 * @param outputToken ERC20 token to send to caller
 * @param outputAmount The exact amount of tokens to send to caller
 * @audit-info missing deadline param in natspec
 */

function swapExactOutput(
    IERC20 inputToken,
    IERC20 outputToken,
    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
    );

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

Recommended Mitigation

The recommended mitigation is to update the natspec comment to include the deadline parameter. This will provide clear documentation for developers and auditors, ensuring they understand the importance of the deadline in this function and its role in time-sensitive operations.

```
 /*
  * @notice figures out how much you need to input based on how much
  * output you want to receive.
  *
  * Example: You say "I want 10 output WETH, and my input is DAI"
  * The function will figure out how much DAI you need to input to get 10 WETH
  * And then execute the swap
  * @param inputToken ERC20 token to pull from caller
  * @param outputToken ERC20 token to send to caller
  * @param outputAmount The exact amount of tokens to send to caller
+* @param deadline The time by which the transaction must be executed
  */

 function swapExactOutput(
     IERC20 inputToken,
     IERC20 outputToken,
     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
     );

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

Updating the natspec comment to include the deadline parameter ensures that the function's documentation is complete and clear, reducing confusion for developers and auditors.

This update will help developers understand that the deadline is a critical part of the function, preventing unintended transactions from being executed after the set deadline.

The updated documentation enhances the clarity of the function's purpose, ensuring that time-sensitive parameters like deadline are not overlooked during contract interactions.

[I-9] Magic Number Used in Price Calculation (Root Cause + Impact)

Description:

The vulnerability in `TSwapPool::getPriceOfOneWethInPoolTokens` and `TSwapPool::getPriceOfOnePoolTokenInWeth` functions arises from the use of a "magic number" (1e18) in the calculation of token prices. Magic numbers refer to hardcoded values with no explanation, making the code less readable and prone to errors or misunderstandings. In this case, the use of `1e18` as the input amount for the price calculations is unclear and should be replaced with a more meaningful, self-explanatory value or a defined constant.

Impact:

Using magic numbers reduces the clarity of the code and makes it harder for developers or auditors to understand the logic behind these functions. If the value `1e18` was used for specific reasons related to precision or the scale of tokens, it should be clearly documented or defined as a constant. Without this context, other developers may inadvertently change the value or misinterpret the code, leading to potential issues in price calculations or future code modifications. Furthermore, this can introduce risks when interacting with the contract, as users may misunderstand how prices are derived or assume that the code behaves differently.

Proof of Concept:

The issue stems from the use of the hardcoded value `1e18` in both `TSwapPool::getPriceOfOneWethInPoolTokens` and `TSwapPool::getPriceOfOnePoolTokenInWeth` functions. The current implementation uses `1e18` as the input amount to calculate the price ratio between WETH and pool tokens, but the rationale behind this specific value is unclear.

```
// Function to get the price of one WETH in terms of pool tokens
function getPriceOfOneWethInPoolTokens() external view returns (uint256) {
    return
        getOutputAmountBasedOnInput(
            1e18, // Magic number
            i_wethToken.balanceOf(address(this)),
            i_poolToken.balanceOf(address(this))
        );
}

// Function to get the price of one pool token in terms of WETH
function getPriceOfOnePoolTokenInWeth() external view returns (uint256) {
    return
        getOutputAmountBasedOnInput(
            1e18, // Magic number
            i_poolToken.balanceOf(address(this)),
            i_wethToken.balanceOf(address(this))
        );
}
```

Recommended Mitigation

To mitigate this issue, the value `1e18` should either be replaced with a clearly defined constant or documented in the code to provide better clarity. If it is used for scaling purposes, the constant should be named accordingly, such as `TSwapPool::PRECISION_SCALE`, and defined at the beginning of the contract. This will enhance code readability and reduce the risk of misunderstandings in the future.

```
+ // Define a constant for precision scale
+ uint256 constant PRECISION_SCALE = 1e18;

// Function to get the price of one WETH in terms of pool tokens
function getPriceOfOneWethInPoolTokens() external view returns (uint256) {
    return
        getOutputAmountBasedOnInput(
-           1e18,
+           PRECISION_SCALE, // Use the defined constant
            i_wethToken.balanceOf(address(this)),
            i_poolToken.balanceOf(address(this))
        );
}

// Function to get the price of one pool token in terms of WETH
function getPriceOfOnePoolTokenInWeth() external view returns (uint256) {
    return
        getOutputAmountBasedOnInput(
-           1e18,
+           PRECISION_SCALE, // Use the defined constant
            i_poolToken.balanceOf(address(this)),
            i_wethToken.balanceOf(address(this))
        );
}
```

The magic number `1e18` should be replaced with a constant like `PRECISION_SCALE` to improve code clarity and avoid confusion.

The constant `PRECISION_SCALE` should be defined at the start of the contract and be used in relevant calculations to standardize the scaling factor.

Documenting or defining magic numbers as constants enhances code maintainability and prevents accidental misuse or misinterpretation of values.

[I-10] Inadequate Error Message for Minimum WETH Deposit.

Description:

The vulnerability arises from a custom error message triggered when the `TSwapPool::wethToDeposit` amount is less than the `TSwapPool::MINIMUM_WETH_LIQUIDITY`. The issue is that the `TSwapPool::MINIMUM_WETH_LIQUIDITY` is a constant in the `TSwapPool` smart contract. However, the custom error might be misleading or unnecessarily emit this information, making it more complex than needed.

Impact:

If the `TSwapPool::wethToDeposit` is lower than `TSwapPool::MINIMUM_WETH_LIQUIDITY`, the error is raised. The constant `TSwapPool::MINIMUM_WETH_LIQUIDITY` is already known and does not need to be passed as part of the error. This can lead to unnecessary gas costs by including the constant value in the error message, which is redundant because the value is fixed and can be inferred by users or developers directly from the contract. Additionally, this unnecessary complexity can be a potential source of confusion for users interacting with the contract.

Proof of Concept:

The `TSwapPool::wethToDeposit` is emitted with the constant `TSwapPool::MINIMUM_WETH_LIQUIDITY` in the following condition:

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

Recommended Mitigation

To mitigate this issue, we can modify the error handling to exclude the constant `TSwapPool::MINIMUM_WETH_LIQUIDITY` from being emitted. This can be done by removing it from the parameters in the revert statement. The error message can still indicate the issue without including the constant value.

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

By passing only the `wethToDeposit` value in the revert message, we reduce unnecessary gas usage and simplify the error message while still informing users of the invalid deposit amount.

[I-11] Liquidity Minting Function Does Not Follow the CEI Pattern

Description:

The function `deposit` in the `TSwapPool` smart contract contains a potential issue related to the order of operations. Specifically, the assignment of the `TSwapPool::liquidityTokensToMint` variable occurs after making an external call to `_addLiquidityMintAndTransfer`. This violates the Checks-Effects-Interactions (CEI) pattern, which is a security best practice. The CEI pattern suggests that state changes (effects) should be made before interacting with external contracts to prevent reentrancy attacks or unintended side effects.

Impact:

If the `TSwapPool` smart contract continues to execute in this manner, it may expose the contract to security vulnerabilities. Specifically, because the `TSwapPool::liquidityTokensToMint` variable is updated after an external call, malicious actors could potentially exploit this order and cause unexpected behavior or vulnerabilities in the contract. This could lead to a failure in the minting process, where liquidity tokens are not properly assigned or could be manipulated by a malicious actor.

Proof of Concept:

The issue lies in the following portion of the `TSwapPool::deposit` function:

```
else {
    // This will be the "initial" funding of the protocol. We are starting
from blank here!
    // We just have them send the tokens in, and we mint liquidity tokens
based on the weth
    _addLiquidityMintAndTransfer(
        wethToDeposit,
        maximumPoolTokensToDeposit,
        wethToDeposit
    );
    // In the function above we make an external call
    // and then we update a local variable, it is probably fine because it's
not a state variable
    // q does not follow CEI pattern
    liquidityTokensToMint = wethToDeposit;
}
```

The vulnerability arises from the order of these two actions, where the state variable liquidityTokensToMint is updated after the external call to _addLiquidityMintAndTransfer.

Recommended Mitigation:

To resolve this vulnerability, the assignment of the TSwapPool::liquidityTokensToMint variable should be done before calling the _addLiquidityMintAndTransfer function. This would ensure that the contract follows the CEI pattern and minimizes the risk of potential vulnerabilities.

```
else {
    // This will be the "initial" funding of the protocol. We are starting
```

```
from blank here!
    // We just have them send the tokens in, and we mint liquidity tokens
based on the weth
+    liquidityTokensToMint = wethToDeposit;  // Update state variable before
external call
    _addLiquidityMintAndTransfer(
        wethToDeposit,
        maximumPoolTokensToDeposit,
        wethToDeposit
    );
-   liquidityTokensToMint = wethToDeposit;  // Update state variable before
external call
}
```

This change ensures that the state variable liquidityTokensToMint is updated before the external call, following the CEI pattern, and improving the security and stability of the contract.

Gas

[G-1] Unused Local Variable in Deposit Function (Root Cause + Impact)

Description:

The vulnerability stems from the declaration of a local variable `TSwapPool::poolTokenReserves` in the `TSwapPool::deposit` function, which is never used within the function.

Impact:

The main issue is that the variable `TSwapPool::poolTokenReserves` is declared but not used, which results in unnecessary gas consumption during execution. This could increase transaction costs without providing any value to the function's logic, leading to inefficiency in the smart contract. Removing or reusing the variable would optimize the gas cost, improving the contract's performance and reducing the unnecessary computational overhead.

Proof of Concept:

In the `TSwapPool::deposit` function, the local variable `TSwapPool::poolTokenReserves` is declared to store the balance of `TSwapPool::i_poolToken` but is never used after its declaration:

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

Recommended Mitigation

To mitigate the issue, the `TSwapPool::poolTokenReserves` variable should either be removed if it's not necessary or integrated into the function logic if it was intended to be used. By doing so, we can avoid wasting gas on an unused calculation.

If the variable is truly not needed, simply remove it:

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

This change would result in more efficient gas usage, reducing transaction costs and making the TSwapPool::deposit function cleaner and more optimized.